These de doctorat de
l'Universite Pierre et Marie Curie

Specialite : Informatique

presented by

Francis Maes

for the Degree of Doctor of Computer Science
at Universite Paris VI — Pierre-et-Marie-Curie

Choose-Reward Algorithms

Learning the inference procedure

soutenue publiquement le XX XX XXXX
devant le jury compose de

XX XX	XX	*examinateur*
Patrick Gallinari	Professeur a l'Universite Pierre et Marie Curie (Paris 6)	*directeur de these*
XX XX	XX	*examinateur*
XX XX	XX	*examinateur*
XX XX	XX	*rapporteur*
XX XX	XX	*rapporteur*

Experience is the name everyone gives to their mistakes, *Oscar Wilde*

Contents

List of Figures

List of Tables

List of Algorithms

List of CR-algorithms

1
Introduction

Contents

During the last decade, statistical machine learning has reached a high level of maturity to solve classification and regression problems with efficient and theoretically well-founded algorithms. This success story has led researchers to move on more advanced learning problems. Classification and regression respectively consist in predicting simple discrete and continuous outputs. In many real-world applications however, learning components have to deal with much more complex data. There are mainly two approaches to tackle complex learning problems. We can either create new models for each new task or we can adopt the reductionist approach, which consists in combining simple learning models to form solutions to more complex tasks. In this thesis, we adopt this last approach and propose a new framework to define complex learning problems in a unified, efficient and simple way: CR-algorithms.

1.1 Introduction

Many practical solutions to deal with complex real-world applications of machine learning rely on the idea of simplifying the problem enough to use classical learning machines, such as probabilistic classifiers. Let us consider the example of a multi-language character recognition system. Such a complex problem can be decomposed into simpler sub problems, such as language classifiers, font classifiers, letter classifiers, layout recognizers and so forth. These sub-problems can then be tackled with traditional learning machine tools.

Systems using machine learning involve a *learning* phase before being able to perform *inference*. Learning aims at training the system, given examples of its desired behavior. *Inference* is the process of running the whole system: to perform character recognition on a given document in our example. In many complex applications of machine learning, *learning is disconnected from inference*. First, each classifier is trained independently, then, the classifiers are grouped to form the whole system. In this approach, each classifier only has a local vision of the system and the specificities of inference are not taken into account during learning. This leads to important

Learning
Inference

learning problems such as error propagation among chains of classifiers. We claim that a key step to learn such complex systems correctly is to model the whole system as a single learning problem instead of modeling each of its learning-based components independently. This way, the empirical behavior and weaknesses of the system can be taken into account during learning. Instead of optimizing each classifier locally, this approach suggests to optimize the whole system globally. Therefore, the dynamic of the system – what happens between two learning-based decisions ? – must be taken into account during learning, through simulation for example.

Integrate learning and inference

The major idea underlying CR-algorithms is to integrate learning and inference into a single object. A CR-algorithm jointly represents an inference procedure and an associated learning problem. The inference procedure is a piece of code that can perform machine-learning based decisions. The learning problem defines a *quality* criterion that should be maximized to find the optimal decision sequences. An important aspect of our approach – that may be disturbing – is that CR-algorithms put less the focus on learning machine, than on the environment in which learning is used. Since computers execute programs sequentially, we put a particular emphasis on the temporal nature of code execution, *i.e.* the quality of an inference procedure depends on the whole sequence of decisions that are performed during its execution. In order to properly model inference procedures, CR-algorithms are defined as sequential decision-making problems and are formalized within the well-establish framework of Markov decision processes.

1.2 Overview

We here briefly introduce the CR-algorithm formalism with a simple example: automatic recognition of handwritten words. In this application, inputs are sequences of bitmaps representing handwritten letters and outputs are words, *i.e.* sequences of recognized letters. Recognizing a word can be done in multiple ways. A simple choice for the inference procedure is *left-to-right* labeling, which consists in recognizing each letter sequentially, from left to right. In this approach, at each time step t, the t-th letter is recognized on the basis of the corresponding input and on the previous recognized letters.

Figure 1.1 illustrates the use of CR-algorithms on the word recognition example. We now briefly describe each step of the proposed methodology:

1) **Problem** The starting point is a problem that involves some kind of learning. In this manuscript, we essentially consider *structured prediction* problems. Structured prediction problems are supervised learning problems – the aim is to learn a mapping given a set of input and associated desired outputs – that involve structured outputs, such as sequences, trees or graphs. Handwritten word is a structured prediction problem where inputs and outputs are respectively sequences of bitmaps and sequences of recognized letters. Structured prediction covers numerous applications ranging from bio-informatics to image processing or web-mining. Although most of this thesis is about structured prediction, CR-algorithms is a general framework that can be relevant to other fields. As an example, we propose prospective work on the learning-for-search problem, which consists in using machine learning to improve the efficiency of combinatorial search algorithms.

2a) **CR-algorithm** A CR-algorithm jointly defines an inference procedure and an associated learning objective:

- **Inference procedure** Inference is a piece of code that solves the problem and that relies on machine-learning based decisions. In order to deal with such decisions, we introduce a new instruction called *choose*. In our example, the *choose* (line 4) defines

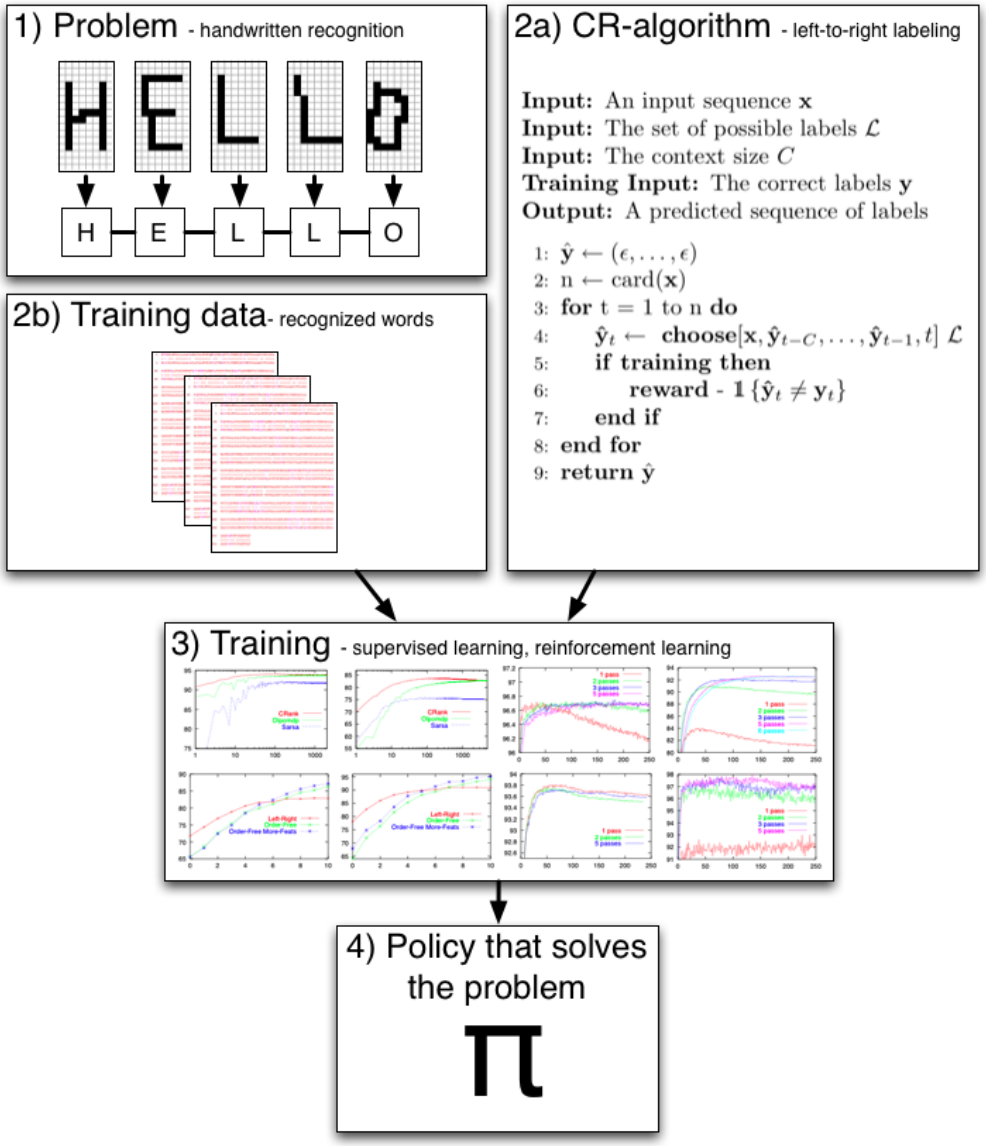


Figure 1.1: **Illustration of the proposed methodology.** 1) The starting point is a problem involving some kind of learning. We consider here a handwritten word recognition task, which consists in recognizing words given 8×16 black-and-white bitmaps corresponding to segmented letters. 2) In order to solve the problem, we have to provide two components: a CR-algorithm (2a) and training data (2b). We here give the CR-algorithm that recognizes letters from left to right. 3) Given the training data and the CR-algorithm, we learn the inference procedure. 4) Learning results in a policy π useable in conjunction with the inference procedure to recognize any new incoming word.

a classification problem: given the input bitmap $\mathbf{x}$ and the previous recognized letters $\hat{\mathbf{y}}_{t-C}, \ldots, \hat{\mathbf{y}}_{t-1}$, choose one label among the set of possible labels $\mathcal{L}$. The core of the CR-algorithm iterates over the letters of the word to recognize (line 3) and invokes this classification problem for each letter. Once all the letters are recognized, the inference procedure returns the predicted word $\hat{\mathbf{y}}$ (line 9).

– **Learning objective** The learning objective of CR-algorithms is defined thanks to *reward* instructions. The aim of learning is to find the sequences of decisions that maximize the sum of rewards. In our example, each time that a letter is wrongly predicted, we give a reward of -1 (line 6), *i.e.* the aim is to take decisions in order to minimize the number of wrongly predicted letters.

2b) **Training data** This step is classical in machine learning: in order to train the inference procedure, the user has to provide a set of training examples. In the case of handwritten word recognition, a training example is a correctly recognized word.

3) **Training** A key aspect of our work is that we develop tools to learn CR-algorithm that support different kind of supervision knowledge. We mostly deal with *perfect supervision* or *imperfect supervision*. Perfect supervision assumes that, for all training examples, we can compute the best decisions on the long-term in any state that can be reached by the inference procedure. This is the case of handwritten word recognition: whatever the previous predictions are, the best decision is to recognize the next letter correctly. Not all applications can be perfectly supervised in this manner. A key aspect of our work is to introduce reinforcement learning techniques to learn CR-algorithm with less a-priori knowledge. We also introduce *post-supervision*, which corresponds to situations where the supervision is only known once the inference procedure is terminated.

4) **Policy that solves the problem** Training results in a policy π: a function that maps states of the inference procedure to decisions. The CR-algorithm, in conjunction to this policy, can be used to recognize any new incoming word.

1.3 Contributions

CR-algorithm formalism

The primary contribution in this thesis is the development of the CR-algorithm formalism, to integrate inference procedures and associated learning problems (see Chapter 4). This framework is, up to our knowledge, the first attempt to explicitly describe learning-based programs as sequential decision-making problems. The main features of our formalism are:

- **Well-founded formalism** Since we model programs that are executed sequentially, we develop a natural connection between CR-algorithms and the well-established Markov decision process formalism. Thanks to this connection, we develop a proper understanding of the temporal nature of programs described with CR-algorithms.

- **Existing learning algorithms** Another fundamental benefit the connection with Markov decision process is to provide a range variety of existing policy learning algorithms that can be applied, without modifications, to learn CR-algorithms. We identify different levels of supervision that corresponds to different applications of CR-algorithms. We show that the structured prediction field provides some algorithms useable to deal with *perfect supervision* situations. In order to deal with *imperfect supervision*, we introduce algorithms from the field of approximated reinforcement learning.

- **Simplicity** CR-algorithms rely on an imperative description of learning problems. This description is close to this of classical programs and learning is introduced in an intuitive manner, which makes the formalism easy to understand for any computer scientist. The learning algorithms that we propose for CR-algorithms are also relatively simple. In our experiments, we only make use of simple linear approximators trained stochastically and show successful results.

- **Expressiveness** CR-algorithms describe complex Markov decision processes in a very concise way. For example, describing the Markov decision processes corresponding to the CR-algorithm given in Section 1.2 takes at least one or two pages of text, whereas the CR-algorithm defines it completely in 9 lines. Chapter 5, Chapter 6 and Chapter 7 provide numerous CR-algorithms corresponding to complex and hard-to-describe Markov decision processes. Appendix B presents an on going work, which is to create a programming language to develop CR-algorithms. The feasibility of such a language gives it whole sense to CR-algorithms. The work presented in this thesis can be though as a preliminary study to create this *learning-based* programming language.

- **Efficient applicative solutions** CR-algorithms lead to efficient solutions, which are most of time competitive with state-of-the-art models, while having much lower complexities. Thanks to their expressiveness, CR-algorithms makes it possible to develop efficient solutions for complex problems, as shown by Chapter 6. As an example of this efficiency, our approaches are able to transform HTML pages into XML trees containing thousands of nodes in less than one second.

Structured prediction with reinforcement learning

From the point of view of structured prediction, our work can be seen as a follow up of [Daumé III et Marcu, 2005, Daumé III *et al.*, 2006]. In addition to these founding approaches, we provide a description formalism and wide the range of possible supervision situations. Our main contribution to the field of structured prediction is to demonstrate the relevancy of reinforcement learning to deal with imperfect supervision. Chapter 5 shows that, although requiring much more training iterations, reinforcement learning algorithm are competitive with supervised approaches for sequence labeling tasks. In Chapter 6, we develop a complex structured prediction task, where perfect supervision is not available. We give successful results for reinforcement learning approaches in this complex domain, on which most previous models fail.

CR^{ank}: action ranking

From the point of view of reinforcement learning, we introduce the idea of *policy as a ranking machine*. Learning to rank is a problem that has received an increasing interest over the last years in the machine learning community. A ranking machine is a learning machine that ranks possible alternatives given situations. We propose to view policy learning as a ranking problem. This leads to policies that, given the current state, rank all the possible actions and select the top-ranked one. We propose a policy-learning algorithm called CR^{ank}, to learn policies through action ranking.

Reinforcement learning sucess

CR-algorithms lead to very particular reinforcement learning problems, which are very large, mostly deterministic but not always fully observable. We give numerous experimental results that show the success of reinforcement learning with simple linear approximators in these large problems. In particular, all our experiments lead to good generalization properties, which is a satisfying result for reinforcement learning.

1.4 Manuscript layout

This thesis is presented in four parts. The first part gives background on supervised learning and sequential decision-making problems and overviews the domain of structured prediction. This domain motivated most of the work leading to CR-algorithms. Although our formalism is not restricted to structured prediction problems, these problems have a central place in this thesis. The second part of the thesis describes the proposed formalism. The third part of the thesis, comprising Chapter 5 though Chapter 7, discusses the applications of CR-algorithms and describes numerous experimental results. The final part of the thesis concludes and gives an overview of the perspectives raised by this work.

The chapters in this thesis are organized as follows:

Part I: Background

Chapter 2 introduces relevant background to understand the remainder of the thesis. It focuses on supervised learning and sequential decision-making and puts a particular emphasis on learning-based methods for sequential decision-making. These methods are able to deal with very large decision processes and are being used as building blocks in the remainder of the thesis.

Chapter 3 describes the field of structured prediction and makes an overview of state-of-the-art models from this field. We distinguish global models from incremental models. The former focus on what good outputs look like, while the latter directly model how to construct good outputs. We show that the latter approach has several advantages over the former. Our work can be considered as follow-up to existing work on incremental models.

Part II: CR-algorithm Framework

Chapter 4 presents the CR-algorithms formalism. It first describes the formalism and its connection with Markov decision processes. It then overviews existing learning algorithm that can be used to learn CR-algorithms. Finally, it presents CR^{ank}, our algorithm to learn policies with action ranking.

Part III: Applications

Chapter 5 describes the application of CR-algorithms to sequence labeling, which is is the generic task of assigning labels to the elements of a sequence. This aim of this chapter is threefold: firstly to introduce CR-algorithms on a simple task, secondly to compare CR-algorithms to state-of-the-art work and thirdly to experiments CR-algorithms with perfect supervision. We give multiple solutions to the sequence-labeling problem, including two original approaches: order-free labeling and multiple-pass labeling with CR-algorithms. The latter approach especially leads to nice results: it is competitive with the best baselines on all our datasets and clearly outperforms them in some experiments, while being very fast both in training and inference.

Chapter 6 describes a challenging structured prediction task: ordered labeled tree transformation. The aim of this chapter is twofold: firstly to demonstrate that CR-algorithms provide relatively simple solutions to complex learning problems and secondly to experiment CR-algorithms in problems with imperfect supervision. CR-algorithms lead to an original family of algorithms to perform tree transformation. One of our most spectacular results is that our approach, which has a linear complexity in the number of tree leaves,

outperforms the baseline model, which has a cubic complexity in the number of tree leaves. In practice, our approach runs about 50 times faster than the baseline on small datasets and scales well to all our large real-world datasets.

Chapter 7 shows that CR-algorithms can be relevant to other problems than structured prediction. The aim of this chapter is twofold: firstly to show the relevancy of CR-algorithms to learning-for-search problems and secondly to experiment CR-algorithms with post-supervision. We give several examples of CR-algorithms dedicated to learning-for-search problems and introduce a general approach to learn such CR-algorithms. We also provide prospective experiments to show the promise of this approach on two search problems.

Part IV: Perspectives

Chapter 8 concludes this work and discusses the perspectives it opens. The perspectives are organized according to three directions: automating the CR-algorithm learning system, extending the CR-algorithm formalism and developing new applications of CR-algorithms.

Appendices

Appendix A presents Nieme, the open-source machine-learning library developed in the context of this thesis. This library is distributed under the GPL license and is published in the Journal of Machine Learning Research.

Appendix B introduces a major on-going work: the creation of aCR-algorithm based programming language. This programming language relies on the work exposed in this thesis and its existence definitely validates the relevancy of CR-algorithms.

Appendix C discusses our work on defining feature generators in the context of Nieme.

Appendix D gives a summary of the mathematical notations used in this thesis.

1.5 Personal bibliography

In the context of this thesis, I participated to the following contributions:

International journals

[1] *Maes (Francis)*
Nieme: Large-Scale Energy-Based Models
Journal of Machine Learning Research, 2009

International conferences

[2] *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
Sequence Labelling with Reinforcement Learning and Ranking Algorithms
18th European Conference on Machine Learning (ECML'07)

[3] Wisniewski (Guillaume), Denoyer (Ludovic), *Maes (Francis)*, Gallinari (Patrick)
Probabilistic Model for Structured Document Mapping
5th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM'07)

International workshops with proceedings

[4] *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
Application of Reinforcement Learning to Structured Prediction
8th European Workshop on Reinforcement Learning, 2008

[5] *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
XML Structure Mapping
In proceedings of 5th Workshop of the INitiative for the Evaluation of XML Retrieval (INEX'06)

[6] Gallinari (Patrick), Wisniewski (Guillaume), *Maes (Francis)*, Denoyer (Ludovic)
Stochastic Models for Document Restructuration
In proceedings of ECML Workshop on Relationnal Machine Learning, 2005

French journals

[7] Wisniewski (Guillaume), *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
Modèle probabiliste pour l'extraction de structures dans les documents Web
Documents Numeriques, pages 89–107, 2007

French conferences

[8] *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
Apprentissage de conversion de documents semi-structurés à partir d'exemples.
5ème Conférence en Recherche d'Information et Applications (CORIA 2008)

[9] Wisniewski (Guillaume), Denoyer (Ludovic), *Maes (Francis)*, Gallinari (Patrick)
Modèle probabiliste pour l'extraction de structures dans les documents semi-structurés, Application aux documents Web.
3ème Conférence en Recherche d'Information et Applications (CORIA 2006)

Submitted

[10] *Maes (Francis)*, Denoyer (Ludovic), Gallinari (Patrick)
Structured Prediction with Reinforcement Learning
Submitted to Machine Learning Journal, 2008

[2] is the work on sequence labeling preceding the contributions of Chapter 5. [6] and [7] describe early work on tree-transformation with generative models. [3] describes our first tree-transformation models with global and incremental structured prediction approaches. We propose to use the Sarsa reinforcement-learning algorithm to learn tree transformation policies in [5] and [8]. We widen our conclusions in [4] and [10] by discussing the relevancy of reinforcement learning to structured prediction in general. [1] describes our machine-learning toolbox Nieme.

Note that all these publications where written before the creation of the CR-algorithm formalism. Most of them rely on formal, often complex, descriptions of MDPs. This thesis revisits this work within the framework of CR-algorithm, which leads to a much more concise presentation.

2

Background

Contents

An important topic in AI concerns the ability of computer programs to *learn from examples*. When looking at human learning, one can observe a lot of different learning tasks: learning to walk, learning to speak, learning to recognize words and, more generally, several forms of behavior learning. Formalizing all these concepts is far from being trivial and multiple scientific domains, which focus on different aspects of the problem, have naturally emerged over the past sixty years.

In this chapter, we focus on two kind of learning problems: supervised learning problems (SL) and sequential decision-making (SDM) problems. SL, presented in Section 2.1, aims at learning a function from training data. The training data consists in pairs of input objects and associated desired output objects. The main challenge of SL is that the learned function should be able to *generalize* to new input objects that were not in the training data.

SDM, which is discussed in Section 2.2, is the problem of taking decisions in a given environment. Usually, an environment is described in terms of states, actions and rewards and the aim is to learn a function that map states to actions in order to maximize a long-term reward. SDM raises multiple challenges: a learning challenge, but also an exploration challenge. In order to find a good behavior, SDM algorithms usually perform some form of search over all the possible ways to act and to observe their long-term quality.

Since SL and SDM correspond to old and very active scientific communities, being exhaustive on these subjects would require more than a book. This chapter is an introduction to these fields

and is only intended to provide the sufficient tools to understand the remainder of this manuscript. For more detail, you can refer to one of the multiple existing books introducing SL ([Vapnik, 1995] for example) and SDM ([Sutton et Barto, 1998] or [J. Si et II, 2004] for example).

2.1 Supervised Learning

This section introduces the SL problem and classical solutions to this problem. We start with a simple use-case that gives an overview of the methodology: *email filtering*. We then describe several tasks that fit in the SL framework in Section 2.1.2. We formalize the learning problem and introduce the *empirical minimization risk* principle in Section 2.1.3. Common learning methods following this principle are discussed in Section 2.1.4.

2.1.1 A use case: email filtering

We consider here the problem of email filtering, where incoming emails have to be classified as being spam emails or normal emails. In this problem, we want to learn a classification function f:

$$f : email \rightarrow \{spam, not\text{-}spam\}$$

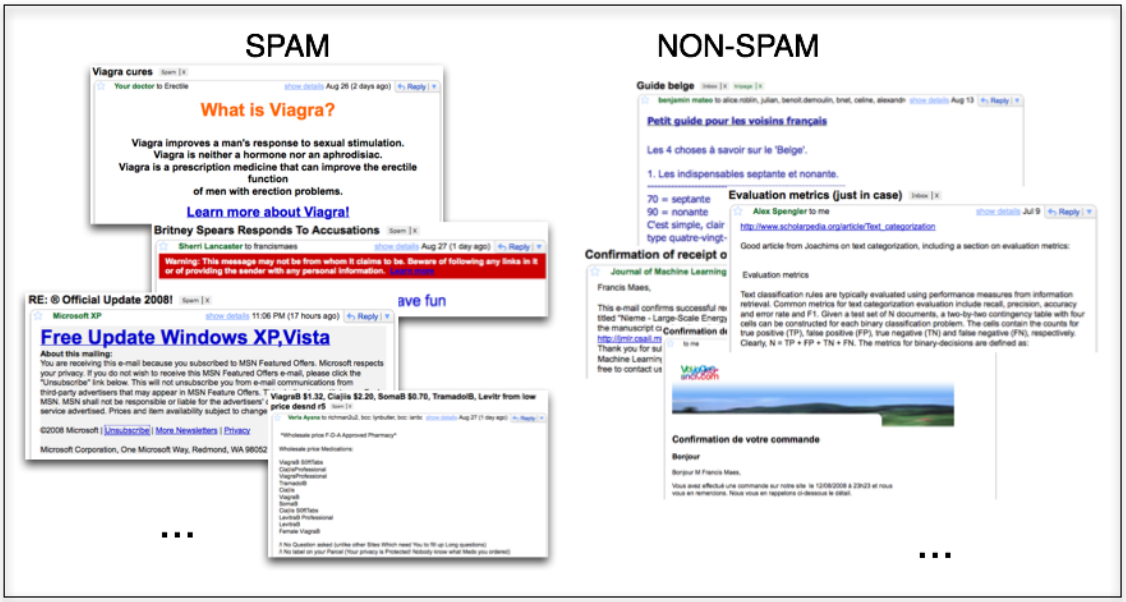


Figure 2.1: **Email filtering training set.** The training set is made of examples containing an input and its associated output. Here, inputs are emails and outputs are categories among *spam* and *not-spam*.

Training dataset

Methodology When using SL, the first step is always to gather a training set. As illustrated in Figure 2.1, such a training set is composed of inputs (emails) and associated desired outputs

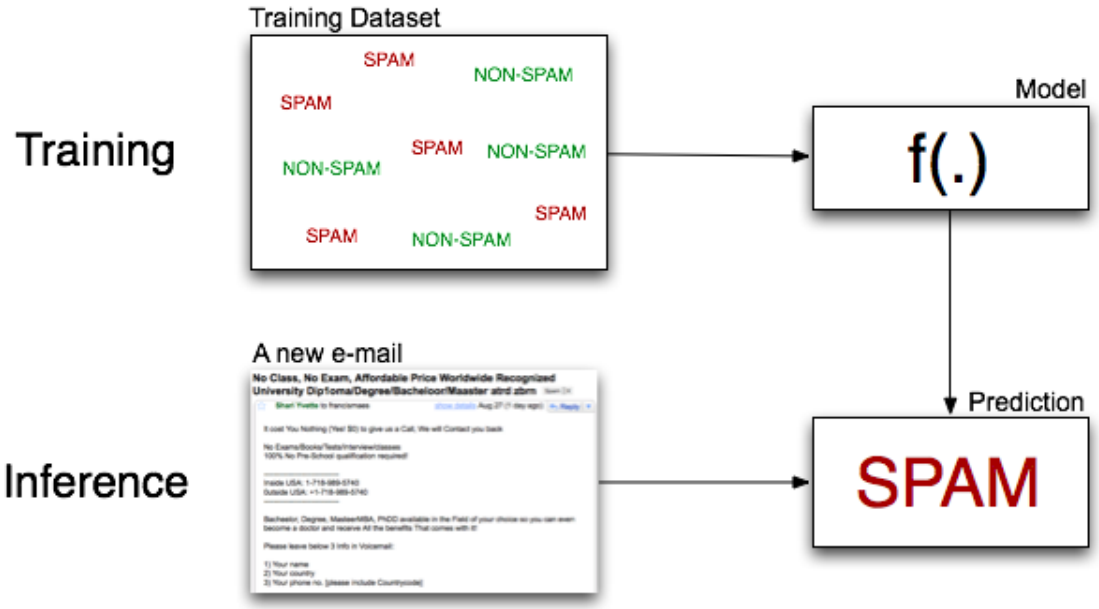


Figure 2.2: **Supervised Learning: Training and Inference.** Given the training set, training aims at finding a model, representing the function f. Given this model, we can make predictions for any new incoming input.

(*spam* or *not-spam*). The training set needs to be representative of the real-world use of the function. The desired outputs are usually constructed by human experts.

Given the training set, we can perform the *training* phase. The result of training is a *model* that represents the learned function $f(.)$. This function can then be applied to perform *inference* on new input objects, which result in *predictions*. The global architecture of the SL methodology is summarized in Figure 2.2. Usually, training is performed only once and may use a lot of computing resources. However, in most of cases, inference will be used frequently and its computation should be done reasonably fast.

Training
Inference

Representation In order to fit in the SL framework, we have to choose a representation for the input objects. Depending on the learning methods, different kind of representations may be used: vectors, attribute-value lists, first-order logic, relational representations and so forth. In this manuscript, we only use vectorial representations: functions that map input objects to vectors. In the case of email filtering, described in Figure 2.3, we define:

Representation

$$\phi : email \rightarrow \mathbb{R}^d$$

The elements of the $\phi(.)$ vectors are often called *features* and may describe any aspect of the input objects. For example, when dealing with textual documents, a common choice is to have one feature per possible word, each feature corresponding to a word frequency.

Features

Choosing an appropriate representation for a given learning problem is a difficult task and usually requires expertise of the domain. From a practical point of view, it is often observed that the quality of learning crucially depends on the representation choice. There must be

Figure 2.3: **Supervised Learning: Vectorial representation of an email.** The representation function ϕ encodes input objects in the form of feature vectors. Each feature describes an arbitrary aspect of the input.

enough features to accurately describe the properties of the input objects, but too many features may lead to hard learning problems. Given the current state-of-the-art of SL, creating a good representation is still much more an art than a science.

The Perceptron

A solution example A funding method of SL is the Rosenblatt's Perceptron (1957). The Perceptron performs inference using a *linear* function: [1]:

$$f(\mathbf{x}) = \begin{cases} spam & \text{if } \langle \theta, \phi(\mathbf{x}) \rangle \text{ is positive} \\ not\text{-}spam & \text{otherwise} \end{cases}$$

where θ are the parameters learned during the training phase and $\langle ., . \rangle$ denotes the dot product operator:

$$\langle \theta, \phi(\mathbf{x}) \rangle = \sum_{i=1}^{d} \theta_i \phi(\mathbf{x})_i$$

Linear separation

Figure 2.4 illustrates inference with the Perceptron in the case of email filtering. With its linear architecture, the Perceptron learns the parameters of a hyper-plan in the feature-space-, separating the examples into *spams* and *not-spams*. The classification is then performed by looking at the sign of the *activation*, which is the result of the dot product between the parameters and the descriptions.

Perceptron Training

Training the Perceptron is done in an *online* way: the training examples are processed one at a time. For each example, we compute the prediction given the current parameters. If this prediction is correct, we continue with the next example. Otherwise, we apply a correction to

[1] We omit here the bias parameter for the sake of clarity. A simple way to introduce a bias is add a unit feature, whose value is always one, to each example.

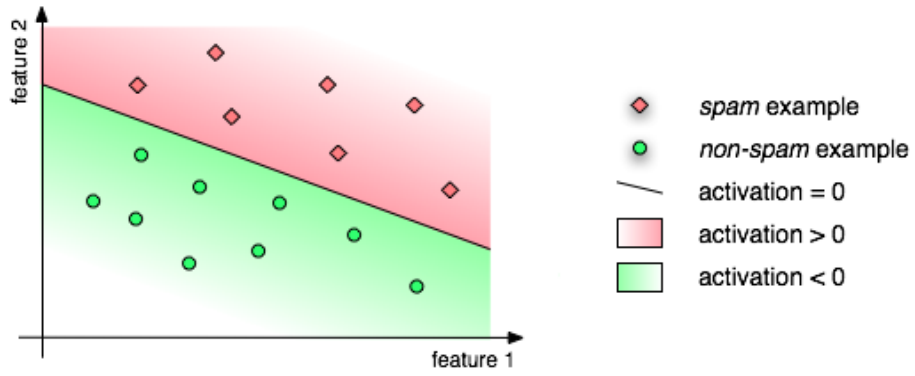


Figure 2.4: **Linear Classification.** We assume minimalist descriptions containing two features *feature 1* and *feature 2* and a unit features whose value is always one. The feature-space is split by a linear separation $\theta_1 f_1 + \theta_2 f_2 + \theta_3 1$. In order to predict the class of an example, we look at which side of the space we are.

the parameters in order to enforce the correct answer:

$$\theta \leftarrow \begin{cases} \theta + \phi(x) & \text{if x is a spam} \\ \theta - \phi(x) & \text{otherwise} \end{cases}$$

If the training dataset is *linearly separable*, *i.e.* there exists a hyper-plane that perfectly separates *spams* from *not-spams* in the feature space, then the Perceptron will converge to such a solution [Rosenblatt, 1958]. Usually, training is iterated over the complete training dataset, until the number of errors cannot be lowered more.

2.1.2 Supervised Learning Tasks

The aim of SL is to induce a function $f : \mathcal{X} \to \mathcal{Y}$ that maps inputs[2] $\mathbf{x} \in \mathcal{X}$ to outputs[3] $\mathbf{y} \in \mathcal{Y}$, given a set of training examples $D = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i \in [1,n]}$. SL is a general framework that encompasses several different tasks. In the following, we briefly review the most frequent ones.

Discrete outputs A central SL problem is the *binary classification* problem: learning problems with two possible outputs. By convention, these two outputs are often qualified as *positive* and *negative* and denoted $+1$ and -1. With these notations, the output space of binary classification is: *Binary Classification*

$$\mathcal{Y} = \{-1, +1\}$$

[2]The inputs are sometimes called *observations*.
[3]The outputs are sometimes called *labels*, or *classes* in the discrete case.

Examples of binary classification problems include the following:

- Email filtering: the example introduced above.
- Medical testing: predicting if a patient has a certain disease or not.
- Gender recognition: predicting if an image represents the face of a woman or a man.
- Ad-hoc retrieval: predicting if a document is relevant or not for a given query[4].

Binary classification is particularly interesting for multiple reasons: it has a simple formalization, it allows extensive theoretical insights and it is historically the first learning problem where computers achieved effective learning.

Figure 2.5: **A multi-class learning problem: sentiment classification.** Each face represent a training example described by two features *feature 1* and *feature 2*.

Multi-class classification

When there are more than two possible discrete outputs, we fall into *multi-class classification* problems. Here are a few examples of multi-class classification problems:

- Sentiment classification: *e.g.* classifying a human face as being *happy*, *neutral*, *sad* or *surprised* (Figure 2.5).
- Genre classification: *e.g.* predicting the genre of a piece of music with the following categories: $\mathcal{Y} = \{classical, rock, techno, funk, rap\}$.
- Email routing: *e.g.* predicting the category of an email between *professional*, *personal* and *spam*.
- Text categorization: *e.g.* predicting the topic of a document between *sports*, *politics* and *people*.

Regression

Continuous outputs Learning problems with scalar outputs are known as *regression* problems:

$$\mathcal{Y} \subseteq \mathbb{R}$$

The regression problem is illustrated in Figure 2.6. Regression problems include the following:

[4]In this example, the input objects are (document, query) pairs.

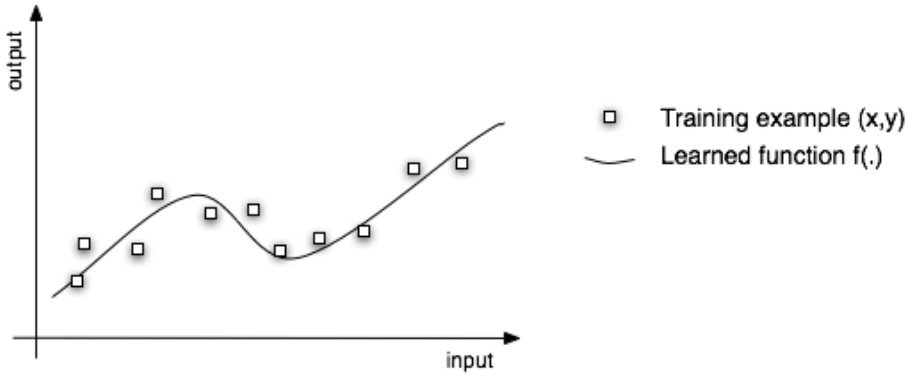


Figure 2.6: **Learning tasks: Regression.**

- Market prediction: predicting the volatility of a stock market.
- Predicting the weight of a person given its age, its height and various informations of her way of life.

There also exist learning problems which need to predict multiple scalar outputs in the same time ($\mathcal{Y} \subseteq \mathbb{R}^n$). Such problems are called *multi-dimensional regression* problems. *Multi-dimensional Regression*

Learning to rank In the last decade, there has been a growing interest on the *ranking* problem[5]. The aim of ranking is to learn preferences about alternatives [Cohen *et al.*, 1998]. We adopt the *label ranking* formalism, where a training example is made of a situation $\mathbf{x}_s$ and a set of preferences over alternatives $\mathbf{x}_a$ in this situation. Figure 2.7 gives the example of a search-engine, where a situation is a query and alternatives are documents that may be relevant for that query. As shown by the figure, the aim of ranking is to learn a scoring function $F(\mathbf{x}_s, \mathbf{x}_a; \theta)$ that defines an order over alternatives[6].

Ranking is close to regression in the sense that we learn a function with scalar outputs. The main difference is that only the order over objects induced by this function matters. Any re-scaling or offset may be applied to a ranking function without changing its meaning.

Ranking is currently receiving a growing interest in many communities. Examples of the ranking problem include:

- Information retrieval: learning to order a set of documents by relevance *w.r.t.* a given query. This example is illustrated in Figure 2.7 in the context of a web search engine.
- Automatic summarization: learning to order sentences of a text by *summarization power*.
- Recommendation systems: *e.g.* learning to order movies to recommend a user.
- Preference learning: *e.g.* learning customer's preferences from transaction data of supermarkets.

[5]Many different names still exist for this learning problem: preference learning, ordering, label ranking, ordinal regression, *etc.*

[6]Formally, this order can be seen as the prediction $\mathbf{y} \in \mathcal{Y}$, where $\mathcal{Y}$ is the set of all possible orders.

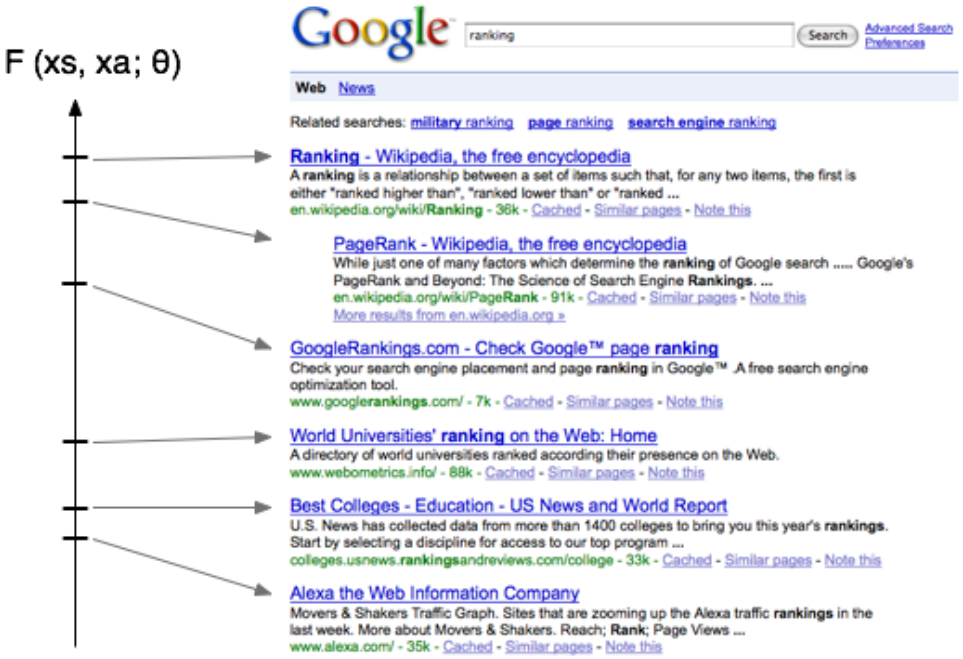


Figure 2.7: **Learning tasks: Ranking.** In information retrieval, a situation $\mathbf{x}_s$ is a query and alternatives $\mathbf{x}_a$ are documents that may be relevant for that query. The alternatives are sorted by the learned function $F(\mathbf{x}_s, \mathbf{x}_a; \theta)$.

Beyond classification, regression and ranking In many different fields such as biology, natural language processing, image processing or chemistry, data may be naturally described as structured objects like sequences, trees, lattices or graphs. Learning problems with arbitrary outputs are called *structured prediction* problems and form one of the main topics of this manuscript. In the following chapters, we introduce methods that reduce hard prediction problems into much more simpler classification, regression or ranking problems. We will show that these three kinds of learning problems can be though as basic building blocks that allow constructing complex learning systems.

Structured prediction

2.1.3 Risk Minimization

Supervised learning aims at learning a function $f : \mathcal{X} \to \mathcal{Y}$ that *performs well* on a given learning problem. Formally, such a learning problem is defined by a joint distribution over input-output pairs $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$. In order to measure the quality of predictions, we assume the existence of a task-specific loss function:

Loss function

$$\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$$

When predicting the correct output, such a loss function has a null value. Otherwise, $\Delta(\hat{\mathbf{y}}, \mathbf{y})$ quantifies how bad it is to predict $\hat{\mathbf{y}} = f(\mathbf{x})$ instead of $\mathbf{y}$. A simple loss function is the 0/1 loss, which equals one if the prediction is incorrect:

0/1 loss

$$\Delta^{0/1}(\hat{\mathbf{y}}, \mathbf{y}) = \mathbb{1}\{\hat{\mathbf{y}} \neq \mathbf{y}\}$$

where $\mathbb{1}\{b\}$ is the indicator function whose value is one if b is true and zero otherwise.

Expected Risk The expectation of loss over the distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ is called the *expected risk* and is defined as follows:

$$R(f) = \mathsf{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}}\{\Delta(f(\mathbf{x}),\mathbf{y})\} \tag{2.1}$$

The expected risk can be related to the task-dependent learning objective by appropriately choosing the loss function. For example, in classification with the 0/1 loss, $R(f)$ is the expectation of classification errors with function f.

Given a function space $\mathcal{F}$, SL aims at selecting the function $f \in \mathcal{F}$ that minimizes the expected risk:

Expected Risk Minimization

$$f^* = \operatorname*{argmin}_{f\in\mathcal{F}} R(f)$$

In practice, the distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ is unknown, so that the expected risk quantity cannot be computed. However, we usually have access to a set of training examples $D = (\mathbf{x}^{(i)},\mathbf{y}^{(i)})_{i\in[1,n]}$. These examples are assumed to be independently and identically drawn (i.i.d.) from $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$.

Training Examples

Empirical Risk The training examples allows us to *approximate* the true risk with the *empirical risk*:

$$\hat{R}(f) = \frac{1}{n}\sum_{i=1}^{n}\Delta(f(\mathbf{x}^{(i)}),\mathbf{y}^{(i)})$$

Selecting the function f^* that minimizes the empirical risk is known as the principle of *empirical risk minimization*:

Empirical risk minimization

$$f^* = \operatorname*{argmin}_{f\in\mathcal{F}} \hat{R}(f) \simeq \operatorname*{argmin}_{f\in\mathcal{F}} R(f)$$

Regularization The relation between empirical risk minimization and expected risk minimization has a central place in modern statistical learning [Vapnik, 1999]. A classical phenomenon when minimizing the empirical risk is that we may perfectly learn the training examples but *generalize* badly to new examples. This is known as *overfitting* and is illustrated in Figure 2.8 (left). Overfitting occurs when the model becomes too strongly tailored to the particularities of the training set. In order to avoid it, we need a way to control the capacity of the learning method.

Overfitting

A flexible way to control overfitting is inspired from the Ockham's razor principle. This principle is often paraphrased as *All other things being equal, the simplest solution is the best.* In other terms, among different functions f whose empirical risks are not significantly different, we should choose the *simplest* f. The common way to introduce such a preference for simpler models is to use a *regularization term* $\Omega(.)$. Such a function returns high scores for complex models and low scores for simple models. Learning then aims at finding a function, which is a good compromise between low empirical risk and simplicity. Finding such a function is known as the *regularized empirical risk minimization* principle[7]:

Regularization term

Regularized empirical risk minimization

$$f^* = \operatorname*{argmin}_{f\in\mathcal{F}} \hat{R}(f) + \lambda\Omega(f) \tag{2.2}$$

where λ is a parameter that allows to control the tradeoff between empirical risk minimization and regularizer minimization.

[7] This is also known as *structural risk minimization.*

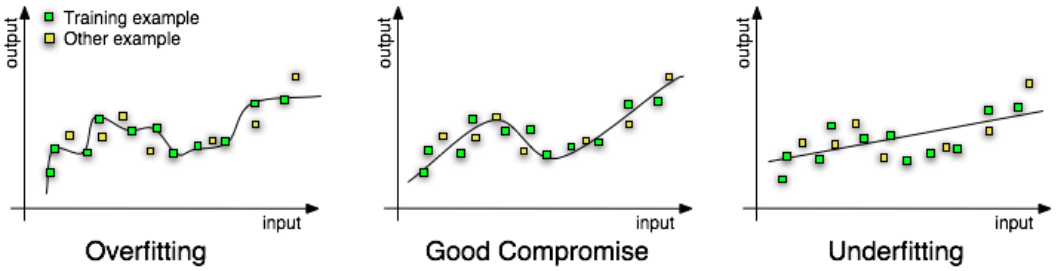


Figure 2.8: **Overfitting and Regularization in a one-dimensional regression problem.**

Figure 2.8 shows the result of learning for three values of λ. On the left, the λ parameter is null and no importance is given to the simplicity of f, this leads to overfitting. The middle case corresponds to a good value of λ: we have learned a simple function that still gives a good mean accuracy. The right part of the figure corresponds to a big λ value: we are only looking for a simple function at the price of accuracy.

2.1.4 Learning Methods

A huge number of SL methods have been developed. A traditional point of view is to classify them into symbolic and numeric methods. Symbolic methods are based on discrete function spaces (*e.g.* decision trees or ensemble of rules) while numeric methods learn in continuous function spaces (*e.g.* artificial neural networks, support vector machines or bayesian statistics). In this following, we only focus on numerical methods[8].

Parameterized functions

Numerical methods Numerical learning methods rely on parameterized functions: the prediction function f is fully defined by a set of scalar parameters $\theta \in \mathbb{R}^d$. In such models, minimizing the regularized empirical risk aims at finding the best parameters θ:

$$\theta^* = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \hat{R}(f_\theta) + \lambda \Omega(f_\theta) \tag{2.3}$$

where f_θ is the prediction function corresponding to parameters θ. In general, such parameterized prediction functions are chosen to be *continuous*: a small change on θ leads to a small change on f_θ. This allows using various numerical optimization techniques to perform training.

Discrete outputs

In order to deal with discrete outputs, numerical methods usually compute *activation* levels $F_\theta(\mathbf{x}) \in \mathbb{R}$ for all possible class. In this approach, predicted classes are those that have the highest activation. A well-known function to compute such activations is the linear function, which computes the dot product between the input's description and the parameters:

$$F_\theta(\mathbf{x}) = \langle \theta, \phi(\mathbf{x}) \rangle$$

In the case of binary classification, the sign of the activation determines the predicted class (-1 or $+1$):

$$f_\theta^{binary}(\mathbf{x}) = sign(F_\theta(\mathbf{x}))$$

[8]Many of these methods are also qualified as *statistical learning* methods.

In order to treat multiclass problems, we have a set of parameters $\theta_{\mathbf{y}}$ per possible class $\mathbf{y} \in \mathcal{Y}$[9]. Prediction is then performed by searching the class which has the highest activation:

$$f_\theta^{multiclass}(\mathbf{x}) = \underset{\mathbf{y}\in\mathcal{Y}}{\operatorname{argmax}} F_{\theta_{\mathbf{y}}}(\mathbf{x})$$

Losses Many loss functions are hard to optimize directly. For example, the 0/1 loss is not continuous (and so not derivable either), which makes the solving of Equation 2.2 intractable. In such cases, many methods minimize a related loss that has better mathematical properties-. Alternatives losses are often upper bounds of the original losses, so that the results of both minimization problems are equivalent. *Alternative Losses*

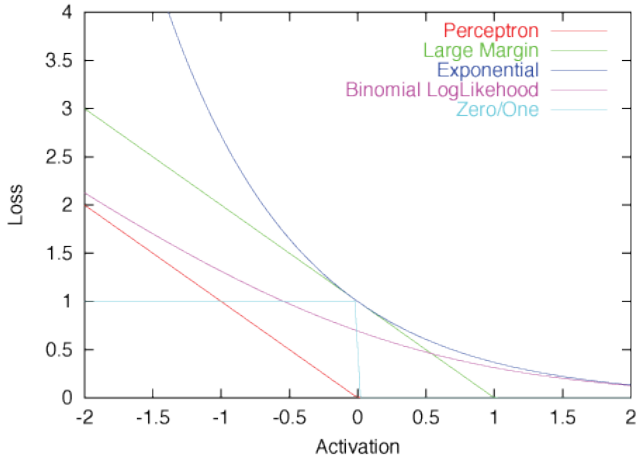


Figure 2.9: **Common alternatives of the 0/1 loss.**

Multiple alternative losses of $\Delta^{0/1}$ are given in Figure 2.9 in the context of binary classification with $\mathbf{y} = +1$. In this case, positive activations lead to correct predictions. The losses have either theoretic or historical justifications:

- The 0/1 loss is the theoretical objective: there is one error when the activation is negative.
- When interpreting the Rosenblatt's Perceptron algorithm in the formalism of empirical risk minimization, each training update can be seen as a stochastic gradient descent step with the *Perceptron loss*.
- The *large-margin loss*[10] enforces a minimum *margin* between the positive examples and the negative examples. Enforcing such a margin leads to stronger theoretical guaranties [Vapnik, 1995]. Furthermore, a lot of experimental results show that large-marge methods improve the generalization performance. The large-margin loss is the loss that is optimized by the well-known Support Vector Machines [Cristianini et Shawe-Taylor, 2000].
- It is possible to interpret the boosting method as an empirical risk minimization method with the *exponential loss*. Boosting [Schapire, 1990] can then be seen as a closed-form solution of the risk minimization problem.

[9] We do not consider *one-against-one* approaches here.
[10] This loss is also known as the *hinge loss*

- Maximum entropy classifiers and the logistic regression models learn log-linear conditional probabilities over classes given inputs. When comparing them with empirical risk minimization techniques, it can be shown that they perform minimization of the *binomial loss* [S. et A, 1985, Hastie *et al.*, 2001].

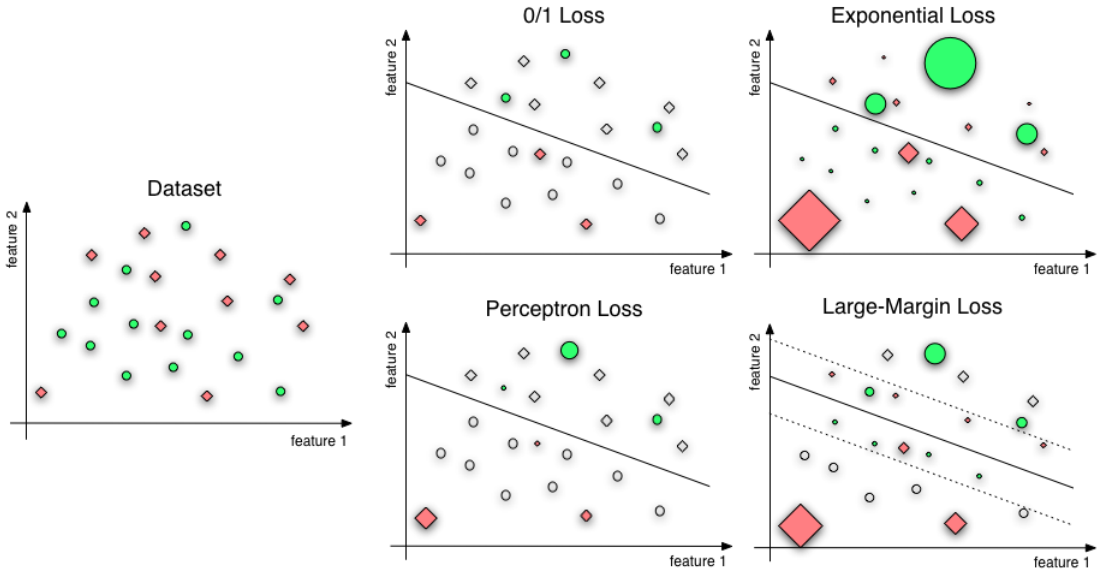


Figure 2.10: **Losses illustrated on a binary classification task.** Empty shapes correspond to examples with a null loss. Otherwise, the size of the filled shapes reflects the amount of loss the examples suffer from. In this example, ideally, all the green examples should be below the line and the red examples should be above.

Figure 2.10 shows how various losses penalize the training examples on a binary classification task. The 0/1 loss penalizes all the errors in the same way, whereas the other loss functions take the activation into account: the farther examples are from being correctly predicted, the more loss they suffer from. The exponential loss increases exponentially in function of the activation, whereas the Perceptron and large-margin losses only grow linearly. As soon as the examples are correctly classified in the Perceptron, their loss becomes null. Instead, the large-margin loss considers that an example that is too near from $F_\theta(\mathbf{x}) = 0$ should be considered as an error.

Training Once we have a continuous objective function, training aims at solving Equation 2.3.

Online Training Online training methods are minimization techniques that process the examples one per one, or small groups per small groups. The simplest minimization technique is the *stochastic gradient descent* algorithm. In this algorithm, the training examples $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ are processed one at a time. In order to process an example, we compute the gradient of the regularized empirical risk restricted to this example, and slightly modify the parameters in the inverse direction of the gradient:

$$\forall i, \theta_i \leftarrow \theta_i - \alpha \frac{\partial(\Delta(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}) + \lambda\Omega(f_\theta))}{\partial\theta_i}$$

The Perceptron training algorithm presented in Section 2.1.1 is the instantiation of stochastic gradient descent with the Perceptron loss, $\lambda = 0$ and $\alpha = 1$. Many variant of stochastic descent exists. In particular, a lot of work has been done on tuning the α parameter.

Mini-batches

Another stochastic descent method that receives a growing interest in SL is the *mini-batch* gradient descent. Mini-batches are small subsets of the training examples that allows approximating the risk and its gradient *w.r.t.* the parameters. Mini-batch methods repeat the following steps: sample a subset of the examples (*e.g.* one hundred examples out of ten thousand), compute the approximated risk gradient and perform a gradient descent step. Pure stochastic gradient descent can be seen as a mini-batch algorithm with mini-batches of size one.

Batch Training

Batch training is another approach that requires all the training examples to be known in advance. Batch training methods compute $\hat{R}(f) + \lambda\Omega(f)$ exactly, which makes it possible to use classical numerical optimization techniques to perform minimization. The loss function and prediction function are often chosen to lead to a convex optimization problem. This makes it possible to use various efficient optimization techniques [Boyd et Vandenberghe, 2004] including BFGS (for continuously derivable losses), SMO ([Platt, 1998], used in SVMs) and various closed-forms methods when available (in regression with the square-loss for example).

2.1.5 Summary

In this section, we introduced the SL problem: given a limited amount of training examples, induce a function mapping inputs to outputs. We have illustrated SL with a simple use-case: email filtering. Such learning problems with two possible outputs are called binary classification problems. When there are more than two discrete outputs, we fall into multiclass classification problems. Supervised learning with scalar outputs is known as regression. We also talked about ranking problems where the aim is to learn to order objects.

The SL problem can be formalized according to the principle of expected risk minimization: the function we are searching is the function that leads to the lowest expected value of a task-specific loss function. Computing the expected risk is not possible, since we do in general not have access to the distribution underlying the learning problem. Instead, we usually have access to a limited amount of training examples that are sampled from this distribution. A key idea of SL is to approximate the expected risk with the empirical risk computed over the set of training examples.

When minimizing the empirical risk, a common phenomenon is overfitting. This happens when the learned function is too tailored to the particularities of the training examples. In order to control overfitting, a common approach is to introduce a regularization term that gives a preference for simple models.

In order to perform training, many methods can be thought of as minimizing the regularized empirical risk. This minimization can be done with online and with batch methods. The former modify the parameters continuously as new training examples are processed, while the latter needs to know all the examples in advance.

2.2 Sequential Decision Making

This section introduces sequential decision making (SDM) problems. This topic has a central place in AI and has been widely studied by researchers since the sixties. We start with a few SDM examples in Section 2.2.1. SDM can be formalized within the *Markov Decision Processes* (MDPs) framework, which is described in Section 2.2.2. The behavior of an agent evolving in an MDP is defined by its policy. Section 2.2.3 introduces *reinforcement learning* which aims at learning such a policy. Classical reinforcement learning methods suffer from scaling problems

to large decision problems. In order to deal with such problems, the decision maker can be represented approximately, thanks to Supervised Learning (SL) techniques. This approach is described in Section 2.2.4.

2.2.1 Agent-environment interaction

Planning a company's production, scheduling an elevator, driving a car autonomously, choosing a travel route or playing a video game are all examples of decision making problems where the aim is to maximize a long-term satisfaction measure. This satisfaction measure depends on the particular decision problems: it could be the turnover of the company, the average waiting time for an elevator or the score in a computer game.

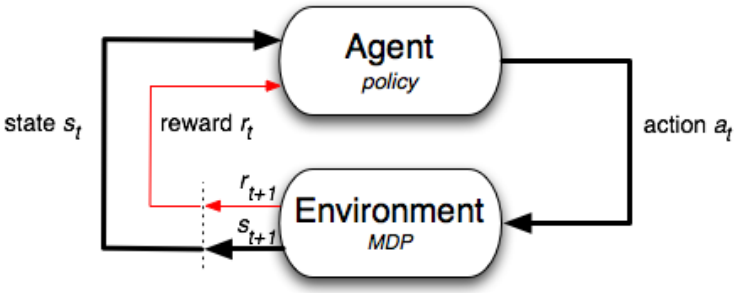


Figure 2.11: The agent-environment interaction in SDM problems.

Agent/Environment interaction

SDM problems involve the interaction between an agent and an environment as illustrated in Figure 2.11. The agent is the decision maker and the environment defines the decision problem. Given a current *state* of the environment, the agent chooses the next *action* to take. Taking an action then modifies the current state of the environment. The environment also transmits a scalar *reward* signal to the agent, that represent goals. Informally, when interacting with the environment, the aim of the agent is to choose the actions that maximize the *long-term reward*, *i.e.* the sum of rewards over a long time period.

GridWorld Consider the grid world example, given in Figure 2.12. Here, the agent is a mouse and the environment is a grid world containing cells, walls, cheese and a mousetrap. A state is a position (x, y) of the mouse in the world. In a given state, the mouse can move in four possible directions $\{left, right, top, bottom\}$. Given one such direction, either the mouse faces a wall and does not move consequently, or it goes deterministically to the new position. There are two states for which the agent perceives non-null rewards: the *mousetrap* gives a very large negative reward, which reflects the cost of dying into the trap. Reaching the *cheese* state is the goal of the mouse and gives a positive reward. Now, from the point of view of the agent, the problem can be reformulated as finding the sequence of directions to take, in order to maximize the reward, *i.e.* finding the cheese without crossing the mousetrap.

Tic-tac-toe Another classical example is the child's game of *tic-tac-toe*[11]. This game involves two players **X** and **O** who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical or diagonal row wins the game.

[11] This example comes from `http://www.cs.ualberta.ca/~sutton/book/ebook/node10.html`.

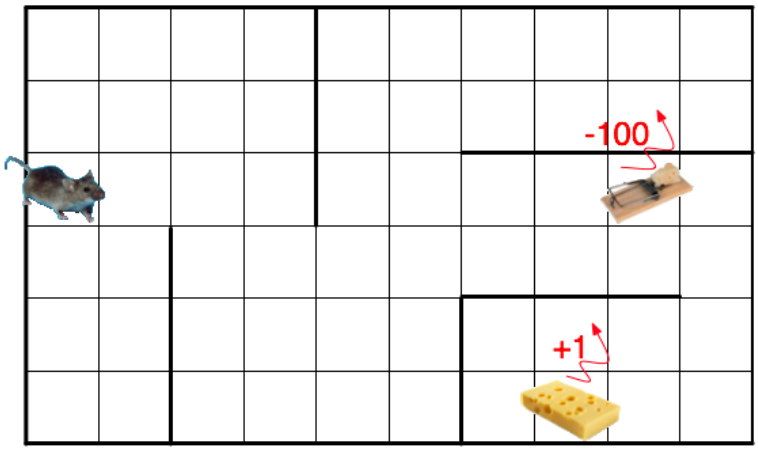


Figure 2.12: **An example decision problem: the grid world.**

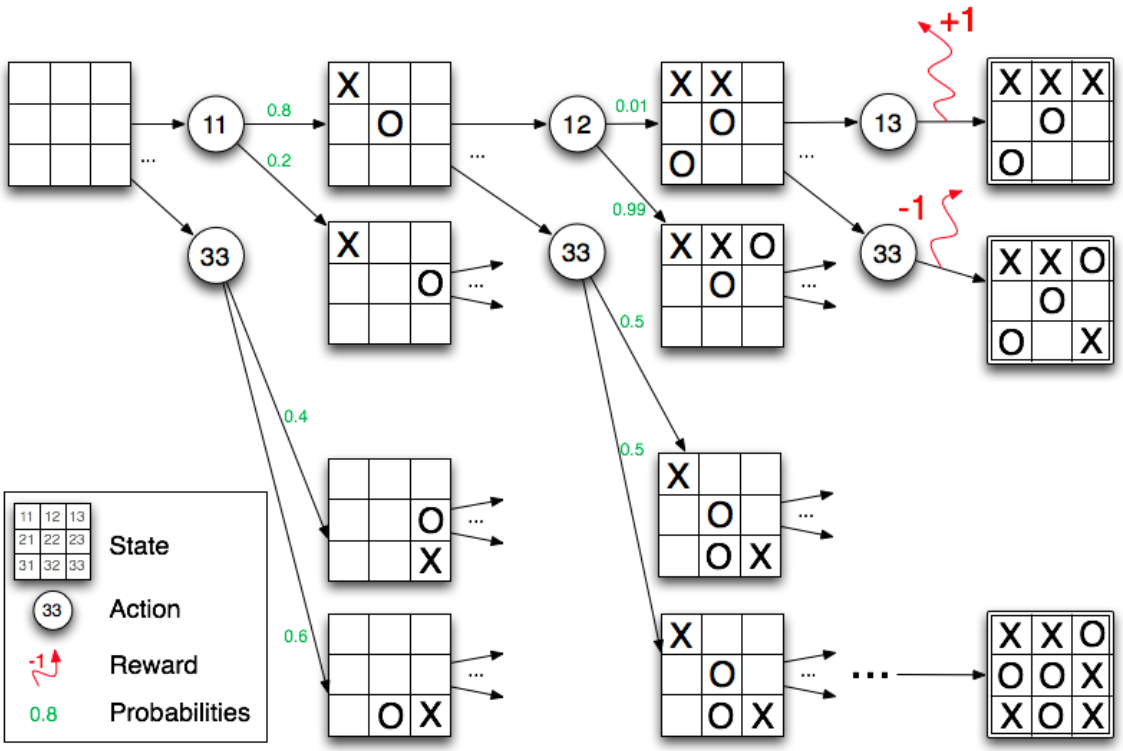


Figure 2.13: **An example decision problem: the tic-tac-toe game.**

Here, the agent is the player **X**. All the other elements of the game are part of the environment: the game board, the opponent player **O** and the game rules. A state of tic-tac-toe is a

vector of nine elements in $\{none, \mathbf{X}, \mathbf{O}\}$ indicating the state of each cell of the game. The aim of our agent is to select positions where to put **X**s. There is thus one action per free position in the game board. If **X** wins, the agent perceives a reward of $+1$. If **O** wins, it perceives a reward of -1. In all other cases, the reward is null. The SDM problem, illustrated in Figure 2.13 is the problem of finding the behavior that maximizes the probability of wining against a given opponent.

In tic-tac-toe, the agent chooses where to put **X**s but it has no control over the **O**s. When playing a turn, the successor state of the game depends on the behavior of the opponent, which may be *stochastic*. In many SDM problems, we assume the environment to be stochastic: a decision leads to a probability distribution over successor states.

2.2.2 Markov Decision Processes

We introduced SDM as an interaction problem between an agent and an environment. We now formalize these concepts: Markov Decision Processes (MDPs) and policies respectively describe environments and agents.

Markov Decision Process Markov Decision Processes [Howard, 1960] provide a mathematical framework for modeling SDM problems. They are used in a variety of areas, including robotics, automated control, economics and manufacturing. Formally, an MDP is defined by:

- A state space $\mathcal{S}$. This set represents all the states in which the environment can be at a given time step. The state space may contain *final states*. The decision processes finishes when the agent enters in such a final state.
- An action space $\mathcal{A}$. This set is composed of the actions among which the agent has to choose at each time step. In some MDPs, the set of possible actions depend on the current state. In such cases, we denote $\mathcal{A}_{\mathbf{s}} \subset \mathcal{A}$ the set of actions available in state $\mathbf{s}$. In final states, the set of possible actions is empty: $\mathcal{A}_{\mathbf{s}} = \emptyset$. In all other states, the set of possible actions must contain at least one action.
- A transition function $T : \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$, with $\Pi(\mathcal{S})$ the set of probability distributions $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, where $\mathbf{s}_t$ is the state at time step t and $\mathbf{a}_t$ is the chosen action at time step t.
- A reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. This function gives recompenses or punishments to the agent. Selecting action $\mathbf{a}$ in state $\mathbf{s}$ leads to a reward of $r(\mathbf{s}, \mathbf{a})$.

Figure 2.14 gives an example MDP with three states S_0, S_1 and S_2 and two actions a_0 and a_1. In each state, the agent can choose between the two actions. Each action leads to a probability distribution over successor states. The probabilities are shown in green in the figure. Some transitions lead to non-null rewards, which are represented by red arrows in the figure.

Policy

Policies The behavior of an agent is defined by its *policy*[12]. Deterministic policies map states to actions : $\pi : \mathcal{S} \to \mathcal{A}$. Stochastic policies define a distribution over actions for each possible state: $\pi : \mathcal{S} \to \Pi(\mathcal{A})$. When dealing with stochastic policies, we will denote $\pi(\mathbf{s}, \mathbf{a})$ the probability of selecting the action $\mathbf{a}$ in state $\mathbf{s}$.

Optimality Criterions Given an MDP, the aim is to find a policy that maximizes a long-term sum of rewards. The criterion that should be optimized can be defined in several ways. A natural definition is the *total reward* criterion. The expected total reward $V_\pi(\mathbf{s})$ when starting

Total reward

[12] We only consider stationary policies.

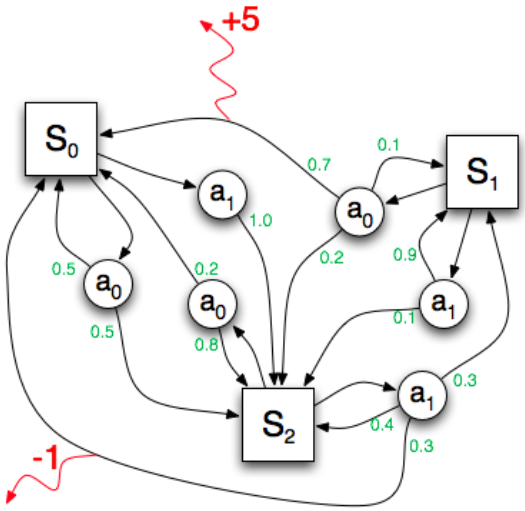


Figure 2.14: A general Markov Decision Process.

from state $\mathbf{s}$ and selecting actions with π is defined in the following way:

$$V_\pi(\mathbf{s}) = \mathsf{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\{\sum_{t=1}^{\infty} r(\mathbf{s}_t,\mathbf{a}_t) \mid \mathbf{s}_1 = \mathbf{s}\}$$

Note that, in the general case, this sum may diverge. In order to avoid divergence, it is common to introduce a discounting factor $\gamma \in [0,1[$ in order to reduce the influence of late rewards. The expected γ-*discounted reward* when starting from state $\mathbf{s}$ given π is defined in the following way: *Discounted Reward*

$$V_\pi(\mathbf{s}) = \mathsf{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\{\sum_{t=1}^{\infty} \gamma^t r(\mathbf{s}_t,\mathbf{a}_t) \mid \mathbf{s}_1 = \mathbf{s}\}$$

The choice of the discount factor γ leads to a continuous range of problems of increasing complexity, from maximizing the immediate reward ($\gamma = 0$) to maximizing the total reward ($\gamma \to 1$). Since it is not always convenient to choose the value of this parameter, it has also been proposed to consider the *average reward* criterion. This criterion focuses on the expected reward per step and is defined in the following way: *Average Reward*

$$V_\pi(\mathbf{s}) = \mathsf{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\pi}\{\lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T} r(\mathbf{s}_t,\mathbf{a}_t) \mid \mathbf{s}_1 = \mathbf{s}\}$$

Note that the V-functions described above are generally called state-value functions[13]. *State values*

Given the state-value function, we can define a partial order over policies. A policy π is said to dominate π' if and only if its state-value is higher for all the states of the problem: *Optimal policies*

$$\pi \geq \pi' \iff \forall \mathbf{s} \in \mathcal{S}, V_\pi(\mathbf{s}) \geq V'_\pi(\mathbf{s})$$

[13]Depending on the communities, this quantity may also be called *utility*, *expected utility* or *expected return*. The opposite quantity may also be considered under the *cost-to-go* name.

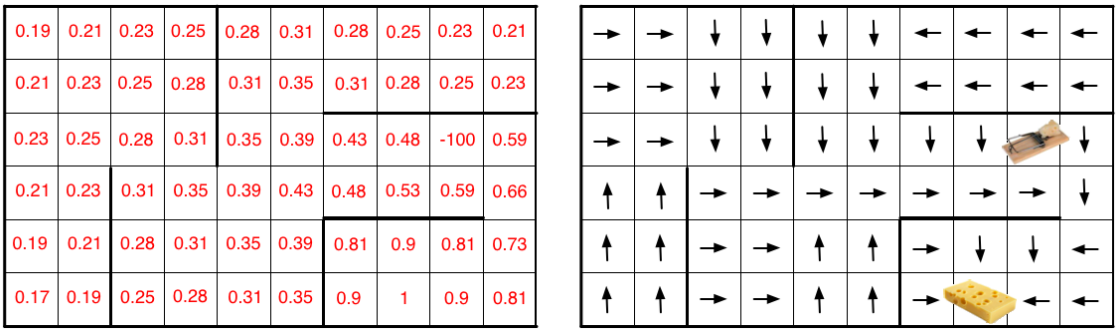


Figure 2.15: **The optimal state-value function V^* and an optimal policy π^* in the grid-world problem with a discount factor of 0.9.** The optimal policy is a greedy policy *w.r.t.* the V^*-function.

An *optimal policy* is a policy that is better or equivalent than any other possible policies. Such a policy π^* is defined by:

$$\forall \pi, \pi^* \geq \pi$$

Greedy Policy

Greedy policies Given a value function $V : \mathcal{S} \to \mathbb{R}$ that assigns scores to each state of an MDP, it is possible to define the greedy policy π_V^{greedy} that performs one-step-ahead search of the best action *w.r.t.* V. Such a policy is defined in the following way:

$$\pi_V^{greedy}(\mathbf{s}) = \underset{\mathbf{a} \in \mathcal{A}_\mathbf{s}}{\operatorname{argmax}} Q^V(\mathbf{s}, \mathbf{a})$$

State-action value

where $Q^V(\mathbf{s}, \mathbf{a})$ is called the state-action value function and reflects how desirable it is to choose action $\mathbf{a}$ in state $\mathbf{s}$. The definition of the state-action value function depends on the choice of optimality criterion. For example, with γ-discounted reward, we have:

$$Q^V(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \sum_{\mathbf{s}'} P_\pi(\mathbf{s}'|\mathbf{s}) V(\mathbf{s}')$$

where $P_\pi(\mathbf{s}'|\mathbf{s}) = P(\mathbf{s}'|\mathbf{s}, \pi(\mathbf{s}))$.

Figure 2.15 illustrates the optimal state-value and one optimal policy in the grid-world problem with a discounted reward criterion. The optimal policy is a greedy policy *w.r.t.* the state-value: it always chooses directions that most increase the state-value.

Note that greedy policies may be defined given any function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that assigns scores to state-action pairs. In the following, we denote π_Q^{greedy} the greedy policy *w.r.t.* Q:

$$\pi_Q^{greedy}(\mathbf{s}) = \underset{\mathbf{a} \in \mathcal{A}_\mathbf{s}}{\operatorname{argmax}} Q(\mathbf{s}, \mathbf{a})$$

2.2.3 Reinforcement Learning

Many different methods have been developed to find optimal policies in MDPs. In the *reinforcement-learning* problem, the transition and reward functions are not known a priori. Such problems involve an agent that *discovers* the environment and has to interact with it in order to maximize the perceived reward. In the tic-tac-toe example, the transition and reward function are determined by the rules and by the behavior of the opponent player. If we do not have a perfect

model of the opponent, we cannot compute the transition probabilities: in order to solve the problem, we have to estimate the opponent's behavior by interacting with him.

Exploration/Exploitation dilemma

Reinforcement learning problems are characterized by the *exploration/exploitation* dilemma: the agent has to *explore* the environment in order to find new promising ways to receive reward, and in the same-while, it has to *exploit* its current knowledge to effectively perceive rewards. Therefore, reinforcement-learning algorithms have to make use of exploration strategies. A common strategy is to use ϵ-greedy policies. Such a policy selects a random action with a small probability ϵ and the greedy actions otherwise:

ϵ-greedy policies

$$\pi_Q^{\epsilon-greedy}(\mathbf{s}) = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \text{argmax}_{\mathbf{a}\in\mathcal{A}}\, Q(\mathbf{s},\mathbf{a}) & \text{otherwise} \end{cases}$$

When selecting random actions, the hope is to find a new way to behave that leads to more reward: this is exploration. Instead, the greedy actions make use of the current knowledge of the agent that is encapsulated in the action-value function Q. This corresponds to exploitation steps.

Algorithm 1 One-step QLEARNING

Require: the learning rate parameter α
1: Initialize $Q(\mathbf{s},\mathbf{a})$ arbitrarily
2: **repeat** ▷ For all episodes
3: $\mathbf{s} \leftarrow$ sample an initial state
4: **while** not isStateFinal(s) **do** ▷ For all states
5: Choose $\mathbf{a}$ from $\mathbf{s}$ using a policy derived from Q (*e.g.* ϵ-greedy)
6: Take action $\mathbf{a}$, observe reward $r(\mathbf{s},\mathbf{a})$ and new state $\mathbf{s}'$
7: $Q(\mathbf{s},\mathbf{a}) \leftarrow (1-\alpha)Q(\mathbf{s},\mathbf{a}) + \alpha[r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}',\mathbf{a}')]$
8: $\mathbf{s} \leftarrow \mathbf{s}'$
9: **end while**
10: **until** convergence of $Q(\mathbf{s},\mathbf{a})$
11: **return** Q

Q-learning

An example solution One of the most important breakthroughs in reinforcement learning was the development of the learning algorithm known as QLEARNING [Watkins, 1989], which is given in Algorithm 1. QLEARNING aims at learning the optimal action-value Q^* iteratively, given a γ-discounted reward criterion. One value $Q(\mathbf{s},\mathbf{a})$ is stored per state-action pair. Those values are initialized arbitrarily (line 1). The algorithm then performs several *episodes*, *i.e.* sequences from an initial state until a final state in the MDP (line 2–10). The core of QLEARNING is the update step (line 7), which slightly modifies the current $Q(\mathbf{s},\mathbf{a})$ value to make it closer to $r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} Q(\mathbf{s}',\mathbf{a}')$. QLEARNING has been shown to converge to Q^* if all state-pairs are visited an infinite number of times. In practice, QLEARNING has been widely used with success in various applications.

Several applets demonstrating its use are available on Internet. Figure 2.16 is a capture of the applet of David Poole[14]. Here, QLEARNING has been applied on a grid-world domain similar to ours. In this example, there are four rewarding states, two that give negative rewards (-10 and -5) and two that give positive rewards ($+5$ and $+10$). Each cell corresponds to a state and four state-action pairs. The numbers displayed in the cells are the learned $Q(\mathbf{s},\mathbf{a})$ values. The blue arrows display the learned policy, which is a greedy policy *w.r.t.* the Q function.

[14] `http://www.cs.ubc.ca/~poole/demos/rl/q.html`

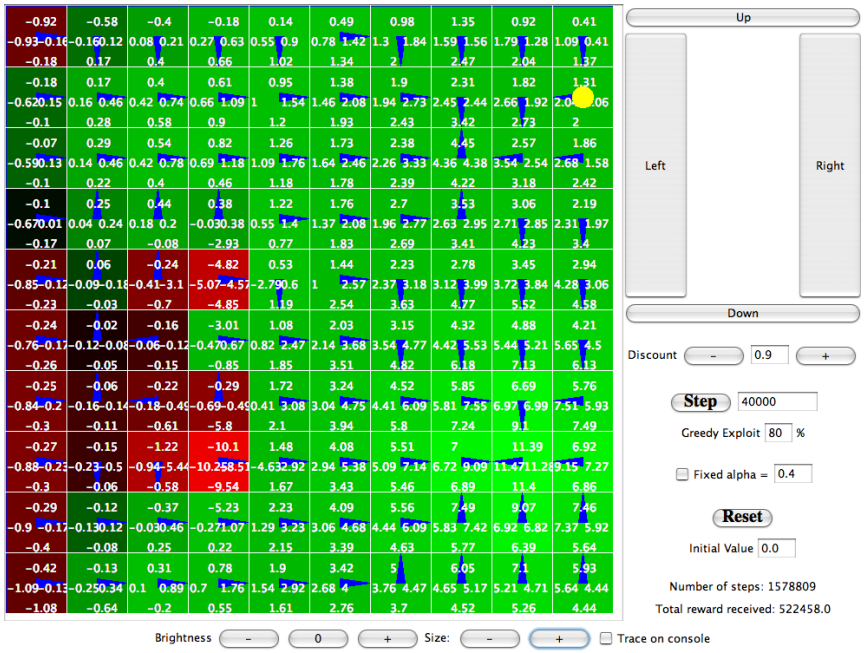


Figure 2.16: An applet demonstrating Qlearning.

Whereas QLEARNING learns the Q function, it is also possible to learn the state-value function V. Algorithms adopting this approach include TD(0) and TD(λ). Other well-known reinforcement learning algorithms that are not discussed here include *Monte Carlo Control* and *Sarsa*.

All these methods compute the optimal state-value (*resp.* action-value) for all states (*resp.* all state-action pairs). This implies that we are able to enumerate all the states of the MDPs. Furthermore, in order to store these values that implicitly encode the policy, we need $\mathcal{O}(\mathcal{S})$ (*resp.* $\mathcal{O}(\mathcal{S} \times \mathcal{A})$) memory size. For small MDPs, such as the grid-world or the tic-tac-toe problems, this is perfectly realistic. However, many MDPs and in particular those we deal with in this manuscript, have very large state spaces (*e.g.* billions of states). In the following, we only introduce approximate methods that are able to deal with such very-large MDPs.

Tabular representation

2.2.4 Function Approximation

In many MDPs, the state space is typically astronomically large, described implicitly and decomposed into factors, or aspects of state. Since storing state or action values explicitly in memory is intractable for these MDPs, various approximate methods have been developed. This has lead to an important bridge between SL techniques, discussed in Section 2.1 and SDM. Instead of storing the state or action-values explicitly, approximate methods make use of learning machines to approximately and compactly represent those values.

Feature Functions

In order to use SL to compactly store policies, the first step is to introduce vectorial descriptions of states or state-action pairs. We therefore use feature functions $\phi(.) \in \mathbb{R}^d$, such as those introduced in Section 2.1.1, for states and for state-action pairs. The latter is illustrated in Figure 2.17 in the context of the tic-tac-toe problem. As shown in this figure, features can describe any joint aspect of the state and the action. For example, the *Number of Os in diagonal* feature

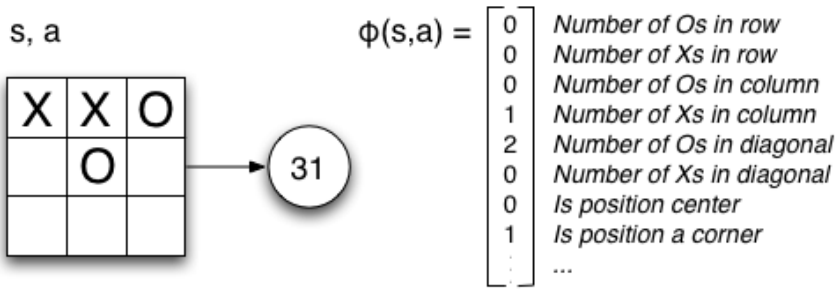


Figure 2.17: **Example of state-action joint description in the tic-tac-toe problem.**

counts the number of **O**s in the diagonal (of the current state) corresponding to position *31* (the envisaged action).

Value-based reinforcement learning Given a vectorial feature function $\phi(.)$, we can use any regression machine to approximately store value functions. Instead of storing one value per state-action pair, the policy is then fully defined by the parameters θ of the learning machine: *Approximate Values*

$$Q_\pi(\mathbf{s},\mathbf{a}) \overset{approx}{=} \hat{Q}_\theta(\phi(\mathbf{s},\mathbf{a}))$$

Any SL method can be used to approximate the value functions. A common and simple choice is to perform linear regression:

$$\hat{Q}_\theta(\phi(\mathbf{s},\mathbf{a})) = \langle \theta, \phi(\mathbf{s},\mathbf{a}) \rangle \tag{2.4}$$

Algorithm 2 Approximated One-step QLEARNING

Require: A supervised learning algorithm for regression.
1: Initialize θ arbitrarily
2: **repeat** ▷ For all episodes
3: $\mathbf{s} \leftarrow$ sample an initial state
4: **while** not isStateFinal(s) **do** ▷ For all states
5: Choose $\mathbf{a}$ from $\mathbf{s}$ using a policy derived from $\hat{Q}_\theta$ (*e.g.* ϵ-greedy)
6: Take action $\mathbf{a}$, observe reward $r(\mathbf{s},\mathbf{a})$ and new state $\mathbf{s}'$
7: Correct θ *w.r.t.* learning example $(\phi(\mathbf{s},\mathbf{a}), r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} \hat{Q}_\theta(\mathbf{s}',\mathbf{a}'))$
8: $\mathbf{s} \leftarrow \mathbf{s}'$
9: **end while**
10: **until** convergence of θ
11: **return** θ

Most reinforcement learning algorithms that were developed to work with tables can be easily extended to this approximate case. As an example, Algorithm 2 shows the approximate version of QLEARNING. The only notable difference with traditional QLEARNING lies at line 7: instead of modifying the current value of $Q(\mathbf{s},\mathbf{a})$, we update the parameters θ of the SL machine. These parameters are updated on the basis of a new learning example: a pair containing the description of the state-action pair and the associated Q value. *Approximate Q-learning*

Online training

Training methods Approximate reinforcement learning fits well with online learning methods (see Section 2.1.4). For example, if we use linear regression (Equation 2.4) with the unregularized squared error loss function (see Section 2.1.3), the line 7 can be replaced by:

$$\theta \leftarrow \theta + \alpha[\underbrace{r(\mathbf{s},\mathbf{a}) + \gamma \max_{\mathbf{a}'} \hat{Q}_\theta(\mathbf{s}',\mathbf{a}')}_{\text{Desired Q-value}} - \underbrace{\hat{Q}_\theta(\mathbf{s},\mathbf{a})}_{\text{Predicted Q-value}}].\phi(\mathbf{s},\mathbf{a})$$

where α is the learning rate parameter.

Batch training

It is also possible to use batch-learning methods within approximate reinforcement learning algorithms. Many algorithms, including the batch version of QLEARNING, which is called *Fitted* QLEARNING, work by iterating the following steps:

1. Make several passes in the MDP and create one training example per visited state, until we reach a predefined number of examples.
2. Train the SL machine with the examples created in step (1).
3. Replace the parameters θ by those learned in step (2) and continue.

One problem of value-based reinforcement learning is that minimizing the regression error is not directly related to the policy's performance. A good approximation of the value function may not lead to a good policy. Inversely, we could obtain a good policy while badly approximating the value function. Recent work in approximated reinforcement learning includes the development of new approximation methods that are not expressed as regression problems. In the following we review two alternative approaches: *policy gradient* and *reinforcement learning as classification*.

Policy Gradient Policy gradient algorithms directly optimize the expected long-term sum of rewards, without using value functions. These algorithms rely on stochastic policies π_θ that are parameterized by a set of parameters θ. A simple example of such policy is the log-linear policy:

$$\pi_\theta(\mathbf{s},\mathbf{a}) = \frac{1}{Z} \exp \langle \theta, \phi(\mathbf{s},\mathbf{a}) \rangle$$

where Z is the normalization factor:

$$Z = \sum_{\mathbf{a} \in \mathcal{A}_\mathbf{s}} \exp \langle \theta, \phi(\mathbf{s},\mathbf{a}) \rangle$$

The aim of policy gradient algorithms is to find the parameters θ that maximize the expectation of reward. Given a distribution over initial states $\mathcal{D}_\mathcal{S}$, the objective function ∇ is defined in the following way:

$$\eta(\theta) = \mathsf{E}_{\mathbf{s} \sim \mathcal{D}_\mathcal{S}}\{V^{\pi_\theta}(\mathbf{s})\}$$

The key idea of policy gradient algorithms is that the gradient $\nabla_\theta \eta(\theta)$ can be estimated empirically through simulation. GPOMDP [Baxter et Bartlett, 2001] is a classical gradient estimation method to deal with the *average reward* criterion. Given the estimates of gradients $\nabla_\theta \eta(\theta)$, any numerical optimization method can be used to optimize the parameters θ. A simple choice is to use stochastic gradient ascent. This leads to the online GPOMDP algorithm, called OLPOMDP [Baxter *et al.*, 2001]. Given an appropriate choice of the learning rate parameter, this algorithm is guaranteed to converge toward a locally optimal policy.

Reinforcement learning as classification In Section 2.1.2, we introduced the classification problem. Classification can be easily understood in the context of policies: the aim of a policy is to choose one action among a set of possible actions given a particular state, while classifiers aims at choosing one class among a set of possible classes given a particular input object. This parallel has led to the development of reinforcement learning algorithms that train a policy represented by a classifier [Lagoudakis et Parr, 2003]. This approach is directly related to the policy learning problem: a classification error correspond to an error of the policy. This strong relation makes it possible to relate the performance of the classifier to the reinforcement learning performance [Langford et Zadrozny, 2005], which was unfeasible with the regression approach.

Reinforcement learning as classification algorithms essentially follow the batch-training sketch given above. The main difficulty is that, in step (1), we have to create classification training examples. [Lagoudakis et Parr, 2003] proposed to determine the *good action* for a given state by using *rollouts* [Bertsekas, 1999]. Given a policy π and a state $\mathbf{s}$, a rollout is a run of π, starting from $\mathbf{s}$ with a maximum number of steps T_{max}. Rollouts are used to estimate empirically the state-value $V_\pi(\mathbf{s})$ by averaging the sum of rewards perceived over multiple runs. Formally, given the γ-discounted reward criterion, the state-values are approximated the following way:

Rollouts

$$V_\pi(\mathbf{s}) \overset{approx}{=} \frac{1}{K} \sum_{k=1}^{K} \sum_{t=0}^{T_{max}} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t) \mid \pi, \mathbf{s}_0 = \mathbf{s}$$

In order to create a classification example for a given state $\mathbf{s}$, the $Q_\pi(\mathbf{s}, \mathbf{a})$ values are estimated for all possible actions by using rollouts. These estimated values are then use to determine if one action clearly is better than the others. If this is the case, a classification example can be created with this *good action*. A particularly interesting case with rollouts is deterministic MDPs (such as the grid-world). With deterministic transitions, a policy will always behave the same way when started from the same state. Therefore, only one rollout ($K = 1$) is enough to exactly compute $V_\pi(\mathbf{s})$[15]. In deterministic MDPs, rollouts can thus be used with a relatively small cost.

2.2.5 Summary

In this section, we introduced Sequential Decision Making (SDM) problems that involve an agent interacting with an environment. Such environments where formalized as Markov Decision Processes (MDPs). An MDP is composed of a state-space, an action-space, a transition function and a reward function. At a given time step t, the agent perceives the current state of the MDP $\mathbf{s}_t$ and it has to choose between multiple possible actions $\mathcal{A}_{\mathbf{s}t}$. Once an action $\mathbf{a}_t$ has been chosen, the environment goes into a new state $\mathbf{s}_{t+1}$ sampled from a distribution $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ and it generates a reward $r(\mathbf{s}_t, \mathbf{a}_t)$. The aim of the agent is to choose actions in order to maximize a long-term sum of rewards. Common definitions of this long-term sum of rewards include the total reward, the γ-discounted reward and the average reward.

Given an MDP, the agent's behavior is described by its policy, which is a function that maps states to actions. The SDM problem consists in finding an optimal policy: a policy that maximizes the long-term reward. Most SDM methods make use of state-value and action-value functions. A state-value function (or V-function) reflects how desirable it is to be in a given state. An action-value function (or Q-function) reflects how desirable it is to choose an action in a given state. Given a state-value or an action-value, we can define greedy policies that always select the best scored actions. A fundamental property of these functions is that, for a given MDP, there is a unique optimal state-value function and a unique optimal action-value that are shared by all the optimal policies.

[15] Assuming that the T_{max} value is big enough.

Finding an optimal policy, when the transition and reward functions of the MDP are unknown, is called a reinforcement-learning problem. In such problems, agents have to continuously make a trade-off between exploiting their current knowledge and exploring new behaviors that may eventually lead to more reward. This is called the exploration/exploitation dilemma and various strategies to make the trade-off have been proposed. An example of such strategies is the ϵ-greedy policy, which, from time to time, selects a random action to enforce exploration.

QLEARNING is a central algorithm of reinforcement learning, which aims at iteratively estimating the optimal Q-function. Therefore, it stores one Q-value per state-action pair and computes these values by visiting each state-action pair several times. Most classical reinforcement-learning algorithms store one value of the V-function (*resp.* Q-function) per possible state (*resp.* state-action pair) in a table.

Storing V-functions or Q-functions explicitly in a table is reasonable for small MDPs, *e.g.* less than 10^6 possible states. However, many MDPs, including those discussed in this manuscript, have much larger state-spaces. Using tables to store value functions becomes then quickly intractable. A key idea to handle very large MDPs, is to replace the explicit table representation by an implicit representation encoded by a learning machine. Using SL techniques to approximately store and learn policies is the essence of approximated reinforcement learning. We overviewed three approaches to perform approximated reinforcement learning: value-based reinforcement learning, policy gradient algorithms and reinforcement learning as classification.

3

Structured Prediction

Contents

Supervised Learning focuses on learning a function given a set of input-output examples. A lot of models have been proposed mainly for continuous outputs (*i.e.* regression) or discrete outputs (*i.e.* classification). Though most Machine Learning (ML) work focuses on regression and classification, in many different fields such as biology, natural language processing, image processing or chemistry, data may be naturally described as structured objects like sequences, trees, lattices or graphs. This data, which does not fit in the classical learning frameworks, has recently witnessed a surge of interest in the ML community. Supervised learning problem where the outputs are structured objects is known as *learning with structured outputs* or mort shortly *Structured Prediction* (SP). Structured Prediction has a lot of real-world applications, including prediction of protein structure in bio-informatics, image understanding, speech processing, handwriting recognition and several natural language processing tasks such as part-of-speech tagging, named entity extraction, sentence parsing or automatic translation.

This chapter introduces the SP problem and makes an overview of existing methods to solve this problem. We first introduce SP formally and illustrate it with various traditional problems in Section 3.1. We will show that the main challenges of SP come from the combinatorial nature of the problem. The need for search in SP has been tackled in two different ways that we contrast in

our overview of state-of-the-art. The first approach relies on a global combinatorial optimization (Section 3.2) while the second approach uses greedy search (Section 3.3).

3.1 Predicting Structured Objects

Many learning problems of the real world naturally express themselves with more complex outputs than discrete classes or simple scalars. Prediction problem with complex outputs such as sequences, trees, graphs or lattices are known as Structured Prediction (SP) problems. We adopt a general definition of SP as a supervised learning problem, where the goal is to learn a mapping from inputs $\mathbf{x} \in \mathcal{X}$ to outputs $\mathbf{y} \in \mathcal{Y}_{\mathbf{x}}$. The outputs are discrete structured objects, such as sequences, trees or graphs. $\mathcal{X}$ is the set of all possible inputs and $\mathcal{Y}_{\mathbf{x}}$ is the set of candidate outputs for a given input $\mathbf{x}$. We denote by $\mathcal{Y} = \cup_{\mathbf{x}\in\mathcal{X}}\mathcal{Y}_{\mathbf{x}}$ the full output space.

For learning, the user supplies a set of examples $D = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i\in[1,n]}$. These examples are supposed to be independently and identically sampled from an unknown distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$. In order to evaluate the quality of a prediction, we use a loss function $\Delta(\hat{\mathbf{y}}, \mathbf{y})$, which quantifies how bad it is to predict $\hat{\mathbf{y}}$ instead of $\mathbf{y}$. Given D and $\Delta(.,.)$, the aim is to learn a model that is able to predict outputs for any new input $\mathbf{x} \in \mathcal{X}$. SP learning methods try to find the model that minimizes the expected loss values between predicted and correct outputs.

3.1.1 Examples of SP tasks

Sequence Labeling

Sequence labeling is the generic task of assigning labels to the elements of a sequence. This task corresponds to a wide range of real-world problems. For example, in the field of Natural Language Processing (NLP), *part of speech tagging* consists in labeling the words of a sentence as nouns, verbs, adjectives, adverbs, etc. Other examples in NLP include: chunking sentences, identifying sub-structures, extracting named entities, etc. Information extraction systems can also be based on sequence labeling models. For example, one could identify relevant and irrelevant words in a text for a query need. Sequence labeling also arises in a variety of other fields (character recognition, user modeling, bioinformatics, ...). See [Dietterich, 2002] for an exhaustive overview of sequence labeling applications.

In sequence labeling, an input $\mathbf{x} \in \mathcal{X}$ is a sequence of elements $(\mathbf{x}_1, \ldots, \mathbf{x}_T)$ and an output $\mathbf{y} \in \mathcal{Y}$ is the corresponding sequence of labels $(\mathbf{y}_1, \ldots, \mathbf{y}_T)$, where each y_t is the label that corresponds to element x_t. The labels belong to a predefined application-dependent dictionary denoted $\mathcal{L}$.

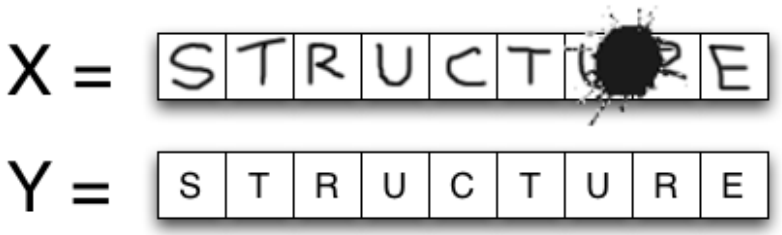


Figure 3.1: **Sequence labeling: handwritten recognition.**

Figure 3.1 illustrates one application of sequence labeling: handwritten characters recognition. In this application, inputs are sequences of handwritten characters (*e.g.* gray-scale bitmaps) and outputs are sequences of labels identifying recognized characters. The set of possible labels is the 26-letters alphabet: $\mathcal{L} = \{A, \ldots, Z\}$.

In order to perform handwritten recognition, one could simply ignore the output structure by treating each letter independently with a traditional multi-class classifier. However, this is not very convincing because we ignore structural information that may help for better predictions. In our example, the letters U and R are mostly hidden. If we remove the entire neighboring context, recognizing them seems very hard. However, given the context – predictions on other letters: the word begins with `STRUCT` and finishes with `E` – predicting the missing letters seems much more easy. Handwritten recognition is an example of *collective classification*: the aim is to solve a set of interdependent classification problems. Such dependencies legitimate SP approaches, which work at the level of the complete structured objects.

Collective classification

In order to cast sequence labeling in the framework of SP, we must select a loss function that measures the quality of predictions. A common loss function for sequence labeling is the *Hamming Loss*, which simply counts the number of wrong predicted labels:

Loss Functions

$$\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y}) = card(\{i \in [1, T], \hat{\mathbf{y}}_i \neq \mathbf{y}_i\})$$

Depending on the applications, other loss function may be used. Such loss function may for example put more focus on particular labels or particular elements of the sequence.

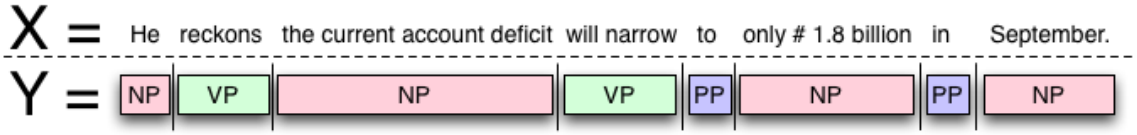


Figure 3.2: **SP example: Text Chunking.**

Figure 3.2 illustrates *text chunking*[1], which consists of dividing a text in syntactically correlated parts of words. In order to evaluate a particular chunker, the F_1 score is usually used. This score is the geometric mean between the precision and the recall of the predicted chunks [van Rijsbergen, 2001]. When applying SP methods on text chunking, we can directly incorporate this quality measure into the loss function $\Delta(.,.)$. Note that this loss function, contrary to the Hamming loss, is not additively decomposable onto the predicted elements.

Chunking

Another kind of natural language analysis is the *dependency-parsing* task, which is illustrated in Figure 3.3. Here, inputs are sentences (*i.e.* sequence of words) that may be enriched thanks to pre-processing and outputs are *dependency trees*. For each word, there is one arc in the dependency tree. Each arc is labeled (*e.g.* body, subject, ...) and linked to a predecessor word.

Dependency Parsing

A well-known problem where SP may be applied is the *machine translation* task. Machine translation may be formalized in several different ways. Figure 3.4 illustrates phrase-based translation[2], where both the input and the output are segmented sentences.

Machine Translation

Document engineering is another field that offers several applications that fit in the SP framework. Figure 3.5 introduces the structure mapping problem: mapping an input tree describing a document (an HTML page) onto an enriched structured representation of the same

Structure Mapping

[1] example from `http://www.cnts.ua.ac.be/conll2000/chunking/`

[2] example from `www.iccs.inf.ed.ac.uk/~pkoehn/publications/tutorial2003.pdf`

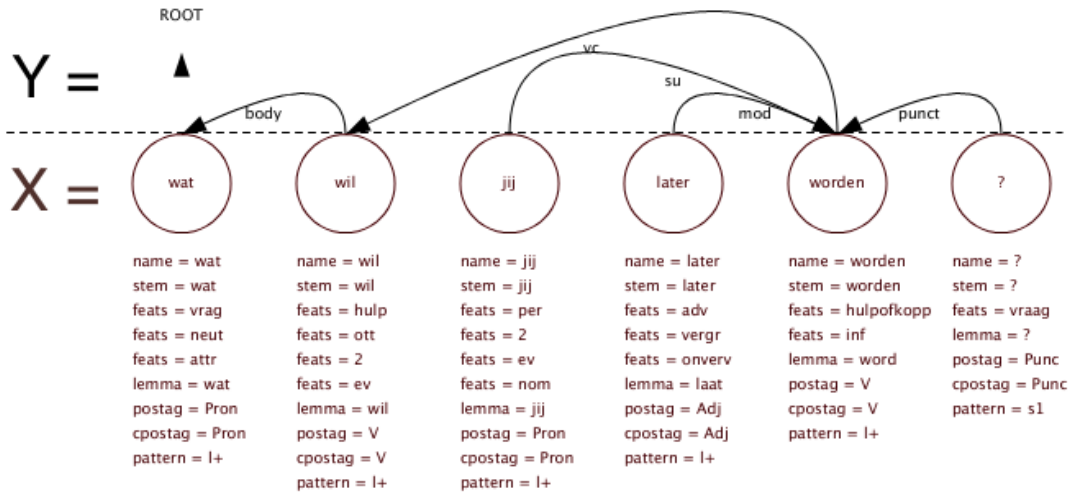


Figure 3.3: **SP example: Dependency Parsing.**

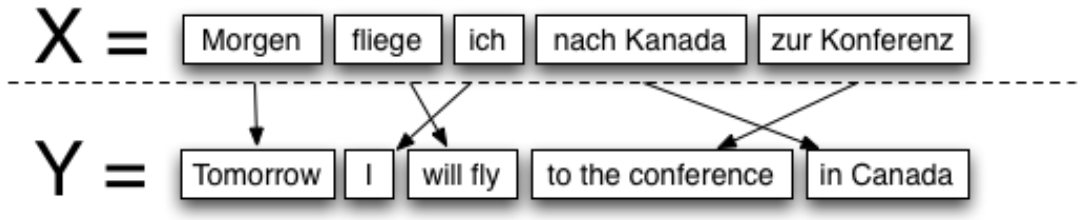


Figure 3.4: **SP example: Phrase-based Machine Translation.**

data (an XML tree with a specific Document Type Definition (DTD) for example). In structure mapping, the output structure is not known in advance. Constructing the output trees involves both to find their structure and to decide the labels of each node.

Document Annotation

Another popular application where SP may be applied is the *document annotation* task. This task, illustrated in Figure 3.6, aims at predicting the structure of a document available as an image. This can be seen as an SP task where the outputs are tree-structured layout and content information.

Beside these few popular examples, SP has several other applications in various fields such as bio-informatics or image processing.

3.1.2 Inference and Training

In order to properly treat the characteristics of the output space, SP methods work at the level of the structured outputs. SP can be seen as a classification problem where each possible output

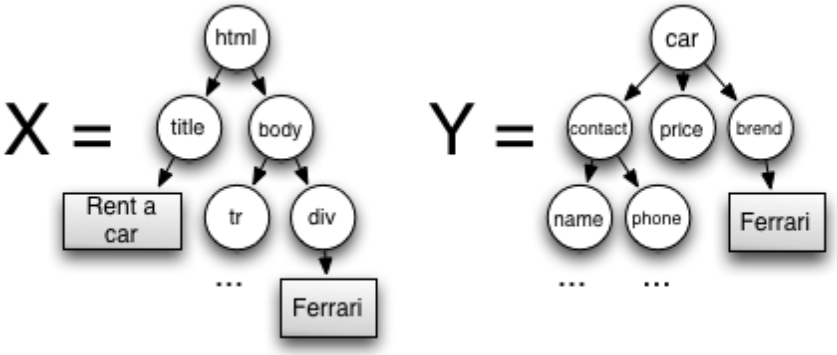


Figure 3.5: **SP example: HTML to XML structure mapping.**

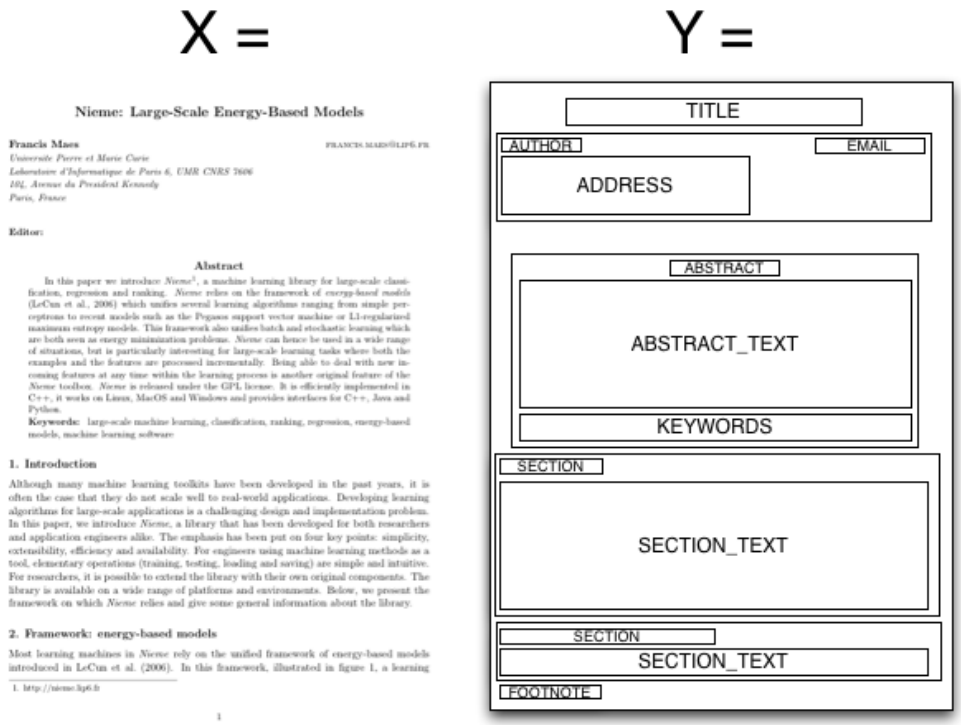


Figure 3.6: **SP example: Document Annotation.**

is a possible class. But, compared to traditional multi-class classification, the number of possible outputs grows, in general, exponentially in the size of the objects for most structured objects. For example, in handwritten character recognition with a 26-letters alphabet, the number of possible outputs of size n is 26^n.

In the following, we discuss SP models that are parameterized by a vector $\theta \in \mathbb{R}^d$ and we denote f_θ the prediction function corresponding to parameters θ. Two problems have to be solved in SP:

- **Inference:** Given the parameters θ and an input $\mathbf{x}$, the inference consists in selecting an output $\hat{\mathbf{y}} = f_\theta(\mathbf{x})$ among all candidates $\mathcal{Y}_\mathbf{x}$. The main challenges of inference are related to the combinatorial nature of the output space.
- **Training:** Given the set of examples D and the loss function Δ, the aim of training is to find the parameters θ that will lead to a *good* inference function. Ideally, we would like to minimize the *expected risk*, which is defined as follows:

$$\theta^* = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} E_{\mathbf{x},\mathbf{y} \sim \mathcal{D}_{\mathcal{X}\times\mathcal{Y}}} \{\Delta(f_\theta(\mathbf{x}), \mathbf{y})\}$$

  Since the distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$ is unknown, the expected risk cannot be computed and one usually minimizes the *empirical risk*[3] [Vapnik, 1995]:

$$\theta^* = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \frac{1}{N} \sum_{(\mathbf{x},\mathbf{y}^*) \in D} \Delta(\hat{\mathbf{y}} = f_\theta(\mathbf{x}), \mathbf{y})$$

A common aspect of all SP methods is that the inference function is launched several times in order to perform training. The intractability of the inference thus systematically leads to the intractability of training.

In the two following sections, we give an overview of the field of Structured Prediction. The apparition of general SP models is relatively recent. Previously, many ad-hoc solutions have been proposed on some instances of the SP problem such as sequence labeling, sequence chunking or natural language parsing. Well-known examples of early work dealing with structure include the Hidden Markov Models [Rabiner, 1990] [Cappé, 2001] for modeling sequences and the Probabilistic Context Free Grammar [Johnson, 1998] for modeling syntax trees. In the following, we only focus on *structure independent* SP models: those that are not restricted to a particular kind of data.

3.2 Overview of Global Models

One of the first ideas of SP was to generalize existing classification methods to structured outputs. The main difficulty to generalize classification is related to the exponential number of possible outputs *w.r.t.* the size of the data. Inference in the methods presented below is a *global combinatorial optimization problem.*

3.2.1 Principle

We introduce here the common concepts underlying global methods.

Compatibility functions

Several methods have been proposed which are based on a compatibility function $F(\mathbf{x}, \mathbf{y}; \theta)$ that measures *how good the output y is, given an input x*. A simple and common choice for the function F is to use a linear function:

$$F(\mathbf{x}, \mathbf{y}; \theta) = \langle \theta, \phi(\mathbf{x}, \mathbf{y}) \rangle$$

where $\langle ., . \rangle$ denotes the scalar product and ϕ is an input-output *joint description* function.

Joint descriptions

In order to assign scores to candidate outputs, the global methods make use of input-output joint feature functions $\phi(.,.)$. Such functions jointly describe an input $\mathbf{x}$ and a corresponding candidate output $\mathbf{y}$ as a vector in $\mathbb{R}^d$. Such vectors are usually called *feature* vectors. A feature is a function that describes one aspect of the input-output pair as a scalar. For example, in the natural language parsing domain, the feature function illustrated in Figure 3.7 has one feature per *production rule*. Each of these features corresponds to the number of times a rule is expressed in the input-output pair[4].

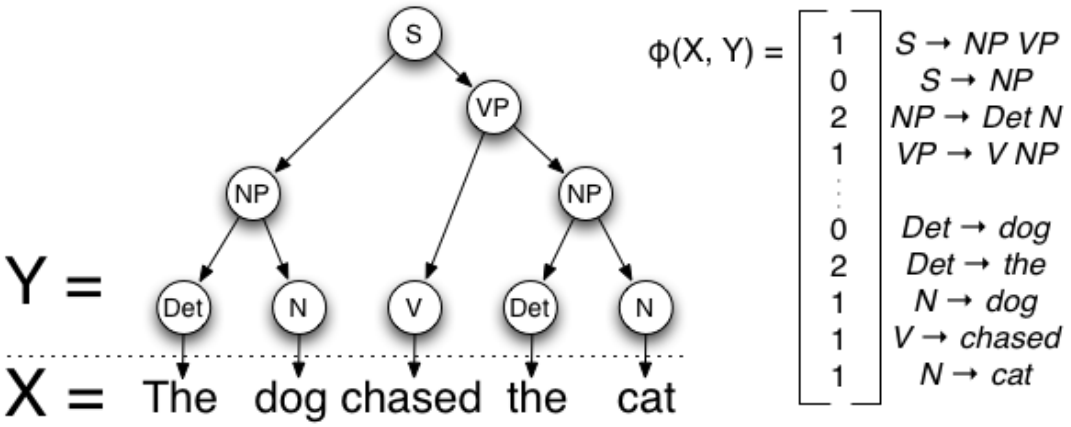


Figure 3.7: **Features example: Text Parsing.**

Inference

Given the compatibility function $F(.,.)$, inference consists in finding the output, which has the best compatibility with the specified input:

$$f_\theta(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}_\mathbf{x}}{\operatorname{argmax}} F(\mathbf{x}, \mathbf{y}; \theta) \tag{3.1}$$

The global methods assume that this equation can be solved efficiently. The way this combinatorial search problem is tackled depends on the properties of the feature function $\phi(.,.)$ and on the output space $\mathcal{Y}$. Usually, global models require that the functions $\phi(.,.)$ and $\Delta(.,.)$ both decompose additively over the structure elements. Such decomposable functions make it possible to use dynamic-programming based inference. Other kind of search algorithms may also be used, such as graph-cut algorithms or approximate inference algorithms.

Training

Global models differ in the meaning that is associated to the compatibility function and in the way they learn the θ parameters. We review below the most-known training methods for global models.

3.2.2 Structured Perceptron

The Structured Perceptron [Collins, 2002] is a generalization of the classical Perceptron, which was first applied to natural language parsing.

Training is described in Algorithm 3. The principle is very similar to training in the classical Perceptron. For a given input $\mathbf{x}^{(i)}$, we predict the output $\hat{\mathbf{y}} = f_\theta(\mathbf{x}^{(i)})$ (line 4). If this output

[3] or some regularized empirical risk.

[4] Example from [Tsochantaridis *et al.*, 2004]

Algorithm 3 Structured Perceptron training algorithm.

Require: a training set D

1: $\theta \leftarrow \mathbf{0}$
2: **while** training **do**
3: **for** $i = 1$ to $card(D)$ **do**
4: $\hat{\mathbf{y}} = \text{argmax}_{\mathbf{y} \in \mathcal{Y}} \langle \theta, \phi(\mathbf{x}^{(i)}, \mathbf{y}) \rangle$
5: **if** $\hat{\mathbf{y}} \neq \mathbf{y}^{(i)}$ **then**
6: $\theta \leftarrow \theta + \phi(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) - \phi(\mathbf{x}^{(i)}, \hat{\mathbf{y}})$
7: **end if**
8: **end for**
9: **end while**

is the correct one, we continue with the next example. Otherwise, we update to the parameters vector (line 6) to enforce the correct output and reduce the (wrong) predicted output. Several runs over the training datasets are performed until the convergence of θ. In the Structured Perceptron, no special meaning is given to the parameters vector, except the requirement that the correct output should have higher scores than wrong outputs.

This algorithm has one advantage: its simplicity. However, it does not take into account the loss function Δ and considers implicitly a 0/1 loss: 0 for the correct output, 1 for all other possible outputs.

3.2.3 Conditional Random Fields

Conditional Random Fields (CRFs) [Lafferty *et al.*, 2001, Wallach, 2004] use a log-linear probability function to model the conditional probability of an output $\mathbf{y}$ given an input $\mathbf{x}$:

$$p(\mathbf{y}|\mathbf{x}, \theta) = \frac{1}{Z_\theta(\mathbf{x})} \exp \langle \theta, \phi(\mathbf{x}, \mathbf{y}) \rangle$$

This form of probability distribution is motivated by the formalism of Markov Random Fields, which are graphical models where nodes are random variables and edges indicate dependencies between those variables. The outputs $\mathbf{y}$ are seen as sets of random variables $(y_1, \ldots, y_n)$ and the graphical model defines a probability distribution that decomposes over a set of cliques $\mathcal{C}$:

$$p(\mathbf{y}|\mathbf{x}, \theta) = \frac{1}{Z_\theta(\mathbf{x})} \exp \Sigma_{c \in \mathcal{C}} \langle \theta_c, \phi_c(\mathbf{x}, \mathbf{y}) \rangle$$

where θ_c is the part of the parameters corresponding to the clique c and ϕ_c is a joint description of the input and the variables forming clique c. This decomposition of the probability distribution is intended to make training and inference tractable. Training works by finding the parameters θ that maximize the conditional likelihood of the outputs given the inputs [S. et A, 1985] [Berger *et al.*, 1996]. Inference involves searching to output with highest probability, given the input. Note that finding the maximum of $p(\mathbf{y}|\mathbf{x}, \theta)$ is equivalent to finding the maximum of $\langle \theta, \phi(\mathbf{x}, \mathbf{y}) \rangle$, *e.g.* Equation 3.1. Similarly to the other methods presented in this section, we assume to have access to an efficient solver of this equation. In the case of *linear-chain* CRFs, that model sequences with a Markov assumption, the model is close to the traditional HMMs and inference can be solved exactly using the Viterbi algorithm.

A large amount of work has been spent on training methods for CRFs. Initially the focus was on batch methods such as *iterative scaling* [Lafferty *et al.*, 2001], BFGS [Sha et Pereira, 2003], or

conjugate gradient descent [Wallach, 2002]. More recently, it has been shown that CRFs can efficiently be trained with online methods including *stochastic meta-descent* [Schraudolph et Graepel, 2003, Vishwanathan *et al.*, 2006] and exponentiated gradient [Globerson *et al.*, 2007]. In order to deal with many potential features, [Dietterich *et al.*, 2004] discuss the use of regression tree in order to represent the clique potentials and propose to use boosting for training. [Liao *et al.*, 2007] also introduce boosting in the context of CRFs in order to simultaneously perform training and feature selection. Apart from training, it has also be shown that CRFs can be kernerlized [Lafferty *et al.*, 2004].

3.2.4 Large margin methods

Several methods extending the ideas of Support Vector Machines to SP have been proposed. SVM for Interdependent and Structured Output spaces (SVMISO, also known as SVM^{struct}) [Tsochantaridis *et al.*, 2004] [Tsochantaridis *et al.*, 2005] is a generalization of maximum margin multi-class classification [Mayoraz et Alpaydin, 1998] to structured outputs. Maximum Margin Markov Network (M^{3}N) [Taskar *et al.*, 2003] is another well known SP model which relies on margin maximization. Both methods try to maximize the margin between correct and wrong outputs. This margin is defined the following way:

$$m(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}, \mathbf{y}; \theta) = \underbrace{\left\langle \theta, \phi(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \right\rangle}_{\text{score of the correct output}} - \underbrace{\left\langle \theta, \phi(\mathbf{x}^{(i)}, \mathbf{y}) \right\rangle}_{\text{score of an incorrect output } \mathbf{y}}$$

Note that this margin can be rewritten as a dot product $\langle \theta, \Phi_i(\mathbf{y}) \rangle$ with $\Phi_i(\mathbf{y}) = \phi(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) - \phi(\mathbf{x}^{(i)}, \mathbf{y})$.

The aim of learning is to find parameters θ that lead to high margins for high loss values. The two methods differ in the way they generalize the max-margin principle and in how they handle the potentially exponential number of constraints of the quadratic programming problem they solve.

The loss function is introduced with a slack rescaling approach in SVMISO, which solves the following problem: *SVM-ISO*

$$\forall i, \langle \theta, \Phi_i(\mathbf{y}) \rangle > 1 - \frac{\xi_i}{\Delta(\mathbf{y}, \mathbf{y}^{(i)})}$$

where the ξ_i are slack variables, similar to those of the traditional SVMs. This approach allows $\sum \xi_i$ to be interpreted as an upper bound on the empirical risk. In order to solve the quadratic problem, which has an exponential number of constraints, the authors propose to find an approximate solution, using only a small number of significant constraints. An extension of SVMISO to semi-supervised learning problems is presented in [Brefeld et Scheffer, 2006].

M^{3}N introduces the loss function with a margin rescaling approach: *M^{3}N*

$$\forall i, \langle \theta, \Phi_i(\mathbf{y}) \rangle > \Delta(\mathbf{y}, \mathbf{y}^{(i)}) - \xi_i$$

In the initial paper describing M^{3}N, the author proposed an adaptation of the SMO algorithm [Platt, 1999] in order to solve the quadratic program. Several more efficient algorithms have been proposed in the following, such as exponentiated gradient [Bartlett *et al.*, 2004], extra-gradient [Taskar *et al.*, 2005], dual extra-gradient [Taskar *et al.*, 2006] or sub-gradient [Ratliff *et al.*, 2006a] methods.

3.2.5 Discussion

All global models assume that inference (Equation 3.1) can be solved efficiently. This optimization step, which is involved in both learning and inference, has mostly been tackled with dynamic programming techniques. This has multiple major drawbacks:

- Dynamic programming requires strong independence assumptions on the feature function. For example, in sequence labeling, it is often assumed that a label only interacts with the previous and next labels. This disables the use of feature functions that captures long-term or global dependencies in the output.
- The use of dynamic programming generally restricts the training methods to decomposable loss functions. Such loss function can be written as sums over a set of sub-structures of the output. However, as shown in Section 3.1.1, this restricting assumption is violated in many tasks.
- The inference algorithm entirely depends on the structure of the outputs and on the assumptions that are made in the loss and feature functions. There is thus a high adaptation cost of the global methods from one SP task to another. In order to apply a given global method to a new SP task, one often needs to fully rewrite an appropriated optimization method. In practice this requires an advanced knowledge of optimization methods and good implementation skills.
- Even with independence assumptions, dynamic programming algorithms may have a prohibitive complexity. For example, when predicting trees, the best dynamic programming algorithms have a cubic complexity in the number of leaf nodes. This limits the use of such methods to small trees, *e.g.* less than 50 nodes. Some domains, such as HTML to XML conversion, deal with trees containing thousands of nodes, which make the use of dynamic programming unrealistic.

In summary, the family of global methods is constructed on task dependents inference procedures. For many applications, solving Equation 3.1 is unrealistic, in particular in the presence of large-scale collections or complicated structured output. In order to circumvent this bottleneck, some approximate inference algorithms have been proposed, but the way approximation in inference interacts with learning is far from being clear [Kulesza et Pereira, 2008]. The next section presents a new family of methods that avoid the need to solve Equation 3.1.

3.3 Overview of Incremental Models

A key difficulty when dealing with SP is the combinatorial nature of the output space. In order to break this complexity, global SP models usually make simplifying assumptions on the nature of the dependencies or on the loss functions used for training. Even though, many models are often restricted to problems with limited complexity and they do not scale, neither for inference nor for learning, with large datasets. We introduce here a new family of methods for SP that has specially been designed to complex cases where global models fail. Much less work exists in the literature on incremental models than on global ones. However, since CR-algorithms are closely related to following methods, we provide a detailed description of them.

3.3.1 Principle

A new vision of the SP process has been recently proposed where SP inference is considered as a sequential decision problem. Instead of modeling *what a good output looks like* and then

searching for the best output, these methods directly model *how to build the good output*. This simple idea suggests thus to integrate learning and searching into a sequential prediction process.

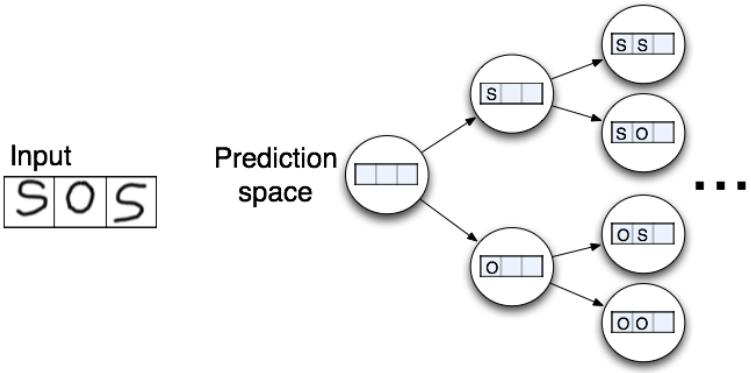


Figure 3.8: **Incremental Structured Prediction for Sequence Labeling.** Given an input, inference consists in constructing the output incrementally. The prediction space is the set of states in which inference can be during this construction process.

For this approach, illustrated in Figure 3.8, the structured output is built incrementally: components are added one at a time to a current *partial output*. Inference then consists in exploring a search space defined by states (partial outputs) and actions (choosing a component to be added to the current state) until a *complete* output is built. Going back to Figure 3.1, decisions for the sequence-labeling example would simply consist in choosing the correct label for each input character. For other examples, decisions may correspond to node creations, displacements or deletions in trees and graphs, and labeling and link creation/deletion in graphs.

Inference

Incremental models share the very nice property to have very fast inference functions. Most of them perform inference in a purely greedy fashion with the method depicted in Algorithm 4. This algorithm works the following way: start with an *initial partial output* ϵ (line 1). While the current partial output is not complete (line 2), enumerate all partial outputs that we can reach in one elementary step by modifying the current partial output (line 3). For each of those new partial outputs, compute a score reflecting its quality. Then, choose and continue with the best-scored partial output (line 4). These decision steps are repeated until a complete output is built.

Algorithm 4 Greedy inference algorithm.

Require: an input $\mathbf{x}$
Require: the parameters vector θ.
1: $\bar{\mathbf{y}} \leftarrow \epsilon$
2: **while** not IsOutputComplete($\bar{\mathbf{y}}$) **do**
3: nexts $\leftarrow$ Successors($\bar{\mathbf{y}}$)
4: $\bar{\mathbf{y}} \leftarrow \text{argmax}_{\bar{\mathbf{y}}' \in nexts} \langle \theta, \phi(\mathbf{x}, \bar{\mathbf{y}}') \rangle$
5: **end while**
6: **return** $\bar{\mathbf{y}}$

The complexity of greedy inference is $\mathcal{O}(T.s)$ where T is the number of steps required to build a complete output and s is the number of successors at each step. Compared to all dynamic-

programming based methods, this complexity is very low. For example, when dealing with trees, each single step adds one node to the tree: the inference is (only) linear in the number of nodes.

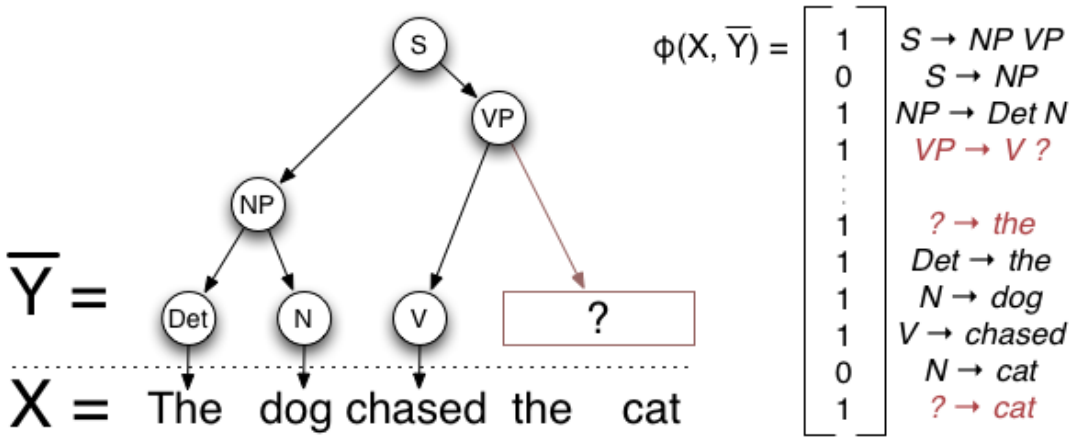


Figure 3.9: **Features example: Text Parsing with a partial output.**

Partial outputs description

As previously, incremental models rely on feature functions $\phi(.,.)$. The descriptions that are used in incremental SP are more general than those of global models: instead of describing input-output pairs, they focus on *input-partial output* pairs. As an example, Figure 3.9 illustrates how the joint-description of Figure 3.7 is augmented to handle partial outputs. The new features, related to the incompleteness of the output, are shown in red.

3.3.2 Incremental Parsing

Incremental Parsing [Collins et Roark, 2004] is one of the first models using the idea of incremental prediction. This model was introduced in the context of natural language parsing, where inputs are sequences of words and outputs are parse trees.

Inference

As most incremental SP methods, Incremental Parsing uses the greedy inference function given in Algorithm 4. Incremental Parsing is built around a Perceptron, which assigns ranking scores to partial outputs.

Training

Training repeats the following process: run the inference function until a wrong decision happens, stop inference and make an elementary correction of the Perceptron:

$$\theta \leftarrow \theta + \phi(\mathbf{x}^{(i)}, \bar{\mathbf{y}}^{(i)}) - \phi(\mathbf{x}^{(i)}, \bar{\mathbf{y}})$$

where $\bar{\mathbf{y}}^{(i)}$ is the correct partial output and $\bar{\mathbf{y}}$ is the predicted partial output. Incremental Parsing does not take the loss function Δ into account and implicitly considers a 0/1 loss.

3.3.3 Learning as Search Optimisation

Incremental prediction was popularized by the *Learning as Search Optimization* (LaSO) algorithm [Daumé III et Marcu, 2005], which is probably the first general Incremental SP model. The authors propose to perform inference with the *beam-search* function given in Algorithm 5. This beam-search algorithm considers multiple partial outputs simultaneously thanks to a queue of partial outputs that are scored by a Perceptron.

Algorithm 5 Beam-search inference algorithm

Require: an input $\mathbf{x}$
Require: the parameters vector θ
Require: a beam size B.
1: partialOutputs $\leftarrow$ MakeQueue(ϵ)
2: **while** partialOutputs $\neq \emptyset$ **do**
3: $\quad\bar{\mathbf{y}} \leftarrow$ RemoveFront(partialSolutions)
4: $\quad$**if** IsOutputComplete($\bar{\mathbf{y}}$) **then**
5: $\quad\quad$**return** $\bar{\mathbf{y}}$
6: $\quad$**end if**
7: $\quad$nexts $\leftarrow$ Successors($\bar{\mathbf{y}}$)
8: $\quad$**for** each $\bar{\mathbf{y}}' \in nexts$ **do**
9: $\quad\quad$score $\leftarrow \langle\theta, \phi(\mathbf{x}, \bar{\mathbf{y}}')\rangle$
10: $\quad\quad$Insert(partialOutputs, $\bar{\mathbf{y}}'$, score)
11: $\quad$**end for**
12: $\quad$KeepBestElements(partialOutputs, B)
13: **end while**
14: **return** *failure*

Inference

The algorithm first initializes the partial outputs queue with the singleton containing the initial partial output (line 1). The algorithm then repeats the following steps until the *partialOutputs* queue is empty (line 2–13). Pop the best-scored partial output from the queue (line 3). If this is a complete output, return it to the user (line 4–6). Otherwise, enumerate the successors of the current partial output (line 7), compute scores for each of them (line 9) and insert them in the queue (line 10). Once the new partial outputs are in the new queue, the algorithm only keeps the B best-scored partial outputs (line 12), where B is the beam-search size parameter.

The parameter B gives a direct control the complexity of the algorithm. Two values of this parameter are of particular interest: $B = 1$ and $B = +\infty$. With a beam-size of one, the beam-search inference algorithm is equivalent to the purely greedy inference algorithm (Algorithm 4). With an infinite beam-size, the algorithm performs exhaustive search – which, in practice, is most of time intractable.

Training

Training in LaSO relies on the concept of *y-good* partial outputs: the partial outputs from which we can reach the correct output. The aim of training is to always have at least one y-good node in the partial outputs queue. Furthermore, the first complete output in the queue should be the correct output. An error is said to occur each time that one of these two conditions are violated.

The training algorithm of LaSO is given in Algorithm 6. The main idea is to launch the inference procedure until an error occurs. For each error, a correction is applied to the parameters, the error is corrected and the inference continues. The authors propose a Perceptron-like update rule:

$$\Delta = \sum_{\bar{\mathbf{y}}' \in Sibs(\bar{\mathbf{y}})} \frac{\phi(\mathbf{x}, \bar{\mathbf{y}}')}{card(Sibs(\bar{\mathbf{y}}))} - \sum_{\bar{\mathbf{y}}' \in partialOutputs} \frac{\phi(\mathbf{x}, \bar{\mathbf{y}}')}{card(partialOutputs)}$$

where $Sibs(\bar{\mathbf{y}})$ is the set of successors of $\bar{\mathbf{y}}$ that are y-good. This rule enforces the scores of the y-good nodes and reduces the scores of the nodes that caused the error. The authors also propose an approximate large-margin update adapted from [Crammer et Singer, 2003]. Training in LaSO does not take the SP loss function into account and implicitly considers a 0/1 loss.

Algorithm 6 LaSO training algorithm

Require: a training set D
Require: a beam size B
1: **while** training **do**
2: $\theta \leftarrow \mathbf{0}$
3: $(\mathbf{x}, \mathbf{y}) \leftarrow$ PickAnExample(D)
4: **while** inference is not finished **do**
5: RunInferenceUntilAnErrorOccur($\mathbf{x}, \theta, B, \mathbf{y}$)
6: $\theta \leftarrow \theta + \Delta$
7: **end while**
8: **end while**
9: **return** θ

3.3.4 Searn

Searn (contraction of *Search-Learn*) [Daumé III *et al.*, 2006] is another general Incremental SP model developed later. In Searn, SP is reduced to multi-class *cost-sensitive* classification. Each decision-making step is seen as a classification problem, with one class per possible successor. This classification problem can be solved with a classifier of any type (*e.g.* Support Vector Machines or Decision Trees).

Inference The inference algorithm of Searn is the greedy algorithm (Algorithm 4), where the choice of the best successor (line 4) is replaced by the classifier decision function.

Training The aim of Searn is to learn a policy (see Section 2.2). In the context of SP, a policy is a decision function to choose between successor partial outputs. Searn uses three kinds of policies:

- **Training policies**. In order to perform training, Searn assumes that we know an *optimal training policy* π^*. For any training input and any partial output, this policy computes:
  - The best(s) successor(s) to choose, *i.e.* the successor from which we can reach the lowest Δ values.
  - The *regret* associated to each decision, *i.e.* the difference between the lowest Δ that we can reach before and after the decision.

  The optimal training policy is only defined for training examples and can make use of the correct outputs. For example, in sequence labeling with Hamming loss, the optimal training policy acts the following way:
  - Whatever the partial output sequence is, choose the next label given by the correct output.
  - The regrets are zero for correct labels and one for other labels, since incorrect label increase the Hamming loss by one.
- **Classifier-based policies**. Any multi-class classifier can be seen as a policy, with one class per possible successor partial output. The aim of Searn is to learn a classifier-based policy that performs well on the SP task.
- **Mixture policies**. The mixture of two policies π_1 and π_2 samples between decisions from π_1 and decisions from π_2:

$$Mixture_{\pi_1,\pi_2,p}(\mathbf{x}, \bar{\mathbf{y}}) = \begin{cases} \pi_1(\mathbf{x}, \bar{\mathbf{y}}), & \text{with probability } p \\ \pi_2(\mathbf{x}, \bar{\mathbf{y}}), & \text{otherwise} \end{cases}$$

Searn can be though as a case of imitation learning. Given an optimal training policy that is *only defined over the training examples*, Searn tries to reach a fully learned policy (*i.e.* a classifier) *able to generalize over unseen examples*. Searn uses an iterative batch-learning approach, which is depicted in Algorithm 7. Each training iteration has three steps: it runs the inference function for all training examples (line 5), it creates a cost-sensitive classification dataset (line 6–12) and updates the current policy (line 14–15).

Algorithm 7 Searn training algorithm

Require: a training set D
Require: an optimal training policy π^*
1: $\pi \leftarrow \pi^*$ ▷ Start with the optimal training policy
2: **while** training **do**
3: $D_{classif} \leftarrow \emptyset$
4: **for** $(\mathbf{x}, \mathbf{y}) \in D$ **do**
5: trajectory $\leftarrow$ RunInference$(\mathbf{x}, \pi)$ ▷ Run inference
6: **for** $\bar{\mathbf{y}} \in$ trajectory **do** ▷ Create classification examples
7: example $\leftarrow$ Create a cost-sensitive multi-class example **where**
8: input $= \phi(\mathbf{x}, \bar{\mathbf{y}})$,
9: classes $=$ Successors$(\bar{\mathbf{y}})$,
10: costs $=$ regrets computed with π^*
11: $D_{classif} \leftarrow D_{classif} \cup$ example
12: **end for**
13: **end for**
14: $\pi' \leftarrow$ TrainClassifier$(D_{classif})$ ▷ Train a new classifier
15: $\pi \leftarrow$ Mixture(π, π') ▷ Mix the new and old policies
16: **end while**
17: **return** $\pi \setminus \pi^*$ ▷ Remove influence of the optimal training policy

Inference Simulation

At the core of training, Searn simulates inference with the current policy π (line 5). Inference returns a trajectory containing the sequence of partial outputs that were explored.

Reduction to classification

For each partial output that has been visited during inference, one multi-class classification example is created (line 7–10). This example has one class per possible successor of the partial output. The cost of predicting each of the possible classes is linked to the regrets computed by the optimal training policy. In order words, the more a classification error reduces our hope to reach low Δ values, the more this error should be penalized.

Policy update

Once a full classification dataset has been built, a new classifier is learned (line 14). This new classifier could directly be used at the next iteration, however for convergence and theoretical issues it is preferable to mix it with the previous policy (line 15), as shown by *Conservative Policy Iteration* reinforcement-learning algorithm [Kakade et Langford, 2002]. In Searn, the authors propose to choose the mixture coefficient with line-search on a validation dataset.

At its first iteration, Searn performs inference with the training optimal policy. This corresponds to an ideal case, where only correct decisions are performed. The following iterations then progressively move away from the training optimal policy to a fully learned policy. After convergence, only the learned part of the policy is kept (line 17).

Searn has been applied to a wide variety of SP tasks and is considered as a state-of-the-art SP method.

3.3.5 Discussion

Incremental models have been developed in order to deal with hard SP problems, when solving Equation 3.1 is intractable. Instead of searching among all possible outputs, they make use of greedy inference. Hence, these methods can cope with arbitrary complex structures, arbitrary dependencies in the joint feature function and any loss function. They can be applied both for large-size problems and for complex dependencies.

However, greedy inference may lead to sub-optimal solutions. For example, when facing local ambiguities, taking an immediate decision is certainly not the good thing to do. Instead, in such cases, global methods explicitly search for the best compromise between all possible solutions.

Furthermore, Incremental Parsing, LaSO and Searn make strong assumptions on the availability of supervision information:

- Incremental Parsing and LaSO assume that we know the *Optimal Learning Trajectories* (OLTs): the set of partial outputs that can lead to the correct outputs (*i.e.* the set of y-good nodes).
- Search assumes that we have access to the *Optimal Learning Policy* (OLP) which an even stronger assumption: for any partial output (that may contain several errors) the optimal learning policy always knows how to choose the best successor.

In some simple SP tasks, the OLTs and the OLP can be computed trivially in $\mathcal{O}(1)$. This is the case for sequence labeling, where, whatever the current partial output is, the best thing to do is to select a correct label. In some other tasks, such as the tree transformation described in XX, OLTs and OLPs computation is a non-trivial or even intractable combinatorial search problem. In particular, computing the OLP means solving the following equation:

$$\pi^*(\bar{\mathbf{y}}) = \underset{\bar{\mathbf{y}}' \in Successors(\bar{\mathbf{y}})}{\operatorname{argmin}} \{ \min_{y \in \mathcal{R}(\bar{\mathbf{y}}')} \Delta(y, y^*) \}$$

where $\mathcal{R}(s) \subset \mathcal{Y}$ is the set of reachable outputs when starting from $\bar{\mathbf{y}}$. In cases where the OLP is not trivially computed, a possibility is to directly solve the search problem given above. This is the solution proposed in [Daumé III *et al.*, 2006] where the authors use a greedy beam-search algorithm for finding an approximate OLP for automatic summarization. However, there are problems where this combinatorial problem can still be too complex.

In summary, the idea of incremental SP is attracting to solve complex SP problems, but suffer from two drawbacks in its current state: the potential sub-optimality of greedy inference and the need for strong supervision assumptions.

3.4 Conclusion

In this chapter, we have introduced the Structured Prediction (SP) problem. SP problems are learning problems where both the input and output objects have an arbitrary structure (*e.g.* sequences, trees, graphs, or lattices). We have detailed on instance of SP: the sequence-labeling task. We have also illustrated the diversity of SP tasks with several examples coming from the natural language processing and document processing fields. We have discussed the two problems of SP: inference (predicting an output given a new input) and training (learning the model given a set of input-output examples).

The remainder of the chapter was an overview of the existing SP models. We have distinguished two families of methods: the *global* methods and the *incremental* methods. Global

methods, including structured Perceptron, CRFs, M^{3}N and SVMISO, rely on the fundamental assumption that we are able to solve the following combinatorial search problem:

$$f_\theta(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}_\mathbf{x}}{\operatorname{argmax}} \langle \phi(\mathbf{x}, \mathbf{y}), \theta \rangle$$

In many applications, solving this search problem is a serious bottleneck, limits the model to small data or imposes restricting assumptions on the description and loss functions.

Instead of modeling *what a good output looks like* and then searching for the best output, the incremental methods directly model *how to build the good output*. Inference is seen as a sequential decision problem. In order to make a prediction, we start with an empty output and grow it incrementally by taking a sequence of construction decisions. We have introduced Incremental Parsing, LASO and SEARN and discussed the assumptions (Optimal Learning Trajectories and Optimal Learning Policy) on which these methods rely. The incremental models have many advantages over global ones, however, when facing local ambiguities, they may suffer from prediction errors due to their greedy inference.

In our overview, we did not detail some particular SP models that are beyond the cope of our work. Let us just cite the work of [Titov et Henderson, 2007], which propose the use of incremental Bayesian network to solve SP. The factor graphs models also deal with some SP problems [Abbeel *et al.*, 2006]. We also skipped the *kernel dependency estimation* model [Weston *et al.*, 2002] were SP is seen as a multi-dimensional regression problem, where both the input space and the output space are kernelized.

In the following chapter, we introduce a new framework for Incremental SP. The two next chapters illustrate our approach experimentally. We first focus on the sequence labeling task introduced in Section 3.1.1. This task, which is probably the simplest one exhibiting a non-trivial structure, allows extensive experiments and comparisons with state-of-the-art work. We show how CR-algorithm can be used to label sequences and discuss the links between our approach and other incremental models. We will show that CR-algorithms profits from all advantages of the incremental approach, while offering multiples ways to bypass the disadvantages of pure greedy inference method. The following chapter will then show the application of CR-algorithms to hard, large-scale real-world SP problems.

4

Choose/Reward Algorithms

Contents

Structured prediction often involves very large search spaces and intensively relies on combinatorial search. A possible approach to solve hard SP problems is to make use of greedy approximate inference. Since greedy optimization may lead to sub-optimal solutions this raises a fundamental question. Do we want to learn a model to accurately describe how good it is to predict $\mathbf{y}$ for the given input $\mathbf{x}$? Or, instead, do we want to learn a model on which our greedy inference procedure gives good outputs? The work of [Wainwright, 2006] suggests that optimal performance is obtained when the methods used for training and testing are appropriately aligned, even if those methods are not independently optimal. For two different inference procedures that solve the same problem, we should learn two different functions that *help* inference to perform well in each case.

In SP, the characteristics of the inference procedure should be taken into account during learning. In order to reach good quality predictions, the central idea of our work is to directly learn the behavior of the inference procedure. We formalize inference procedures as sequential decision making (SDM) problems and investigate the use of SDM learning methods to learn the

inference behavior. The central aspect of our work is the idea of *learning the inference procedure*: we directly relate the learning problem to the form of the inference procedure.

This chapter describes four major contributions of our work:

• **CR-algorithms framework** CR-algorithms is a new framework for describing learning-based inference procedures. Our framework is particularly tailored to the needs of Structured Prediction and gives a high flexibility to describe SP inference procedures. Compared to previous work in Incremental SP, CR-algorithms put the focus on the way to describe inference procedures and associated learning problems simultaneously. CR-algorithms are traditional algorithms augmented with two learning-related instructions: *choose* and *reward*. The former define classification problems and the latter define learning objectives. Learning in CR-algorithms aims at finding the classifiers – corresponding to *choose* instructions – that maximize the expectation of *reward*. This way of writing learning-based inference procedures proves to be simple and concise. Thanks to CR-algorithms it becomes easy to describe a wide range of inference procedures for possibly very complex tasks.

• **Connecting CR-algorithms and MDPs** CR-algorithms can be seen as SDM problems. We propose to formalize the relation between CR-algorithms and SDM problems, through Markov Decision Processes (MDPs). Given a set of inputs, a CR-algorithm implicitly defines an MDP where each *choose* corresponds to decision-making step that should be treated *w.r.t.* the current state of the CR-algorithm. This connection makes it possible to formally define the CR-algorithm learning problem as a policy-learning problem, bridging the gap with various existing work in the field of SDM learning.

• **Reinforcement Learning for Structured Prediction** We propose a classification of learning methods for CR-algorithms. Thanks to the tight relation between CR-algorithms and MDPs, we can bridge the gap between traditional reinforcement learning algorithms and CR-algorithms training. In particular, we focus on approximated reinforcement learning algorithms and show their usefulness in the context of general SP. This reinforcement-learning based approach for SP is an original idea of our work, which requires fewer assumptions than previous methods and is thus applicable to strictly more tasks.

• **An action-ranking policy learning algorithm** CR^{ank} is a new policy learning algorithm that is based on the idea of ranking actions. A ranking machine is a supervised learning machine that learns to rank alternatives given situations. CR^{ank} relies on an original connection between ranking and decision-making: taking a decision can be performed by ranking all the possible actions in the current state and by picking the best one. We show how to learn such an action-ranking function and discuss the multiple advantages of this approach compared to traditional value-base reinforcement learning.

The chapter is structured as follows. The CR-algorithms formalism is introduced in Section 4.1. In particular, Section 4.1.3 connects CR-algorithms to MDPs and Section 4.1.4 states the CR-algorithm learning problem as a policy-learning problem. Section 4.2 makes an overview of various existing methods to solve the CR-algorithm learning problem. We discuss two classes of methods: approximated reinforcement learning methods and incremental SP methods. Section 4.3 presents CR^{ank} : a new policy learning algorithm that tries to capture the best from the various learning methods discussed previously.

4.1 CR-algorithms formalism

CR-algorithms aims at simultaneously defining inference procedures and associated learning problems. CR-algorithms put the inference procedure at the central place, which make them close to classical algorithms describing the inference procedure. In order to introduce learning, CR-algorithms rely on two original constructs, called *choose* and *reward*, that respectively describe classification problems and learning objectives.

4.1.1 Choose and Reward

The *choose* instruction corresponds to a choice between multiple possible actions. Its syntax is the following: *choose instruction*

choose[state] action1, ..., actionN

where *state* are variables that describe the current state of the CR-algorithm and $action_1, \ldots, action_N$ are possible actions among which a learning component should choose. A *choose* could be though as a *switch-case* in C, where the condition will be automatically constructed by a training algorithm:

```
1: switch (??)                          ▷ The condition will be learned
2:     {
3:         case 1: action1; break;
4:         ...
5:         case N: actionN; break;
6:     }
```

The missing conditions of *choose* instructions are learned during a training phase, in order to maximize a learning objective. In order to define this objective into a CR-algorithm, we make use of the *reward* instruction. The syntax of *reward* is given below: *reward instruction*

reward scalarReward

where $scalarReward \in \mathbb{R}$ is a number reflecting the quality of previously made choices. The better the inference procedure is, the higher the rewards will be.

A CR-algorithm is not a classical algorithm, in the sense that it requires a training phase before being executable. This training phase is illustrated in Figure 4.1. Training aims at learning the *choose* functions, in order to maximize the expectation of *reward*s.

4.1.2 Examples of CR-algorithms

Binary Classification CR-algorithm 1 gives one of the simplest use cases of CR-algorithms: the binary classification problem with the 0/1 loss. This CR-algorithm has two inputs: an input $\mathbf{x}$ and its associated class $\mathbf{y}$. In practice, we only know the class $\mathbf{y}$ for a finite amount of training

Figure 4.1: **Training phase in CR-algorithms.** This figure illustrates the training phase in CR-algorithms: given a CR-algorithm and training data, the aim is to learn the *choose* functions of the CR-algorithm. The result of training is a ready-to-use inference procedure.

Training Inputs

data. The correct class $\mathbf{y}$ is called a *Training Input*: an input of the CR-algorithm that is only known during the training phase. CR-algorithm 1 has one output: the predicted class.

CR-algorithm 1 A binary classification problem as a CR-algorithm

Input: An input $\mathbf{x}$
Training Input: The correct class $\mathbf{y} \in \{-1, +1\}$
Output: A predicted class

1: $\hat{\mathbf{y}} \leftarrow$ **choose**[$\mathbf{x}$] -1, +1 ▷ Choose
2: **if training then**
3: **reward** $\mathbb{1}\{\hat{\mathbf{y}} = \mathbf{y}\}$ ▷ Reward
4: **end if**
5: **return** $\hat{\mathbf{y}}$

Choose

Line 1 shows the use of a *choose* instruction to describe the classification problem: given $\mathbf{x}$, the algorithm should choose a class between -1 or $+1$. The result of this choice is then stored in the predicted class $\hat{\mathbf{y}}$. The lines 2–4 form a *training block*: a part of the CR-algorithm that only exists during the training phase. In order to be valid, CR-algorithms can only use training inputs inside such training blocks. The binary-classification learning objective is described thanks to

Reward

the *reward* instruction of line 3. Here, a reward of $+1$ is given if the predicted class is correct and a reward of 0 is given otherwise. If we are not in the training phase, the training inputs are not available and the lines 2–4 are skipped consequently. In any case, line 5 returns the result of inference: the predicted class among -1 or $+1$.

Sequence labeling A fundamental characteristic of CR-algorithms is that *chooses*[1] can be made sequentially. Furthermore, the *state* of a *choose* may depend on previously made choices.

Sequence labeling

CR-algorithm 2 illustrates this on a simple inference procedure for the sequence-labeling task (*e.g.* handwriting recognition or part-of-speech tagging, see Chapter 3).

This CR-algorithm has two normal inputs and one training input. Normal inputs are the input sequence $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ and the set of possible labels $\mathcal{L}$ (*e.g.* all the letters from 'a'

[1] *chooses* is an abuse of language to abbreviate *choose instructions*.

to 'z'). The training input is the correct label sequence $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$. The aim of the CR-algorithm is to predict and return a sequence of labels $\hat{\mathbf{y}}$.

CR-algorithm 2 Left-Right Sequence labeling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

1: $\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
2: **for** t = 1 to card($\mathbf{x}$) **do**
3: $\quad \hat{\mathbf{y}}_t \leftarrow$ **choose**$[\mathbf{x}_t, \hat{\mathbf{y}}_{t-1}]$ $\mathcal{L}$ ▷ Choose the next label
4: **end for**
5: **if training then**
6: $\quad$ **reward** $-\Delta(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Learning objective
7: **end if**
8: **return** $\hat{\mathbf{y}}$

Left-right labeling

Line 1 initializes $\hat{\mathbf{y}}$ with an empty prediction: all the labels $\hat{\mathbf{y}}_t$ are set to the ϵ value to denote that they are not chosen yet. The lines 2–4 predict the sequence of labels in the left-right order: the first label is decided at the first step, the second label is decided at the second step and so forth. Line 3 describes the classification problem: given the current input element $\mathbf{x}_t$ and the previous predicted label $\hat{\mathbf{y}}_{t-1}$, *choose* a label among the set of possible labels $\mathcal{L}$. The result of this choice is then stored into $\hat{\mathbf{y}}_t$. The lines 5–7 are the training-specific part of the CR-algorithm. During training, once all the labels have been decided, the *reward* instruction (line 7) tells how good these choices were. In the case of sequence labeling, this *reward* is directly related to the SP loss function $\Delta(.,.)$.

CR-algorithms can be seen as defining SDM problems: the aim is to find sequences of decisions that maximize the expectation of reward. This parallel leads us to the formalism of Markov Decision Processes (MDPs).

4.1.3 Formalization with Markov Decision Processes

Let $\mathcal{P}$ be a CR-algorithm. We denote $\mathcal{I}_\mathcal{X}$ the Cartesian product of all possible normal inputs of $\mathcal{P}$. Similarly, $\mathcal{I}_\mathcal{Y}$ is the cartesian product all possible training inputs of $\mathcal{P}$.

Given a particular set of inputs $\mathbf{i_x} \in \mathcal{I}_\mathcal{X}$ and $\mathbf{i_y} \in \mathcal{I}_\mathcal{Y}$, the CR-algorithm $\mathcal{P}$ can be seen as implicitly defining an MDP denoted $MDP(\mathcal{P}, \mathbf{i_x}, \mathbf{i_y})$. We adopt the view of an agent that evolves in these MDPs by taking decisions that corresponds to *choose* instructions. Formally, $MDP(\mathcal{P}, \mathbf{i_x}, \mathbf{i_y})$ is defined in the following way:

- The state space $\mathcal{S}$ is the union of all possible values that can be put in the *state* part of *choose* instructions. States correspond to *choose*s: each time their is a *choose*, the agent is in a corresponding state and has to select between various possible actions. The initial state of a CR-algorithm is computing by running the algorithm until the first *choose* occurs. This state is denoted $\mathbf{s}^{initial}(\mathcal{P}, \mathbf{i_x})$. When a CR-algorithm finishes, we enter in a special state: the *final state*. This state is unique and denotes the end of the sequential decision-making problem.
- The set of possible actions $\mathcal{A}_\mathbf{s}$ in a given state $\mathbf{s} \in \mathcal{S}$ is the set of parameters given to the current *choose*. The whole action space $\mathcal{A}$ is the union of possible choices in every possible

state: $\mathcal{A} = \cup_{\mathbf{s}\in\mathcal{S}}\mathcal{A}_\mathbf{s}$. $\mathcal{A}_\mathbf{s}$ is empty for the final state, otherwise it always contains at least one action.

- The transition function $T(\mathbf{s}, \mathbf{a})$ is *deterministic*. The transition function executes the chosen instruction and runs the CR-algorithm until the next *choose* occurs. If the CR-algorithm finishes before any other *choose* occurrence, the transition function lead to the final state.
- The reward function is computed by summing all the scalar values given by *reward* instructions between two states. If there is no *reward* between two successive *choose*s, the reward function has a null value.

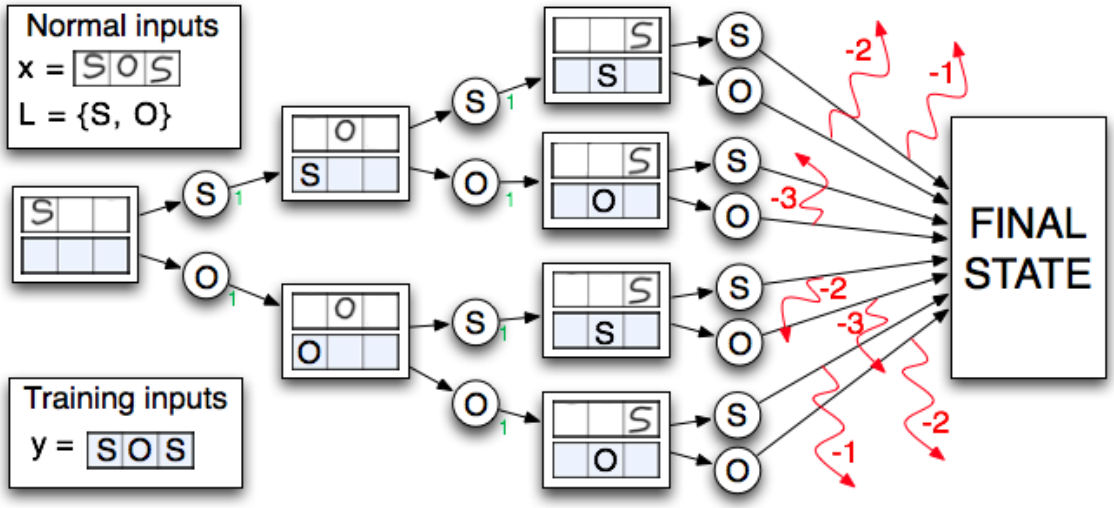


Figure 4.2: **The MDP corresponding to CR-algorithm 2 with given input parameters.** In this example, the input sequence $\mathbf{x}$ and the set of possible labels $\mathcal{L}$ are normal inputs ($\mathbf{i_x} = (\mathbf{x}, \mathcal{L})$) and the correct sequence of labels $\mathbf{y}$ is a training input ($\mathbf{i_y} = (\mathbf{y})$). Boxes are states, circles are actions and arrows correspond to transitions. Green numbers are transition probabilities. Here we only have deterministic transitions. Red numbers are rewards.

An example of $MDP(\mathcal{P}, \mathbf{i_x}, \mathbf{i_y})$ is given in Figure 4.2. This MSP is fully deterministic: the transitions are computed by deterministically executing $\mathcal{P}$ and rewards are also computed deterministically by $\mathcal{P}$. However, the rewards depend on the training inputs, which are only observable during training. From the point of view of the agent, which never sees the training inputs, the rewards are stochastic functions that depend on the conditional probability of training inputs given normal inputs $P[\mathbf{i_y}|\mathbf{i_x}]$.

4.1.4 The CR-algorithm learning problem

The connection with MDPs developed above allows us to define the learning problem of CR-algorithms as a policy-learning problem.

Learning Problem

CR-algorithm learning problem Formally, a CR-algorithm learning problem is a pair $(\mathcal{P}, \mathcal{D}_{\mathcal{I}_\mathcal{X}\times\mathcal{I}_\mathcal{Y}})$ where $\mathcal{P}$ is the CR-algorithm and $\mathcal{D}_{\mathcal{I}_\mathcal{X}\times\mathcal{I}_\mathcal{Y}}$ is a joint distribution over normal inputs

and training inputs. We assume that, in order to be valid, a CR-algorithm always finishes in a finite time. The MDPs induced by valid CR-algorithm have thus a bounded horizon. This makes it possible to use the total reward criterion (see Section 2.2.2). Given a CR-algorithm learning problem, let $\eta(\pi)$ be the expectation of total reward *w.r.t.* policy π:

$$\eta(\pi) = \mathsf{E}_{(\mathbf{i_x},\mathbf{i_y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{\sum_t r(\mathbf{s}_t,\mathbf{a}_t) \mid \pi, \mathbf{i_y}, \mathbf{s}_0 = \mathbf{s}^{initial}(\mathcal{P}, \mathbf{i_x})\}$$

The aim of learning is to find a policy π^* that maximizes η:

$$\pi^* = \operatorname*{argmax}_\pi \eta(\pi)$$

In general the process that generates inputs is unknown, which makes the exact computation of $\eta(\pi)$ unfeasible. Instead, we assume that we have access to a finite number of independently and identically drawn (*i.i.d.*) samples from $\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}$. These samples compose a *training set* denoted $D = \{\mathbf{i}_\mathbf{x}^{(i)}, \mathbf{i}_\mathbf{y}^{(i)}\}_{i\in[1,n]}$. Thanks to the training set, it is possible to approximate the true expectation of cumulative reward with the empirical cumulative reward:

Sampling Assumption

$$\eta(\pi) \simeq \hat{\eta}(\pi) = \frac{1}{n}\sum_{i=1}^{n}(\sum_t r(\mathbf{s}_t,\mathbf{a}_t) \mid \pi, \mathbf{i}_\mathbf{y}^{(i)}, \mathbf{s}_0 = \mathbf{s}^{initial}(\mathcal{P}, \mathbf{i}_\mathbf{x}^{(i)}))$$

CR-algorithms and expected risk minimization There is a direct link between some CR-algorithms and well-known supervised learning problems. For example, the binary classification learning problem (see Chapter 2) is defined through a joint distribution $\mathcal{D}_{\mathcal{X}\times\{-1,+1\}}$ and the aim is to learn a classifier f that minimizes the expectation of classification errors: $\mathsf{E}_{\mathbf{x},\mathbf{y}\sim\mathcal{D}_{\mathcal{X}\times\{-1,+1\}}}\{\mathbb{1}\{f(\mathbf{x}) \neq \mathbf{y}\}\}$. In CR-algorithm 1, the normal input space is $\mathcal{I_X} = \mathcal{X}$ and the training input space is $\mathcal{I_Y} = \mathcal{Y}$. Learning the CR-algorithm aims at maximizing the expectation of total reward $\eta(\pi)$. In this degenerated example of CR-algorithms, we only have one decision-making step, *i.e.* $\eta(\pi)$ only depends on the first state and action. We can thus rewrite $\eta(\pi)$ in the following way:

Binary Classification

$$\begin{aligned}
\eta(\pi) &= \mathsf{E}_{(\mathbf{i_x},\mathbf{i_y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{r(s_1,\pi(s_1)) \mid \pi, \mathbf{i_y}, \mathbf{s}_1 = \mathbf{s}^{initial}(\mathcal{P}, \mathbf{i}_\mathbf{x}^{(i)})\} && (4.1)\\
&= \mathsf{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{r(\mathbf{x},\pi(\mathbf{x})) \mid \pi, \mathbf{y}, \mathbf{x}\} && (4.2)\\
&= \mathsf{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{\mathbb{1}\{\pi(\mathbf{x}) = \mathbf{y}\}\} && (4.3)
\end{aligned}$$

By choosing $\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}} = \mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$, $\eta(\pi)$ becomes the expectation of good classification and the maximization of $\eta(\pi)$ is equivalent to the expectation of classification errors minimization.

In this manuscript, we are particularly interested by CR-algorithms that perform structured prediction. The aim of SP is to learn a prediction function f that minimizes the expected loss $\mathsf{E}_{\mathbf{x},\mathbf{y}\sim\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}}\{\Delta(f(\mathbf{x}),\mathbf{y})\}$. In order to solve the SP problem with CR-algorithms, we usually make use of *reward*s that sum to the negative loss $\Delta(f(\mathbf{x}),\mathbf{y})$ such as in CR-algorithm 2. This way, *maximizing the cumulative reward expectation* becomes equivalent to *minimizing the expected SP loss*:

Structured Prediction

$$\begin{aligned}
\operatorname*{argmax}_\pi \eta(\pi) &= \operatorname*{argmax}_\pi \mathsf{E}_{(\mathbf{i_x},\mathbf{i_y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{\sum_t r(\mathbf{s}_t,\mathbf{a}_t) \mid \pi, \mathbf{i_y}, \mathbf{s}_0 = \mathbf{s}^{initial}(\mathcal{P}, \mathbf{i_x})\} && (4.4)\\
&= \operatorname*{argmax}_\pi \mathsf{E}_{(\mathbf{i_x},\mathbf{i_y})\sim\mathcal{D}_{\mathcal{I_X}\times\mathcal{I_Y}}}\{-\Delta(\mathcal{P}^\pi(\mathbf{x}),\mathbf{y})\} && (4.5)\\
&= \operatorname*{argmin}_\pi \mathsf{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}}\{\Delta(\mathcal{P}^\pi(\mathbf{x}),\mathbf{y})\} && (4.6)
\end{aligned}$$

where $\mathcal{P}^\pi : \mathcal{X} \rightarrow \mathcal{Y}$ is the inference function defined by CR-algorithm $\mathcal{P}$ with policy π.

Other problems

Note that, although most of our work is about SP, CR-algorithms is a rather general framework, in which many MDPs corresponding to other tasks can be described. See chapter [XX Discussion] for examples of CR-algorithms that are not related to SP.

4.2 Learning methods for CR-algorithms

The CR-algorithm formalism is a general way to describe inference procedure and associated learning problems. We have shown that CR-algorithms implicitly describe MDPs and that the learning problem of CR-algorithms can be seen as a policy-learning problem. We now focus on existing learning methods that can be used to solve this policy-learning problem. This sections aims at giving an overview of the spectrum of learning methods that could be applied to CR-algorithms. In particular, we develop a unified view of methods coming from two different fields: reinforcement learning and incremental SP.

We first focus on approximated reinforcement learning methods in Section 4.2.1 whose use in the context of SP is original. We then discuss the adaptations of existing Incremental SP algorithms to the case of CR-algorithms in Section 4.2.2. The fundamental difference between these two approaches is the supervision assumptions they rely on. Different possible level of supervisions are described and discussed in Section 4.2.3. Section 4.2.4 discusses another fundamental difference between the various learning methods for CR-algorithms, which is is the kind of policy representation they rely on.

4.2.1 Approximated reinforcement learning

Approximated Reinforcement Learning

Approximated Reinforcement Learning algorithms (see Section 2.2.4) use function approximation to compactly store policies. The traditional motivation of these algorithms is to cope with large state-spaces and with the curse of dimensionality. When learning CR-algorithmswe put a special emphasis on *generalization*. Contrary to traditional SDM problems, CR-algorithms involve two distinct phases: a training phase and an inference phase. Given only a finite amount of training data, a CR-algorithm policy should be able to generalize on any unseen inputs. The reward function may depend on training inputs and thus be available only during the training phase. This problem is illustrated in Figure 4.3.

General approximate reinforcement learning can be applied to the MDPs induced by CR-algorithms by respecting the two following constraints on the policies:

- The learned policy should not depend on the training inputs. Otherwise, it would not be able to generalize to situations where the training inputs are not observable.
- The learned policy should not depend on the perceived rewards[2]. Since, in general, CR-algorithm rewards depend on the training inputs, they are only available during the training phase. In other words, after training, the environment stops sending reward feedback to the policy. These feedbacks should therefore not be necessary to compute the policy.

In practice, it is sufficient to choose a feature functions for states or state-action pairs that only depends on the current state and the normal inputs. As soon as this property is verified in the feature function, a policy learned by a approximated reinforcement learning algorithm is automatically defined over the whole set of $MDP(\mathcal{P}, \mathbf{i_x}, \mathbf{i_y})$'s.

[2]unless the reward computations do not depend on training inputs. Chapter [XX discussion] gives examples of CR-algorithms where the rewards are available at anytime.

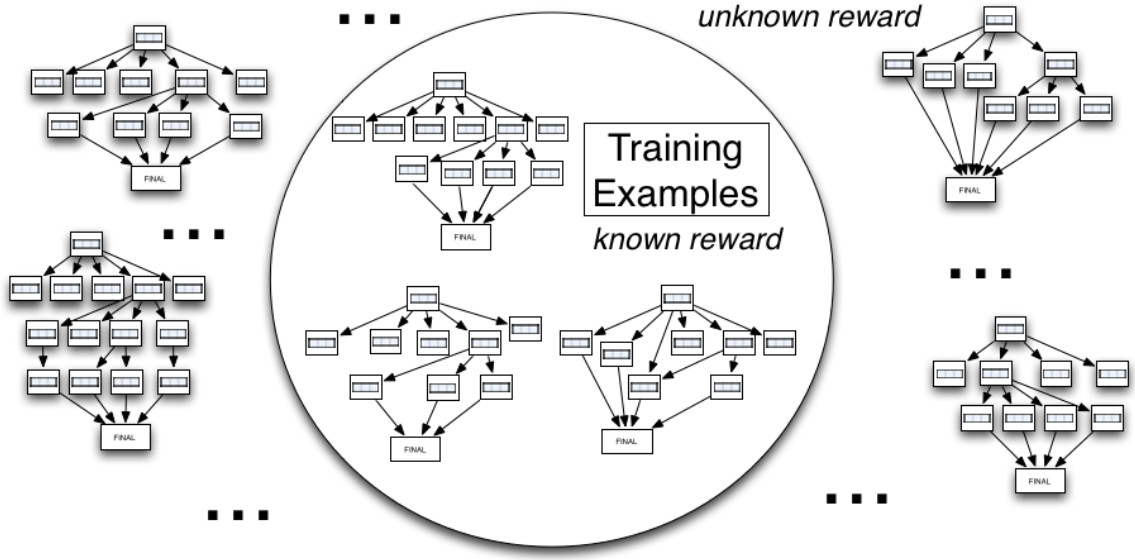


Figure 4.3: **This figure illustrates a whole set of $MDP(\mathcal{P}, \mathbf{i_x}, \mathbf{i_y})$'s.** The big circle denotes states that correspond to training examples: those where the reward function is known. For all MDP's outside this circle, the reward function is unknown. More than a way to cope with the curse of dimensionality, we use function approximation in order to obtain policies able to generalize on the whole set of MDPs given only a small training subset.

Many reinforcement learning algorithms either optimize the γ-discounted reward or the average reward. In order to optimize the total reward, algorithms using γ-discounted reward can be used with $\gamma = 1$, since we assume MDPs with a bounded horizon. The average reward can also be connected to the total reward criterion, see [Garcia et Ndiaye, 1998].

4.2.2 Incremental structured prediction

Reinforcement learning algorithms only use observed *reward*s in order to learn. In many CR-algorithms for SP however, the structure of the problem makes it possible to compute much richer supervision information. It may for example be possible to immediately know what the best choice is, given the current state and the training inputs. The Incremental SP algorithms that were described in the previous chapter, namely Incremental Parsing, LaSO and Searn, use such additional supervision. *Incremental SP*

CR-algorithms are more general than previous Incremental SP problems:

- Previous methods focused on selecting partial outputs until reaching a complete output, *i.e.* states were composed of input and partial outputs. Here, we consider the more general problem of selecting actions in an arbitrary inference procedure described as a CR-algorithm. Any relevant variable of the inference procedures may be included in the states.

- The SP loss Δ is replaced by the more general concept of *reward*. Rewards may be computed at any time in the CR-algorithm and learning involves the maximization of expected cumulative reward. While the SP loss only focuses on the result of inference – the prediction – , rewards can be given at any time during inference. This offers a convenient way to introduce *a priori knowledge* of the problem. It is for example possible to dispatch the loss among the successive decisions, in order to make learning easier. Furthermore, rewards can be given to intermediate states of the CR-algorithms that do not directly have a consequence on the SP losses. An example of such CR-algorithm is the multiple-pass sequence labeling, presented in Chapter 5.

It is possible to adapt the previous Incremental SP models to the framework of CR-algorithms by replacing partial outputs by CR-algorithm *states* and losses by *total rewards*. For example, instead of describing partial outputs $\phi(\mathbf{x}, \bar{\mathbf{y}})$, CR-algorithms involve describing states $\phi(\mathbf{s})$.

In order to perform learning, the Incremental SP models make use of strong supervision assumptions. Depending on the tasks and on the CR-algorithms, such strong supervision may be or may not be available. In the following, we classify the learning methods for CR-algorithms according to the kind of supervision they rely on.

4.2.3 Supervision assumptions

Differences The fundamental difference between the reinforcement learning approach and the Incremental SP approach concern the supervision assumptions that are made. The former approach uses weak supervision in the form of *reward* feedbacks, whereas the latter relies on much stronger supervision, such as – ideally – a function that immediately computes the long-term cost of actions. Depending on the tasks and on the CR-algorithms such stronger supervision may be or may not be easily computed. We overview here multiple supervision assumptions that are relevant for CR-algorithms:

• **Optimal Learning Policy** Availability of the *Optimal Learning Policy* (OLP) is the strongest assumption. Intuitively, it is assumed that for any possible state of the CR-algorithm, *given the training inputs*, we exactly know what the best choice to perform is. The sequence labeling CR-algorithm 2, is an example problem, where the optimal learning policy is easily computed. Indeed, whatever the current state is, the best thing to do is to choose the next label correctly. The best choice at time step t is thus simply determined by taking the correct label $\mathbf{y}_t$ from the training input. Formally the optimal policy is a greedy policy *w.r.t.* the optimal action value:

$$\pi^*(\mathbf{s}, \mathbf{a}) = \pi^{greedy}_{Q^*}(\mathbf{s}, \mathbf{a}) = \operatorname*{argmax}_{\mathbf{a} \in \mathcal{A}_\mathbf{s}} Q^*(\mathbf{s}, \mathbf{a})$$

Having access to the OLP is the ideal case, in which learning a CR-algorithm becomes *imitation* problem: we know the good behavior of the CR-algorithm on a set of training examples and the aim is to generalize this behavior to any possible input.

• **Optimal Learning Trajectories** In many tasks, computing the OLP is either unfeasible or very slow. One weaker assumption is called *Optimal Learning Trajectories*. Here, we assume that for each training example, we have access to the set of *optimal* states $\mathcal{S}^{opt} \subset \mathcal{S}$, which are the states reachable by the optimal policies. Formally, $\mathcal{S}^{opt}$ is the union of the optimal states at all time steps t:

$$\mathcal{S}^{opt} = \bigcup_{t \in [1,T]} \mathcal{S}^{opt}_t$$

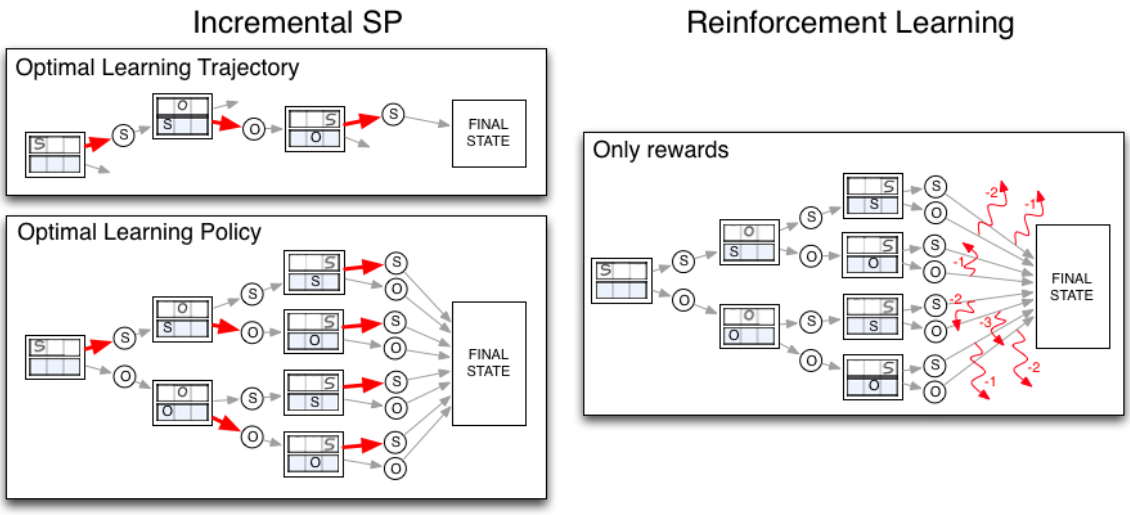


Figure 4.4: **Various supervision assumptions illustrated on the sequence labeling task.** The red color indicate supervision information available in various cases. The left part of the figure illustrates the OLT and OLP assumptions that come from the field of Incremental SP. The right part of the figure illustrates the general reinforcement learning case.

where the optimal states $\mathcal{S}_t^{opt}$ at time step t are defined recursively:

$$\mathcal{S}_{t+1}^{opt} = \bigcup_{\mathbf{s} \in \mathcal{S}_t^{opt}, \pi^*} T(\mathbf{s}, \pi^*(\mathbf{s}))$$

starting from the initial states of the CR-algorithm:

$$\mathcal{S}_1^{opt} = \{\mathbf{s}^{initial}(\mathcal{P}, \mathbf{i}_{\mathbf{x}}^{(i)})\}_{i \in [1, card(D)]}$$

The OLTs assumption is weaker than the OLP assumption. In the latter, the best decisions must be known for every learning state, while, in the former, it is only sufficient to know the best decisions to take in the *optimal* states.

• **Heuristics** In some other tasks, such as the one described in Chapter 6, computing the OLP or OLTs is a non-trivial or even intractable combinatorial search problem. A weaker assumption is the knowledge of *Heuristics*: policies $\hat{\pi}_1, \ldots, \hat{\pi}_n$ that are known to behave *not too bad*. Such policies can be used as a starting point for learning.

• **None** When neither the optimal choices can be computed, nor we have any idea of correct heuristics, we fall in the traditional reinforcement learning case, where only the *reward* can be used for learning.

Figure 4.4 illustrates the various supervision assumptions on the sequence-labeling task. Depending on the tasks and on the CR-algorithms strong supervision may be or may not be available. The choice of the learning method thus crucially depends on which kind of supervision we can compute in a reasonable time. Compared to reinforcement learning based methods, the

Incremental SP methods are less applicable due to their dependence to strong supervision. However, when OLP or OLTs are known, this additional knowledge makes the learning problem much simpler in general.

The generality of CR-algorithms leads to situations ranging from pure reinforcement learning to imitation learning. In order to deal with these various situations, we believe that a general CR-algorithm learning method should be able to deal with any kind of available supervision. CR^{ank}, which is presented in the next section, is one such algorithm, which can deal with various supervision information in a unified manner.

4.2.4 Policy representations

An important difference between the learning methods discussed previously, is the way policies are represented and learned. Reinforcement learning algorithms generally rely on values regression and use value-based greedy policies. Incremental SP, thanks to their rich supervision, make use of discriminative techniques and represent policies thanks to classification or ranking learning machines. Table 4.1 proposes a classification of various learning methods, according to their supervision assumptions and their policy representation.

	OLP	OLTs	Heuristics / None
State regression	-	-	TD(0), TD(λ), LSTD
Action regression	-	-	Fitted QLEARNING, SARSA
State ranking	-	Incremental Parsing, LASO	-
Action ranking	CR^{ank}	CR^{ank}	CR^{ank}
Classification	SEARN	-	RL as classification

Table 4.1: **Learning approaches for CR-algorithms.** The generalizations of existing Incremental SP methods are shown in blue. Incremental Parsing, LASO and SEARN are the algorithms respectively described in Section 3.3.2, Section 3.3.3 and Section 3.3.4. All the other methods are original in the context of SP. Approximated reinforcement learning algorithms are shown in red and are described in Section 2.2.4. Our algorithm CR^{ank}, described in Section 4.3, is shown in black. The rows correspond to various ways to represent and learn policies. The columns correspond to supervision assumptions. OLP stands for Optimal Learning Policy and OLTs stands for Optimal Learning Trajectories.

State/Action regression

Various ways exist to represent a policy as a learning-machine. Most of the approximated reinforcement-learning algorithms work by approximating value functions. In these algorithms, approximation is seen as *State regression* or *Action regression* learning problems. Given an approximated value function, these algorithms make use of the *greedy* policy: the policy that maps states to highest valued actions or successor states.

State/Action ranking

Instead of learning the exact values, a simpler alternative consists in learning ordering functions. *State-ranking* and *Action-ranking* are instances of the ranking problem that was introduced in Section 2.1.2. Both approaches involve learning functions that assign scalar scores to states or actions, similarly to value functions. Only the learning criterion changes between State value (*resp.* Action value) and State ranking (*resp.* Action-ranking): the former is a regression problem and the latter is a ranking problem.

Classification

A last approach, *Classification*, makes the parallel between decision-making (mapping a state to an action) and multi-class classification (mapping an input to a class). Thanks to this connection, it is possible to use standard multi-class classifiers (*e.g.* support vector machines, decision trees) in order to represent policies.

Classification The *Classification* solution is not entirely convincing, since *choose*s allow a number of situations that are not handled by standard multi-class classifiers:

- The set of available choices may depend on the current state. The following chapters will give examples such as choosing words in a sentence or choosing nodes in a tree. In these examples, the *choose* instructions have dynamic lists of choices that depend on the sentence length or on the tree size. *State-dependent classes*
- At any time, new choices may appear depending on the current state. Consequently, the whole set of choices may not be known before execution. *Apriori unknown classes*
- Choices may not only concern discrete values such as in the previous examples, but also general programming instructions. Examples of such instructions include adding/removing nodes from a tree, creating new variables, breaking or continuing a loop and so forth. In general, choices concern structured instructions which are in relation with various other elements of the program. These relations must be taken into account by the learning system. *Structured classes*

Although our setting does not fit with standard multi-class classifiers, representing a policy as a classifier is still a good idea. However, we must replace standard classifier by learning machine able to solve our *generalized* classification problem. In the following, we introduce the action-ranking approach, which is a practical way to handle the generalized classification problem.

Action vs States In order to deal with complex actions, two approaches are possible: learning state values $\hat{V}_\theta(.)$ or learning action values $\hat{Q}_\theta(.,.)$. In the first approach, the greedy policy selects the action that leads to the best-scored successor state[3]:

$$\pi^{greedy}_{\theta(\mathbf{s})} = \underset{\mathbf{a}\in\mathcal{A}_\mathbf{s}}{\operatorname{argmax}}\, \hat{V}_\theta(\phi^{state}(T(\mathbf{s},\mathbf{a}))) \tag{4.7}$$

where ϕ^{state} is a state feature function. In the second approach, the greedy policy directly scores the possible actions, without computing successor states:

$$\pi^{greedy}_{\theta(\mathbf{s})} = \underset{\mathbf{a}\in\mathcal{A}_\mathbf{s}}{\operatorname{argmax}}\, \hat{Q}_\theta(\phi^{action}(\mathbf{s},\mathbf{a})) \tag{4.8}$$

where ϕ^{action} is a joint state-action feature function. From a practical point of view, action-value based policies have two major advantages over state-value based policies. Firstly, they are in general much faster than state-value based policies, since they do not require transitions computations, *i.e.* there is no need for one-step-ahead search. Secondly, they are more general than state-value policies. Indeed, action values can be made equivalent to state values by choosing to describe actions with the description of their successor state $\phi^{action}(\mathbf{s},\mathbf{a}) = \phi^{state}(T(\mathbf{s},\mathbf{a}))$. Instead of computing successor state, it is often more efficient in practice to describe actions by only considering the part of the state their execution would change.

Regression vs Ranking Value functions, during inference, are only used to *sort* possible actions and to *pick the best* one at each step. The exact values of the state or actions do not rely matter; only the order that is induced by the value functions is used. Instead of learning the exact values, the ranking approach directly learn functions $\hat{V}_\theta(.)$ and $\hat{Q}_\theta(.,.)$ that can be used to rank state or actions. As illustrated in Figure 4.5, the exact values of these functions do not matter

[3]We assume deterministic transitions.

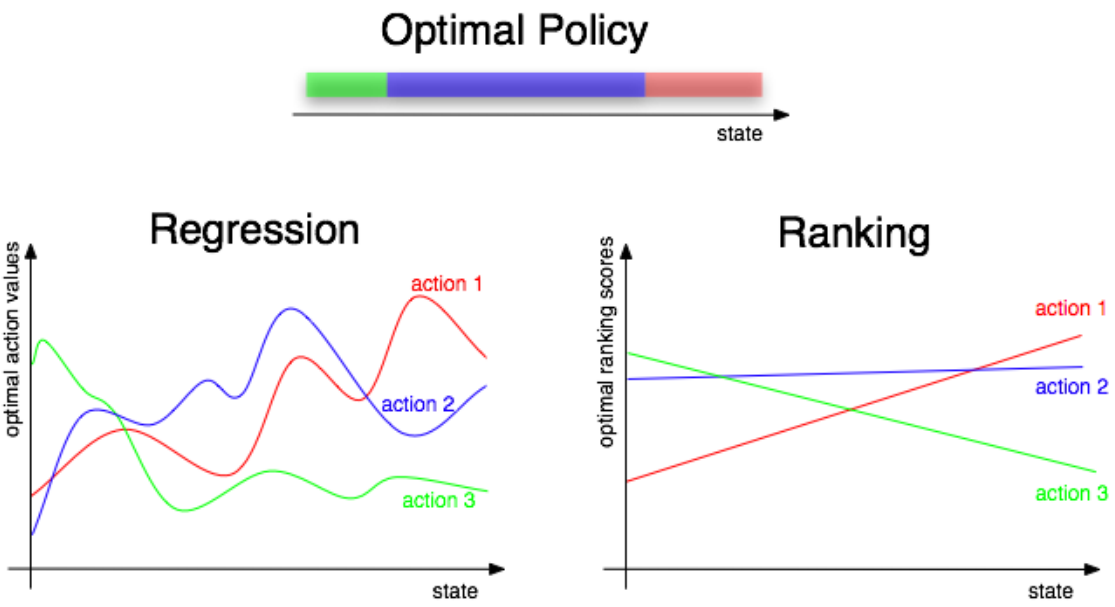


Figure 4.5: **Action value *regression* versus action *ranking*.** We consider an MDP with a one-dimensional state space and three actions *action*1, *action*2 and *action*3. The optimal policy is given on the top of the figure. The optimal action value is illustrated on the left part. In our approach, only the order that is induced by the action-ranking function matters. The right part of the figure gives *one possible* optimal action-ranking function. Since their exist an infinity of optimal action-ranking function, ranking can be solved with much simpler functions than the traditional action-values.

as long as the induced order is correct. Contrary to traditional action values, MDPs admit infinities of optimal ranking functions. This set of optimal ranking functions contains all the possible re-scalings of the optimal action value function. More interestingly, it also contains much more simpler functions than the traditional values. This makes the ranking learning problem *simpler* than the value regression problem, such that we expect better learning performances. Furthermore, ranking models can be trained with discriminative learning methods, such as large-margin methods, which provide good generalization guarantees.

4.3 Action-ranking policy learning: CR^{ank}

In this section, we introduce CR^{ank}: a new algorithm for learning to rank actions in MDPs. CR^{ank} is particularly motivated by the particularities of the MDPs induced by CR-algorithms, but it may be used as a general scheme for learning policies in any finite-length MDPs.

The main novelty of CR^{ank} is to use ranking of actions in the context of sequential decision-making. In order to take decisions, the policies learned with CR^{ank} rank the set of possible actions and pick the top-ranked one. These action ranking functions trained in order to minimize the expectation of *action costs*. The function that computes action costs is a parameter of the algorithm that gives the possibility to handle the various supervision assumptions discussed previously in a unified manner.

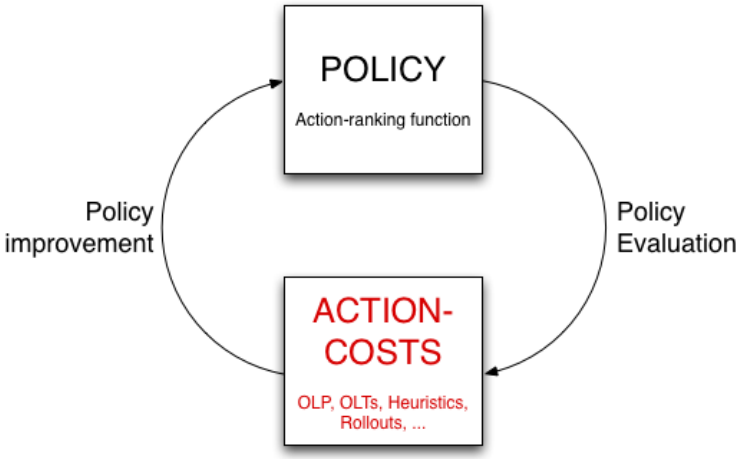


Figure 4.6: **CR^{ank} viewed as generalized policy iteration.** Generalized policy iteration algorithms interleave two kinds of steps: policy evaluation and policy improvement. In CR^{ank}, evaluation is carried out by the action-cost function and improvement corresponds to learning updates on an action ranking function.

CR^{ank} belongs to the family of *generalized policy iteration* algorithms. As illustrated in Figure 4.6, CR^{ank} is constructed around policy evaluation and policy improvement steps. Policy evaluation steps involve the computation of action costs that encapsulate supervision. Policy improvement steps update a set of parameters θ that defines an action ranking function.

Our presentation of CR^{ank} is structured as follows. First, we describe the *ranking of actions* learning problem in Section 4.3.1. In order to solve this learning problem, CR^{ank} make use of a simple stochastic descent gradient rule, which is described in Section 4.3.2. Learning in CR^{ank} aims at minimizing the expectation of the top-ranked action costs. We show in Section 4.3.3 how to exploit available supervision information in the definition of action cost functions. Finally, Section 4.3.4 puts all together a describes the CR^{ank} training loop, which interleaves action cost computation steps with learning steps.

4.3.1 Ranking of actions

CR^{ank} represents policies with action ranking functions $\hat{Q}_\theta(.)$. Such functions induce an order over the possible actions $\mathcal{A}_\mathbf{s}$ given the state $\mathbf{s}$. This order is used to sort the actions and pick the best one at each decision step. In the following, we describe how CR^{ank} makes use of action-costs to learn action-ranking functions.

CR^{ank} makes use of the most common form of ranking which uses a linear prediction function: *Linear ranking*

$$\hat{Q}_\theta(\phi(\mathbf{s},\mathbf{a})) = \langle \theta, \phi(\mathbf{s},\mathbf{a}) \rangle \tag{4.9}$$

The main advantages of linear ranking functions are their simplicity and their good scaling properties. Linear functions make it possible to deal with very large feature sets (we have up to 10^6 distinct features in our applications). Furthermore, linear functions may be non-linear in function of the input, by adding additional features in the feature function $\phi(.,.)$.

For example, if the state is composed of two scalar numbers x_1 and x_2, it possible to include derived quantities such as $x_1 x_2$, $(x_1 * x_2)/(x_1 + x_2)$ or $\sqrt{x_1^2 + x_2^2}$ into the features. With such an augmented representation, while being linear *w.r.t.* the parameters, we are able to express non-linear functions of x_1 and x_2 in this example.

Good action	Bad action	Importance
action1	action3	1
action1	action4	1
action1	action6	4
action2	action1	1
action2	action3	2
action2	action4	2
action2	action6	5
action3	action6	3
action4	action6	3
action5	action1	1
action5	action4	2
action5	action3	2
action5	action6	5

Figure 4.7: **Ranking and binary preferences.** The left part of the figure gives an example ranking over actions in a given state. Each action has an associated cost. Learning aims at top-ranking actions with the lowest cost. The right part of the figure shows the decomposition of the order into binary preferences. Each such preference is made of an action $\mathbf{a}_1$ that should be preferred over action $\mathbf{a}_2$. The importance of each binary preference is the difference of action costs. The binary preferences that are violated in the current ranking are displayed on a red background.

Ranking Example

A *ranking example* in CR^{ank} is composed of a state $\mathbf{s}$, a list of actions $(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ and a list of associated action costs $\mathbf{c} = (\mathbf{c}_1, \ldots, \mathbf{c}_n)$. The costs of actions are computed through a function $c : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that is provided by the user. We thus have: $\mathbf{c} = (c(\mathbf{s}, \mathbf{a}_1), \ldots, c(\mathbf{s}, \mathbf{a}_n))$.

Given a set of parameters θ, we are interested by the cost associated to top-ranked actions, *i.e.* those that the greedy policy π_θ^{greedy} would select. The ranking problem of CR^{ank} consists in the minimization of the expectation of the top-ranked action cost.

Ranking Loss

Ranking losses Ranking can be formalized following the principle of expected risk minimization (see Section 2.1). This implies the use of ranking loss function that measures the quality of possible orders. Ranking losses compute scalar penalties associated to the current ranking scores *w.r.t.* the action costs. In the following, the set of current ranking scores is denoted $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ with $\mathbf{y}_i = \langle \theta, \phi(\mathbf{s}, \mathbf{a}_i) \rangle$. Formally, the loss $\Delta^r(\mathbf{y}, \mathbf{c}) \in \mathbb{R}$ is the amount of penalty that is given to scores $\mathbf{y}$ *w.r.t.* costs $\mathbf{c}$.

Binary Preferences

We focus on ranking losses that decomposes themselves over binary preferences. A binary preference (i, j, β) indicates that action i should be preferred over action j with an importance weight of β. The quality of the current ranking function *w.r.t.* a binary preference depends on the margin $m = \mathbf{y}_i - \mathbf{y}_j$, which is the difference between the two predicted scores. We use a discriminant loss $\Delta^d : \mathbb{R} \to \mathbb{R}^+$ that penalizes negative or too small margins. An example of such loss is the large-margin function that gives a linear penalty in function of the margin violation:

Discriminant Loss

$$\Delta^d(m) = \begin{cases} 1 - m & \text{if } m < 1 \\ 0 & \text{otherwise} \end{cases}$$

Minimizing $\Delta^d(\mathbf{y}_i - \mathbf{y}_j)$ aims at making the score $\mathbf{y}_i$ significantly higher than the score $\mathbf{y}_j$. See Figure 2.9 for other examples of discriminative losses.

The general form of our ranking losses is the following:

$$\Delta^r(\mathbf{y}, \mathbf{c}) = \sum_{(i,j,\beta)} \beta \Delta^d(\mathbf{y}_i - \mathbf{y}_j)$$

We have investigated three decomposition strategies. Each of these strategies represents a family of ranking loss functions, depending on which discriminant loss Δ^d is used. The most frequent idea in ranking is to learn on the basis of the whole set of binary preference [Cohen *et al.*, 1998, Freund *et al.*, 2003]. The *all-pairs* decomposition strategy Δ^{ap} involves one binary preference per pair of actions that have different costs:

All-pairs

$$\Delta^{ap}(\mathbf{y}, \mathbf{c}) = \sum_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i)\Delta^d(\mathbf{y}_i - \mathbf{y}_j)$$

Figure 4.7 illustrates a ranking situation with the corresponding *all-pairs* binary preferences. The *all-pairs* strategy gives an equal importance to the whole order. However, in our case, we are only interest by the top-ranked action. In order to give more importance to the top of the list, we propose two new decomposition strategies: *most-violated-pair* and *best-against-all*.

The *most-violated-pair* strategy Δ^{mvp} only considers the most violated binary preference:

Most-violated-pair

$$\Delta^{mvp}(\mathbf{y}, \mathbf{c}) = \max_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i)\Delta^d(\mathbf{y}_i - \mathbf{y}_j)$$

For example, if we have only good action (with a cost of 0) and bad actions (with a cost of 1), the *most-violated-pair* is composed of the lowest good action and the highest bad action. When Δ^d is a convex decreasing function, the Δ^{ap} and Δ^{mvp} decompositions lead to convex learning problem.

The *best-against-all* strategy Δ^{baa} particularly focuses on the top-ranked action. It creates binary preferences between the top-ranked action and all the other actions that have different costs. Formally, Δ^{baa} is defined the following way:

Best-against-all

$$\Delta^{baa}(\mathbf{y}, \mathbf{c}) = \underbrace{\sum_{i|\mathbf{c}_i<\mathbf{c}_{top}} (\mathbf{c}_{top} - \mathbf{c}_i)\Delta^d(\mathbf{y}_i - \mathbf{y}_{top})}_{i \text{ is better than } top} + \underbrace{\sum_{i|\mathbf{c}_{top}<\mathbf{c}_i} (\mathbf{c}_i - \mathbf{c}_{top})\Delta^d(\mathbf{y}_{top} - \mathbf{y}_i)}_{top \text{ is better than } i}$$

where *top* denotes the top ranked element $\text{argmax}_i \mathbf{y}_i$. Although it is particularly tailored to our problem, the Δ^{baa} strategy has a drawback: it leads to non-convex learning problem, due to the dependency on the top-ranked element *top*. From a theoretical point of view, this is not very satisfactory. However, we will relativize this problem with experimental results that demonstrate that the Δ^{baa} strategy often outperforms the other decomposition strategies.

4.3.2 Learning to rank

In order to learn the ranking function, we rely on the regularized empirical risk minimization principle. In order to minimize the ranking loss, we use a gradient descent scheme. This involve the computation of ranking loss gradients. Let $\Phi_i = \phi(\mathbf{s}, \mathbf{a}_i)$ be the descriptions corresponding to actions $\mathbf{a}_i$. Since we use a linear prediction function, the gradient of all-pair losses can be written:

$$\nabla_\theta \Delta^{ap}(\mathbf{y}, \mathbf{c}, \Phi) = \sum_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i) \nabla_\theta \Delta^d(\mathbf{y}_i - \mathbf{y}_j) \tag{4.10}$$

$$= \sum_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i) \frac{\partial \Delta^d(\mathbf{y}_i - \mathbf{y}_j)}{\partial(\mathbf{y}_i - \mathbf{y}_j)} \nabla_\theta (\mathbf{y}_i - \mathbf{y}_j) \tag{4.11}$$

$$= \sum_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i) \frac{\partial \Delta^d(\mathbf{y}_i - \mathbf{y}_j)}{\partial(\mathbf{y}_i - \mathbf{y}_j)} (\Phi_i - \Phi_j) \tag{4.12}$$

Similarly, the gradient of most-violated-pair losses can be written:

$$\nabla_\theta \Delta^{mvp}(\mathbf{y}, \mathbf{c}, \Phi) = (\mathbf{c}_j - \mathbf{c}_i) \frac{\partial \Delta^d(\mathbf{y}_i - \mathbf{y}_j)}{\partial(\mathbf{y}_i - \mathbf{y}_j)} (\Phi_i - \Phi_j) \mid (i,j) = \max_{i,j|\mathbf{c}_i<\mathbf{c}_j} (\mathbf{c}_j - \mathbf{c}_i) \Delta^d(\mathbf{y}_i - \mathbf{y}_j)$$

The naive computation of these gradients has a complexity in $\mathcal{O}(n^2)$. However, by observing that these gradients are linear combinations of the Φ_i vectors, it is possible to perform only n weighted vector additions, which makes implementation much faster. Furthermore, with discriminative losses such as the large-margin loss, many correctly ranked binary preferences will have zero loss. In general, during training, the better the ranking function is, the more null values appear in the ranking loss. This can be exploited by sorting the set of actions by predicted score. The gradient can then be computed by quickly pruning all binary preferences that do not suffer from loss.

The computation of the best-against-all loss gradient requires only $\mathcal{O}(n)$ steps:

$$\nabla_\theta \Delta^{baa}(\mathbf{y}, \mathbf{c}, \Phi) = \sum_{i|\mathbf{c}_i<\mathbf{c}_{top}} (\mathbf{c}_{top} - \mathbf{c}_i) \frac{\partial \Delta^d(\mathbf{y}_i - \mathbf{y}_{top})}{\partial(\mathbf{y}_i - \mathbf{y}_{top})} (\Phi_i - \Phi_{top}) + \sum_{i|\mathbf{c}_{top}<\mathbf{c}_i} (\mathbf{c}_i - \mathbf{c}_{top}) \frac{\partial \Delta^d(\mathbf{y}_{top} - \mathbf{y}_i)}{\partial(\mathbf{y}_{top} - \mathbf{y}_i)} (\Phi_{top} - \Phi_i)$$

Online learning

CR^{ank} relies on online learning. This choice is motivated by simplicity and scalability concerns. Online learning scales can deal with much bigger training sets than batch methods. Furthermore, online learning allows *continuous* learning: the same ranking function is incrementally improved during the whole training process. Finally, online learning can be implemented very simply by using stochastic gradient descent.

Learning Steps Frequency

A recurrent question in approximated reinforcement learning concerns the frequency of learning steps. Should we update the parameters θ after each decision step, after each episode (*i.e.* each time we reach the final state) or after a whole pass on the training set? Online learning at the *training set* level may be very slow, particularly with large datasets. At the opposite, learning at the *decision* level introduces may introduce a local over-fitting phenomenon: learning on the firsts decisions of an episode may have a direct effect on the behavior of the greedy policy

on latter decisions. The risk is that the greedy policy takes decisions in function of specificities of the current episode instead of using *generalization* over episodes. Due this local over-fitting, the observed value of the current policy may be highly superior to its real performance in generalization. In order to avoid this phenomenon, CR^{ank} performs learning at the *episode* level: it makes one descent gradient step per episode, each time that the final state is reached. *Episode Level*

In order to perform learning at the *episode* level, CR^{ank} relies on the *episode loss*. This quantity is used as an approximation of the standard regularized empirical risk, based on the training examples of the current episode:

$$\hat{R}(\mathbf{y}^{(1)}, \mathbf{c}^{(1)}, \ldots, \mathbf{y}^{(T)}, \mathbf{c}^{(T)}) = \frac{1}{T} \sum_{t=1}^{T} \Delta^r(\mathbf{y}^{(t)}, \mathbf{c}^{(t)}) + \lambda \Omega(\theta)$$

where $\mathbf{y}^{(t)}$ is the vector of predicted scores at time step t and $\mathbf{c}^{(t)}$ is the vector of corresponding costs. The learning steps of CR^{ank}– gradient descent steps on the episode loss – are defined the following way:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \hat{R}(\{\mathbf{y}^{(i)}, \mathbf{c}^{(i)}\})$$

where α is the learning rate parameter that defines the descent speed.

4.3.3 Supervision

In order to deal with the various supervision assumptions that are relevant for CR-algorithms, CR^{ank} is parameterized by an *action-cost* function $c : \mathbf{s} \times \mathbf{a} \rightarrow \mathbb{R}$. Learning in CR^{ank} aims at minimizing the expectation of the costs of the top-ranked actions.

Depending on the kind of supervision that is available, action-cost can take various forms:

- **Optimal Learning Policy**: If we have access to an OLP, we can compute the regrets of actions. A regret $c(\mathbf{s}, \mathbf{a})$ is the amount of reward that is definitively lost when taking action $\mathbf{a}$ in state $\mathbf{s}$. In deterministic MDPs, such as ours, the regret function is defined the following way:

  $$c(\mathbf{s}, \mathbf{a}) = V^*(\mathbf{s}) - V^*(T(\mathbf{s}, \mathbf{a}))$$

  where V^* is the optimal value function: the value function of the OLP. The regret of an action is difference between two terms: the maximum of reward that can be perceived, when starting from $\mathbf{s}$ and the maximum of reward that can be perceived after action $\mathbf{a}$'s execution.

- **Optimal Learning Trajectories**: If we have access to the OLTs, or equivalently to the set of optimal states $\mathcal{S}^{opt}$, we can penalize actions to lead to sub-optimal states:

  $$c(\mathbf{s}, \mathbf{a}) = \begin{cases} 0 & \text{if } T(\mathbf{s}, \mathbf{a}) \in \mathcal{S}^{opt} \\ 1 & \text{otherwise} \end{cases}$$

- **None**: When no additional supervision than the reward is available, we can relate the costs to the regrets *w.r.t.* the current policy. In order to compute these regrets, we must be able to evaluate the current value function V^π. This can be done with rollouts, as described in Section 2.2.4:

  $$V^\pi(\mathbf{s}_t) = \sum_{i \geq 0} r(\mathbf{s}_{t+i}, \mathbf{a}_{t+1}) \mid \mathbf{s}_{t+1} = T(\mathbf{s}_t, \pi(\mathbf{s}_t))$$

Given the V^π value function, the regrets *w.r.t.* π are defined the following way:

$$c(\mathbf{s}, \mathbf{a}) = V^\pi(\mathbf{s}) - V^\pi(T(\mathbf{s}, \mathbf{a}))$$

Note that, since the MDPs induced by CR-algorithms are deterministic, it is sufficient to perform a single rollout to compute the exact value of state. In order to use CR^{ank} in a non-deterministic context, it is possible to estimate state values empirical by averaging the results of several rollouts.

- **Heuristics**: Multiple heuristics $\pi_1, \ldots, \pi_n$ can be combined into a single action-cost function, by performing rollouts of the heuristic and selecting each time the best value:

$$c(\mathbf{s}, \mathbf{a}) = \max_i V^{\pi_i}(\mathbf{s}) - \max_i V^{\pi_i}(T(\mathbf{s}, \mathbf{a}))$$

4.3.4 Algorithm

Algorithm 8 CR^{ank}

Require: a CR-algorithm $\mathcal{P}$
Require: a training set D
Require: an action cost function $c : \mathbf{s} \times \mathbf{a} \to \mathbb{R}$
Require: a learning rate α
Require: a regularizer value λ (default: 0)
Require: a ranking loss Δ^r

$\theta \leftarrow \mathbf{0}$
while training **do**
  $(\mathbf{i_x}, \mathbf{i_y}) \leftarrow$ sampleTrainingData(D)
  $\mathbf{s} \leftarrow \mathbf{s}_0(\mathcal{P}, \mathbf{i_x})$ ▷ initial state
  $\delta \leftarrow \mathbf{0}$ ▷ initialize current gradient
  $t \leftarrow 0$
  while $\mathbf{s}$ is not the final state **do** ▷ perform steps in the MDP
    for $i \in [1, card(\mathcal{A}_\mathbf{s})]$ **do**
      $\Phi_i \leftarrow \phi(\mathbf{s}, \mathbf{a})$ ▷ action's description
      $\mathbf{y}_i \leftarrow \langle \phi(\mathbf{s}, \mathbf{a}), \theta \rangle$ ▷ action's score
      $\mathbf{c}_i \leftarrow c(\mathbf{s}, \mathbf{a})$ ▷ action's cost
    end for
    $\delta \leftarrow \delta + \nabla_\theta \Delta^r(\mathbf{y}, \mathbf{c}, \Phi)$ ▷ update current gradient
    $\mathbf{a} = \pi^{greedy}_{\theta(\mathbf{s})}$ or an exploratory action ▷ select action
    $\mathbf{s} \leftarrow T(\mathbf{s}, \mathbf{a})$ ▷ take action
    $t \leftarrow t + 1$
  end while
  $\theta \leftarrow \theta - \alpha[\frac{1}{t}\delta + \lambda \nabla_\theta \Omega(\theta)]$ ▷ gradient descent step
end while
return π^{greedy}_θ

Parameters CR^{ank} is given in Algorithm 8. It has five parameters. The CR-algorithm $\mathcal{P}$ and the training set $D = \{(\mathbf{i}^{(i)}_\mathbf{x}, \mathbf{i}^{(i)}_\mathbf{y})\}$ implicitly define the family $\{MDP(\mathcal{P}, \mathbf{i}^{(i)}_\mathbf{x}, \mathbf{i}^{(i)}_\mathbf{y})\}$. We have formulated CR^{ank}

in the context of CR-algorithms , but these two parameters may be replaced by any initial states generator, in order to deal with other reinforcement learning problems. The next parameter is the action-cost function $c(.,.)$ that was introduced in Section 4.3.3. The two last parameters are related to learning: α is the learning rate parameter and Δ^r is the ranking loss function.

CR^{ank} is an iterative algorithm that repeats the following three steps: sample a decision problem, run an episode and improve the policy:

• **Initial state sampling** (line 3–4). When using stochastic descent, it is often a good idea to randomize the order of training examples in order to avoid local over-fitting problems. In all our experiments with CR^{ank} , the training set is shuffled each time that all the examples have been processed since the last shuffle. This procedure ensures that all the training examples appear the same number of times.

• **Running an episode** (line 5–17). This step aims at computing the gradient of the episode loss *w.r.t.* the current parameters θ. This is performed incrementally, by summing per-decision gradients over all steps of the episode. For each state that is encountered, CR^{ank} computes the descriptions (line 9), ranking scores (line 10) and costs (line 11) of all available actions, which makes it possible to update the current sum of per-decision gradients (line 13). Once this has been performed, an action is selected depending on the current state (line 14). This can either be the greedy action (the action maximizing $scores[i]$) or an exploratory action (*e.g.* following an ϵ-greedy policy).

• **Policy improvement** (line 18). Once the gradient of the episode loss has been computed, CR^{ank} improves the current policy by applying a gradient-descent step to the parameters θ.

In order to treat one decision-making step, CR^{ank} makes $card(\mathcal{A}_{\mathbf{s}})$ computations of descriptions, ranking scores and action-costs. In practice, most of computing time is spent in $\phi(\mathbf{s}, \mathbf{a})$ computations and, depending on which supervision is used, in $c(\mathbf{s}, \mathbf{a})$ computations. Anyway, these complexities are still low compared to most SP approaches that use global inference.

4.4 Conclusion

In this chapter, we have described the major contribution of our work: CR-algorithms, a formalism for simultaneously describing inference procedures and associated learning problem. Compared to classical algorithms, CR-algorithms make use of two learning-related constructs: *choose* and *reward*. *choose*s correspond to classification problems and *reward*s represent the objective that learning try to maximize. A fundamental characteristic of CR-algorithms, is that *choose*s can be made sequentially before receiving any *reward*. CR-algorithms can be seen as implicitly defining MDPs and the CR-algorithm learning problem can be cast as a policy-learning problem.

We have proposed an overview of various methods that can be applied to learn CR-algorithms. Two sources of learning methods were developed: adapting previous Incremental SP algorithms and using general reinforcement learning algorithms. The main difference between these two approaches concerns the supervision assumptions that are made. Strong supervision assumptions, such as the knowledge of an Optimal Learning Policy, makes learning easier and faster but restrict the applicability of the method. The weakest supervised case is the general reinforcement learning setting, where only the observed rewards can be used for learning. In order to deal with these various situations, we believe that a general CR-algorithm learning method should be able to deal with any kind of available supervision.

The second major contribution that was presented in this chapter is the CR^{ank} algorithm. CR^{ank} is a policy-learning algorithm that is particularly tailored to the particularities of the MDPs induced by CR-algorithms. CR^{ank} relies on a linear ranking machine that defines an order over actions. Up to our knowledge, using action raking in the context of reinforcement learning is a novelty of our work. In order to handle various supervision assumptions in a unified manner, CR^{ank} relies on an action-cost function. By changing the content of this function, CR^{ank} can range from the classical reinforcement learning setting (no supervision assumptions) to imitation learning (OLP).

In the two next chapters, we illustrate the CR-algorithm methodology on two SP tasks: sequence labeling and tree structure mapping. The former is a simple task that allows us to perform extensive experiments and comparisons with state-of-the-art sequence labeling models. The latter involve complex structured domains (XML trees) and deals with very large amounts of data. This task is intended to demonstrate the scalability of the CR-algorithm in a general reinforcement learning case.

5

Sequence Labeling with CR-algorithms

Contents

Sequence labeling is the generic task of assigning labels to the elements of a sequence. This structured prediction (SP) task corresponds to a wide range of real world problems. For example, in the field of natural language processing, *part of speech tagging* consists in labeling the words of a sentence as nouns, verbs, adjectives, adverbs, etc. Other examples in NLP include: chunking sentences, identifying sub-structures, extracting named entities, etc. Information extraction systems can also be based on sequence labeling models. For example, one could identify relevant and irrelevant words in a text for a query need. Sequence labeling also arises in a variety of other fields[1] (character recognition, user modeling, bioinformatics, ...).

In this chapter, we illustrate the use of CR-algorithms in the context of sequence labeling and demonstrate experimentally the multiple advantages of this approach. We use the same notations as in Chapter 3: input sequences are denoted $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, correct output sequences are denoted $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ and predicted output sequences are denoted $\hat{\mathbf{y}} = (\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_n)$. The set of possible labels is denoted $\mathcal{L}$. The aim is to learn a prediction function $f : \mathcal{X} \to \mathcal{Y}$ that leads to low expected risk. In the following, we consider sequence labeling with the Hamming loss.

[1] See [Dietterich, 2002] for an exhaustive overview of sequence labeling models and applications.

In order to measure the quality of a prediction, this loss function counts the number of wrong predicted elements:

$$\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y}) = card(\{i \in [1, T], \hat{\mathbf{y}}_i \neq \mathbf{y}_i\})$$

This chapter is structured as follows. Section 5.1 introduces a first CR-algorithm for sequence labeling, which performs left-to-right labeling. This simple CR-algorithm is compared with six baseline models on five datasets. CR-algorithms is a general framework, in which more complicated inference procedures than the left-to-right labeling can be described. We show experiments with two other CR-algorithms: order-free labeling and multiple-pass labeling, and discuss several possible extensions in Section 5.2. Various aspects of the learning process are discussed in Section 5.3.

5.1 Left-to-right sequence labeling

A simple way to predict a sequence of labels is to predict the labels sequentially, from left to right. In this approach, at each time step t, a label $\hat{\mathbf{y}}_t$ is predicted on the basis of the input $\mathbf{x}$ and the previous predicted labels. In the following, we describe the left-to-right labeling CR-algorithm and present various experimental results with CR^{ank}.

5.1.1 Left-to-right CR-algorithm

In order to fully describe the left-to-right sequence labeling model, we describe the CR-algorithm (the inference procedure and associated learning problem) and the way to perform learning: the description and supervision functions.

CR-algorithm Left-to-right labeling is given in CR-algorithm 3: we start with an empty prediction (line 1). Then, for each element of the sequence, we *choose* a label among the set of possible labels $\mathcal{L}$ (line 4). Once all the labels have been predicted, the CR-algorithm returns the predicted sequence $\mathbf{y}$ (line 9).

CR-algorithm 3 Left-to-right Sequence labeling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

1: $\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
2: n $\leftarrow$ card($\mathbf{x}$)
3: **for** t = 1 to n **do**
4: $\quad \hat{\mathbf{y}}_t \leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{t-C}, \ldots, \hat{\mathbf{y}}_{t-1}, t]$ $\mathcal{L}$ ▷ Choose the next label
5: **end for**
6: **if training then** ▷ Learning objective:
7: $\quad$ **reward** - $\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Hamming Loss
8: **end if**
9: **return** $\hat{\mathbf{y}}$

Usually, sequence models (Hmm, Crfs, SvmIso) rely on the first-order Markov assumption: a label $\mathbf{y}_t$ only interacts with its previous label $\mathbf{y}_{t-1}$ and its next label $\mathbf{y}_{t+1}$. This assumption is

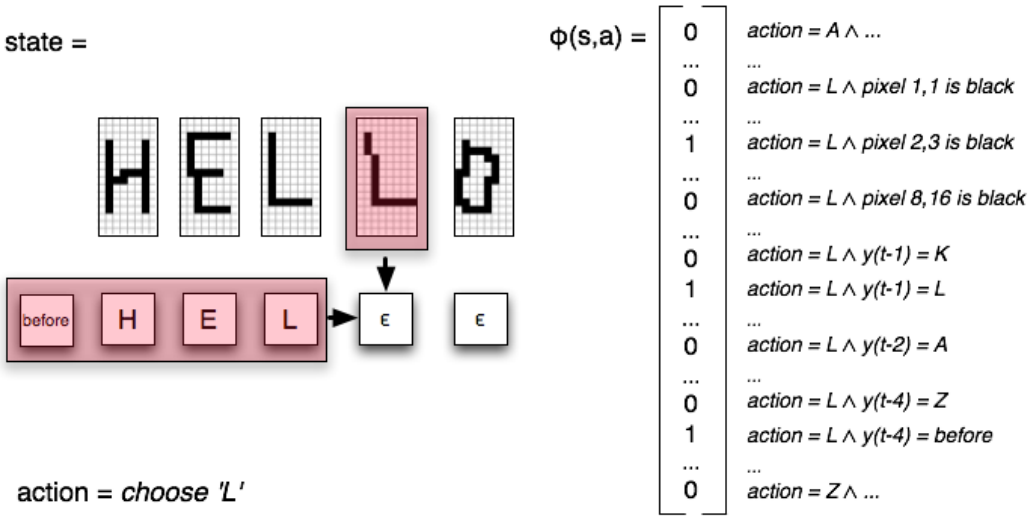


Figure 5.1: **Feature function for CR-algorithm 3.** The current state contains the input sequence $\mathbf{x}$ (handwritten letters) and the current prediction $\hat{\mathbf{y}} = (H, E, L, \epsilon, \epsilon)$. The current action consists in recognizing the next letter as a L. The joint description of this state-action pair is given on the right. It contains content features (those related to the handwritten digits) and structural features (those related to previous predictions).

required in order to perform exact inference with the Viterbi algorithm. When performing greedy inference, we are not limited by such concerns. Hence, we can very easily introduce long-term dependencies. In CR-algorithm 3, the parameter C controls the length of the dependencies on which the labeling process relies.

Long-term dependencies

Action Descriptions In order to apply CR^{ank} or any other action-value based learning algorithms, we need a joint state-action feature function: $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$. The feature function that we use in our experiments is illustrated in Figure 5.1. We use content features and structural features:

Feature function

- The content features compute a joint aspect of the action and the current input element $\mathbf{x}_t$. Features related to input elements depend on the tasks: they might correspond to pixel values in handwritten recognition or word prefixes and suffixes in part-of-speech tagging. In Figure 5.1, input elements are black-and-white bitmaps and we use one feature $f_{l,p}$ per possible label $\in \mathcal{L}$ and per possible pixel position p:

Content Features

$$f_{l,p}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if chosen label} = l \wedge \text{pixel } p \text{ is black in } \mathbf{x}_t \\ 0 & \text{otherwise} \end{cases}$$

- The structural features focus on previously predicted labels. In left-to-right labeling, we use one feature per possible label pair $(l_1, l_2) \in \mathcal{L}^2$ and per possible context position[2]

Structural Features

[2]Note that $\hat{\mathbf{y}}_{t-\delta}$ may sometimes not be defined due to border effects. In order to handle all cases in a uniform way, we introduce a special label, called `Before`, to denote elements that are before the beginning of the sequence (*i.e.* $\delta > t$).

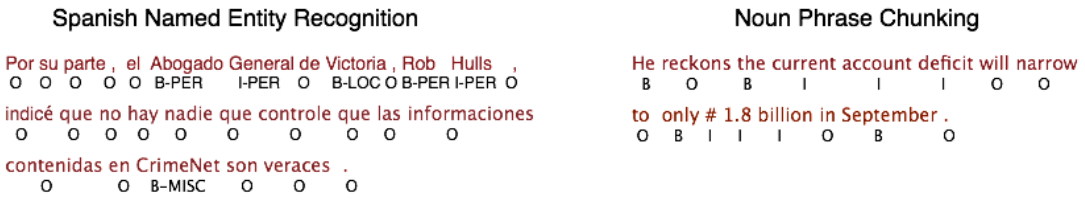


Figure 5.2: **Examples of training sequences.** The left part of the figure gives an example sequence from the NER corpus. This sequence contains the following labels: Outside, Begin-PERson, Inside-PERson, Begin-LOCation and Begin-MISC. The right part of the figure gives an example from the Chunk corpus. Three labels are possible: Begin, Inside and Outside a noun-phrase.

$\delta \in [1, C]$:

$$f_{l_1,l_2,\delta}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if chosen label} = l_1 \wedge \hat{\mathbf{y}}_{t-\delta} = l_2 \\ 0 & \text{otherwise} \end{cases}$$

Sparsity

The feature space induced by content and structural features may be very large. With $card(\mathcal{L})$ distinct labels and d input features, we have $card(\mathcal{L}) \times d$ content features and $card(\mathcal{L})^2 \times C$ structural features. A major characteristic of the feature functions we use is *sparsity*: for a given state-action pair, most of the features have null values. In practice, we only consider *active features*: those that have non-null values. As an example, on the $card(\mathcal{L})^2 \times C$ structural features, only C features are active in a given state-action pair. Sparseness can be exploited through sparse vectors data structure, which, in practice leads to quite efficient implementations.

Optimal Learning Policy

Supervision Left-to-right labeling with the Hamming loss has a very nice property: it is easy to compute the optimal policy function on training examples. Indeed, whatever the current state is, the best action (to maximize the reward) is to select the next label correctly. If we start from time step t and then choose all the succeeding labels correctly, the optimal value is:

$$V^*(\mathbf{s}) = V^*((\mathbf{x}, \hat{\mathbf{y}}, \mathbf{y}, t)) = -card(\{i \in [1, t], \hat{\mathbf{y}}_i \neq \mathbf{y}_i\})$$

Optimal Regrets

i.e. the optimal values only depend on the errors that occurred before time step t. Since we know the optimal value function, we can compute regrets associated to actions. We consider two cases: either the chosen label is correct or we made a prediction error. In the former case, the hamming loss will not increase and the regret equals zero. In the latter case, the optimal value decreases by one: the regret equals +1. We can thus supervise CR^{ank} with optimal regrets:

$$c(\mathbf{s}, \mathbf{a}) = c((\mathbf{x}, \hat{\mathbf{y}}, \mathbf{y}, t), l) = \begin{cases} 0 & \text{if } \mathbf{y}_t = l \\ 1 & \text{otherwise} \end{cases}$$

5.1.2 Datasets

We performed experiments on three standard sequence labeling datasets that correspond to five train/test splits.

	Training Set			Evaluation Set				
	Sequences	Elements	Elts/Seq	Sequences	Elements	Elts/Seq	Labels	Features
NER-SMALL	300	7059	≈ 23.5	8,323	264,715	≈ 31.8	9	175,531
NER-LARGE	8,323	264,715	≈ 31.8	1,517	51,533	≈ 34.0	9	188,248
HANDWRITTEN-SMALL	626	4,617	≈ 7.4	6,251	47,535	≈ 7.6	26	128
HANDWRITTEN-LARGE	6,251	47,535	≈ 7.6	626	4,617	≈ 7.4	26	128
CHUNK	8,936	211,727	≈ 23.8	2,012	47,377	≈ 23.5	3	143,310

Table 5.1: **Statistics of the Sequence Labeling corpora.** From left to right: number of sequences and elements for the training and testing sets, number of input features and number of labels.

• **Spanish Named Entity Recognition (Ner)** This dataset, illustrated in Figure 5.2 (left), is composed of sentences in Spanish, labeled with named entities. Named entities are phrases that contain the names of persons, organizations, locations, times and quantities (9 labels). This dataset was introduced in the CoNLL 2002 shared task[3] where the aim was to develop language-independent NER taggers. We used two train/test splits: NER-LARGE is the original split, composed of 8,324 training sentences and 1,517 test sentences. In order to compare our model with baseline methods that cannot handle such a large dataset, we also used the NER-SMALL split, with a random selection of 300 training sentences, the 9541 remaining sentences forming the test set. This corresponds to the experiments performed in [Daumé III *et al.*, 2006] and [Tsochantaridis *et al.*, 2004]. Input features include word descriptions, suffixes and prefixes.

• **Noun phrase chunking (Chunk)** This dataset comes from the CoNLL-2000 shared task[4]. As illustrated in Figure 5.2 (right), the aim is to divide sentences into non-overlapping phrases. In this task, each *chunk* consists of a noun phrase. This task can be seen as a sequence-labeling task thanks to the "BIO encoding": each word can be the Beginning of a new chunk, Inside a chunk or Outside chunks. This standard dataset put forward by [Ramshaw et Marcus, 1995] consists of sections 15-18 of the Wall Street Journal corpus as training material and section 20 of that corpus as test material. Input features are similar to the previous ones and we consider one additional feature per surrounding word that corresponds to the *part of speech* of the word.

• **Handwriting Recognition (HandWritten)** This corpus was created for handwriting recognition and was introduced by [Kassel, 1995]. Here, sequences are handwritten words and labels are possible characters of the 26-letter alphabet. The dataset corresponds to a *limited vocabulary* setting: there are only 55 different words. Most of the words have been written by 150 subjects leading to a total of 6900 sequences of handwritten characters (more than 52,000 characters). Each character is a 8×16 black-and-white pixels images. As in [Daumé III *et al.*, 2006], we used two variants of the set: HANDWRITTEN-SMALL is a random split of 10% words for training and 90% for testing. HANDWRITTEN-LARGE is composed of 90% training words and 10% testing words. For both splits, the 55 possible word appear at least once in the training set. Letters are described using one feature per pixel, as in Figure 5.1.

The statistics of our five corpora are summarized in Table 5.1.

[3] `http://www.cnts.ua.ac.be/conll2002/ner/`
[4] `http://www.cnts.ua.ac.be/conll2000/chunking/`

	Independent Classification			Structured Prediction		
	L2-MAXENT	L1-MAXENT	SVM	CRF	SVMISO	Simple Searn
NER-SMALL	93.00	92.87	93.26	91.86	93.45	**93.8**
NER-LARGE	96.57	96.75	-	**96.96**	-	96.3
HANDWRITTEN-SMALL	71.65	71.20	**77.59**	66.86	76.94	64.1
HANDWRITTEN-LARGE	78.69	78.56	**82.78**	75.45	-	73.5
CHUNK	96.36	96.56	-	**96.71**	-	95.0

Table 5.2: **Baselines scores for Sequence Labeling.** We considered two approaches: *independent classification* and *structured prediction*. The former treats each element as an independent prediction problem, while the latter try to capture interdependencies between the neighboring elements. All the scores are percentages of correctly predicted labels in the test set. The − symbol denotes experiments that failed due to their excessive memory or CPU time requirement.

5.1.3 Baselines

In order to compare our models, we have computed baselines that correspond to two approaches: *independent classification* and *structured prediction*.

Independent Classification When applied to sequence labeling, SP methods try to exploit the sequential structure in order to improve the predictions accuracy. In order to measure the benefit of using this structure, we consider a simple series of baselines which ignore the interactions between labels. In this case, any traditional multi-class classifier may be used to predict the labels independently. We give results for maximum entropy classifiers and support vector machines. L1-MAXENT and L2-MAXENT[5] are maximum entropy classifiers, which differ in the regularizer and learner they use. The former is regularized with the L2-norm of the parameters and trained with the L-BFGS optimization procedure. The latter is regularized with the L1-norm of the parameters and trained with the OWLQN method proposed recently in [Andrew et Gao, 2007]. We tried regularizer values varying from 10^{-7} to 10^{-2} and selected the best scores. *LibSVM*[6] is the traditional batch support vector machine classifier with linear kernels. We tried C values ranging from 10^{-4} to 10^{4} and selected the best scores.

Maxent

Support Vector Machines

Structured Prediction We have used three baseline methods, which are described in Chapter 3, in order to compare our models: CRFs, SVMISO and SEARN. For CRFs, we used the FlexCRFs [Phan et Nguyen, 2005] implementation – which is freely available online – with default parameters. We compared with discriminant training of the SVMISO approach, thanks to the implementation given by the authors: SVM^{Hmm}[7]. For each dataset, we tried three values of the C parameter: 0.01, 1, and 100, and kept only the best results. Our last baseline is a "*very simply and stripped down implementation of Searn*"[8]. This implementation is limited to sequence labeling with Hamming loss and its base learner is an averaged Perceptron.

We trained each baseline and computed the percentage of correctly labels on the test set. The results are summarized in Table 5.2. Surprisingly, the SP methods do not always perform better than the much more simpler independent classifiers. As it will be shown in the following,

[5]Implementation from `http://nieme.lip6.fr`.
[6]Implementation from `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.
[7]Implementation from `http://svmlight.joachims.org/svm_struct.html`.
[8]Implementation from `http://searn.hal3.name`.

	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
Best baseline	93.8	96.96	77.59	82.78	96.71
CR-algorithms	93.78	97.03	75.78	83.06	96.66

Table 5.3: **Left-to-right sequence labeling with CR-algorithms compared to the best baseline results.** The first row gives the test-accuracies (the percentage of correctly labeled elements in the test set) of the best baseline. The second row gives the test-accuracies of the left-to-right CR-algorithm with a context size of $C = 1$, trained with CR^{ank}.

the contribution of the structure to the accuracy differs following the datasets. Similar results were obtained in [Nguyen et Guo, 2007].

5.1.4 Experiments

Due to their use of Viterbi, CRFs and SVMISO rely on the first-order Markov assumption. Searn, for comparison, was also executed with a first-order Markov assumption. To be comparable, we thus also start with first-order dependencies (*i.e.* in CR-algorithm 3: C = 1).

First Order Dependencies Table 5.3 compare CR^{ank} to the best baselines models presented above. It can be seen that the CR-algorithm approach is competitive with state-of-the-art on four datasets over the five. On two datasets, we have slightly better results than the baselines (+0.07 % on NER-LARGE and +0.28 % on HANDWRITTEN-LARGE). Figure 5.3 shows the training behavior of CR^{ank} . For each dataset, we display the train and test accuracies (*i.e.* the percentage of correctly labeled elements) in function of the number of training iterations. A training iteration corresponds to a whole pass on the dataset.

Depending on the datasets, most of the learning is generally done during the ten or twenty first iterations (*e.g.* On NER-SMALL, we have 92.8% accuracy after 10 iterations, 93.4% accuracy after 20 iterations, and 93.8% at convergence). As usual in supervised learning, training scores are higher than test scores. This effect is related to over-fitting and its amount reduces with the number of training examples. For example in the NER-LARGE and HANDWRITTEN-LARGE corpora, the differences between train and test scores are much smaller than in the corresponding NER-SMALL and HANDWRITTEN-SMALL corpora.

Parameters In order to obtain the previous results, we have selected the following parameters for CR^{ank}:

- Since we deal with corpora having from 128 to 188,248 input features, it is crucial to properly tune the learning rate in each case. Experimentally, we found that the invert of the average L1-norm of feature vectors is a good heuristic that works in all cases:[9] *Learning Rate*

$$\alpha = \mathsf{E}_{\mathbf{s},\mathbf{a}\sim\pi}\{\sum_i |\phi(\mathbf{s},\mathbf{a})_i|\}$$

- Ranking losses define the way ranking scores are updated in CR^{ank}. In the previous chapter, we introduced three decomposition strategies: all-pairs, most-violated-pair and best-against-all. Each of these strategies can be combined with a discriminant loss Δ^d. By default, we use the Δ^{mvp} ranking loss with the large-margin discriminative loss. The impact of ranking losses is studied below. *Ranking Loss*

[9] This heuristic works well in practice, furthermore it is easy to compute incrementally.

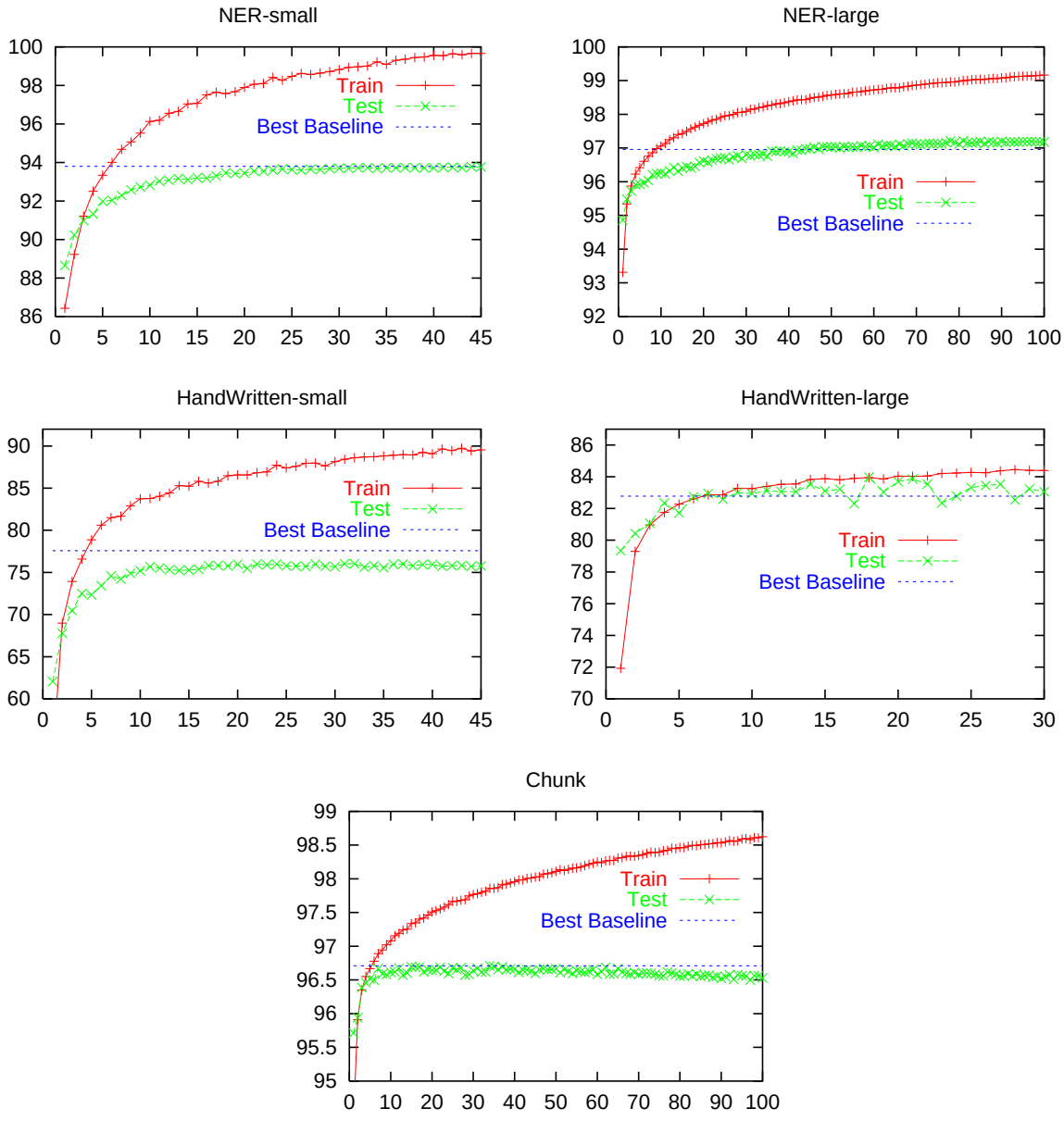


Figure 5.3: **Train and test accuracies during CR^{ank} training with the left-to-right labeling CR-algorithm.** The X-axis is the number of passes on the training dataset that have been performed. The Y-axis is the accuracy: the percentage of correctly predicted labels. We compare CR^{ank} train and test accuracies with the best baseline test accuracy from Section 5.1.3.

Loss		Context = 0		Context = 2		Context = 8	
Base Loss	Ranking Loss	Train	Test	Train	Test	Train	Test
Large-margin	Most violated pair	83.56	72.56	92.77	77.98	96.56	**83.04**
	All pairs	81.91	68.52	92.81	75.70	**98.51**	80.51
	Best against all	**85.16**	**72.60**	**93.29**	77.88	97.36	82.51
Perceptron	Most violated pair	77.41	65.24	89.43	73.37	96.4	79.42
	All pairs	72.10	63.05	88.07	70.93	97.64	77.72
	Best against all	72.75	64.69	83.73	72.43	88.87	76.78
Log-binomial	Most violated pair	81.16	71.03	88.11	77.21	93.63	82.32
	All pairs	78.90	69.02	88.95	76.25	93.42	80.71
	Best against all	82.93	72.09	89.86	**78.10**	93.02	82.10
Exponential	Most violated pair	78.06	68.61	88.76	76.73	96.53	83.02
	All pairs	/	/	/	/	/	/
	Best against all	/	/	/	/	/	/

Table 5.4: **Impact of the ranking loss, HandWritten-Small dataset.** This table gives the train and test accuracies of CR^{ank} after 50 iterations with various ranking losses. Ranking losses are combinations of decomposition strategies Δ^r and discriminative losses Δ^d. For each possible combination, we give train and test accuracies for three different context sizes: $C = 0$, $C = 2$ and $C = 8$. The / symbol denotes experiments where the parameters θ diverged.

- We observed experimentally that, on our datasets, adding a regularization term does not significantly improve the test-accuracies. Therefore and for the matter of simplicity, we only deal with unregularized CR^{ank} ($\lambda = 0$) in the following.

Higher order dependencies One of the major advantages of greedy inference over exact inference is that we can easily incorporate higher-order dependencies in the feature function. The impact of the context size C – the number of predicted labels on which the feature function depends– on the training and testing accuracies of CR^{ank} are shown in Figure 5.4. These results are similar to those that can be found in [XX effet taille du contexte]. In some datasets, such as NER and CHUNK, the optimal values of C are small (0, 1 or 2) and too much context may slightly degrade the performances. However, on some other datasets such as HANDWRITTEN, increasing the context size has a major impact. Thanks to long-term dependencies, CR^{ank} significantly outperforms the baselines: +5.8% accuracy on HANDWRITTEN-SMALL and +8.3% accuracy on HANDWRITTEN-LARGE.

Ranking Losses We have tried all the combinations between the three decomposition strategies Δ^{mvp}, Δ^{ap} and Δ^{baa}, and four disciminative losses: the *Perceptron* loss, the *large-margin* loss, the *log-binomial* loss and the *exponential loss*. The train and test accuracies for all possible combinations are given in Table 5.4 for the HANDWRITTEN-SMALL dataset and Table 5.5 for the NER-SMALL dataset.

Most of our experiments using the *exponential* loss failed due to parameters divergence. This is related to our learning method: stochastic gradient descent. The exponential loss may lead to very big gradients, which introduce a lot of stochastically in the learning process. In an ideal world, this should not diverge. We believe that the problem comes from numerical error, which are propagated and amplified by the *exponential* loss. *Exponential*

The *large-margin* loss always significantly outperforms the *Perceptron* loss on test accuracies. *Large-margin*

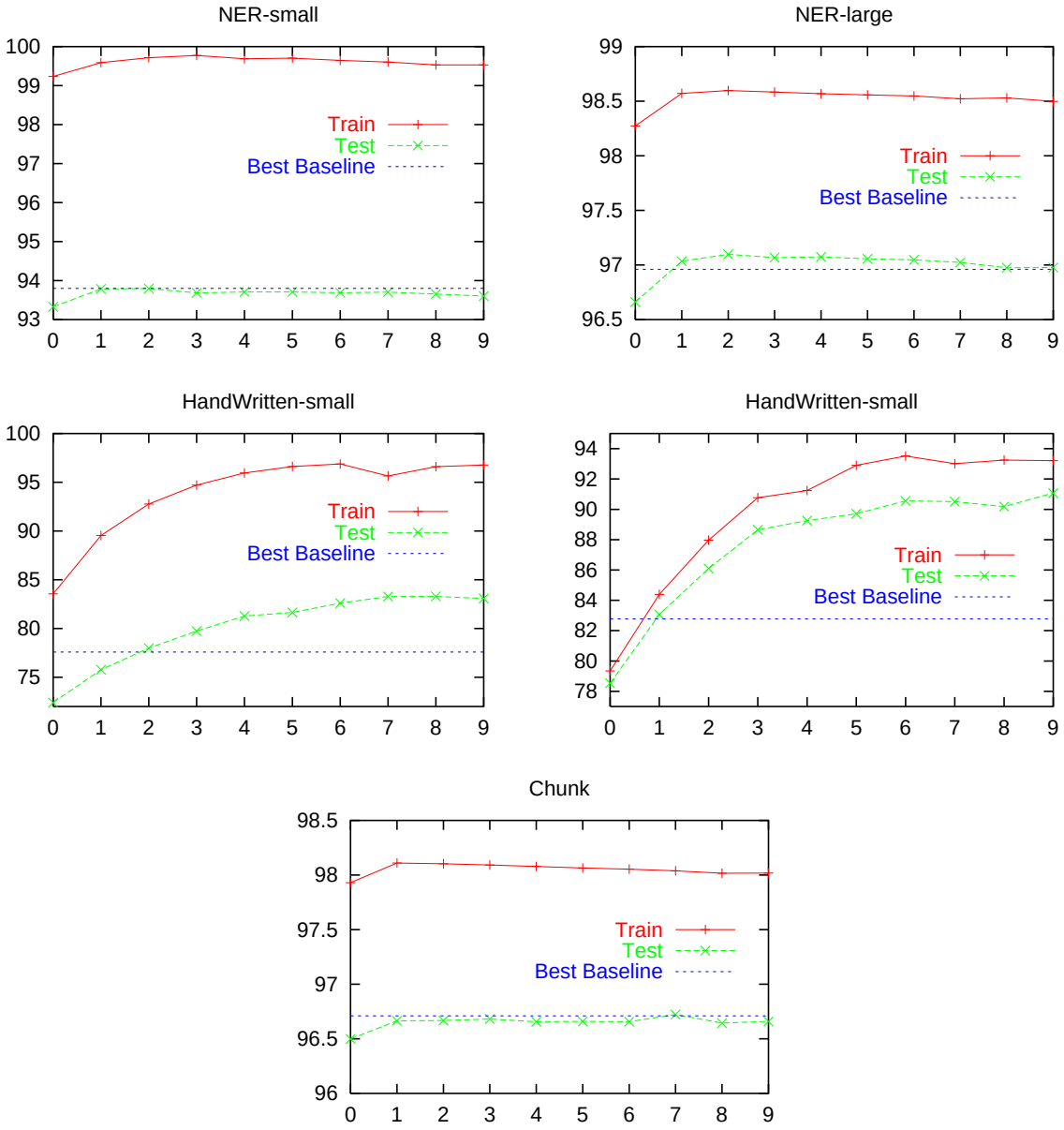


Figure 5.4: **Impact of the context size.** The X-axis corresponds to the parameter C: the number of previous predicted labels on which predictions rely. The Y-axis is the percentage of correctly predicted labels after 50 iterations of CR^{ank}on the left-to-right CR-algorithm with first order dependencies. We compare the train and test accuracies with the best baseline test-accuracy.

Loss		Context = 0		Context = 2		Context = 8	
		Train	Test	Train	Test	Train	Test
Large-margin	Most violated pair	98.99	**93.27**	99.52	93.69	99.12	93.64
	All pairs	99.59	93.18	99.9	93.69	99.83	93.62
	Best against all	98.8	93.25	99.16	**93.79**	99.08	**93.66**
Perceptron	Most violated pair	**100**	92.27	**100**	92.78	**100**	92.61
	All pairs	**100**	91.91	**100**	92.47	**100**	92.65
	Best against all	99.9	92.27	**100**	92.77	**100**	92.78
Log-binomial	Most violated pair	96.19	92.11	96.47	92.33	96.02	92.18
	All pairs	96.76	92.3	97.96	92.83	96.78	92.41
	Best against all	95.57	92.4	96.27	92.75	96.12	92.58

Table 5.5: **Impact of the ranking loss, Ner-Small dataset.** This table gives the train and test accuracies of CR^{ank} with various ranking losses, similarly to Table 5.4.

Intuitively, the *large-margin* loss requires a minimal margin between the good actions and the bad actions, whereas the *Perceptron* only requires their order to be good. For a same computation cost, minimizing the *large-margin* loss leads to much better predictions. *Perceptron*

Most of time, the *log-binomial* loss leads to lower accuracies than the *large-margin* loss. Furthermore, it has a higher computation cost than the *large-margin* and *Perceptron* losses, since it requires the computation of an exponential. *Log-binomial*

In most of our experiments, we use *large-margin* based losses since they lead to good accuracies while being simple and fast to compute.

Concerning the decomposition strategies, the two datasets exhibit different behaviors. In HandWritten-Small, the *all-pairs* strategy give significantly lower test accuracies than the two other strategies. In Ner-Small, the difference is less significant but the best losses are still those using *most-violated-pair* and *best-against-all* decompositions. Although the *best-against-all* with *large-margin* loss seems to give slightly better results, we mostly use the *most-violated-pair*/*large-margin* combination in the following. This choice is motivated by the fact that *most-violated-pair* leads to convex learning problems, which, from a theoretical point of view, is much more preferable. *Decomposition Strategies*

5.2 Improving the inference procedure

In the previous section, we introduced the left-to-right labeling CR-algorithm and showed its competitiveness *w.r.t.* state-of-the-art. Left-to-right is a simple approach for sequence labeling that can be improved in various ways thanks to our formalism. In this section, we introduce two alternative CR-algorithms: order-free labeling and multiple-pass labeling. We show experimentally that CR^{ank} is able to learn these CR-algorithms and that, for some datasets, these new inference procedures significantly improve our best previous results.

5.2.1 Order-free CR-algorithm

Instead of labeling from left to right, we consider here a CR-algorithm that is able to label in any order. The underlying idea is that it may be easier to first label *easy-to-recognize* elements, in order to enrich the context for the harder remaining labels. For example, in a handwritten recognition task, some letters may be very noisy whereas others are clear and well drawn. If the

algorithm starts to recognize some letters with a high confidence, it will then be able to use these labels, as an additional context to decide how to label the remaining letters.

CR-algorithm 4 Order-free Sequence labeling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

1: $\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
2: n $\leftarrow$ card($\mathbf{x}$)
3: labelings $\leftarrow \mathcal{L} \times [1, n]$ ▷ All labelings
4: **for** t = 1 to n **do**
5: (label, pos) $\leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{pos-\lfloor C/2 \rfloor}, \ldots, \hat{\mathbf{y}}_{pos-1}, \hat{\mathbf{y}}_{pos+1}, \ldots, \hat{\mathbf{y}}_{pos+\lceil C/2 \rceil}]$ labelings
6: $\hat{\mathbf{y}}_{pos} \leftarrow label$ ▷ Put *label* at position *pos*
7: labelings $\leftarrow$ labelings $\setminus \mathcal{L} \times \{pos\}$ ▷ Removes labelings corresponding to *pos*
8: **end for**
9: **if training then** ▷ Learning objective:
10: **reward** - $\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Hamming Loss
11: **end if**
12: **return** $\hat{\mathbf{y}}$

CR-algorithm Order-free labeling is given in CR-algorithm 4. The main difference with left-to-right labeling is that each *choose* corresponds to both a position and a corresponding label. The CR-algorithm first creates the set of all possible labeling: all possible (position,label) pairs (line 3). Then, at each step, it chooses simultaneously a position and an associated label (line 5). This decision is made on the basis of the input sequence $\mathbf{x}$ and on the neighboring predicted labels $\mathbf{y}_t$. Each time an element has been labeled, we remove the possibility for future choices to relabel this element (line 7). This ensures that, after n steps, the whole sequence is labeled.

Action Features As in left-to-right labeling, the CR-algorithm has a context size parameter C. In both cases cases, C is the total number of predicted labels that are incorporated in the feature function. We propose two feature functions for the order-free labeling CR-algorithm. The first one is illustrated in Figure 5.5. Compared to left-to-right labeling, we introduce two new labels: *non-decided* and *after*. These two labels respectively denote unlabeled elements ($\hat{\mathbf{y}}_t = \epsilon$) and elements that are beyond the end of the sequence ($t > n$).

Our second feature function incorporates one more information in the features: the current context *completion*. The completion is the number of elements in the context that are already labeled. In order to include this additional information, we replace the content features by:

$$f_{c,l,\ldots}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if context completion} = c \wedge \text{chosen label} = l \wedge \textit{input feature} \\ 0 & \text{otherwise} \end{cases}$$

and structural features by:

$$f_{c,l_1,l_2,\delta}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if context completion} = c \wedge \text{chosen label} = l_1 \wedge \hat{\mathbf{y}}_{t-\delta} = l_2 \\ 0 & \text{otherwise} \end{cases}$$

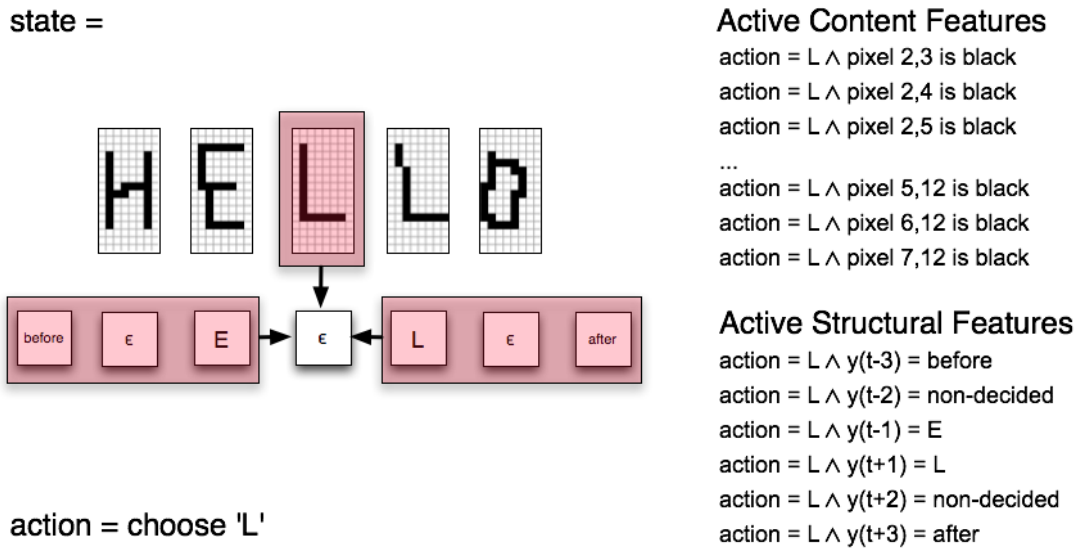


Figure 5.5: **Order-free labeling features.** The left part of the figure represents the current state and action. The set of features corresponding to this situation is given in the right part of the figure. We only display active features: those that have a non-null value. Since this example only deals with binary features, all the active features have a value of 1.

Given a context size C, the set of possible completions c is the interval $[0, C]$. We thus multiply the number of possible features by $C + 1$. However, the number of active features per state-action pair does not change. In the following, order-free with the extended feature function will be called *order-free more-feats*.

Supervision Similarly to left-to-right labeling, the Optimal Learning Policy of order-free labeling is easy to compute. This makes it possible to supervise CR^{ank} with the optimal regrets:

$$c(\mathbf{s}, \mathbf{a}) = c((\mathbf{x}, \hat{\mathbf{y}}, \mathbf{y}), (label, pos)) = \begin{cases} 0 & \text{if } \mathbf{y}_{pos} = label \\ 1 & \text{otherwise} \end{cases}$$

Training Behavior

Experiments Figure 5.6 compares the training behavior of CR^{ank} for left-to-right labeling, order-free labeling and order-free labeling with the extended description. On the handwritten recognition task, order-free labeling significantly improves the prediction accuracies over left-to-right labeling: +2.8% for HandWritten-Small and +2.9% for HandWritten-Large. Furthermore, the order-free method seems to be less sensible to over-fitting than the left-to-right method (especially on the HandWritten-Small dataset). The *more-feats* feature function seems to lead to better results than the normal feature function. However, these improvements are done at the cost of the number of necessary training iterations. On HandWritten-Small, left-to-right reaches its best performance after 40 iterations, order-free needs about 100 iterations and order-free *more-feats* needs about 200 iterations.

Impact of Context size

Figure 5.7 compares the three approaches *w.r.t.* the context size. A surprising phenomenon

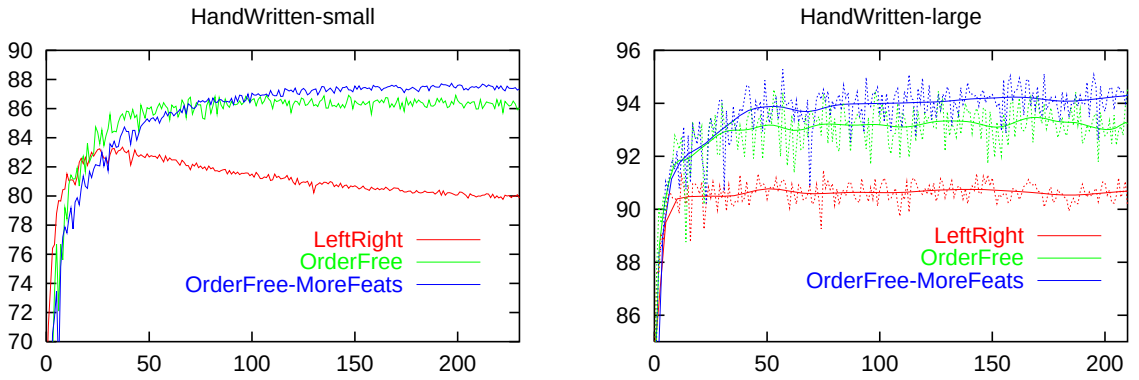


Figure 5.6: **Left-to-right and order-free during CR^{ank} training.** The X-axis is the number of passes over the training set and the Y-axis is the test accuracy of the methods. We compare three labeling methods: left-to-right labeling, order-free labeling and order-free labeling with the extended description. For all methods, we used a context size of 10 (*i.e.* the ten previous predicted labels for left-to-right and the five previous and five next predicted labels for OrderFree). Since the test set of HANDWRITTEN-LARGE is very small, the test accuracies are very noisy. We therefore display smoothed versions of these curves.

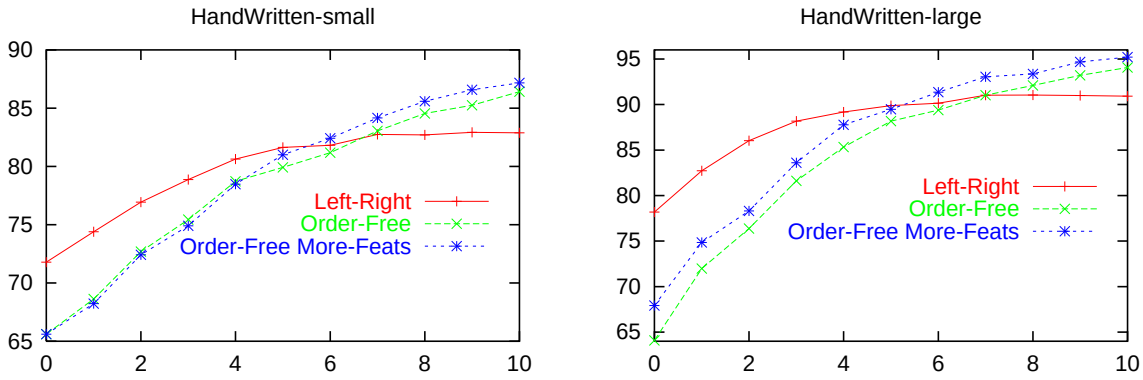


Figure 5.7: **Left-to-right and order-free in function of the context size.** The X-axis corresponds to the context size parameter C and the Y-axis corresponds to test accuracies. For each context size, we give the results of left-to-right labeling after 50 training iterations, order-free labeling after 100 training iterations and order-free *more-feats* after 200 training iterations.

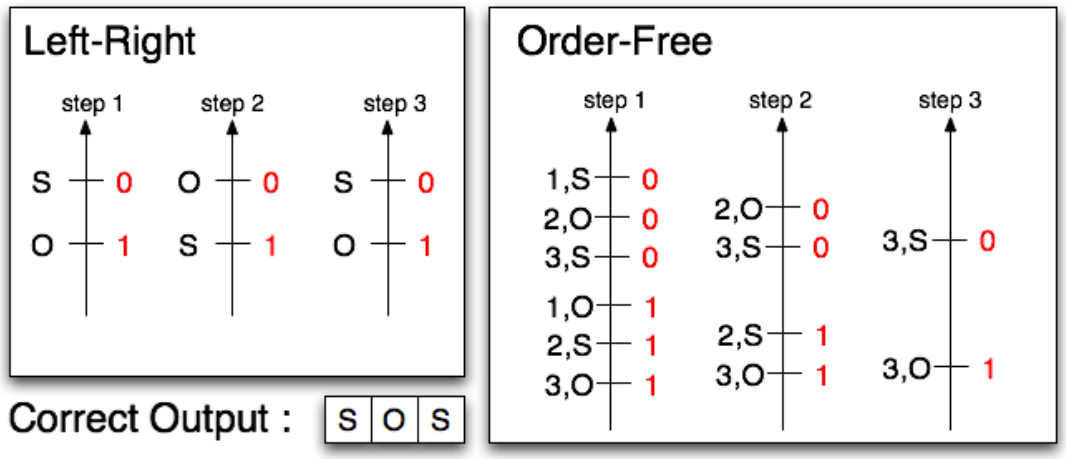


Figure 5.8: **Left-to-right vs order-free ranking problems.** We consider the sequence (S, O, S) in a problem with two possible labels $\mathcal{L} = \{S, O\}$. The left part of the figure illustrates the ranking problem corresponding to left-to-right labeling. The right part of the figure illustrates the ranking problem corresponding to order-free labeling. Actions are written in black and the corresponding red numbers are their action-costs. We assume a perfect ranking situation: all the top-ranked actions have a cost of 0.

is that, for small context sizes, order-free accuracies are significantly lower than their left-to-right counterpart. We believe that this is due to the nature of the underlying ranking problems. Figure 5.8 illustrates the ranking problems corresponding to left-to-right labeling and order-free labeling. A major difference between both problems is the following: the left-to-right ranking problem is homogeneous – each ranking list corresponds to a single element – whereas the order-free ranking problem is heterogeneous – multiple different elements are mixed within ranking lists. We believe that the latter problem is harder to solve, which could explain cases where order-free gives lower accuracies than left-to-right. If the order-free ranking problem is really harder than its left-to-right counterpart, we may need a richer description space to accurately learn the ranking function. This could explain why the extended description works better than the normal description. This is also consistent with the effect of context size observed in Figure 5.7. When the context size is increased, the ranking machine can use more and more structural features, which seems to be crucial to solve the ranking problem accurately.

left-to-right vs order-free

Table 5.6 compares left-to-right labeling with order-free labeling on all our datasets. In most of them, order-free methods slightly degrade the test-accuracy compared to left-to-right methods. This is probably related to the learning problem described above: order-free ranking functions seems more hard to learn than left-to-right ranking functions.

On the HandWritten dataset, where large contexts help a lot, order-free methods clearly outperform the left-to-right methods. With the Normal description, we have up to +4.98% improvement over the left-to-right method. With the *more-feats* feature function, order-free works even better with +5.52% improvement over left-to-right.

The feature function *more-feats* gives, in general, better results than the Normal description. This is consistent with our previous argument: since order-free ranking functions seem more

Description	Loss	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
Normal	mvp	-0.61	-0.28	**+1.95**	**+3.25**	-0.60
	ap	-1.37	-0.62	**+1.39**	**+2.96**	-1.07
	baa	- 0.56	-0.05	**+2.75**	**+4.98**	-0.21
more-feats	mvp	-0.33	-0.17	**+2.84**	**+2.94**	-0.16
	ap	-1.18	-0.48	**+1.55**	**+3.14**	-0.17
	baa	-0.69	-0.07	-0.94	**+5.52**	-0.10

Table 5.6: **Differences of test-accuracies between order-free and left-to-right labeling.** We compare left-to-right labeling (with 50 training iterations) with order-free labeling. We have used two action feature functions: Normal (with 100 training iterations) and *more-feats* (with 200 training iterations). For each dataset, we chose the context size where left-to-right best performed and used the same context size for all methods. Each result is the difference between the order-free test-accuracy and the left-to-right test-accuracy. Positive numbers, shown in bold, correspond to situations where order-free labeling outperforms left-to-right labeling.

complex than left-to-right ranking functions, we need more degrees in freedom in the ranking function to learn it accurately.

As previously, the *all-pairs* decomposition strategy gives the worst results. The *best-against-all* strategy generally gives the best results. However, in one case (HANDWRITTEN-SMALL / *more-feats*), the test-accuracy is surprisingly low. In this experiment, the algorithm has reached 100% training accuracy after only 60 iterations. We believe that the -0.94% accuracy difference is due to overfitting.

5.2.2 Multiple-pass labeling

When performing left-to-right or order-free labeling, each label is decided once for all. If a prediction is wrong, there is no chance for the CR-algorithm to correct it afterwards. This is a drawback of greedy inference procedures that may lead to sub-optimal predictions. Instead of deciding each label once, we can give the CR-algorithms the opportunity to relabel elements multiple times.

CR-algorithm CR-algorithm 5 shows a simple sketch for revising labels multiple times: multiple-pass left-to-right sequence labeling. Compared to traditional left-to-right labeling, this CR-algorithm has a new parameter: the number of passes P. When selecting $P = 1$, the CR-algorithm is equivalent to the previous left-to-right labeling. With $P > 1$, we give inference the opportunity to revise each label up to P times. At a given pass $P > 1$, the predictions can depend both on the previous predicted label and on the next predicted labels of previous pass $P - 1$.

Action Descriptions As the ranking of actions may change depending on the current pass, we incorporate the pass number p into the features. We use the following content features:

$$f_{p,l,\ldots}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if current pass} = p \wedge \text{chosen label} = l \wedge \textit{input feature} \\ 0 & \text{otherwise} \end{cases}$$

CR-algorithm 5 Multiple Pass Left Right Sequence labeling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Input: The number of passes P
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

1: $\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
2: n $\leftarrow$ card($\mathbf{x}$)
3: **for** p = 1 to P **do** ▷ For each pass
4: **for** t = 1 to n **do** ▷ For each element
5: $\hat{\mathbf{y}}_t \leftarrow$ **choose**[p,$\mathbf{x}, \hat{\mathbf{y}}_{pos-C/2}, \ldots, \hat{\mathbf{y}}_{pos-1}, \hat{\mathbf{y}}_{pos+1}, \ldots, \hat{\mathbf{y}}_{pos+C/2}]$ $\mathcal{L}$
6: **end for**
7: **end for**
8: **if training then** ▷ Learning objective:
9: **reward** - $\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Hamming Loss
10: **end if**
11: **return** $\hat{\mathbf{y}}$

and the following structural features:

$$f_{p,l_1,l_2,\delta}(\mathbf{s},\mathbf{a}) = \begin{cases} 1 & \text{if current pass} = p \wedge \text{chosen label} = l_1 \wedge \hat{\mathbf{y}}_{t-\delta} = l_2 \\ 0 & \text{otherwise} \end{cases}$$

Compared to traditional left-to-right labeling, the number of possible features is multiplied by P. However, for a given state-action pair, the number of active features does not change.

Supervision As in our previous CR-algorithms, it is easy to compute an Optimal Learning Policy in the multiple-pass case. A particularity of multiple-pass is that the final *reward* only depends on the predictions of the last pass. Any policy that performs the last pass correctly leads to the maximum of reward. The optimal regrets of the actions are the following:

$$c(\mathbf{s},\mathbf{a}) = c((\mathbf{x},\hat{\mathbf{y}},\mathbf{y},t,p),l) = \begin{cases} 1 & \text{if } p = P \wedge \mathbf{y}_t \neq l \\ 0 & \text{otherwise} \end{cases}$$

The optimal regrets does not give any information on how to perform the first $P-1$ passes. Thus, if we want to learn to predict the good labels as soon as possible, supervising with the optimal regrets is not a satisfying solution. Instead, we propose to supervise with action costs that do not depend on current pass:

$$c(\mathbf{s},\mathbf{a}) = c((\mathbf{x},\hat{\mathbf{y}},\mathbf{y},t,p),l) = \begin{cases} 1 & \text{if } \mathbf{y}_t \neq l \\ 0 & \text{otherwise} \end{cases}$$

Experiments Figure 5.9 shows the training behavior of CR^{ank} on the multiple-pass labeling CR-algorithm. For some datasets (Ner and Chunk), performing multiple passes does not lead to better accuracies. For some other datasets (HandWritten), multiple-passes significantly *Training Behavior*

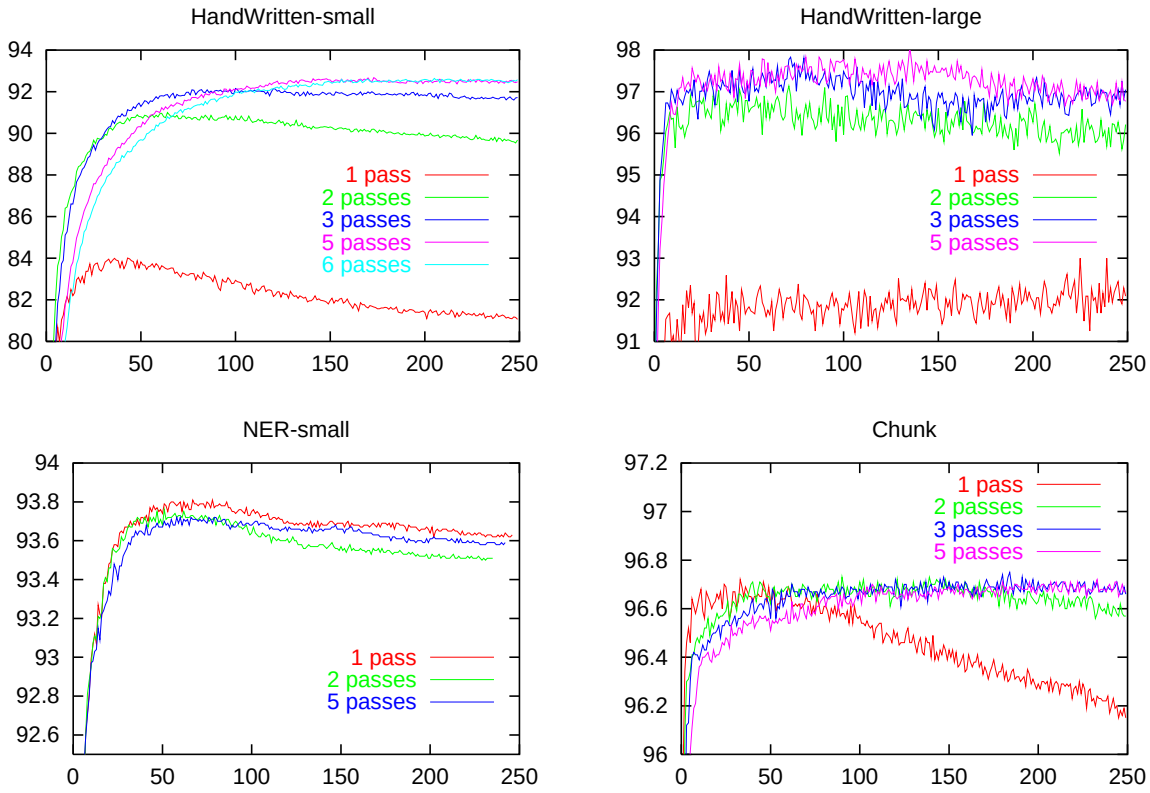


Figure 5.9: **Multiple-pass training behavior.** The X-axis corresponds to number of training iterations and the Y-axis corresponds to the test accuracy. For each dataset, we compare multiple-pass CR-algorithms with varying number of passes P.

Num. Passes	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
2	-0.02	+0.12	+7.01	+4.14	+0.08
3	-0.02	+0.11	+8.12	+5.14	+0.08
4	-0.03	+0.13	+8.53	+5.29	+0.06
5	-0.03	+0.08	+8.66	+5.40	+0.04
10	-0.04	+0.09	+9.57	+6.69	+0.02
10, C=16	-	-	+11.77	+ 7.30	-

Table 5.7: **Differences of test-accuracies between multiple-pass and single-pass labeling.** We compare multiple-pass labeling with single-pass labeling. For each possible value of P, we show the test-accuracy difference between multiple-pass and single-pass after $50 \times P$ training iterations.

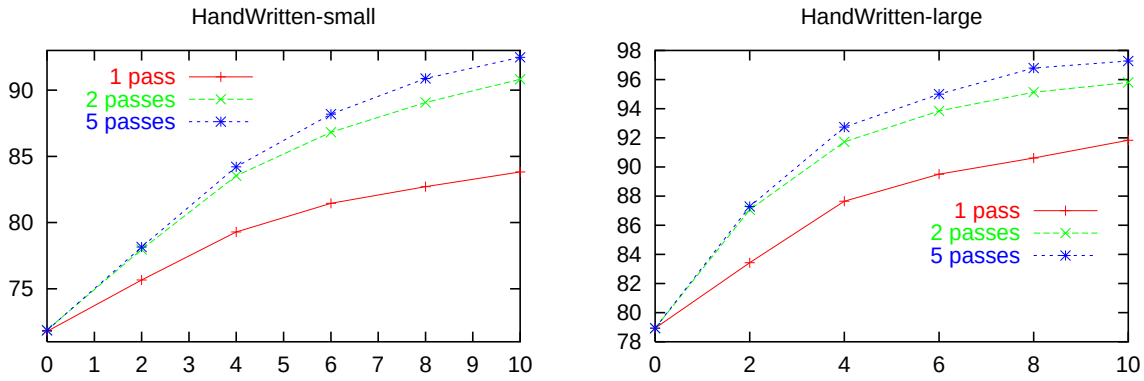


Figure 5.10: **Interaction of the context size and the number of passes.** The X-axis correspond to the context size parameter C and the Y-axis corresponds to the test accuracies. We report results for one pass, two passes and five passes ($P = 1, 2, 5$). For each model, we used $50 \times P$ training iterations.

improve the quality of predictions. In all cases, the more passes we use, the more training iterations are required to learn the ranking function.

Table 5.7 gives the relative performances of multiple-pass and single-pass labeling on our five datasets. In most cases, multiple-pass improves the test-accuracies. In the handwritten recognition task, growing the number of passes dramatically improves the test-accuracies: up to +8.66% improvement on HANDWRITTEN-SMALL and +5.40% improvement on HANDWRITTEN-LARGE for a constant context size $C = 10$. These major improvements are probably related to the limited vocabulary setting of the HANDWRITTEN dataset. Since there are only 55 different words, the multiple-pass approach makes it possible to implicitly learn the vocabulary through the structural features. Note that, with CR-algorithms, this is automatically and transparently integrated into the inference process and does not require an external dictionary or any natural language grammar.

Number of passes and Context size

Figure 5.10 shows how the number of passes interacts with the context size. There is a significant margin between the one-pass and the two-passes methods in both train/test splits.

The next passes seems to help a little bit more, but most of the correction work seems to be done in the second pass.

5.2.3 Extensions

Although we only studied left-to-right, order-free and multiple-pass labeling experimentally, it is easy to imagine various extensions to these CR-algorithms.

Two-step order-free

We have seen that the order-free ranking problem seems to be harder than its left-to-right counterpart. In order to avoid this, one approach could be this split the position and label choice into two *choose*s. The following example first chooses a position and then chooses a label for this position:

pos $\leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{p-C/2}, \ldots, \hat{\mathbf{y}}_{p+C/2}]$ {the set of unlabeled positions p} ▷ Choose a position
label $\leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{pos-C/2}, \ldots, \hat{\mathbf{y}}_{pos+C/2}]$ $\mathcal{L}$ ▷ Choose a label

Instead of having one ranking problem, *Two-step order-free* leads to two different ranking problems: ranking the possible positions and ranking the labels for a given position. The latter is similar to the traditional left-to-right ranking problem. The former is a bit particular: the aim is to rank the positions in order to make further predictions easier. Supervising this problem is not trivial: the quality of positions depends on the error distribution of the label-ranking function. As this might be hard to quantify, one possibility is to supervise the position-ranking problem thanks to rollouts. In this approach, in order to quantify the quality of a position, we perform several labeling steps starting from that position and observe the quality of the predicted label sequence.

multiple-pass order-free

We presented above the multiple-pass left-to-right approach for sequence labeling. It is easy to introduce the multiple-pass idea in an order-free CR-algorithm. The following inference loop performs S labeling steps. Any element of the sequence can be labeled or re-labeled in each such step:

for t=1 to S **do**
  (label, pos) $\leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{pos-C/2}, \ldots, \hat{\mathbf{y}}_{pos+C/2}]$ $\mathcal{L} \times [1, n]$
end for

The optimal policy of the multiple-pass order-free approach is any policy that, after the S labeling or re-labeling steps, leads to the correct label sequence. Similarly to our supervision for multiple-pass left-to-right, we suggest the use of a strong action-cost function. For example, we could use the following supervision function:

$$c(\mathbf{s}, \mathbf{a}) = c((\mathbf{x}, \hat{\mathbf{y}}, \mathbf{y}, t), (label, pos)) = \begin{cases} 0 & \text{if } (\hat{\mathbf{y}}_{pos} \neq \mathbf{y}_{pos}) \wedge (label = \mathbf{y}_{pos}) \\ 1 & \text{if } (\hat{\mathbf{y}}_{pos} = \mathbf{y}_{pos}) \wedge (label = \mathbf{y}_{pos}) \\ 1 & \text{if } (\hat{\mathbf{y}}_{pos} \neq \mathbf{y}_{pos}) \wedge (label \neq \mathbf{y}_{pos}) \\ 2 & \text{if } (\hat{\mathbf{y}}_{pos} = \mathbf{y}_{pos}) \wedge (label \neq \mathbf{y}_{pos}) \end{cases}$$

This supervision function is derived from the immediate change in Hamming Loss, *i.e.* the reward that would be obtained if the CR-algorithm was stopped immediately after the current *choose*. The best thing to do is to correctly label elements that are unlabeled or wrongly labeled (cost = 0). Replacing a wrong label by another wrong label and replacing correct labels by themselves has a cost of 1. The worst actions are those that replace a correct label by a wrong label (cost = 2).

We have seen that performing multiple-passes dramatically improves accuracy on some datasets. However, this improvement is made at the cost of inference (and training) time. In order to make

inference faster, we could imagine CR-algorithms that incorporate a *learning-to-stop* problem. Practically, this problem can be formalized as a *choose* between continuing or stopping the algorithm: *Learning to stop*

choose$[\mathbf{x}, \hat{\mathbf{y}}]$ **break**, **continue** ▷ Stop or continue?

This choice can be inserted after each pass in CR-algorithm 5, or after each *choose* in a multiple-pass order-free approach. In order to enforce fast inference, we must modify the *reward*s to take the inference time into account. Since we have two different objectives: making good predictions and making them fast, there is a trade-off to do between both. This can be done simply, by adding a penalty Γ after each inference pass:

reward $-\Gamma$

The learning problem then consists in learning to stop, as soon as we expect less than Γ improvement on the final loss. This can be supervised by comparing the current Hamming loss with the optimal Hamming loss. If the difference between both is less than Γ, we should choose to **break**.

5.3 Additional results

In this section, we discuss various aspects of training on the basis of several additional experimental results. In particular, we discuss the use of reinforcement learning algorithms (Section 5.3.1), alternative *exploration strategies* (Section 5.3.2) and the use of rollouts in learning (Section 5.3.3). We also discuss the links between multiple-pass labeling and existing methods from the field of collective classification (Section 5.3.4)

5.3.1 Reinforcement learning

The CR-algorithms described in this chapter share a nice property: they are easy to supervise. In order to quantify the contribution of supervision in the training process, we have made a set of experiments with reinforcement learning algorithms that only use the information conveyed by rewards.

Experiments have been performed with the *left-to-right* and *order-free* CR-algorithms combined with two reinforcement learning algorithms: Sarsa and Olpomdp (see Section 2.2.4). Both methods make use of linear learning machines trained with stochastic ascent/descent. The learning rate is an inversely proportional function of the time. In both Sarsa and Olpomdp, we performed softmax exploration with Gibbs distributions [Sutton et Barto, 1998]. The temperature of this distribution was chosen to be an inversely proportional function of the time. For each dataset and labeling method, we tuned the following parameters using grid search: *Models*

- Sarsa: learning rate, temperature, discount
- Olpomdp: learning rate, β (the bias-variance tradeoff)

To evaluate each set of parameters, we performed 100 training iterations with 75% of training data and evaluated the learned policy on the remaining 25% of training data.

Until now, we considered CR-algorithms with *per-episode* rewards: the whole negative loss was given at the end of episodes. From the point of view of reinforcement learning, the credit assignment problem is maximally difficult in this setting. In order to make learning easier, we considered an alternative definition of rewards called *per-decision* rewards. Instead of giving the whole Hamming loss at the end of inference, we give rewards of 0 or -1 immediately after each choose. This new reward function is illustrated for left-to-right labeling in CR-algorithm 6. Note *Per-decision rewards*

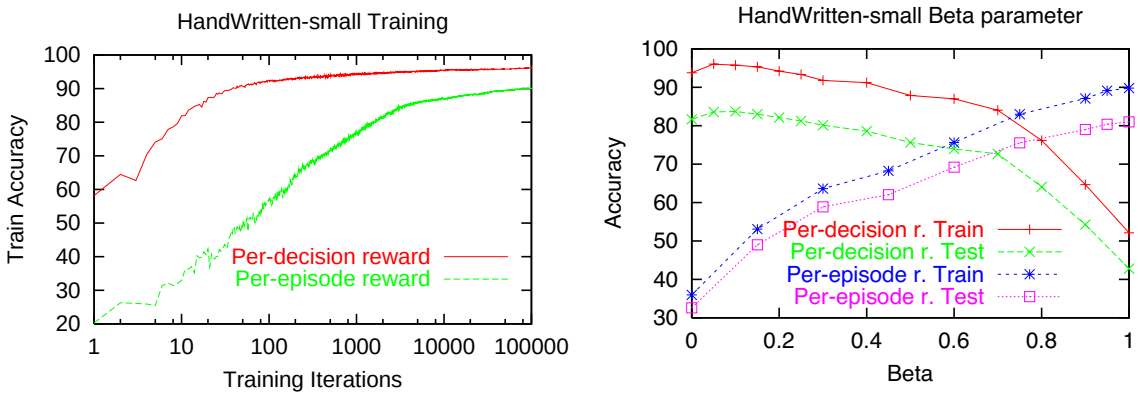


Figure 5.11: **Behavior of Olpomdp depending on which reward function is used.** The figures illustrates experiments performed on the HandWritten-Small dataset. Left: train accuracy as a function of the number of training iterations. Right: impact of the β parameter with both rewards on the train and test scores of Olpomdp.

that when maximizing the total reward criterion, per-decision and per-episode rewards are both equivalent to the empirical sequence labeling risk minimization problem.

CR-algorithm 6 Left-to-right Sequence labeling with per-decision reward.

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

$\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
n $\leftarrow$ card($\mathbf{x}$)
for t = 1 to n **do**
  $\hat{\mathbf{y}}_t \leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}_{t-C}, \ldots, \hat{\mathbf{y}}_{t-1}, t]$ $\mathcal{L}$ ▷ Choose the next label
  if training then ▷ Learning objective:
    reward - $\mathbb{1}\{\hat{\mathbf{y}}_t \neq \mathbf{y}_t\}$
  end if
end for
return $\hat{\mathbf{y}}$

Per-episode vs. Per-decision Rewards

Figure 5.11 shows the behavior of Olpomdp depending on which reward function is used. Although the per-episode correspond to a much more difficult learning problem, Olpomdp is still able to learn a good policy. However, training requires much more iterations, which makes the per-decision rewards hard to apply in practice. With per-episode rewards, since the whole loss is given at the final states, the best value of the β parameter of Olpomdp is one, *i.e.* the algorithm requires maximal propagation of the rewards to perform effective learning. At the opposite, when using the per-decision reward, only few propagation of the reward is required, which leads to low optimal β values. In all the remaining experiments, we use the per-decision reward.

	left-to-right			order-free		
	SARSA	OLPOMDP	CR^{ank}	SARSA	OLPOMDP	CR^{ank}
NER-SMALL	91.90	93.73	**93.83**	91.28	93.63	93.68
NER-LARGE	96.31	96.87	**97.43**	96.32	96.64	97.19
HANDWRITTEN-SMALL	75.20	82.46	83.13	81.56	84.36	**87.56**
HANDWRITTEN-LARGE	85.88	89.74	90.39	92.21	90.75	**94.10**
CHUNK	96.08	96.50	**96.79**	96.17	96.14	96.65

Table 5.8: **Comparison of Sarsa, Olpomdp and CR^{ank} on left-to-right and order-free sequence labeling.** Each row corresponds to a dataset and each column corresponds to a learning method. For each combination, we give the percentage of correctly predicted labels on the test-set. The best test scores are shown in bold.

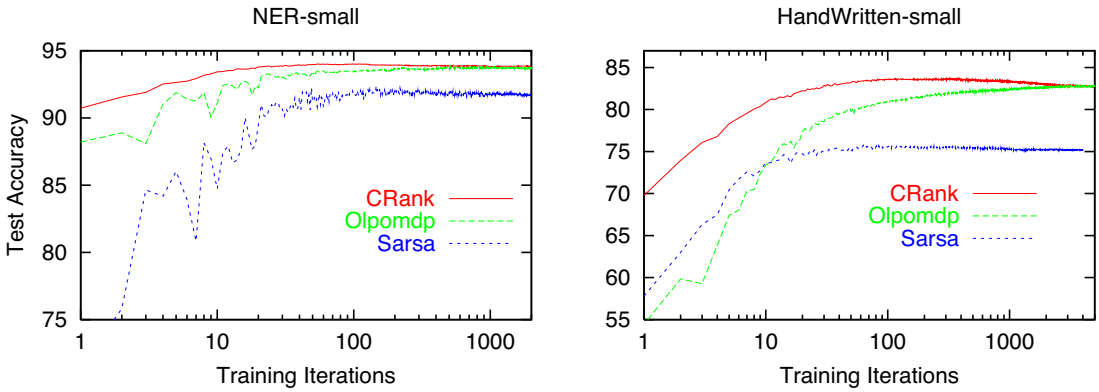


Figure 5.12: **Training behavior of Sarsa, Olpomdp and CR^{ank}on left-to-right sequence labeling.** For each method, we display the number of correctly predicted labels on the test-set as a function of the number of passes over the training set.

Table 5.8 compares the test-accuracies obtained with SARSA, OLPOMDP and CR^{ank}. The main difference between these algorithms is the supervision information that is used during training. Our results show that, although using much weaker supervision, reinforcement learning methods reach nearly the same accuracy as the supervised method on most datasets. Figure 5.12 gives the training curves for each method with left-to-right labeling. On nearly all datasets, we observed the following behavior: SARSA converges towards a sub-optimal solution and OLPOMDP converges to nearly the same result as CR^{ank}, but requires one or two orders of magnitude more training iterations. In summary, removing the rich supervision required by CR^{ank} impacts much more on the required training time than on the accuracy of the model.

RL vs. Supervised

In order to maximize the total reward criterion, SARSA should be used with a discount factor of 1. We also performed experiments with smaller discount values since this sometimes lead to an increased performance. Figure 5.13 shows the behavior of SARSA as a function of the discount value on the HANDWRITTEN dataset. In practice, the best discount values were close to zero for most corpora. Since maximizing the immediate per-decision reward leads to an optimal behavior, a null discount seems natural for left-to-right labeling. Using a discount factor greater than zero would suggest that some decisions should perhaps go against the correct label for the sake of future reward. In the order-free labeling case, discounting makes more sense: discount factors

SARSA *Discount*

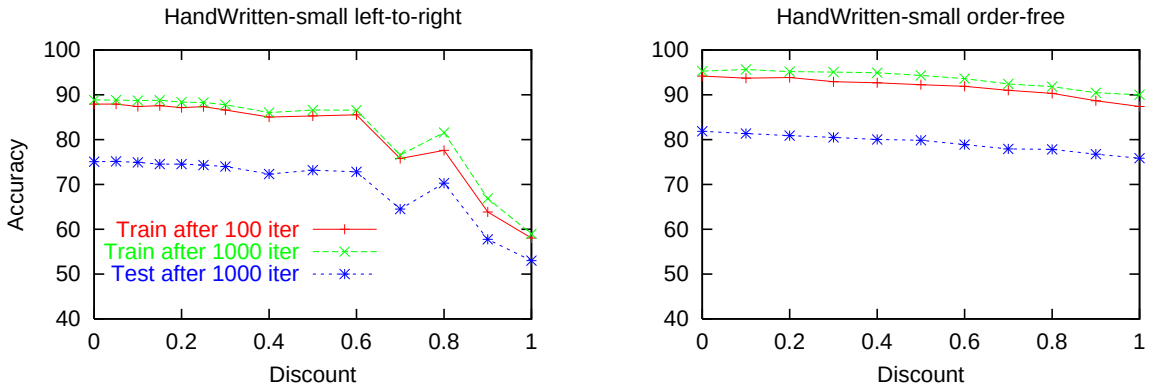


Figure 5.13: **Impact of the discount parameter in Sarsa with left-to-right labeling and order-free labeling.** The curves correspond to the training scores after 100 and 1000 training iteration and to the test score after 1000 iterations.

greater than zero tend to enforce actions that make further predictions easier, *i.e.* between two different correct labeling actions, we should prefer the position that most disambiguate the remaining prediction problems.

Olpomdp β parameter

Figure 5.14 illustrates the impact of the β parameter of Olpomdp. The impact of β on Olpomdp is similar to the one of the discount on Sarsa (see Figure 5.11 for an example). The tuning process led to small, but non-null, values of β: typically from 0.05 to 0.2 for left-to-right labeling and less than 0.15 for order-free labeling.

Ranking vs Regression

In the previous chapter, we argued that ranking actions is an easier learning problem than the traditional action value regression problem. In left-to-right and order-free labeling, if we assume a null discount factor, the value function can be written in the following way:

$$V^{\pi}(\mathbf{s}) = V^{\pi}((\mathbf{x}, \mathbf{y}, \hat{\mathbf{y}}, t)) = -\mathbb{1}\{\pi_t \neq \hat{\mathbf{y}}_t\}$$

i.e. the value functions equals 0 if the policy chooses the next label correctly and -1 otherwise. In this configuration, Sarsa is very close to CR^{ank} : both methods use the Predicted exploration strategy, both of them have immediate supervision and both of them are learned with stochastic gradient descent. Only the learning problem changes: Sarsa tries to predict -1 or 0 values with a regression machine, whereas CR^{ank} learns an ordering function with action-costs 1 or 0. The previous comparisons between Sarsa and CR^{ank} can thus be seen as fair comparisons between regression and ranking for sequence labeling problems.

5.3.2 Exploration

In the previous chapter, we have introduced various learning methods for CR-algorithms. A fundamental characteristic of these methods is the *exploration strategy* they rely on. Exploration strategies define the way actions are selected during training.

We have made a first set of experiments comparing CR^{ank} with its normal behavior – the Predicted exploration strategy – and CR^{ank} with the Optimal exploration strategy:

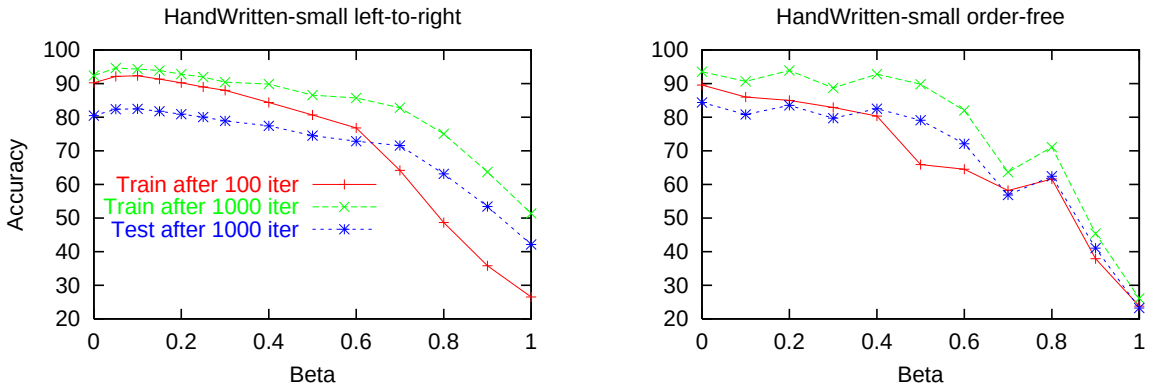


Figure 5.14: **Impact of the β parameter in Olpomdp with left-to-right labeling and order-free labeling.** The curves correspond to the training scores after 100 and 1000 training iteration and to the test score after 1000 iterations.

• **Predicted** The predicted strategy always selects the actions that are predicted by the currently learned policy. This correspond the strategy of CR^{ank}, which always takes the top-ranked actions.

• **Optimal** The optimal strategy assumes that we know the Optimal Learning Trajectories. Given these trajectories, it only selects optimal actions. This is the strategy that underlies Incremental Parsing and LaSO.

Predicted Vs Optimal

Figure 5.15, Figure 5.16 and Figure 5.17 respectively give the results for left-to-right labeling, order-free labeling and multiple-pass labeling. In nearly all cases, the Predicted strategy leads to better results than the Optimal strategy. In order to understand this result, let us consider the way both exploration strategies handle prediction errors during learning. In left-to-right labeling for example, the Optimal strategy leads to perfect contexts: $(\hat{\mathbf{y}}_{t-C}, \ldots, \hat{\mathbf{y}}_{t-1}) = (\mathbf{y}_{t-C}, \ldots, \mathbf{y}_1)$, while the Predicted strategy leads to contexts that may contain the errors made by the currently learned policy. The latter is much more close to the real use of the policy: in practice, learning is often imperfect, and we should learn our policies to recover from their previous errors. The main advantage of the Predicted exploration strategy is that if the system makes an error it will then have been trained to predict the best possible sequence of labels given this error.

Algorithms from the field of reinforcement learning suggest the use of explicit exploration during learning. A simple approach consists in selecting random actions from time to time:

• **Epsilon Greedy** The Epsilon Greedy strategy mixes exploitation and exploratory steps with a $\epsilon-$greedy policy (see Section 2.2.3). Exploitation steps correspond to the Predicted strategy and exploratory steps select actions randomly.

Epsilon Greedy

We have made experiments using ϵ-greedy policies with various values of ϵ. The results are given in Figure 5.18. The more ϵ is big, the more there is noise into the action selection process. Not surprisingly, the biggest values of ϵ particularly degrade the test-accuracies. However, it can be seen that for small values ($\epsilon = 5\%$), the results seems to be competitive with the pure Predicted strategy ($\epsilon = 0\%$). Anyway, we believe that approximation errors are a sufficient

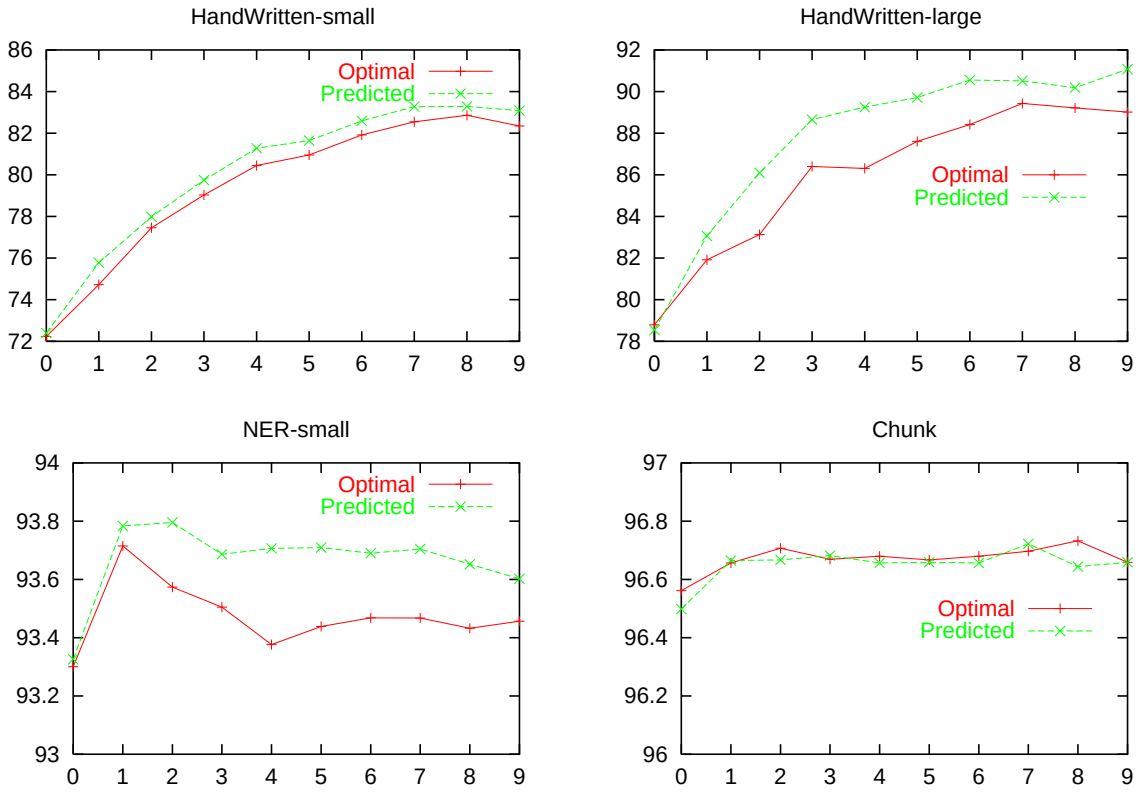


Figure 5.15: **Optimal vs Predicted exploration in left-to-right labeling.** The X-axis corresponds to the context size parameter C and the Y-axis corresponds to test accuracies. We compare two exploration strategies: Predicted and Optimal.

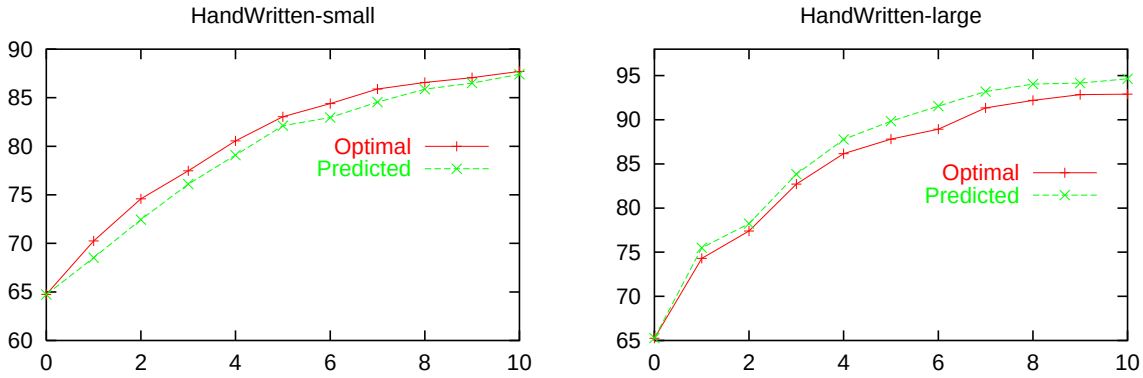


Figure 5.16: **Optimal vs Predicted exploration in order-free labeling.** The X-axis corresponds to the context size parameter and the Y-axis corresponds to test accuracies. We compare two exploration strategies: Predicted and Optimal.

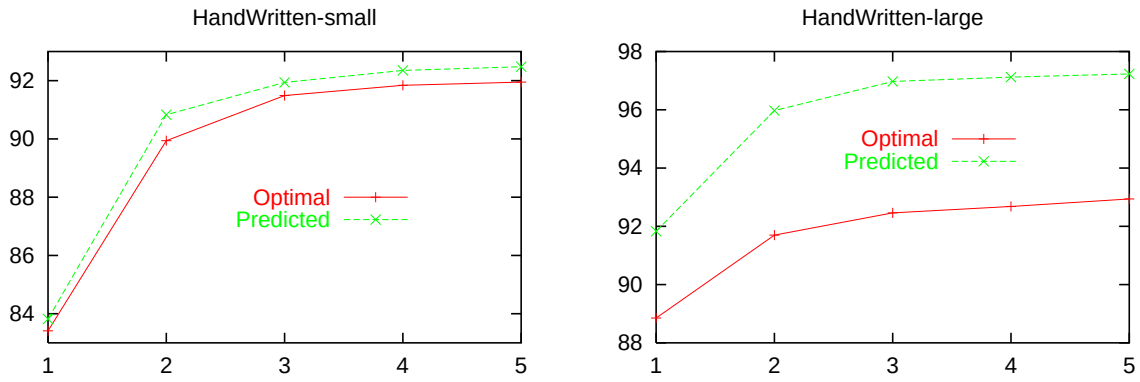


Figure 5.17: **Optimal vs Predicted exploration in multiple-pass labeling.** The X-axis corresponds to the number of passes P and the Y-axis corresponds to test accuracies. We use a context size of $C = 10$ and compare two exploration strategies: Predicted and Optimal.

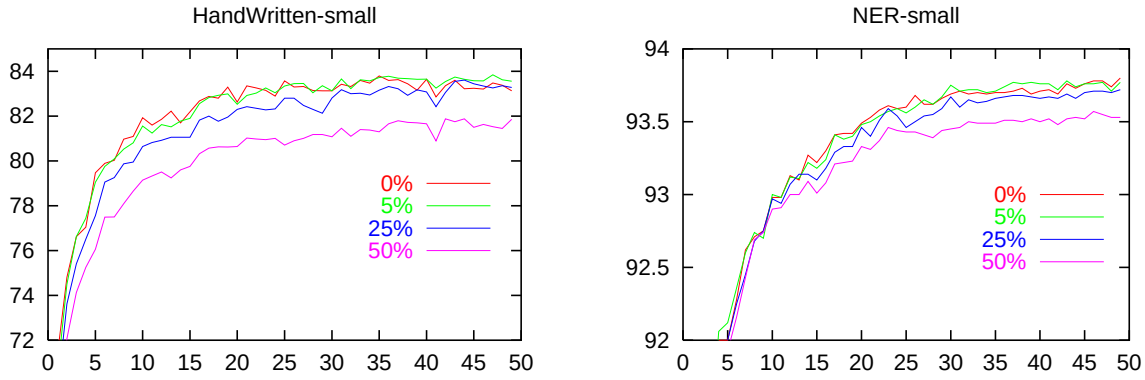


Figure 5.18: **Epsilon Greedy sampling in left-to-right labeling.** The X-axis corresponds to the number of training iterations and the Y-axis corresponds to test-accuracies. We compare ϵ-greedy policies with various values of ϵ. We use a context size of 10 for HandWritten-Small and 2 for Ner-Small.

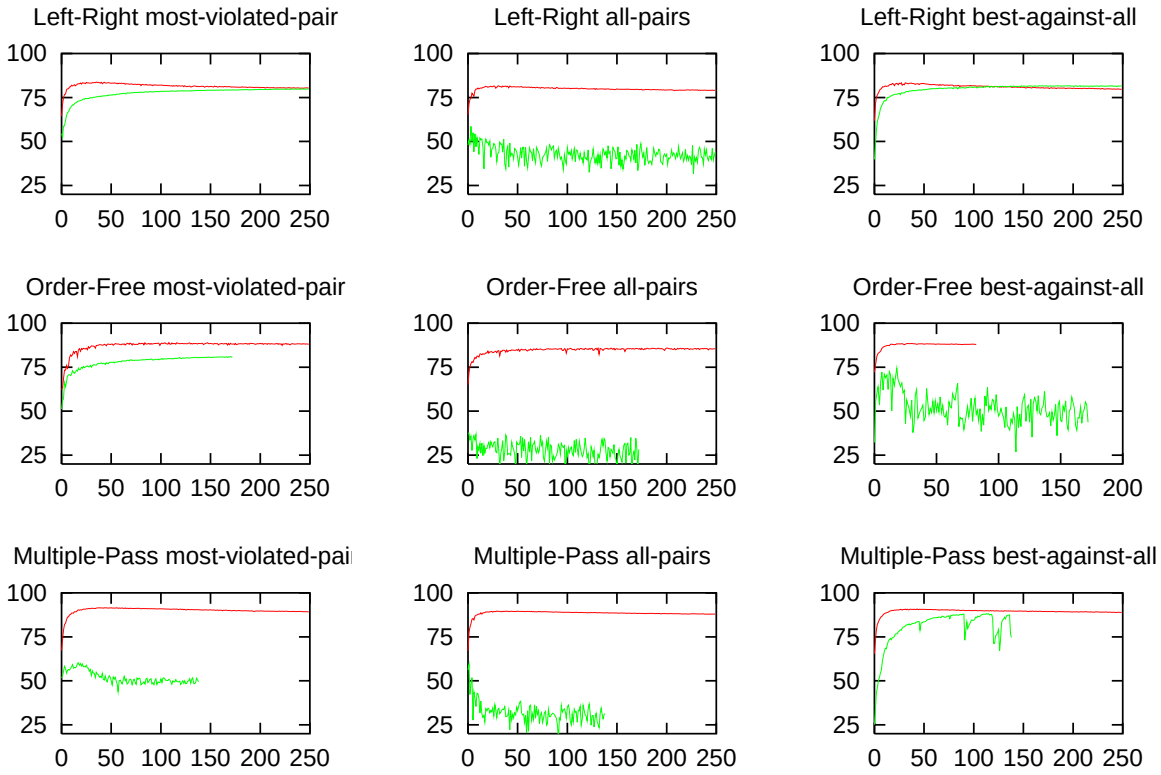


Figure 5.19: **Training behavior of CR^{ank} with rollouts.** Each curve represents a particular ranking loss / CR-algorithm combination on the HANDWRITTEN-SMALL dataset. The X-axis corresponds to the number of training iterations and Y-axis corresponds to test-accuracies. Red curves show the behavior of the normally supervised CR^{ank}. Green curves show the behavior of CR^{ank} with rollouts.

source of exploration in our CR-algorithms. Since we deal with many training examples, the policy visits continuously new states, which reduces the importance of explicit exploration.

5.3.3 Learning with rollouts

In problems where action costs cannot be simply computed, one must learn by using the rewards only. In Section 5.3.1, we discussed the use of classical reinforcement learning in such cases. In order to use CR^{ank}, an alternative consists in using empirically estimated action-costs. Such as discussed in Chapter 4, these regrets can be evaluated empirically by using rollouts.

Figure 5.19 compares the training behavior of CR^{ank} with normal supervision and CR^{ank} with rollouts on the HANDWRITTEN-SMALL dataset. Much can be said about these curves. First, in some cases, rollouts work well and make it possible to reach the same accuracy level as the fully supervised method. In other cases (*e.g.* multiple-pass most-violated-pair), the training process converges toward a local minimum. In the worst cases (typically with the all-pairs decomposition strategy), the training process does neither converge, nor lead to good accuracies. It should be

noted that no tuning on the learning rate was performed for these experiments[10]. It would not be surprising that this parameter has a crucial effect on the training behavior with rollouts.

The kinds of results we obtain with rollouts are well known in approximated reinforcement learning (see for example [Bertsekas et Tsitsiklis, 1996], for some experiments where approximated RL methods diverge). How to ensure convergence toward a good solution is still a very open question in this field. One direction for future work would be to apply a conservative policy iteration scheme to CR^{ank} [Kakade et Langford, 2002]. This would make it possible to ensure convergence after a finite time.

5.3.4 Collective classification

Our multiple-pass labeling approach closely resemble to existing methods in collective classification. In particular, the stacked learning approach [Cohen et Carvalho, 2005] and the Gibbs sampling [XX citation] methods can be used for sequence labeling. A possible approach to compare these methods with our work consists in writing their inference procedures as CR-algorithms. The way inference is performed in Gibbs sampling is described in CR-algorithm 7. Similarly the stacked learning inference procedure is given in CR-algorithm 8. The main difference between our approach and these related method is related to the way learning is performed.

CR-algorithm 7 CR-algorithm corresponding to Gibbs sampling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Input: The number of passes P
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

$\hat{\mathbf{y}} \leftarrow (\epsilon, \ldots, \epsilon)$
n $\leftarrow$ card($\mathbf{x}$)
$\forall t, l \in [1, n] \times \mathcal{L}, M[t, l] \leftarrow 0$
for p = 1 to P **do** ▷ For each pass
  for t in order(1,n) **do** ▷ For each element
    $\hat{\mathbf{y}}_t \leftarrow$ **choose**$[\mathbf{x}, \hat{\mathbf{y}}^{p-1}_{pos-C/2}, \ldots, \hat{\mathbf{y}}^{p-1}_{pos+C/2}]$ $\mathcal{L}$
    $M[t, \hat{\mathbf{y}}_t] \leftarrow M[t, \hat{\mathbf{y}}_t] + 1$
  end for
end for
for t = 1 to n **do**
  $\hat{\mathbf{y}}_t \leftarrow \text{argmax}_{l \in \mathcal{L}} M[t, l]$
end for
if training then ▷ Learning objective:
  reward - $\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Hamming Loss
end if
return $\hat{\mathbf{y}}$

Stacked learning relies on a cross-validation based method for generating the intermediate labels. Gibbs sampling learn classifiers by assuming perfectly predicted neighboring labels. The CR-algorithms key idea is to incorporate inference into the learning process. With algorithms

[10] We kept using the heuristic described in Section 5.1.4

CR-algorithm 8 Stacked Multiple Pass Left Right Sequence labeling

Input: An input sequence $\mathbf{x}$
Input: The set of possible labels $\mathcal{L}$
Input: The context size C
Input: The number of passes P
Training Input: The correct labels $\mathbf{y}$
Output: A predicted sequence of labels

1: $\hat{\mathbf{y}}^0 \leftarrow (\epsilon, \ldots, \epsilon)$
2: n $\leftarrow$ card($\mathbf{x}$)
3: **for** p = 1 to P **do** ▷ For each pass
4: **for** t = 1 to n **do** ▷ For each element
5: $\hat{\mathbf{y}}_t^p \leftarrow$ **choose**[p,$\mathbf{x}, \hat{\mathbf{y}}_{pos-C/2}^{p-1}, \ldots, \hat{\mathbf{y}}_{pos+C/2}^{p-1}$] $\mathcal{L}$
6: **end for**
7: **end for**
8: **if training then** ▷ Learning objective:
9: **reward** - $\Delta^{hamming}(\hat{\mathbf{y}}, \mathbf{y})$ ▷ Hamming Loss
10: **end if**
11: **return** $\hat{\mathbf{y}}^P$

	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
Best baseline	93.8	96.96	77.59	82.78	96.71
left-to-right	93.78	**97.19**	83.04	91.49	**96.72**
order-free	93.19	97.11	85.28	96.47	96.47
order-free more-feats	93.46	97.09	85.88	97.01	96.58
multiple-pass	**93.82**	**97.19**	**94.81**	**98.79**	**96.72**

Table 5.9: **Summary of sequence labeling results.** This table summarizes the test-accuracies that we obtained with our CR-algorithms. These results are contrasted with the scores of the best baselines from Section 5.1.3.

such as CR^{ank}, the classifiers of stacked learning or Gibbs sampling would be learned by simulating the inference procedure. We plan to make an exhaustive comparison of the CR-algorithm approach against stacked learning and Gibbs sampling in our future work.

5.4 Conclusion

In this chapter, we have illustrated the use of CR-algorithms for sequence labeling and we demonstrated experimentally the multiple advantages of this approach:

- CR-algorithms can model *high order dependencies* between the labels. Contrary to most SP approaches, CR-algorithms are not restricted by the Markov assumption. On many datasets, increasing the length of the dependencies dramatically increases the performance of the models. Thanks to these long-term dependencies, CR-algorithms are competitive against state-of-the-art. In particular, we have shown that greedy inference performs as well as dynamic programming based inference.

	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
left-to-right CR^{ank}	2.59 ms	3.04 ms	1.10 ms	1.09 ms	1.17 ms
order-free CR^{ank}	16.10 ms	16.31 ms	3.14 ms	3.04 ms	3.19 ms
multiple-pass CR^{ank}	6.36 ms	6.92 ms	3.04 ms	3.04 ms	3.42 ms

Table 5.10: **Inference times for various sequence labeling methods.** We give the average inference cpu-time per sequence for the left-to-right, order-free and multiple-pass CR-algorithms (with P = 3). All the experiments were performed on a standard desktop machine.

	NER-SMALL	NER-LARGE	HANDWRITTEN-SMALL	HANDWRITTEN-LARGE	CHUNK
left-to-right CR^{ank}	$\approx$ 30 s	$\approx$ 20 min	$\approx$ 40 s	$\approx$ 7 min	$\approx$ 10 min
order-free CR^{ank}	$\approx$ 6 min	$\approx$ 5h	$\approx$ 4 min	$\approx$ 40 min	$\approx$ 1h
multiple-pass CR^{ank}	$\approx$ 4 min	$\approx$ 3 h	$\approx$ 5 min	$\approx$ 1h	$\approx$ 90 min
CRF	-	$\approx$ 8h	-	$\approx$ 2h	-
SVMISO	-	$>$ 3 days	-	$>$ 3 days	-
Simple Searn	$\approx$ 30 min	$\approx$ 6h	$\approx$ 20 min	$\approx$ 3h	$\approx$ 2h
L2-MAXENT	$\approx$ 30 s	$\approx$ 1 h	$\approx$ 60 s	$\approx$ 8 min	$\approx$ 15 min
L1-MAXENT	$\approx$ 40 s	$\approx$ 25 min	$\approx$ 70 s	$\approx$ 8 min	$\approx$ 8 min
SVM	$\approx$ 40 s	-	$\approx$ 15 s	$\approx$ 15 min	-

Table 5.11: **Training times for various sequence labeling methods.** Approximate training times for various sequence labeling methods. Top: CR-algorithms learned with CR^{ank}. We use $P = 3$ for multiple-pass labeling. Middle: structured prediction baselines. Bottom: independent classification baselines.

- CR-algorithms is a very *expressive framework* in which many different inference procedures can be written. We studied three CR-algorithms for sequence labeling: left-to-right labeling, order-free labeling and multiple-pass labeling. Learning order-free and multiple-pass inference procedure is an original idea of this work. Table 5.9 summaries the results corresponding to these CR-algorithms. On some datasets, our new inference procedures significantly outperform state-of-the-art results.

- We did not discuss execution times until now. Since they rely on greedy inference, CR-algorithm approaches are *very fast*. Table 5.10 summarizes the inference times of various models that were learned with CR^{ank}. Most of time, sequences are inferred in less than 5 ms, which is very satisfying for applications.

- CR^{ank} has been conceived to be a fast training algorithm able to deal with large-scale tasks. Table 5.11 summarizes the training times for the various approaches we compare. It can be seen that CR^{ank} based approach perform training in a relative small time, which makes it possible to deal with *very large corpora*.

The CR-algorithms that we introduced in this section were easy to supervise: we had access to Optimal Learning Policies that were used to compute the optimal regrets associated to actions. The knowledge of an Optimal Learning Policy is a strong assumption that is not always verified. In the next chapter, we introduce a new task, which is much harder than sequence labeling, where it is not possible to compute the Optimal Learning Policy. We will show that CR-algorithm can still be used, only the learning methods must be changed.

6

Tree Transformation with CR-algorithms

Contents

This chapter introduces a new challenging SP task: ordered labeled tree transformation. This task deals with large trees (thousand of nodes), complex transformations (structure and text processing, node creations, deletions and displacements) and very high dimensional learning (often more than one million distinct features).

This tree transformation task is motivated by the amount of semi-structured available on the Web and the diversity of existing formats used to describe such data. Most documents available on the Web are expressed in layout-oriented formats (flat text, wiki-text and HTML), while document-processing applications require more and more semantic-oriented information, in the form of XML dedicated formats for example. This leads to the need of automatic conversion tools between semi-structured document formats [Wisniewski *et al.*, 2007] with application in domains like document engineering [Chidlovskii et Fuselier, 2005] and information retrieval [Denoyer et Gallinari, 2007, Jousse *et al.*, 2006]. We propose here to model such conversions as an SP task, where both inputs and outputs are ordered labeled trees. From the point of view of the user, it is enough to provide a set of documents expressed in both source and target format, to learn the conversion process.

Figure 6.1: **Example of XML heterogeneity.** The same movie description extracted from three sources: two HTML styles and one XML general movie schema.

6.1 Introduction

We introduce here the ordered labeled tree transformation task. Section 6.1.1 describes the context that motivates the task. Section 6.1.2 describes related work in connected fields. The task is formalized as an SP problem in Section 6.1.3. Finally, we discuss the relevancy of Incremental SP methods for this problem in Section 6.1.4.

6.1.1 Context

Semi-structured data

Semantically rich data like textual or multimedia documents tend to be encoded using semi-structured formats. Content elements are organized according to some structure that reflects logical, syntactic or semantic relations between these elements. For instance, XML and, to a lesser extent, HTML allow us to identify elements in a document (like its title or sections) and to describe relations between those elements (e.g. we can identify the author of a specific part of the text). Additional information such as metadata, annotations, etc., is often added to the content description leading to richer descriptions.

Heterogeneity

The question of heterogeneity is central for semi-structured data: documents often come in many different formats and from heterogeneous sources. Web data sources for example use a large variety of models and syntaxes, as illustrated in Figure 6.1. Although XML has emerged as a standard for encoding semi-structured sources, the syntax and semantic of XML documents following different DTDs or schemas will be different. For managing or accessing an XML collection built from several sources, a *correspondence* between the different document formats has to be established. Note that in the case of XML collections, the schemas themselves may be known or unknown depending on the source. For HTML data, each site will develop its own presentation and rendering format. Thus even in the case of HTML where the syntax is homogeneous across documents, there is a large variety of formats. Extracting information from different HTML web sites also requires to specify some type of mapping between the specific Web sites formats and the predefined format required by an application.

Tree transformation

Designing tree transformations, in order to define correspondences between the different schemas or formats of different sources is thus a key problem to develop applications exploit-

ing and accessing semi-structured sources. This problem has been addressed for some times by the database and to a lesser extent by the document communities for different conversion tasks and settings. Anyway, the real world solution is to perform a manual correspondence between heterogeneous schemas or towards a mediated schema via structured document transformation languages, like XSLT. Although special tools have been developed for helping programmers at this task, the process remains complex, it requires expert skills and the resulting mappings are often very complicated. This is not adapted to situations where document sources are multiple and change frequently. Furthermore, languages such as XSLT are limited by two important properties of real world document collections:

- The schema for large document collections are often very loose and impose only few constraints on valid documents[1]. In this case, the schema itself does not provide enough information and writing an XSLT script for document transformations can be very difficult and time consuming.
- Many sources, especially on the Web, come without schema and the only evidence comes from the document itself. The transformation problem has then to be studied from a *document centric* perspective as opposed to the *data centric view developed for databases*. This means that the semantic of the document is important and that transformations shall take into account both the textual content and the structure of the document. Here, the order of the document leaves and nodes is meaningful and shall be taken into account by the transformation.

Automatic Tree Transformation

Automating the design of these transformations has rapidly become a challenge. Several approaches have been explored ranging from syntactic methods based on grammar transformations or tree transducers to statistical techniques. However, these methods are still hard to use in practical cases. Many of them heavily rely on task specific heuristics. Current approaches to document transformation are usually limited to one transformation task or to one type of data. A majority of techniques only consider the structural (logical or syntactic) document information and do not exploit content nodes. Even this structural information is used in a limited way and most methods exploit only a few structural relationships. Besides, most proposed techniques do not scale to large collections.

In this chapter, we propose to model automatic tree transformation as a SP task. Instead of writing complex transformation scripts, in this framework, the user has only to provide a set of examples of the transformation, *i.e.* pairs composed of input and associated target documents. The tree transformation task encompasses a large variety of real-world applications like:

• **Semantic Web** conversion from raw HTML to semantically enriched XML. Example sources include forums, blogs, wiki-based sites, domain specific sites (music, movies, houses, ...).

• **Wrapping of Web pages** conversion from pages coming from many sources to a unified format.

Document Annotation/Conversion

• **Legacy Document conversion** The document-engineering field were document mapping has been at the heart of many applications like document annotation or document conversion of flat text, loosely structured text, HTML or PDF formats onto a predefined XML schema [Chung *et al.*, 2002, Chidlovskii et Fuselier, 2005].

[1] This is the case for example for the Wikipedia [Denoyer et Gallinari, 2006] or the IEEE INEX XML collections.

• **Heteregeneous Search** Tree transformation is also relevant to the field of Information Retrieval. When searching in XML collections, targets are no more documents but document elements. Queries may address either the content of the document, or both its content and structure. In both cases, the goal will be to retrieve the most specific relevant elements in a document. The INEX initiative [Fuhr *et al.*, 2002a] launched in 2002 has focused on the development of XML search engines. The initial collections at INEX were homogeneous, all documents sharing a common DTD. Recently a heterogeneous search track has been launched where the goal is to query collections coming from different sources and with different formats.

6.1.2 Related Work

Three domains have been mainly concerned up to now with the document conversion problem : Information Retrieval, Document Engineering and Databases. We briefly review below related work in these three domains.

Information Retrieval and Document engineering models

Structured document transformation is a relatively new topic in the document community. Automating XML document transformation from one source format onto a target schema is a key issue for document reuse and several approaches have been recently proposed. Work in this area only consider schema transformations and thus requires that the input schema is provided. Content information is completely ignored. This class of methods is thus restricted to collections with well-defined schema and few content like bibliographic data. It does not apply to large document collections. Leinone and al. in [Leinonen, 2003] for example propose a syntax directed approach based on finite state tree transducers. The system automatically generates mappings when the user specifies a correspondence between document leaves. Su et al. in [Su *et al.*, 2001] propose a tree-matching algorithm, which decomposes the mapping into a sequence of basic operation. The algorithm relies on heuristics for exploring the "'action"' space and on user provided semantic information. Boukotaya and al. in [Boukottaya et Vanoirbeek, 2005] also propose a heuristic algorithm, which combines multiple criteria (semantic relationships between label names, data types compatibility, path similarity, etc). Another interesting topic in the document and web communities is document annotation, which is the transformation of rendering formats like HTML or PDF formats onto a target XML schema. Annotation is a special case of structure mapping where for example the input and output documents have similar leave sequences. Yip Chung and al. in [Chung *et al.*, 2002] consider an HTML to XML conversion problem. They use unsupervised machine learning and manually defined rules to discover tree patterns in the input document and to produce the output structure. Closest to us is the work by Chidlovskii and colleagues [Chidlovskii et Fuselier, 2005] which has been used here as a baseline model for comparison. They also consider the conversion of HTML or PDF documents to a target XML schema. They use classifiers and probabilistic grammars to label input document elements and build output trees. The complexity of these methods however limits their application to small collections (e.g. Shakespeare corpus in the tests presented here).

It is worth mentioning here the work of Collins et al. [Collins et Roark, 2004] who recently proposed an incremental method based on machine learning for parsing. This method, which achieved impressive results, also uses a sequence of actions quite similar to ours for building the parse tree from an input sentence but it used for natural language processing and cannot be easily adapted to our structure mapping task.

Schema Matching (DB) Models

In the database community automatic or semi-automatic data integration — known as *schema matching* — has been a major concern for many years and there is a profusion of work and topics in this area. Surveys of these techniques can be found in [Rahm et Bernstein, 2001] and [Shvaiko et Euzenat, 2005]. We briefly review below three groups of matching models.

- Schema-based models: they transform schemas and ignore the document content ([Palopoli *et al.*, 1998, Doan *et al.*, 2000, Castano *et al.*, 2001]).
- Instance-level approaches: they consider both the schema and the meaning of the schema elements. They are used when the schema information is limited ([Doan *et al.*, 2001, Li et Clifton, 2000]). Some of these models use basic machine learning techniques like rule learners, neural networks, ...
- Multiple-Matchers approaches: they use several type of matchers in order to compute the matching of two schemas ([Doan *et al.*, 2001, Embley *et al.*, 2001]). Some approaches in this field have explored the inference of mapping using machine learning techniques. A remarkable series of work has been developed in this direction by Halevy, Doan and colleagues. In [Doan *et al.*, 2003], Doan and al. propose a methodology which combines several sources of evidence in the documents (tag names, schema description, data types, local structure, etc), using a regression function learned from a dataset. This method has been developed for different matching problems, and has been used in Section **??** for comparison on the *RealEstate* corpus.

.

Past work on schema matching has mostly been done in the context of a particular application domain. As far as we know, there is no benchmark in this domain and no comparison of the different existing approaches.

6.1.3 Formalization as an SP problem

The tree transformation task can be formalized as an SP task, which consists in learning a mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$ from input documents $\mathbf{x} \in \mathcal{X}$ to target documents $\mathbf{y} \in \mathcal{Y}_{\mathbf{x}}$. Both input and target documents are modeled as rooted labeled ordered trees. In the following, we denote $\mathbf{x}_i$ the i-th leaf of the input document $\mathbf{x}$ and $\mathbf{y}_i$ the i-th leaf of the target document $\mathbf{y}$. The user provides a set of examples of the transformation: $D = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i \in [1,n]}$. In order to restrict the set of possible target documents $\mathcal{Y}_{\mathbf{x}}$ for a given input $\mathbf{x}$, we consider a target schema $\mathcal{G}$. The information contained by this schema, such as the set of possible labels a target document may contain, will be detailed below. The schema can either be provided by the user (with a DTD in the case of XML documents for example) or induced automatically given a set of target documents. *Inputs and Outputs*

Many different kinds of transformations may appear in real-world applications. In the following, we discuss two kinds of transformations that are illustrated in Figure 6.2: *one-to-one tree transformation* and *tree transformation with unaltered text*. In one-to-one tree transformation, the input and target document leaves are aligned: each leaf $\mathbf{x}_i$ of the input corresponds to the leaf $\mathbf{y}_i$ of the target. Furthermore, the textual content of the leaves is preserved during the transformation: the text appearing in $\mathbf{x}_i$ is the same as this appearing in $\mathbf{y}_i$. A particularly interesting instance of the one-to-one tree transformation problem consists in extracting the structure of a flat-segmented textual document. *One-to-one tree transformation*

In many real world tree transformation problems, the one-to-one assumptions are not verified-. Ideally, a tree transformation system should handle as many kinds of transformations as possible. In particular, during the transformation, leaves may appear and disappear or they may be reordered. We could even imagine applications where the document text may be altered. Transformations on the text may range from low-level operations (*e.g.* text splitting, grouping, capitalizing) to high-level semantic operations (*e.g.* text summarizing, automatic translation). In *General tree transformation*

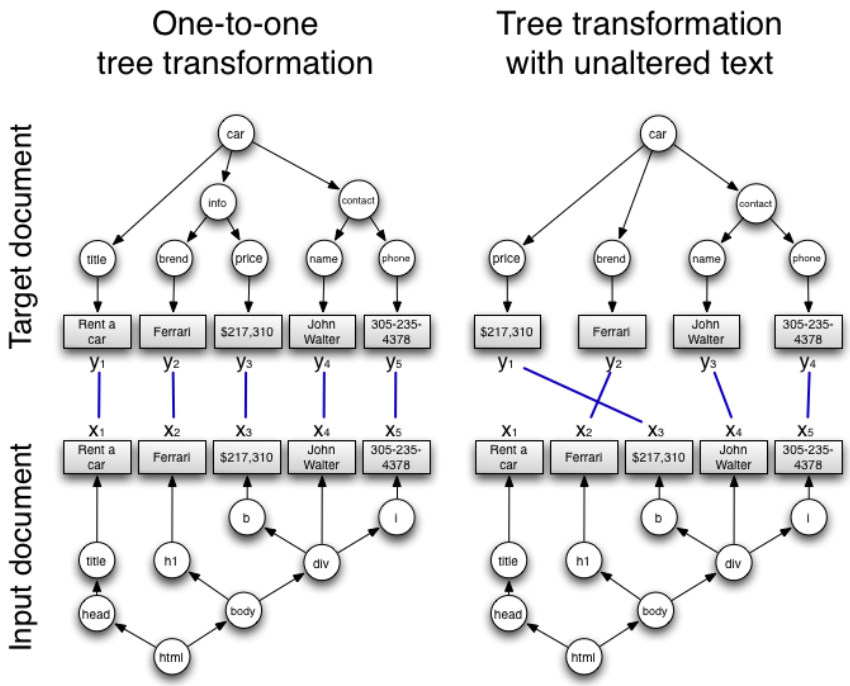


Figure 6.2: **One-to-one tree transformation and tree transformation with unaltered text.** This figures illustrates two tree transformation problems. Pairs of input and target leaves sharing the same textual content are shown with blue links. Left: one-to-one tree transformation, each target leaf $\mathbf{y}_i$ corresponds to the input leaf $\mathbf{x}_i$. Right: tree transformation with unaltered text. Leaves may be reordered or suppressed, however, the text contained by each target leaf $\mathbf{y}_i$ exists in at least one input leaf $\mathbf{x}_j$.

the following, we restrict ourself to tree transformation with unaltered text: each text contained in a target leave $\mathbf{y}_i$ should exist in at least one input leave $\mathbf{x}_j$.

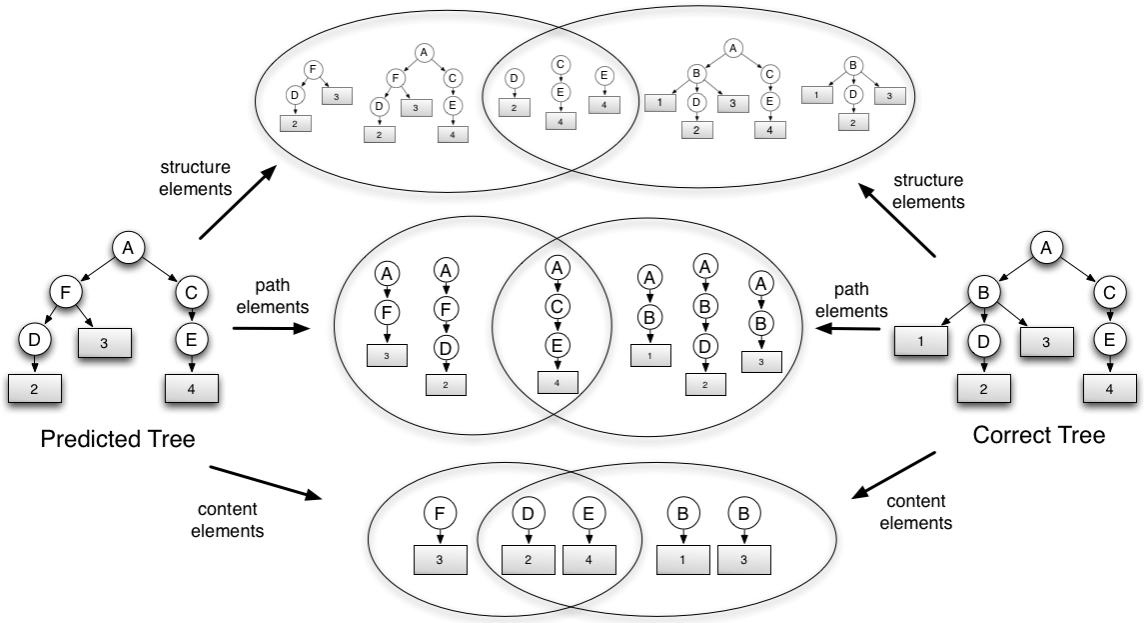
Loss Function

The aim in SP is to minimize the expectation of the loss function Δ. Several such loss functions may be relevant to tree transformation. Ideally, evaluating the quality of a tree transformation should be made in the context of a specific application such as a search system on heterogeneous collections or a specific document conversion tasks. Our approach was developed to solve a large variety of tasks and we propose here to evaluate the general performance of the method by measuring generic tree-similarities between predicted output documents and correct output documents.

F1 Scores

In our experiments, we make use of three tree-similarity *F1* scores, which are illustrated in Figure 6.3. The $F_{structure}$, F_{path} and $F_{content}$ measures are computed by decomposing each tree into a set of elements and by computing the F_1 scores[2] on these decompositions. The three F_1 scores are always in the interval $[0, 1]$ and perfectly predicted trees lead to similarity scores of 1. The three measures give complementary information: $F_{content}$ focuses on the proportion of correctly labeled leaves, F_{path} concerns the proportion of correctly recovered paths and $F_{structure}$

[2]The F_1 score between two sets a and b is: $F_1(a, b) = \frac{2 \times card(a \cap b)}{card(a) + card(b)}$.



$$F_{structure} = \frac{2 \times 3}{5+5} = 60\% \qquad F_{path} = \frac{2 \times 1}{3+4} \simeq 28.57\% \qquad F_{content} = \frac{2 \times 2}{3+4} \simeq 57.14\%$$

Figure 6.3: **Computation of the $F_{structure}$, F_{path} and $F_{content}$ similarity measures.** The left and right parts of the figure respectively correspond to the predicted tree and the correct tree. For each similarity measure, the trees are decomposed into set of elements (structure elements, path elements or content elements). The bottom part of the figure gives the similarity scores, which are the F_1 score between predicted and correct elements.

concerns the proportion of correctly recovered subtrees. By default, we maximize the $F_{structure}$ measure by using the following loss function:

$$\Delta(\hat{\mathbf{y}}, \mathbf{y}) = 1 - F_{structure}(\hat{\mathbf{y}}, \mathbf{y})$$

6.1.4 Relevancy of the Incremental SP approach

Tree transformation is a particularly challenging large-scale SP task with several real-world applications. We provide below a short discussion on why global SP algorithms cannot cope with the complexity or the scale of such a task.

Global SP models assume that the *argmax* problem (see Section 3.2) can be solved efficiently:

$$f_\theta(\mathbf{x}) = \underset{\mathbf{y} \in \mathcal{Y}_\mathbf{x}}{\operatorname{argmax}} F(\mathbf{x}, \mathbf{y}; \theta)$$

In tree transformation, this *argmax* computation is intractable for the following reasons:

- The size of $\mathcal{Y}_\mathbf{x}$ – the set of potential solutions for a given input – is, in general, exponential *w.r.t.* the number of nodes of input documents.

- The loss function $F_{structure}$ is not additively decomposable. This disables the use of dynamic programming algorithms. Solving the argmax leads to a hard combinatorial optimization problem.
- Even with a simpler loss and the one-to-one tree transformation assumptions, the optimization is intractable with dynamic programming algorithms. For example, [Wisniewski *et al.*, 2007] and [Chidlovskii et Fuselier, 2005] propose methods using dynamic programming for one-to-one tree transformation. The complexity of the dynamic programming algorithm is:

$$\mathcal{O}((\text{number of leaves in the input document})^3 \times C)$$

  where C is a constant that measure the complexity of the output schema[3]. This complexity is prohibitive even for moderate size documents.

Incremental SP approaches are built around greedy inference procedures that lead to much lower complexities. For example, if there is one decision step per input leaf, the complexity will remain linear *w.r.t.* the number of leaves of the input document.

6.2 CR-algorithms for tree transformation

In this section, we show how to use the CR-algorithm formalism in the context of tree transformation. We first describe CR-algorithms for one-to-one tree transformation in Section 6.2.1 and tree transformation with unaltered text in Section 6.2.2. We then discuss a feature function able to describe tree transformation actions in Section 6.2.3. We finally discuss supervision issues in Section 6.2.4.

6.2.1 One-to-one tree transformation

We start with a CR-algorithm that solves the one-to-one tree transformation problem. This CR-algorithm processes the leaves $\mathbf{x}_i$ one by one, in a similar way to the left-to-right sequence labeling approach presented in the previous chapter. In order to process a leaf $\mathbf{x}_i$, it enters in a node-creation loop that aims at adding new nodes to the current partial output. This node-creation loop creates at least a leaf node $\mathbf{y}_i$ that has the same textual content as $\mathbf{x}_i$. Where and what nodes to create is expressed thanks to *choose* instructions.

The proposed solution is given in CR-algorithm 9. The core of the CR-algorithm proceeds in the following way. The main loop (line 5–35) iterates over input leaves. For each such leaf, the node-creation loop (line 7–34) constructs new nodes in the current output tree. A key variable in the node-creation loop is the *currentNode* variable. This variable represents the *current position* in the *current output tree*. When starting processing a leaf, we initialize the current position to the root of the current output tree (line 6) and then repeat the following four steps:

1. **Create choices** (line 8–14) This step creates the set of currently available choices. There are two kinds of choices: `create` and `enter`. The aim of `create` is to append a new node to the childrens of *currentNode* and then to set *currentNode* to the created node. The aim of `enter` is only to change the value of *currentNode*. Entering the current node means selecting its last children as being the new current position. This makes it possible to move into the output tree without creating new nodes. There is one `create` choice per possible label. The set of possible labels may be restricted by information contained in the output

[3] often greater than the number of possible output node labels

CR-algorithm 9 One-to-one tree transformation CR-algorithm

Input: An input segmented text $\mathbf{x}$
Input: An output schema $\mathcal{G}$
Training Input: The correct output tree $\mathbf{y}$
Output: A predicted output tree $\hat{\mathbf{y}}$

```
1: ŷ ← emptyTree(G)
2: if training then
3:     currentLoss ← Δ(ŷ, y)
4: end if
5: for i = 1 to card(x) do                                  ▷ For each input leave
6:     currentNode ← getRootNode(ŷ)
7:     while currentNode ≠ null do
8:         choices ← ∅                                      ▷ Create choices
9:         for label ∈ possibleLabels(G, currentNode) do
10:            choices.insert((create, label))
11:        end for
12:        if currentNode.getNumSubTrees() > 0 then
13:            choices.insert((enter))
14:        end if
15:        choice ← choose[x_i, ŷ, currentNode] choices      ▷ Choose
16:        if choice = (create, label) then                  ▷ Apply selected choice
17:            if isInternalNode(G, label) then
18:                newNode ← createInternalNode(label)
19:                currentNode.addSubTree(newNode)
20:                currentNode ← newNode
21:            else
22:                newNode ← createTextualNode(label, x_i)
23:                currentNode.addSubTree(newNode)
24:                currentNode ← null
25:            end if
26:        else if choice = (enter) then
27:            currentNode ← currentNode.getLastSubTree()
28:        end if
29:        if training then                                  ▷ Reward
30:            newLoss ← Δ(ŷ, y)
31:            reward -(currentLoss - newLoss)
32:            currentLoss ← newLoss
33:        end if
34:    end while
35: end for
36: return ŷ
```

schema (line 9). In order to have access to the `enter` choice, *currentNode* must contains at least one children node (line 12).

2. **Choose** (line 15) Given the current input leaf $\mathbf{x}_i$, the current output tree $\hat{\mathbf{y}}$ and the current position *currentNode*, choose between the currently available choices. Each action of the MDPs induced by CR-algorithm 9 corresponds to one choice. We therefore equivalently use the *action* and *choice* terms in the following.

3. **Apply selected choice** (line 16–28) Once a choice as been selected, this step modifies the current output tree and the current output position accordingly. We distinguish two kind of `create` choices: those creating internal nodes (line 18–20) and those creating leaf nodes (line 22–24). When creating a leaf node, the current input text is added to the leaf and *currentNode* is set to **null**, in order to break the node-creation loop.

4. **Reward** (line 29–33) This step computes a reward corresponding to the previously made choice. A simple solution to define rewards would be to give the whole negative loss $\Delta(\hat{\mathbf{y}}, \mathbf{y})$ at the end of the episodes. However, as discussed in Section 5.3.1, this kind of rewards leads to hard learning problems, where the credit assignment problem is maximal. In order to ease learning, rewards should be dispatched among the successive decision steps. We here dispatch the reward by computing the delta of loss $\Delta(\hat{\mathbf{y}}, \mathbf{y})$ induced by choices. The *currentLoss* variable, initialized in line 3 and updated in line 32, keeps a trace of the current loss in order to compute these rewards.

The number of decision-steps required to construct a target document is bounded by the maximal height of target documents times the number of leaves contained by the documents. The complexity of inference when using this approach is thus:

$$\mathcal{O}(maximalHeight \times card(\mathbf{x}))$$

6.2.2 Tree transformation with unaltered text

In many real world tree transformation problems, the one-to-one assumptions are not verified. Thanks to the expressiveness of our formalism, the previous CR-algorithm can easily be extended to support new kind of transformations. We have explored two such extensions: node skipping and node reordering. These extensions makes it possible to manage tree transformation problems with unaltered text, such as the one illustrated in the right part of Figure 6.2.

The new tree transformation inference process is given in CR-algorithm 10. The differences with CR-algorithm 9 are displayed in italic text. Changes related to node reordering are shown in blue and changes related to node skipping are shown in red.

• **Node reordering** In order to support node reordering, the `create` and `enter` choices are now parameterized with a *position* argument. This position either specifies *where* a new node should be created, or *which* existing node should be entered. The cost of node reordering is to multiply the number of possible choices at each decision step by roughly the number of children of *currentNode*.

• **Node skipping** In order to suppress leaf nodes during the transformation, we introduce a new choice available at anytime: `skip`. The effect of `skip` is to set *currentNode* to **null**, which breaks the current node-creation loop. This choice makes it possible to completely ignore an

CR-algorithm 10 CR-algorithm for tree transformation with node reordering and skipping.

Input: An input sequence of leaves $\mathbf{x}$
Input: An output schema $\mathcal{G}$
Training Input: The correct output tree $\mathbf{y}$
Output: A predicted output tree $\hat{\mathbf{y}}$

```
1:  ŷ ← emptyTree(G)
2:  if training then
3:      currentLoss ← Δ(ŷ, y)
4:  end if
5:  for i = 1 to card(x) do                                  ▷ For each input leave
6:      currentNode ← getRootNode(ŷ)
7:      while currentNode ≠ null do
8:          choices ← ∅                                      ▷ Create choices
9:          for position = 0 to currentNode.getNumSubTrees() + 1 do
10:             for label ∈ possibleLabels(G, currentNode, position) do
11:                 choices.insert((create, label, position))
12:             end for
13:         end for
14:         for position = 0 to currentNode.getNumSubTrees() do
15:             choices.insert((enter, position))
16:         end for
17:         choices.insert((skip))
18:         choice ← choose[x_i, ŷ, currentNode] choices      ▷ Choose
19:         if choice = (create, label, position) then       ▷ Apply selected choice
20:             if isInternalNode(G, label) then
21:                 newNode ← createInternalNode(label)
22:                 currentNode.addSubTree(newNode, position)
23:                 currentNode ← newNode
24:             else
25:                 newNode = createTextualNode(label, x_i)
26:                 currentNode.addSubTree(newNode, position)
27:                 currentNode ← null
28:             end if
29:         else if choice = (enter, position)) then
30:             currentNode ← currentNode.getSubTree(position)
31:         else if choice = (skip) then
32:             currentNode ← null
33:         end if
34:         if training then                                 ▷ Reward
35:             newLoss ← Δ(ŷ, y)
36:             reward -(currentLoss - newLoss)
37:             currentLoss ← newLoss
38:         end if
39:     end while
40: end for
41: return ŷ
```

input leaf, or to create only a set of internal nodes, without inserting the current textual content in the output tree.

Note that, although we did not experiment them, many other extensions of the previous CR-algorithms are possible. One direction in particular, would be to add text-processing operations, such as `capitalize-the-current-text` or `split-the-current-text-by-spaces`.

6.2.3 Action descriptions

In order to learn the previous CR-algorithms, we have to provide a feature function $\phi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^d$. We describe here the feature function that has been used in our experiments. This feature function shares a number of important properties with the feature functions presented in the previous chapter. Firstly, it leads to very high dimensional feature spaces: there are often more than 10^6 distinct features. Secondly, it leads to sparse descriptions. For a given state-action pair, the number of active features, *i.e.* features that have a non-null value, is relatively low. Finally, sparsity can be exploited through a smart implementation that focuses only on the set of active features. Our implementation contain *feature generation* functions that in a given situations directly lists the set of active features. The features are thus generated automatically from the data. In general, the complexity of feature generation functions is linear *w.r.t.* the number of active features.

Sparse high-dimensional description

Figure 6.4 illustrates the feature function for the action $\mathbf{a} = create(phone, 2)$. We use only binary features. Each features is a conjunction made of conditions on the action and conditions on the state. The action is described through its kind (`enter`, `create` or `skip`) and the label it is related to, which is either the label of the created node or the label of the entered node. In our example, all the active features share the following form:

Conjunctions

$$f_{l,\ldots}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if } create \wedge label = l \wedge \ldots \\ 0 & \text{otherwise} \end{cases}$$

i.e. only features corresponding to "`create` with label l" actions may be active.

There are four kinds of features used in the experiments:

Input Content Features describe the textual content of the current input leaf. The input leaf content is first summarized by the C first and the C last leaf word where C is a context parameter, which depends on the application. Then, features are computed only for these words. There are three types of features called *word*, *character type* and *pattern*:

- **Word features** are computed for each first or last word: there is one feature denoted $f_{k,l,j,w}$ for each action kind $k \in \{enter, create, skip\}$, output label l, word position j in the leaf word sequence and word w:

$$f_{k,l,j,w}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if } k \wedge label = l \wedge \text{inputWord}[j] = w \\ 0 & \text{otherwise} \end{cases}$$

  where $inputWord[j]$ is the word number at position j in the current leaf.

- **Character type features** correspond to the type of characters used in words. The type of a character can be a digit, an upper case or lower case letter of the alphabet. The character type features of a word are constructed by replacing each lower letter of the word by 'l', each upper letter by 'u' and each digit by 'd'.

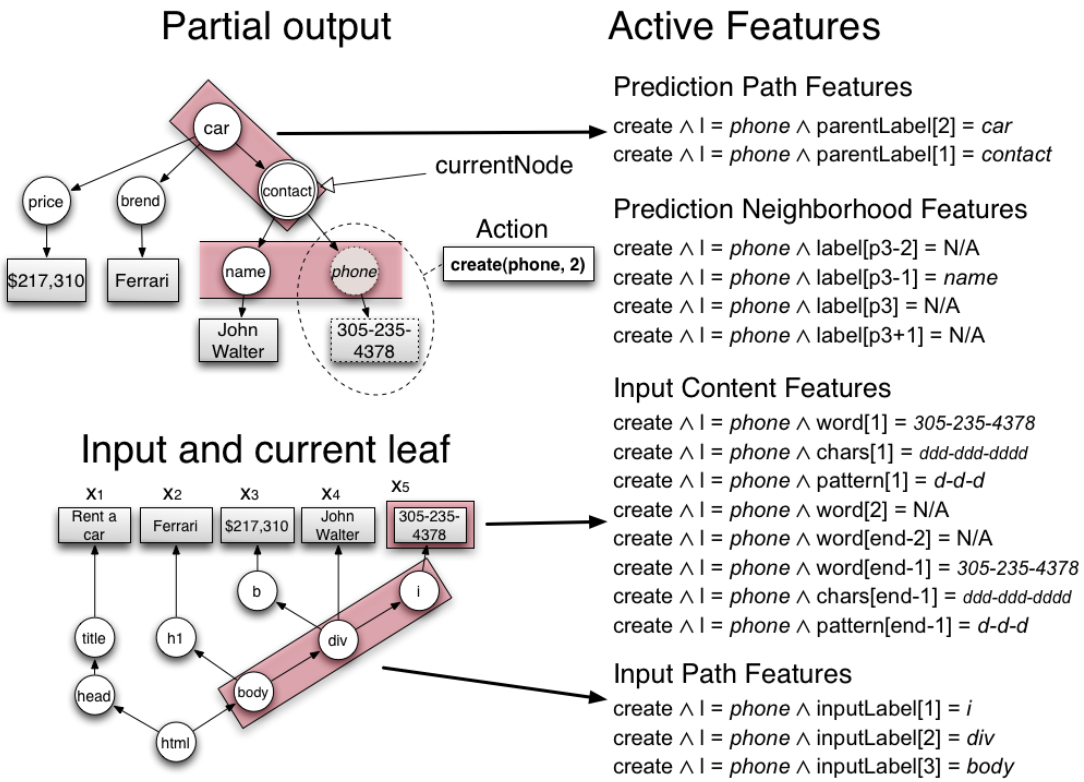


Figure 6.4: **Tree transformation action features.** Top-left: the current partial output and a candidate action. The current node is the double-circled *contact* node. The candidate action consists in creating a *phone* node into this node. Bottom-left: the input tree and the current input leaf $\mathbf{x}_5$. Right: list of the active features corresponding to the action `create`(contact, 2). The active features are those that have a value of 1. The rectangles in red indicate the context considered for each feature type. This context is the number of ancestors, siblings or adjacent words that appear in the features. Since there is only one word in the current textual content, the same word appears both in the *first words* features and the *last words* features.

- **Pattern features** of a word are built based on the **Character type features** by replacing successive occurrences of the same character type by a unique occurrence. This roughly corresponds to a regular expression over the type of characters of a particular word. Figure 6.4 gives examples of such features.

Input Path Features Input Path Features correspond to the structural context of the current input leaf $\mathbf{x}_i$. They encode the labels of the ancestors of the input leaf. There is one such feature per action kind $k \in \{enter, create, skip\}$, output label l, input label l' and height h in the input tree:

$$f_{k,l,l',h}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if } k \wedge label = l \wedge \text{ inputLabel}[h] = l' \\ 0 & \text{otherwise} \end{cases}$$

where $inputLabel[h]$ is the label of the h-th ancestor of the current leaf.

The last two types of features encode the current partial output tree.

Prediction Path Features Prediction Path Features encode the label of the ancestors of *currentNode*. There is one such feature per action kind $k \in \{enter, create, skip\}$, pair of labels (l, l') and ancestor height h:

$$f_{k,l,l',h}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if } k \wedge label = l \wedge \text{ parentLabel}[h] = l' \\ 0 & \text{otherwise} \end{cases}$$

where parentLabel$[h]$ is the label of the h-th ancestor of the node concerned by triplet i.

Prediction Neighborhood Features These features are similar to the prediction path features, except that they encode the children labels of *currentNode* instead of its parent labels. These features are relative to the position of the node to be created, or the position of the node to be entered. There is one such feature per action kind $k \in \{enter, create, skip\}$, label pair (l, l') and context position δ:

$$f_{k,l,l',\delta}(\mathbf{s}, \mathbf{a}) = \begin{cases} 1 & \text{if } k \wedge label = l \wedge \text{ neighboringLabel}[\delta] = l' \\ 0 & \text{otherwise} \end{cases}$$

where neighboringLabel$[\delta]$ is the label of the δ-th sibling of node concerned by the action.

At last we introduce a special label called `N/A` to denote out-of-bounds elements, *i.e.* elements that do not exists in the tree. Typically, if we consider a leaf with only one word and we want to compute the feature of the second word (see *Input Content Features*), which does not exist, we will use this special label. This is illustrated in Figure 6.4.

With these definitions, the number d of active features for a given state-action pair is:

$$\begin{aligned} d = 3 \times 2 \times \text{word context size} \; &+ \; \text{input path context size} \\ + \; \text{prediction path context size} \; &+ 2 \times \text{prediction neighborhood context size} \end{aligned}$$

where *word context size*, *input path context size*, *prediction path context size* and *prediction neighborhood context size* are parameters of the feature function ϕ. The values given to these parameters will be detailed below. In general, the number of active features for a given state-action pair is relatively low (*e.g.* less than 50).

6.2.4 Supervision

In the previous chapter, we introduced sequence labeling, which had the nice property to be easy to supervise. Indeed, we only considered CR-algorithms where the optimal learning policy (OLP) was easy to compute. In the case of tree transformation, computing the OLP is not a trivial problem (see Figure 6.5 for example).

Beam-search of the best action

In such cases, the authors of SEARN suggest to use a search-procedure to (approximately) find the best action. Consider for example a greedy beam-search procedure. The complexity of searching the optimal actions for one training trajectory[4] is $\mathcal{O}(T^2 \times b \times a^2)$, where b is the size

[4]Given a state-action pair, finding the best completion of the partial output, takes $\mathcal{O}(T \times b \times a)$ time. This search has to be launched for each action $a \in \mathcal{A}_{\mathbf{s}}$ in each visited state s of the trajectory: $\mathcal{O}(T \times a)$ times.

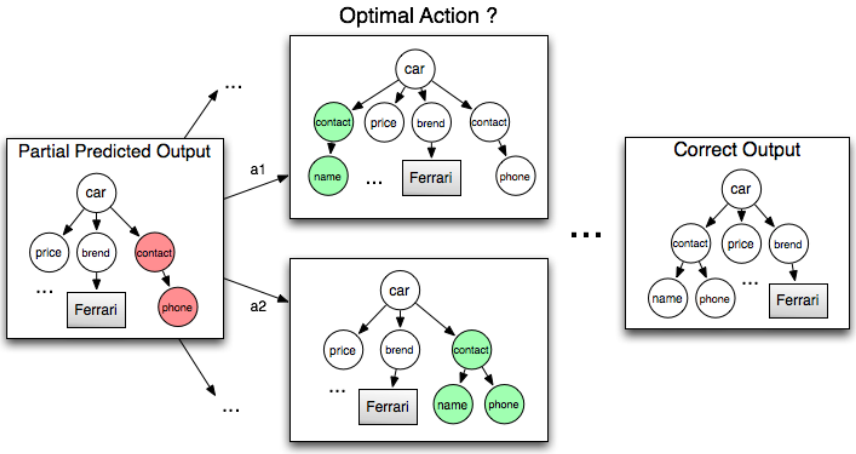


Figure 6.5: **Illustration of the difficulty of finding the optimal actions in tree transformation.** Left: the current partial output which is partially wrong. The red nodes *(contact, phone)* have not been put at the correct location. Right: the correct output tree. Center: two possible sequences of actions that have both advantages and drawbacks. a_1 puts the *(contact, name)* nodes at the good location *w.r.t.* the correct output tree, but at the cost of duplicating the *contact* node. a_2 correctly reconstructs the *(contact, name, phone)* sub-tree, but does not solve the location problem. In this example, selecting the optimal action is a non-trivial problem.

of the beam, T is the depth of the search (in our case the number of input leaves) and a is the mean number of available actions per state.

On large-scale tasks such as tree transformation, determining an approximate OLP with beam search has a prohibitive complexity. Let us perform a naive computation of the time required to process one typical document with $T = 100$ leaves and up to $a = 5,000$ actions per state with a beam size of 5. On such documents, our implementation performs inference – with $\mathcal{O}(T \times a)$ complexity – in $\approx 1s$. The time required to search the best action in each visited state for one document is thus of the order of $1s \times 100 \times 5000 \times 5 \approx 1$ month computation time. This solution is then clearly intractable.

One major contribution of the CR-algorithm formalism is to provide new supervision means, when the OLP assumption is too strong. In particular, we investigated two alternatives: *Alternative supervision means*

• **Content-driven heuristic** We describe below a heuristic to select *not too bad* actions. This heuristic can be used within CR^{ank} to quickly find a *not too bad* policy.

• **Pure reinforcement learning** Reinforcement learning techniques are particularly relevant to tree transformation problems. Indeed, we will show that general approximate reinforcement learning are able to find good policies on a wide range of problems, using only the reward feedbacks.

We now describe the content-driven heuristic $h^{content} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ that was used in some of our experiments. Note that the details of this heuristic are slightly subtle and are not of primary importance for the remaining of this chapter. We therefore only give a brief description of the ideas underlying it. *Content-driven heuristic*

Algorithm 9 Content-driven Heuristic for tree transformation

Require: A state $\mathbf{s} = (\mathbf{x}_i, \bar{\mathbf{y}}, currentPosition)$
Require: An action $\mathbf{a}$
Require: The correct output $\mathbf{y}$
Ensure: A score $\in \mathcal{R}$

```
1: currentText ← getTextualContent(x_i)
2: occurencesInOutput ← findOccurencesOfText(currentText, y)
3: matchings ← matchPositions(currentNode, occurencesInOutput)
4: if a = skip then
5:     return 𝟙{card(matchings) = 0}
6: else
7:     update machings with respect to a
8:     s' ← executeAction(s, a)
9:     retrieve the new value of currentNode from s'
10:    res ← 0
11:    for each correctOutputNode ∈ machings do
12:        res ← res + similarityOfNeighborhingLabels(currentNode, currentOutputNode)
13:    end for
14:    return res
15: end if
```

The pseudo-code of the content-driven heuristic is given in Algorithm 9 and illustrated in Figure 6.6. The central idea of the heuristic is to use the textual content of the documents to guess the quality of a given action. This is performed thanks to the following steps:

1. **Retrieve the current input text** (line 1).

2. **Find occurrences of the text in the correct output tree** (line 2). The *findOccurencesOfText* function returns the set of positions of the leaves $\mathbf{y}_i$ that contains *currentText*. This set can empty if *currentText* does not appear in $\mathbf{y}$. It can also contain more than one element, if *currentText* appear more than once in $\mathbf{y}$.

3. **Match the current position to correct-tree positions** (line 3). Since the current predicted document might be partially wrong, we need a heuristic matching mechanism in this step. The *matchPositions* function tries to determine which nodes of the correct tree best match *currentNode*. Two nodes can be matched if they share the same label. If multiple nodes have the same label, we select the node of the correct tree that maximizes a neighborhood similarity function:

$$\text{matching} = \underset{candidateNode}{\text{argmax}} \ \text{similarityOfNeighborhingLabels(currentNode, candidateNode)}$$

   where *similarityOfNeighborhingLabels* roughly computes the percentage of identical labels between the neighborhood of a predicted node and this of a correct node.

4. `skip` **action** (line 5). The current input leaf should be skipped if it does not appear in the correct output tree. We therefore give a score of 1 to the `skip` action if the set of matching output nodes is empty.

5. `enter` **and** `create` **actions** (line 7–13). The score assigned to `enter` and `create` actions is computing in the following way. First, each matching is updated *w.r.t.* the action $\mathbf{a}$ (line

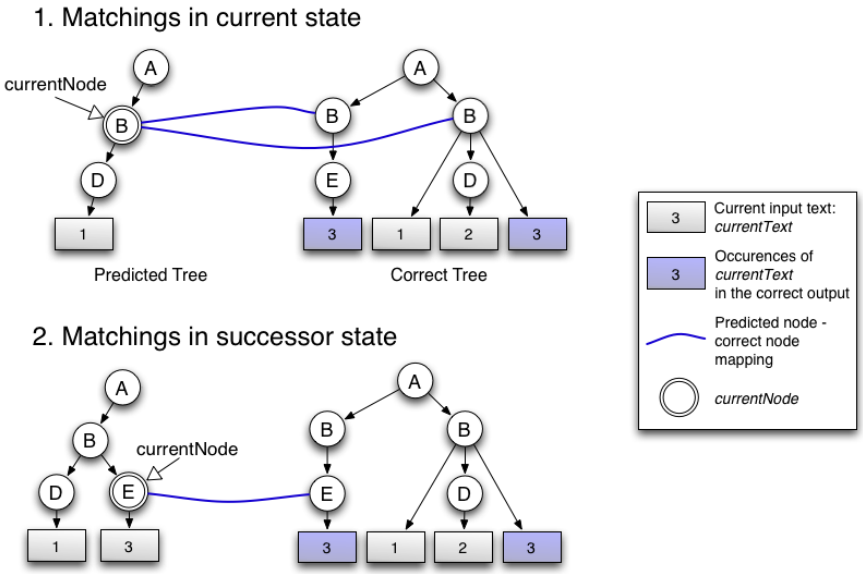


Figure 6.6: **Illustration of the content-driven heuristic for tree transformation.** This figures illustrates the computation of the content-driven heuristic for the action `create`(E, 2), *i.e.* the action that creates a new node with label E at position 2 under the current node B. Top: the matchings computed by the heuristic in line 3. Bottom: the matchings as updated by lines 7–9 of the heuristic. Here, the action `create`(E, 2) is a good choice, since its leads to one valid matching with a good neighborhood similarity: in both trees, the parent of the matched nodes have the label B and their grand-parents have the label A.

Corpus	RealEstate	Mixed-Movie	Shakespeare	Inex-Ieee	Wikipedia
Input Format	XML	HTML	Segmented Text	Segmented Text	HTML
Target Format	XML	XML	XML	XML	XML
Node skipping	yes	yes	no	no	yes
Node reordering	yes	yes (±10)	no	no	yes (±10)
Num. Documents	2,367	13,048	750	12,107	10,681
Internal Nodes	$\simeq 33$	$\simeq 64$	$\simeq 236$	$\simeq 650$	$\simeq 200$
Leaf Nodes	$\simeq 19$	$\simeq 39$	$\simeq 194$	$\simeq 670$	$\simeq 160$
Depth	$\simeq 6$	5	$\simeq 4.3$	$\simeq 9.1$	$\simeq 7.7$
Labels	37	35	7	139	256

Table 6.1: **Tree transformation corpora properties and statistics.** The top part of the table gives properties of the tree transformation problem. From top to bottom: the name of the corpus, the input format, the target format and two flags that tell if the transformation requires node skipping and if it requires node reordering. The bottom part of the table gives statistics on the target documents of the dataset. From top to bottom: the number of documents, the mean number of internal nodes per tree, the mean number of leaves per tree, the mean tree depth and the number of output labels.

7), such as illustrated in Figure 6.6. We then compute the new state, *i.e.* the new predicted tree and the new value of *currentNode* (line 8). The heuristic score is then computing by summing the *similarityOfNeighborhingLabels* measure over all matchings between the new *currentNode* and the corresponding nodes of the correct output.

The quality of the heuristic $h^{content}$ depends on the tree transformation problems. When dealing with one-to-one transformation problems, the heuristic generally performs nearly perfectly, *i.e.* good actions have better score than bad actions. With more complex transformations, the quality of the heuristic may seriously be degraded, as shown by some of our experiments. Failures of the heuristic typically happen when the current predicted tree is partially wrong, or when the current text appears in several different places of the correct output tree.

6.3 Experiments

In this section, we describe the set of experiments that was performed on the challenging tree transformation task. We first present the datasets that were in Section 6.3.1. We then develop three series of experiments. First, we deal with one-to-one tree transformation in Section 6.3.2. We then give results for tree transformation with node skipping and reordering in Section 6.3.3. Finally, we introduce a technique called *action collapsing* that greatly improves the results for reinforcement learning approaches in Section 6.3.4.

6.3.1 Datasets

We used three medium-scale datasets and two large-scale datasets that are described below:

• REALESTATE [Doan *et al.*, 2003]. This corpus, proposed by Anhai Doan[5] is made of 2,367 data-oriented XML documents. The documents are expressed in two different XML formats. The aim is to learn the transformation from one format to the other.

• MIXED-MOVIE [Denoyer et Gallinari, 2007]. The second corpus is made of more than 13,000 movie descriptions available in three versions: two mixed different XHTML versions and one XML version. This corresponds to a scenario where two different websites have to be mapped onto a predefined mediated schema. The transformation includes node suppression and some node displacements.

• SHAKESPEARE [Chidlovskii et Fuselier, 2005]. This corpus is composed of 60 Shakespeare scenes[6]. These scenes are small trees, with an average length of 85 leaf nodes and 20 internal nodes over 7 distinct tags. The documents are given in two versions: a flat segmented version and the XML version. The tree transformation task aims at recovering the XML structure using only the text segments as input.

• INEX-IEEE [Fuhr *et al.*, 2002b]. The INEX-IEEE corpus is composed of 12,017 scientific articles in XML format, coming from 18 different journals. The tree transformation task aims at recovering the XML structure using only the text segments as input.

• WIKIPEDIA [Denoyer et Gallinari, 2006]. This corpus is composed of 12,000 wikipedia pages. For each page, we have one XML representation dedicated to wiki-text and the HTML code. The aim is to use the layout information available in the HTML version, for predicting the semantical XML representation.

The properties and statistics of our corpora are summarized in Table 6.1. For each corpus, we mention if node skipping and node reordering are required to fulfill the transformation.

For each corpus, we randomly extracted 100 examples to form the training set. The schema $\mathcal{G}$ is constructed automatically given these 100 examples. The schema is composed of a set of valid sibling bi-grams and parent bi-grams, which have the following form:

• **Sibling bi-gram** : Label l_1 may appear before Label l_2, or equivalently: label l_2 may appear after label l_1

• **Parent bi-gram** : Label l_1 may appear as a child of label l_2

The set of valid bi-grams in $\mathcal{G}$ are those that appear at least once in the 100 training examples. When enumerating the set of possible labels for a new node in CR-algorithm 9 and CR-algorithm 10, the function *possibleLabels* restricts the set of possible labels to those that respect all the bi-grams of $\mathcal{G}$. Depending on the corpora, we applied some simple rules to further limit the number of possible actions per state:

Constraints on actions

- We always used the simplest CR-algorithm that is able to fulfill the required tree transformation, *i.e.* if node skipping is not required, we do not use `skip` actions.

[5] `http://www.cs.wisc.edu/~anhai/`
[6] `http://metalab.unc.edu/bosak/xml/eg/shaks200.zip`

Corpus	RealEstate	Mixed-Movie	Shakespeare	Inex-Ieee	Wikipedia
Content	0	6	6	6	6
Input Path	2	6	0	0	6
Pred. Path	2	6	5	5	6
Pred. Neighborhood	2	4	6	6	4

Table 6.2: **Tree transformation feature function parameters.** From top to bottom: the dataset name, the context size for content features, the context size for input path features, the context size for prediction path features and the context size for prediction neighborhood features.

- We limited the size of the possible moves in node re-ordering on the Mixed-Movie and Wikipedia corpora. This is performed by restricting the range of the parameter *position* in CR-algorithm 10 to be at a maximum distance of 10 from the lastly created node in *currentNode*.
- On the Wikipedia corpus, we restricted the schema $\mathcal{G}$ to bi-grams that appeared at least 5 times in the training set.

Feature function parameters

Table 6.2 gives the context size parameters that were used into the feature function ϕ. Few tuning effort was spent on these parameters. For the RealEstate dataset, the transformation that has to be learned only depends on the labels, so we removed the content features. For the Shakespeare and Inex-Ieee datasets, the input documents are flat-segmented documents, which makes input path features unavailable, *i.e.* the only available input features are those coming from the text.

6.3.2 One-to-one tree transformation

Our first set of experiments explores the use of CR^{ank} with CR-algorithm 9 to perform one-to-one tree transformation.

Artificial Problems

Since most of our datasets do not respect the one-to-one assumptions, we created artificial problems denoted RealEstate $1 \rightarrow 1$, Mixed-Movie $1 \rightarrow 1$ and Wikipedia $1 \rightarrow 1$ in the following way: given a target document $\mathbf{y}$ (an XML file), we extract the sequence of text blocks contained by leaves $\mathbf{y}_i$, to automatically create the associated input document $\mathbf{x} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$. Input documents are flat-segmented texts and the aim is to predicted the missing XML structure. Each corpus is composed of 100 randomly selected training examples and evaluation is performed on all the remaining documents.

CR^{ank} with heuristic

In these experiments, we supervise CR^{ank} with the content-driven heuristic $h^{content}$. In order to introduce $h^{content}$ into the action cost function, we simply compute the difference between the best action's heuristic score and the current action's heuristic score:

$$c(\mathbf{s}, \mathbf{a}) = \big(\max_{\mathbf{a}' \in \mathcal{A}_\mathbf{s}} h^{content}(\mathbf{s}, \mathbf{a}') \big) - h^{content}(\mathbf{s}, \mathbf{a})$$

We use same learning rate and regularization parameters as in Chapter 5. The resulting training behavior of CR^{ank} is given in Figure 6.7. The results show that, for the one-to-one tree transformation problem, even with an approximated heuristic, CR^{ank} is able to learn policies in a relative few number of iterations. We now describe baselines, which makes it possible to appreciate the quality of these policies.

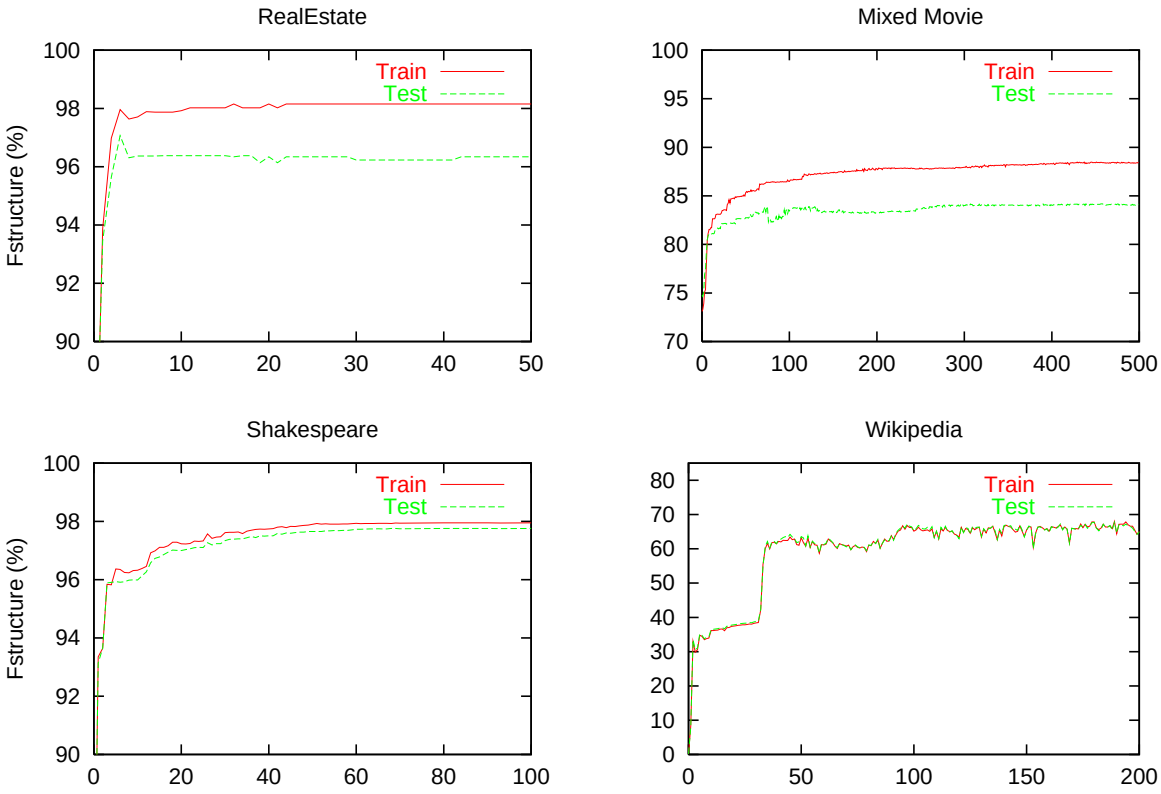


Figure 6.7: **CR^{ank} training behavior for one-to-one tree transformation.** The X-axis corresponds to number of training iterations, *i.e.* the number of passes over the whole training set. The Y-axis corresponds to the mean $F_{structure}$ score between predicted documents and correct documents. For each dataset, the curves are the train and test $F_{structure}$ scores as functions of the number of training iterations.

	Sequence		Constraint Sequence		One to one transformation		
	F_{leaf}	$F_{subtree}$	F_{leaf}	$F_{subtree}$	F_{leaf}	F_{path}	$F_{subtree}$
RealEstate $1 \rightarrow 1$	**98.32**	82.95	97.03	82.42	96.07	95.81	**96.74**
Mixed-Movie $1 \rightarrow 1$	**91.63**	74.31	87.22	77.95	84.92	84.39	**84.13**
Shakespeare	40.16	16.24	26.35	50.93	**98.44**	98.10	**97.95**
Wikipedia $1 \rightarrow 1$	41.98	08.07	32.93	52.07	**79.21**	65.01	**68.13**

Table 6.3: **Sequence labeling models vs one-to-one tree transformation.** This table gives the mean F_{leaf}, F_{path} and $F_{subtree}$ scores between predicted documents and correct documents of the test-set. We compare three approaches: left-to-right sequence labeling, constraint left-to-right sequence labeling and one-to-one tree transformation. For each corpus, the best scores are shown in bold.

Sequence

We performed a set of experiments with the left-to-right sequence labeling CR-algorithm described in Chapter 5. Sequence labeling models are only able to predict the sequence of labels of the leaf nodes $\mathbf{y}_i$. Since they do not reconstruct the full structure of $\mathbf{y}$, only the F_{leaf} scores are truly comparable between sequence labeling and tree transformation approaches. The F_{path} score is generally 0 for sequence labeling models, since these models do not construct full paths from the root to the leaves. The $F_{subtree}$ score reflects how much the correctly predicted leaves contribute to the whole output structure.

Constraint Sequence

In addition to the left-to-right sequence-labeling baseline, we have tried a constraint version of left-to-right labeling. This version restricts the set of available labels at each step by using the sibling bi-grams from the output schema $\mathcal{G}$.

Results

Table 6.3 gives the comparison between the sequence labeling approaches and the one-to-one tree transformation approach. Sequence labeling methods optimize the Hamming loss, which is closely related to F_{leaf}, while one-to-one tree transformation focuses on the $F_{subtree}$ score. It seems thus natural that the sequence labeling approaches behave better *w.r.t.* F_{leaf} than the one-to-one transformation approach on RealEstate and Mixed-Movie. However, on Shakespeare and Wikipedia our results exhibit a fully different behavior. Here, sequence labeling models perform poorly compared to the tree transformation model. This result tends to demonstrate that structure plays a major role in these datasets. Structure may be exploited in two ways. Firstly, the schema provides a set of constraints to delimit the set of valid outputs, which leads to a smaller search-space. Secondly, structure – even if it has been predicted in the previous steps – provides a rich context through the feature function, which may be decisive for some decision steps.

On all our datasets, the $F_{subtree}$ score of the one-one-transformation method significantly outperforms those of the sequence labeling approaches. CR^{ank} supervised by $h^{content}$ proves thus to be an efficient solution for one-to-one tree transformation. Furthermore, the complexity of our approach is reasonable: inference time is linear *w.r.t.* the number of leaves of the trees. In practice, inference takes most of the time less than one second per document.

6.3.3 Node skipping and reordering

This section explores CR-algorithm 10 that provides the support for node skipping and node reordering. We compare two approaches: CR^{ank} supervised with $h^{content}$ and reinforcement learning with Sarsa and Olpomdp.

Tuning

CR^{ank} is configured as in Section 6.3.2 and the reinforcement learning algorithms are tuned

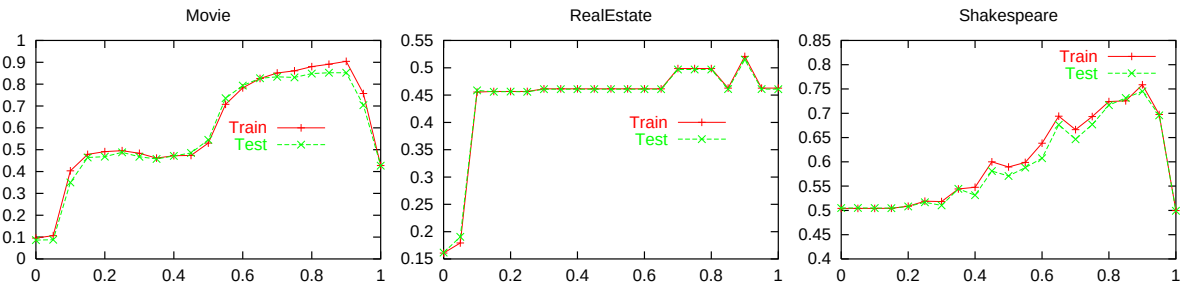


Figure 6.8: **Tree transformation: impact of the discount parameter on Sarsa.** The X-axis corresponds to the value of the discount factor and the Y-axis corresponds to mean $F_{subtree}$ percentages. We give the train and test scores for three datasets: MIXED-MOVIE, REALESTATE and SHAKESPEARE.

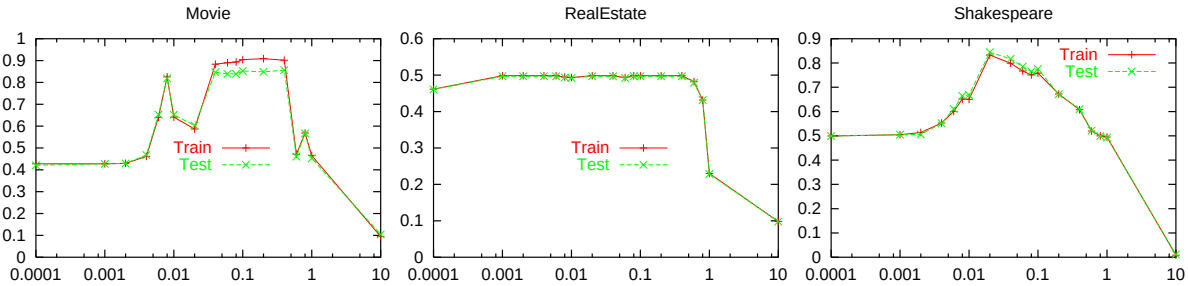


Figure 6.9: **Tree transformation: impact of the discount parameter on Sarsa.** The X-axis corresponds to the value of the learning rate parameter and the Y-axis corresponds to mean $F_{subtree}$ percentages. In our experiments we use learning rates that are inversely proportional functions of the time. The learning rate parameter is a scaling parameter of this function.

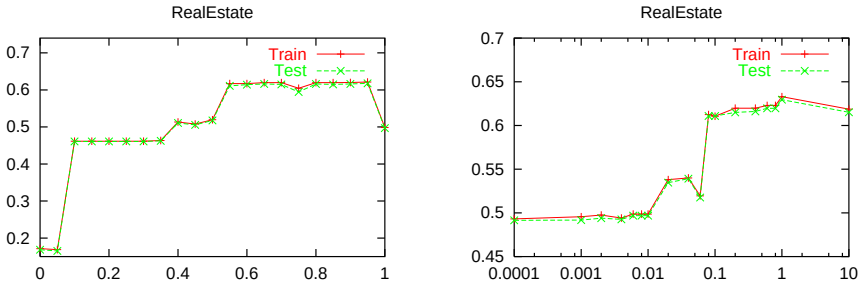


Figure 6.10: **Tree transformation: impact of the parameters of Olpomdp.** Left: the impact of the discount factor on the mean $F_{subtree}$ score. Right: the impact of the learning rate parameter on the mean $F_{subtree}$ score.

	Supervised		Reinforcement learning	
	CR^{ank}	$\pi^{greedy}_{h-content}$	Sarsa	Olpomdp
Inex-Ieee	60.09	100	**61.73**	**61.73**
Mixed-Movie	63.60	69.45	76.68	**87.33**
Wikipedia	**50.69**	67.97	29.95	31.95
RealEstate	**99.94**	100	51.10	81.97
Shakespeare	**97.95**	100	77.52	61.27

Table 6.4: **CR^{ank} with heuristic vs reinforcement learning for tree transformation.** We give the mean $F_{subtree}$ scores on the test-set for the following methods: CR^{ank} supervised with $h^{content}$, the optimistic bound $\pi^{greedy}_{h-content}$ and two reinforcement learning algorithms Sarsa and Olpomdp. For each corpus, the best scores of the learning-based methods in shown in bold.

in the same way as in chapter **??**, with 100 training iterations for each tuning trial. Figure 6.8 and Figure 6.9 show the impact of the discount factor and learning rate parameters on Sarsa. Figure 6.10 shows the impact of the β parameter on Olpomdp. On most datasets, the quality of learned policies crucially depends on these parameters. The reinforcement learning approach thus requires a careful tuning of the hyper-parameters.

Heuristic-greedy policy

We have also computed scores for the greedy policy *w.r.t.* the $h^{content}$ heuristic. This policy, denoted $\pi^{greedy}_{h-content}$ is defined in the following way:

$$\pi^{greedy}_{h-content}(\mathbf{s}) = \underset{\mathbf{a}\in\mathcal{A}_\mathbf{s}}{\text{argmax}}\, h^{content}(\mathbf{s},\mathbf{a})$$

If multiple actions are optimal, one of those actions is selected randomly. Note that the greedy policy *w.r.t.* $h^{content}$ makes use of the correct outputs, does not rely on learning and is thus not able to generalize to new inputs. The scores given by $\pi^{greedy}_{h-content}$ are optimistic bounds of the performance that can be reached by the learning process of CR^{ank}.

Results

Table 6.4 summaries compares the reinforcement learning approach with the supervised approach for CR-algorithm 10. As shown by $\pi^{greedy}_{h-content}$, the heuristic does not perform very well on Mixed-Movie and Wikipedia. This motivates the use of reinforcement learning techniques that are theoretically not limited by this bound. In practice, reinforcement learning significantly outperforms the supervised method on the Mixed-Movie corpus with + 23.7 % improvement of the $F_{subtree}$ score. However, on three other corpora (Wikipedia, RealEstate and Shakespeare), it is significantly worst than CR^{ank}.

The mixed results of reinforcement learning on tree transformation may be due to the huge amount of exploration which is required in this problem. Indeed, most tree transformation actions have long-term consequences. Furthermore, the immediate reward is only poorly informative on the quality of actions. For example, `enter` and `skip` actions lead to an immediate reward of 0 (the predicted tree remains unchanged by these actions). Only `create` actions lead to non-null rewards. But, even in this case, the immediate rewards may be poorly correlated to the long-term quality of the action. The next section proposes an alternative way to define tree transformation actions to ease the reinforcement-learning problem.

6.3.4 Action collapsing

In the previous section, we showed that reinforcement learning give mixed results on CR-algorithm 9 and CR-algorithm 10. We propose here an alternative CR-algorithm, which makes the reinforcement learning problem much easier.

Action collapsing

The new CR-algorithm relies on the idea of *action collapsing*. In order to get rewards closer to related actions, we use a new set of actions by collapsing those of CR-algorithm 9 or CR-algorithm 10. Each collapsed action corresponds to a whole sequence of node-creation actions for a given input leaf. Thanks to collapsed actions, a non-null reward is perceived after each decision step[7]. Compared to the previous CR-algorithms, the actions still have long-term consequences, but the immediate rewards are much more related to the true long-term quality of actions.

CR-algorithm 11 Tree transformation CR-algorithm with collapsed actions.

Input: An input segmented text $\mathbf{x}$
Input: An output schema $\mathcal{G}$
Training Input: The correct output tree $\mathbf{y}$
Output: A predicted output tree $\hat{\mathbf{y}}$

```
1:  ŷ ← emptyTree(G)
2:  if training then
3:      currentLoss ← Δ(ŷ, y)
4:  end if
5:  for i = 1 to card(x) do                          ▷ For each input leave
6:      currentNode ← getRootNode(ŷ)
7:      choices ← all possible node-creation action-sequences of CR-algorithm 9 or 10.
8:      (a_1, ..., a_n) ← choose[x_i, ŷ, currentNode] choices          ▷ Choose
9:      for j = 1 to n do                            ▷ Apply collapsed action
10:         apply a_j on ŷ and on currentNode
11:     end for
12:     if training then                                              ▷ Reward
13:         newLoss ← Δ(ŷ, y)
14:         reward -(currentLoss - newLoss)
15:         currentLoss ← newLoss
16:     end if
17: end for
18: return ŷ
```

The idea of collapsed actions is depicted in CR-algorithm 11. Line 7 first enumerates the set of all possible node-creation action-sequences $(\mathbf{a}_1, \ldots, \mathbf{a}_n)$ for the current input leaf. Line 8 then chooses a node-creation action-sequence among this set. Lines 9–11 finally apply the chosen action-sequence iteratively.

Non-learning baselines

Thanks to collapsed actions, the predicted tree changes after each decision step. We can thus expect the greedy policy *w.r.t.* the immediate reward to perform not too bad. Furthermore, since each action adds a full path in the predicted tree, maximizing the immediate F_{path} gain also makes sense. We thus introduced two new non-learning baselines $\pi^{greedy}_{structure}$ and π^{greedy}_{path}.

[7] Except for `skip` actions.

Corpus	Score	RL		Baselines			
		Sarsa	Olpomdp	$\pi^{greedy}_{structure}$	π^{greedy}_{path}	π^{random}	Pcfg+Me
RealEstate	$F_{structure}$	99.54	**99.99**	87.09	**97.09**	3.27	49.8
	F_{path}	99.87	**99.99**	84.42	**100**	3.91	7
	$F_{content}$	99.88	**100**	**100**	**100**	5.10	99.9
Mixed-Movie	$F_{structure}$	86.22	**86.50**	**47.04**	44.15	3.54	/
	F_{path}	91.53	**91.88**	52.02	**52.18**	5.29	/
	$F_{content}$	91.53	**92.05**	52.02	**52.18**	5.67	/
Shakespeare	$F_{structure}$	**96.03**	95.88	**98.65**	75.16	11.34	94.7
	F_{path}	**97.88**	97.72	98.91	**100**	16.47	97.0
	$F_{content}$	**98.87**	98.40	99.83	**100**	18.25	98.7

Table 6.5: **Tree transformation with collapsed actions, medium-scale datasets.** For each dataset and each method, we give the three average similarity scores $F_{structure}$, F_{path} and $F_{content}$ between the predicted and correct trees of the test set. The two first columns correspond to the Sarsa and Olpomdp reinforcement learning algorithms. The next three columns are non-learning baselines. The last column is the Pcfg+Me [Chidlovskii et Fuselier, 2005] baseline. The / symbol denotes results that could not be computed due to the complexity of Pcfg+Me.

These greedy policies make use of the correct output and select actions whose execution most increase the immediate $F_{structure}$ or F_{path} similarity scores. The scores of $\pi^{greedy}_{structure}$ could be considered as upper bounds for the performance reachable by learning to maximize the immediate reward, *e.g.* with Sarsa and a discount factor of 0. We also computed the scores of the random policy π^{random}, which selects actions randomly.

Learning-based baseline

We only have one *learning-based* baseline on two of these datasets for the complexity reasons explained in Section **??**. The baseline model Pcfg+Me [Chidlovskii et Fuselier, 2005] is a global model for *one-to-one* tree transformation. It models the probability of outputs by using probabilistic context free grammars (PCFG) and maximum-entropy (ME) classifiers. Inference is then performed with a dynamic-programming algorithm that has a cubic complexity in the number of input leaves. This model can only operate on the two smallest datasets RealEstate and Shakespeare.

Medium-scale results

Our experimental results on the three medium-scale datasets are given in Table 6.5. On RealEstate and Mixed-Movie, the reinforcement learning approaches give significantly better results than the greedy policy baselines. This result is very satisfactory and has two major implications. Firstly, it shows that reinforcement-learning algorithms are able to find a *better strategy than greedily moving* toward the correct output. Secondly, it shows that the algorithms perform an *effective learning and generalization* of this strategy. On Shakespeare, the scores of reinforcement learning algorithms is slightly inferior to those of the greedy policies. Since greedy policies perform nearly perfectly on this corpus, the main difficulty here is more related to *generalization* than *exploration*.

Comparison with state-of-the-art

The RealEstate collection corresponds to an *XML database* where we need to label the leaves correctly and to put them in the correct order. The task is easy, and most RL approaches achieve $> 99\%$ on the different scores. Pcfg+Me only performs 7 % on the F_{path} score and about 50 % on the $F_{structure}$ score because it does not handle node skipping and node reordering, which are required on this collection. On the Shakespeare corpus, Pcfg+Me gives slightly lowers results than RL methods. This result might be related to the different kind of features

exploited by the methods. Since PCFG+ME relies on a PCFG, it cannot deal with some of the features that we use here (e.g. the prediction path features, see Section 6.2.3).

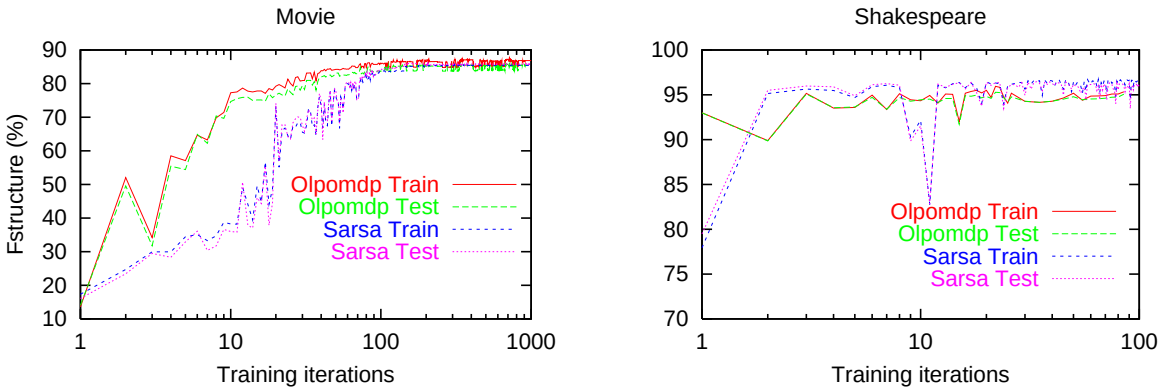


Figure 6.11: **Training behavior of Sarsa and Olpomdp on tree transformation with collapsed actions.** We give the train and test $F_{structure}$ scores for each method as a function of the number of training iterations. Left: results for MIXED-MOVIE. Right: results for SHAKESPEARE.

Training behavior

The training behavior of SARSA and OLPOMDP is illustrated by Figure 6.11. We have often observed that the training behavior of SARSA is slightly noisier than that of OLPOMDP. On our datasets, the number of training iterations needed to converge was reasonable in all cases.

Impact of parameters

The impact of the discount factor on SARSA is illustrated in Figure 6.12. The optimal discount factor values highly depend on the corpus. On some corpora, such as MIXED-MOVIE, discount factors close to 1 are required to outperform the greedy policy baselines. In some other cases, such as SHAKESPEARE, the optimal discount factors are close to 0.5. As for sequences, the optimal discount factors are not 1. We believe that discount factors smaller than 1 lead to simpler exploration problems, which makes learning easier. This may also be related to the use of SARSA with a limited amount of training material. Figure 6.13 gives the impact of the β on OLPOMDP. It can be seen that β has a similar impact on OLPOMDP as the discount on SARSA has. For the MIXED-MOVIE and SHAKESPEARE corpora, the best β value were respectively 0.85 and 0.5.

Large-scale tree transformation

In order to demonstrate the scalability of our approach, we have performed experiments with SARSA on the two large-scale corpora. Since, in our implementation, episodes with OLPOMDP take about 50 more time than episodes with SARSA[8], we did not use OLPOMDP on the large-scale datasets. These experiments required $\approx$ 5 days training time in order to perform 1000 training iterations, *i.e.* 10^5 SARSAepisodes, for each dataset. This huge amount of time has to be contrasted with the scale of the task: these corpora involve particularly large documents (the biggest documents in these corpora contain up to 10.000 nodes), complex operations (nodes displacements or nodes deletions), highly heterogeneous documents and large number of labels (139 labels for INEX-IEEE and 256 labels for WIKIPEDIA).

[8]SARSA runs much faster thanks to a particularity of our implementation, which is able to compute $\langle \phi(\mathbf{s}, \mathbf{a}), \theta \rangle$ dot products without storing the $\phi(\mathbf{s}, \mathbf{a})$ vectors in memory. In OLPOMDP, the main cpu-time bottleneck is the allocation and deletion of data structures for storing these vectors.

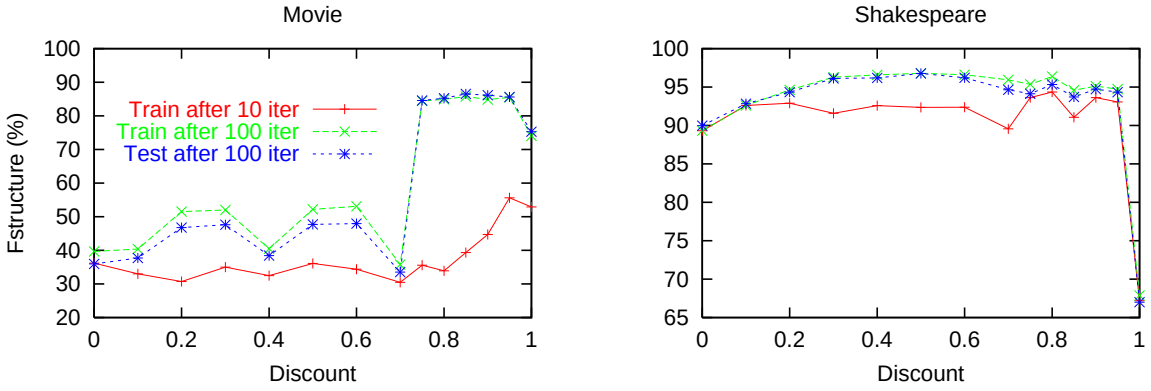


Figure 6.12: **Impact of the discount factor in tree transformation with Sarsa and collapsed actions.** The curves correspond to the training scores after 100 and 1000 training iterations and the test scores after 1000 training iterations.

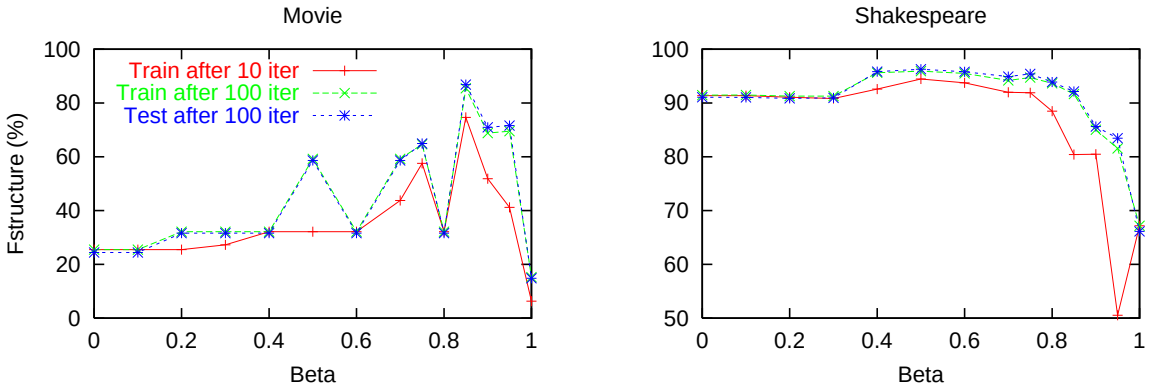


Figure 6.13: **Impact of the β parameter in tree transformation with Olpomdp and collapsed actions.** The curves correspond to the training scores after 100 and 1000 training iterations and the test scores after 1000 training iterations.

Corpus	Score	RL Sarsa	Baselines $\pi^{greedy}_{structure}$	π^{greedy}_{path}	π^{random}
Inex-Ieee	$F_{structure}$	67.5	**76.32**	49.94	2.17
	F_{path}	74.4	39.23	**97.20**	1.00
	$F_{content}$	75.8	82.91	**97.20**	8.62
Wikipedia	$F_{structure}$	**65.6**	57.37	23.53	5.51
	F_{path}	**74.3**	2.28	32.28	0.12
	$F_{content}$	**80.2**	72.92	39.34	12.35

Table 6.6: **Tree transformation with collapsed actions, large scale datasets.** For each method and each dataset, we give the three average similarity scores on the test set. We compare Sarsa with the baseline policies, which do not use learning.

The results for large-scale tree transformation are given in Table 6.6. On Wikipedia, Sarsa outperforms the scores of the greedy policy baselines, which is very satisfactory, given the large number of labels on this corpus. On Inex-Ieee, Sarsa does not reach the level of the greedy policy baselines. However, this corpus contains a huge amount of noise, which could explain this result.

6.4 Conclusion

In this chapter, we have illustrated the use of CR-algorithms for the tree transformation task. We have focused on two tasks: one-to-one tree transformation and tree transformation with unaltered text. We developed a set of experiments, which led to the following main conclusions:

• **One-to-one tree transformation** CR^{ank} supervised with the content-driven heuristic $h^{content}$ is an efficient solution to the one-to-one tree transformation problem. Its complexity is only linear *w.r.t.* the number of leaves. Previous state-of-the-art models for the one-to-one transformation tasks had a restricting cubic complexity *w.r.t.* the number of leaves.

• **Tree transformation with unaltered text** On some problems, $h^{content}$ is not very satisfying. We showed that reinforcement learning can be applied to solve such weakly supervised problems. Up to our knowledge, there are not yet other approaches than ours able to solve the tree transformation with unaltered text task on real world data.

• **Collapsed actions** We introduced a new CR-algorithm, where each action directly constructs a whole path in the predicted output. On this new problem, reinforcement learning gave very satisfying results. In particular, reinforcement-learning algorithms find better strategies than the greedy behavior and succeed in learning and generalizing these strategies.

• **Scalability and inference speed** All the proposed solutions have low complexities and good scalability properties. In practice, most documents are processed in less than one second, which, on the small datasets, is about 50 times faster than Pcfg+Me inference. The Sarsa method is the only method able to learn with all our large-scale real-world datasets.

One key feature of the CR-algorithm formalism that was not fully exploited in this chapter is its expressiveness. It is easy to conceive many extensions to CR-algorithm 9 or CR-algorithm 10, in order to deal with more advanced transformations. Furthermore, we could conceive totally different architectures for tree transformation. For example, one could think at tree transformation as the problem of finding a sequence of tree operators that maps the input tree to the target tree (*e.g.* relabel all A labels into Bs, suppress all nodes that have label C, ...). Another approach would be to adopt a *divide-and-conquer* approach, by decomposing the tree transformation problem into simpler sub-problems recursively. For example, in order to transform a whole document, start by transforming each section and then put the glue between the transformed sections. The next chapter presents some on-going work, including a CR-algorithm compiler. Such a compiler should greatly ease the development of new CR-algorithms such as those evoked here.

7

Learning for search with CR-algorithms

Contents

Heuristic search algorithms are used for a large variety of artificial intelligence problems like constraint solving, planning, diagnosis or natural language processing. General heuristics allow us to deal with a large family of problems but are limited to small problem sizes. On the opposite, domain dependent heuristics use additional information and scale much better with the problem size but are restricted to specific areas. A promising direction for developing efficient and general search methods is the *learning-for-search* approach, which aims at using machine learning to improve existing search techniques or to learn search heuristics.

Although most of this manuscript is dedicated to structured prediction, CR-algorithms may be applicable to a wider range of problems. In particular, we show in this chapter that CR-algorithms prove to be well adapted to deal with learning-for-search problems. Our core contribution is a general methodology to learn CR-algorithms for learning for search. In order to illustrate this methodology, we focus on a particular search algorithm – best-first search – and show how to learn an optimal heuristic for this algorithm. Note that the work that is presented here is still *prospective* and may be completed in several ways. In particular, the validation of our approach is still not fully satisfactory and would require experiments on much more different domains.

The chapter is structured as follows. Section 7.1 introduces the leaning-for-search problem and describes how to use the CR-algorithm framework for this problem. Section 7.2 describes

a set of experiment related to the learning problem of best-first search heuristics. Section 7.3 describes another set of experiment dealing with the approximate computation of tree-edition distances.

7.1 Learning for search

This section introduces the learning-for-search domain and our contributions related to this domain. We first overview related work in Section 7.1.1. We then describe CR-algorithms that correspond to various search problems in Section 7.1.2. We present a general approach to learn such CR-algorithms in Section 7.1.3. In particular, this leads us to an original solution to the problem of learning the optimal heuristic of a best-first search procedure. This solution is experimented in the next section.

7.1.1 Context

The learning-for-search domain has already motivated a substantial amount of work to deal with three major problems: combinatorial search, planning and continuous optimization.

Combinatorial Search

Early efforts in the 80^{es} make use of symbolic machine learning for learning rules to improve search with practice [Langley, 1983]. Several models relying on statistical machine learning have been proposed since. [Doan et Wong, 1997] describes SHAPES, an algorithm to learn a linear heuristic for best-first search. [Petrovic *et al.*, 2007] proposes to learn a linear mixture of heuristic to drive depth-first search for solving constraint satisfaction problems. Linear learning is also at the core of LASO* [Xu et Fern, 2007]. This algorithm extend the SP method LASO, which was presented in Section 3.3.3, to perform supervised learning of beam-search heuristics. Non-linear learning methods have also been proposed in the context of learning-for-search. [Hou *et al.*, 2002] explores the use of artificial neural networks (ANN) and k-nearest neighbors to learn search heuristics. [Ernandes et Gori, 2004] and [Samadi *et al.*, 2008] extend the ANN approach to bias the learning process toward approximately admissible heuristics, in order to maximize the probability to find optimal solutions quickly.

Planning

Using machine learning to improve planning methods has also been studied. [Veloso *et al.*, 1995] presents a whole architecture called PRODIGY, to integrate learning and planning. More recent work includes [Yoon *et al.*, 2006] where the author propose to learn a plan-rewriting heuristic by combining generic features of a database extracted from solved problems. This approach, which relies on linear regression, sometimes outperforms the state-of-the-part planning baseline [Hoffmann et Nebel, 2001]. Instead of using regression, it has been proposed to perform discriminative learning, for example by applying the LASO* algorithm to planning [Xu *et al.*, 2007]. This approach was shown to outperform [Hoffmann et Nebel, 2001] on all tested domains. In order to ensure better generalization guaranties, [Ratliff *et al.*, 2006b] extends previous work by modeling the supervised learning problem of a planning policy as a maximum-margin SP problem.

Continuous optimization

Machine learning has also led to interesting results for continuous optimization problems. A funding method to integrate learning with continuous optimization is STAGE [Boyan et Moore, 2001]. The main idea of this method is to learn a evaluation function that predicts the outcome of local search algorithm, given features of the current search state. This function is then used to bias future search toward better optima on the same problem. Extensive empirical results show the effectiveness of STAGE on several large-scale search domains.

XX "Tuning Local Search by Average-Reward Reinforcement Learning"

7.1.2 CR-algorithms for learning-for-search

In this part, we describe how to formalize various learning-for-search problems within the formalism of CR-algorithms. The problem of learning these CR-algorithms is discussed in Section 7.1.3. We use the following notations to denote search problems. The search space $\mathcal{N}$ is the set of search nodes the problem. A search node is denoted $\mathbf{n} \in \mathcal{N}$ and the initial search node is denoted $\mathbf{n}_0 \in \mathcal{N}$. Search nodes are arranged into a tree and we have access to the successor function $succ : \mathbf{n} \to \mathcal{P}(\mathcal{N})$, where $\mathcal{P}(\mathcal{N})$ is the power set of $\mathcal{N}$.

Best-first search (BFS) is a simple, yet widely used, search algorithm. It usually relies on a domain-specific user-defined heuristic $h : \mathcal{N} \to \mathbb{R}$ that defines an order over search nodes. Given this heuristic, BFS works by visiting the most promising nodes first. The availability of a good heuristic function is the key condition to have an efficient BFS procedure. Automatically learning such a prioritizing function can be done within the framework of CR-algorithms. *Best-first search*

CR-algorithm 12 Simple best-first search CR-algorithm

Input: A search problem $(\mathbf{n}_0, \mathcal{N}, succ)$
Input: A time limit T_{max}
Output: A solution or *no solutions*

```
1: openedNodes ← ∅
2: closedNodes ← {n_0}
3: while card(openedNodes)< T_max do
4:     n ← choose closedNode                          ▷ Choose
5:     if isSolution(n) then
6:         return s
7:     end if
8:     closedNodes ← (closedNodes \ {n}) ∪ succ(n)
9:     openedNodes ← openedNodes ∪ {n}
10:    reward −1                                      ▷ Reward: cost of one step
11: end while
12: return no solutions
```

CR-algorithm 12 describes the learning problem of BFS with a *binary* search problem: nodes $\mathbf{n}$ are either solutions or non-solutions. BFS relies on a set of closed nodes and a set of opened nodes. At each search step, the *choose* instruction (line 4) selects one node to explore among the set of closed nodes. If this node is a solution, search stops and return the solution (line 6). Otherwise, the node becomes an opened node and its successors are added to the set of closed nodes (lines 8–9). Each search step has a cost of 1 (line 10), *i.e.* the aim of learning is to minimize the number of search steps required to find a solution.

Policies of CR-algorithm 12 are functions that choose which candidates to explore during search. Any search algorithm that proceeds by iteratively exploring candidates in a tree-structured search space can be seen as such a policy. For example, random search is the policy that selects actions randomly and breadth-first search is the policy that selects actions corresponding to search nodes at lowest depth. The classical BFS procedure, which rely on the user-defined heuristic $h : \mathcal{N} \to \mathbb{R}$, can also be seen as a policy of CR-algorithm 12 defined in the following way: *Policies of CR-algorithm 12*

$$\pi_h(\mathbf{s}) = \underset{\mathbf{a} \in \mathcal{A}_\mathbf{s}}{\operatorname{argmax}}\, h(\mathbf{a})$$

i.e. at each step this policy selects the closed node that maximizes the heuristic function.

CR-algorithm 13 Multiple-solutions best-first search CR-algorithm

Input: A search problem $(\mathbf{n}_0, \mathcal{N}, succ)$
Input: A time limit T_{max}
Output: *A set of solutions*

```
1:  solutions ← ∅
2:  openedStates ← ∅
3:  closedStates ← {n_0}
4:  while card(openedStates)< T_max do
5:      n ← choose closedStates                                  ▷ Choose
6:      if isSolution(n) then
7:          reward qualityOfSolution(n)                          ▷ Reward
8:          solutions ← solutions ∪ n
9:      end if
10:     closedStates ← (closedState \ {n}) ∪ succ(n)
11:     openedStates ← openedStates ∪ {n}
12:     reward −1
13: end while
14: return solutions
```

Extensions

The previous CR-algorithm was restricted to finding a unique solution in a binary search problem. These restrictions can easily be removed. As an example, CR-algorithm 13 performs search with real-scored solutions and multiple-solutions. The differences with the previous CR-algorithm are shown in blue italic text. We consider the following search-optimality criterion:

$$R = \sum_{\mathbf{n} \in solutions} \text{qualityOfSolution}(\mathbf{n}) \; - \; \text{number of search steps}$$

i.e. the aim is to find a maximum of good quality solutions in a minimum of time. Note that this only one example of possible optimality criterions. Depending on the application, several other optimality criterions may be of interest.

Depth-first search

Another popular form of search is *depth-first search*. Similarly to BFS, this search algorithm can formalized with CR-algorithms. CR-algorithm 14 describes depth-first search for a binary search problem. The algorithm relies on a stack of search nodes that makes it possible to implement backtracking. Furthermore, it uses a set of marked nodes to keep a trace of the already explored nodes. Prioritization of the search space relies on the *choose* instruction of line 15. The aim here is to choose a successor node of $\mathbf{n}$ that has not been explored yet ($\mathbf{n}' \notin$ markedNodes). If no such successor exists, backtracking is performed in line 11. If the whole search space has been explored without finding any solutions, the algorithm stops (line 8). When a node $\mathbf{n}_{succ}$ has been chosen, we test if it is a solution (line 16). If not, we store the current node in the stack and continue with $\mathbf{n}_{succ}$ (lines 19–20).

Extensions

Depending on the application specificities, the depth-first search CR-algorithm may be modified in several ways. Support for real-valued solutions or multiple solutions may be taken into account, such as in CR-algorithm 13. Another particularly extension is the *learning-to-backtrack* problem. Instead of performing one-step backtracks, search may be fastened thanks to multiple-steps backtracking. This raises a new learning problem, which is to choose the number of

CR-algorithm 14 Depth-first search CR-algorithm

Input: A search problem $(\mathbf{n}_0, \mathcal{N}, succ)$
Input: A time limit T_{max}
Output: A solution or *no solutions*

```
1: n ← n_0
2: stack ← ∅
3: markedNodes ← ∅
4: for t = 0 to T_max do
5:     candidates ← {n' ∈ succ(n), n' ∉ markedNodes}
6:     if candidates = ∅ then
7:         if stack = ∅ then
8:             break                                  ▷ Exhaustive search failed
9:         else
10:            markedNodes ← markedNodes ∪ {n}         ▷ n has been fully explored
11:            n = stack.pop()                        ▷ Backtrack
12:            continue
13:        end if
14:    else
15:        n_succ ← choose candidates                 ▷ Choose
16:        if isSolution(n_succ) then
17:            return n_succ                          ▷ Search success
18:        else
19:            stack.push(n)                          ▷ Go forward
20:            n ← n_succ
21:        end if
22:        reward −1
23:    end if
24: end for
25: return no solutions
```

backtracking steps that should be performed. This problem can be expressed thanks to a *choose* instruction: when backtracking, *choose* one node to continue with, among those in the current stack.

Other search algorithms

We described CR-algorithms for best-first search and depth-first search. We believe that many other approaches to combinatorial search may be written as CR-algorithms. Here are some example approaches that could be revisited within the framework of CR-algorithms :

• **Bi-directional search** Bi-directional search runs two simultaneous searches: one forward from the initial state, and one backward from the goal and stops when the two meet in the middle. A bi-directional search CR-algorithm could for example be centered on a *choose* that selects which side of the search is the most promising. Another way to tackle bi-directional search, would be to directly choose among the union of the forward-search successors and the backward-search successors at each search step.

• **Divide-and-conquer search** Many search problems can be solved by breaking them recursively into simpler search sub-problems, until these become simple enough to be solved directly. In this context, *choose* instructions could be used to choose among different kind of problem decompositions. A *choose* instruction could also be used to select among different resolution methods for the sub-problems.

• **Branch and bound search** Branch and bound is a search algorithm for finding optimal solutions. It consists of a systematic enumeration of all search nodes, where large subsets of nodes are discarded by using upper and lower estimated bounds of the quantity being optimized. Depending on the order in which the search space is explored, this bounding mechanism may be more or less efficient. A *choose* could be used here to prioritize the search space, in order to optimize the quality of the bounding mechanism.

7.1.3 Learning and supervision

In the previous part, we have introduced several CR-algorithm for learning for search. We now discuss the key problem of learning these CR-algorithms. As discussed in the previous chapters of this manuscript, we mainly have the choice between two approaches: supervised learning and reinforcement learning. The former assumes the knowledge of optimal learning trajectories (OLTs) or of the optimal learning policy (OLP), while the latter only makes use of the information conveyed by rewards. In theory, reinforcement learning is applicable to all the previous CR-algorithms. However, in practice, it is often desirable to inject supervision knowledge to improve or to speed-up learning. We propose here a general methodology to inject such supervision knowledge.

Post-episode supervision

Our key idea to learn search CR-algorithms is *post-episode supervision*: each time that a solution has been found, we can supervise the whole sequence of actions that led to this solution. For example, in CR-algorithm 12, if we know that $\mathbf{n}^* \in \mathcal{N}$ is a solution, we know that the ancestors of $\mathbf{n}^*$ are better choices than the other nodes of the search space. This knowledge can then be injected in a supervised learning such as CR^{ank} with a heuristic action-cost.

CR^{ank} with post-supervision

Since the action costs must be available immediately in CR^{ank}, the learning algorithm has to be modified to support post-episode supervision. The modified algorithm is called CR^{ank}_{post}, and it repeats the following steps:

1. Sample a new initial state of the CR-algorithm.

2. Perform an episode by storing the trace of the visited state sequence $(\mathbf{s}_1, \ldots, \mathbf{s}_t)$.
3. If solutions were found:
   (a) Given the structure of the search algorithm, extract supervision information from their knowledge.
   (b) Convert this supervision information into a heuristic action-cost function applicable to the visited states $(\mathbf{s}_1, \ldots, \mathbf{s}_t)$.
   (c) Given the action-cost function and the visited state sequence, computes the episode gradient of CR^{ank} and update the parameters θ.

The policy used to perform episodes in step (2) may either be an initial heuristic to solve the search problem, or the currently learned policy π_θ. The only requirement of the CR^{ank}_{post} approach is that some solutions should be found from times to times in order to initiate learning in step (3). The steps (3.a) and (3.b) are dependent from the target CR-algorithm.

In the remaining of this part, we instantiate the post-supervision approach on the BFS CR-algorithm. We first discuss supervision and then discuss feature functions for this CR-algorithm.

Best-first search supervision

In best-first search, the knowledge of a solution $\mathbf{n}^*$ gives the information that all the ancestors of $\mathbf{n}^*$ are good nodes to explore. Thus, actions that do not open ancestors of $\mathbf{n}^*$ should have a higher cost than those that do. A simple way to incorporate this information in the action-cost function $c_{\mathbf{n}^*}$ is the following one:

$$c_{\mathbf{n}^*}(\mathbf{s}, \mathbf{a}) = \begin{cases} 0 & \text{if the node } \mathbf{a} \text{ is an ancestor of } \mathbf{n}^* \\ 1 & \text{otherwise} \end{cases}$$

i.e. each action that is not necessary to complete search has a cost of 1. A key property of $c_{\mathbf{n}^*}$ action costs is that they are tightly related to the regrets $c^*(\mathbf{s}, \mathbf{a})$ of the actions *w.r.t.* optimal policies. Most of the time, both functions are equivalent: $c_{\mathbf{n}}(\mathbf{s}, \mathbf{a}) = c^*(\mathbf{s}, \mathbf{a})$. Approximation occurs in the following cases:

1. If there are multiple solutions, $c_{\mathbf{n}^*}$ may assign a cost of 1 to a node $\mathbf{n}'$ leading to another solution than $\mathbf{n}^*$.
2. If multiple paths connect $\mathbf{n}_0$ to $\mathbf{n}^*$, the optimal behavior is to open only nodes that belong to shortest paths to the solution. This information is not taken into account by $c_{\mathbf{n}^*}$ that may assign a cost of 0 to search nodes that do not belong to the shortest paths to the solution.

Although our action-costs are not exactly equal to the regrets *w.r.t.* optimal policies, these approximates seems to be reasonable in the experiments performed on BFS in Section 7.2.

Feature function

In order to learn CR-algorithm 12, we have to provide an action feature function $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$. Here, states are composed of the *openedNodes* and *closedNodes* sets and actions are nodes of the search space. Two kinds of feature functions are possible for best-first search:

• **State-action description** The ϕ function may incorporate any feature related to the set of opened nodes, the set of closed nodes and a candidate node. This makes it possible to take into account some general statistics of the search process, such as search progression or statistics of the set of opened nodes.

• **State-independent description** A *state-independent* ϕ_{si} function is a feature function that only depends on the current action: $\phi(\mathbf{s}, \mathbf{a}) = \phi_{si}(\mathbf{a})$. In the context of BFS, where actions are search nodes, a state-independent feature function is a function from nodes to features: $\phi : \mathcal{N} \rightarrow \mathbb{R}^d$. Given a node description function, we can thus automatically induce a state-independent feature function.

Optimal heuristic for BFS

An optimal BFS heuristic $h : \mathcal{N} \rightarrow \mathbb{R}$ is a heuristic that minimizes the expected number of required steps to find a solution. A key property of CR-algorithm 12 is that it can be used to learn optimal heuristics through the use of state-independent descriptions. Indeed, thanks to this property, the policies π_θ learned by CR^{ank} are also state-independent:

$$\pi_\theta(\mathbf{s}) = \underset{\mathbf{a} \in \mathcal{A}}{\text{argmax}} \langle \phi_{si}(\mathbf{a}), \theta \rangle$$

Since the policy only depends on a node $\mathbf{n}$, the key idea is that the scoring function $\langle \phi_{si}(\mathbf{a}), \theta \rangle$ can be re-interpreted as a classical BFS heuristic $h_\theta : \mathcal{N} \rightarrow \mathbb{R}$, defined in the following way:

$$h_\theta(\mathbf{n}) = \langle \phi_{si}(\mathbf{n}), \theta \rangle$$

After learning, the function h_θ can thus be returned to the user to perform classical BFS search. Furthermore, state-independance makes it is possible to implement inference in CR-algorithm 12 exactly in the same way as in classical BFS (with a queue of nodes sorted by their heuristic score), which leads to *choose*s with a logarithmic complexity.

In summary, given a node description function $\phi : \mathcal{N} \rightarrow \mathbb{R}^d$, CR^{ank}_{post} provides a solution the learn the heuristic minimizing the expected number of search steps required to find a solution. The key idea of CR^{ank}_{post} is post-supervision: once we know a solution, we can re-visit all the previous search steps and update a ranking functions to prefer ancestors of the solution over all other nodes of the search space. Although we only detailed learning in CR-algorithm 12, we believe that the CR^{ank}_{post} approach generalizes relatively well to other search problems. This is a major direction of future work.

7.2 Experiments with best-first search

We describe here a set of preliminary experiments that show the promise of the learning approach described in the previous section. We illustrate CR^{ank}_{post} on a little but complex search problem: LCeB. This problem is first introduced in Section 7.2.1. We then describe a feature function to describe LCeB search nodes in Section 7.2.2. Finally, we give a set of experimental results in Section 7.2.3.

7.2.1 The LCeB problem

The LCeB (Le Compte est Bon) problem comes from the French game show *Des chiffres et des lettres* (Numbers and Letters), created by Armand Jammot. Over 4,000 episodes of this game have been presented on the British game show *Countdown*, which makes it one of the longest-running game shows in the world[1].

LCeB rules

The rules of LCeB are the following. Six numbers are randomly sampled from the set $\{1, 1, 2, 2, 3, 3, \ldots, 10, 10, 25, 50, 75, 100\}$ and a target number is randomly sampled in $[100, 999]$. The goal is to combine the six numbers by using *additions*, *subtractions*, *multiplications* and

[1] from *wikipedia*, Countdown (game show).

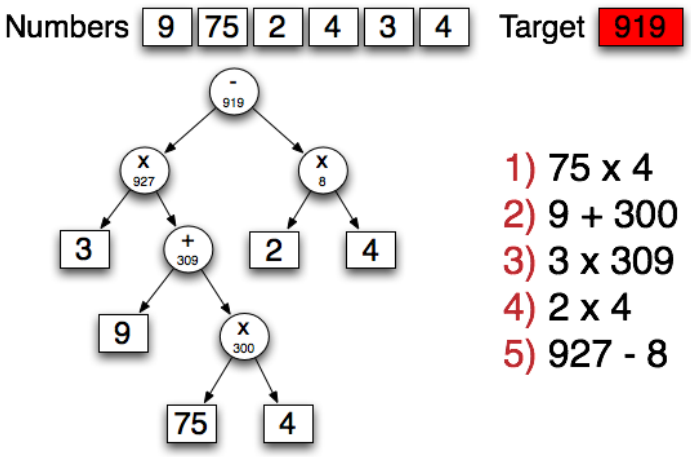


Figure 7.1: **An instance of the LCeB search problem.** Top: A LCeB game composed of six numbers and a target. Bottom: One of the solutions of the game and the sequence of computations to construct this solution.

Mean size of the search space	$\simeq 10^6$
Mean number of solutions	$\simeq 370$
Median number of solutions	17
Percentage of solvable games	94%

Branching factor at depth 1	$\simeq 42,8$
Branching factor at depth 2	$\simeq 29,8$
Branching factor at depth 3	$\simeq 18,7$
Branching factor at depth 4	$\simeq 9,7$
Branching factor at depth 5	$\simeq 3,3$

Table 7.1: **Statistics of the LCeB search space.** Left: statistics on the search space and on the solutions, Right: branching factors – the average number of successors depending the depth of a search node.

divisions in order to produce the target number[2]. Numbers can be used as many times as they appear in the selection, and need not all be used. Only integers may be used at any stage of the computation. Figure 7.1 gives an example game with one solution.

LCeB *search space*

Each LCeB search node $\mathbf{n} \in \mathcal{N}$ contains a set of numbers and nodes are solutions of the problem if one of their numbers is equal to the target. The initial search node contains the six game numbers. The successors of a node are all the nodes that can be reached by replacing two numbers by their combination using an elementary operation ($+, -, \times$ and $/$).

We apply the following simple constraints to the successor function. *Rule 1*: Since decimals and fractions are forbidden in LCeB, an operation can only be performed if its result is a strictly positive integer. *Constraint 1*: A number that appears multiple times is only considered once. *Constraint 2*: We remove symmetries in additions $a + b$ and multiplications $a \times b$ by only considering cases where $a >= b$. These choices lead to the statistics[3] given in table 7.1. Note that only 94% of the possible search problems have a solution and no method is able to do better than 94% of resolved games.

Defining search-space thanks to CR-algorithms

An interesting aspect of the CR-algorithm formalism is that, since search spaces are special

[2]We only consider the binary search case; in the traditional rules the aim is to get as near as possible to the target.

[3]More statistics about the LCeB game can be found on http://www.crosswordtools.com/numbers-game/.

CR-algorithm 15 CR-algorithm defining LCeB search space

Input: The six input numbers $n_1, \ldots n_6$
Input: The target number *target*
Output: A candidate solution

```
1: expressions ← {leaf(n_i), i ∈ [1,6]}
2: while card(expressions) > 1 do
3:     choices ← ∅
4:     markedValuePairs ← ∅
5:     for each exprLeft ∈ expressions do
6:         for each exprRight ∈ expressions, exprRight ≠ exprLeft do
7:             valueLeft ← compute(exprLeft)
8:             valueRight ← compute(exprRight)
9:             if (valueLeft, valueRight) ∈ markedValuePairs then
10:                continue                                   ▷ Constraint 1
11:            else
12:                markedValuePairs ← markedValuePairs ∪ {(valueLeft, valueRight)}
13:            end if
14:            if valueLeft ≥ valueRight then                 ▷ Constraint 2
15:                choices.insert( (plus, exprLeft, exprRight) )
16:                choices.insert( (times, exprLeft, exprRight) )
17:                if valueLeft > valueRight then             ▷ Rule 1
18:                    choices.insert( (minus, exprLeft, exprRight) )
19:                end if
20:                if valueLeft is a multiple of valueRight then   ▷ Rule 1
21:                    choices.insert( (divide, exprLeft, exprRight) )
22:                end if
23:            end if
24:        end for
25:    end for
26:    (operation, left, right) ← choose[expressions] choices
27:    expressions ← (expressions \ {left, right}) ∪ {operation(left, right)}
28:    for each expr ∈ expressions do
29:        if compute(expr) = target then
30:            reward +1                                      ▷ A solution was found
31:            return
32:        end if
33:    end for
34: end while
```

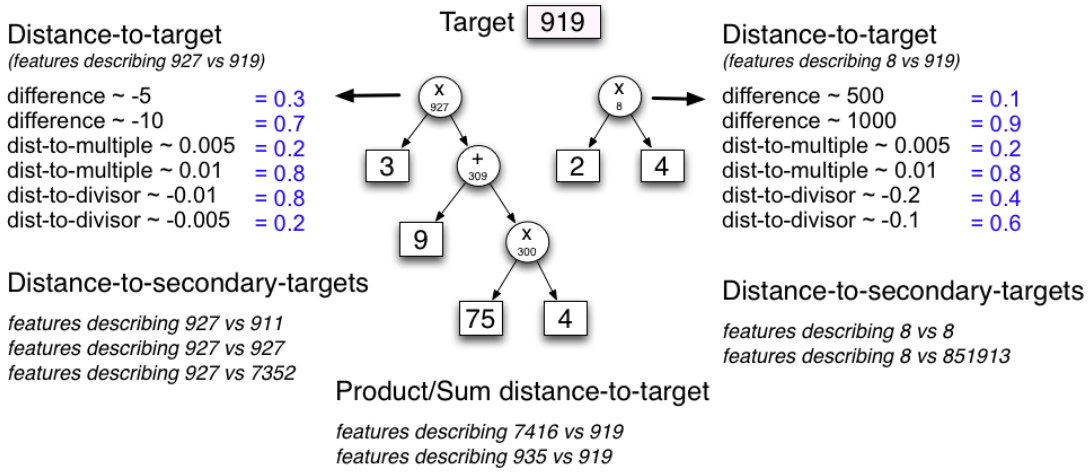


Figure 7.2: **Feature function for the LCeB search problem.** This figure illustrates the feature function $\phi : \mathcal{N} \rightarrow \mathbb{R}^d$ that we used to describe LCeB search nodes. The center part of the figure illustrates the current search node, which is composed of two expressions resulting to 927 and 8. Here, the aim is build the target number 919. The left and right parts of the figure respectively give the features for the first and second expression. The bottom part of the figure gives product and sum features. Only active features are shown (*i.e.* features that have a non-null value). The blue numbers correspond to (approximate) feature values.

cases of MDPs, a search space can be defined formally with a CR-algorithm. In such a CR-algorithm, states are interpreted as search nodes, the initial search node corresponds to the initial state and the successor function is defined in the following way:

$$succ(\mathbf{n}) = \{T(\mathbf{s}, \mathbf{a}), \mathbf{a} \in \mathcal{A}_\mathbf{s}\} \text{ with } \mathbf{s} = \mathbf{n}$$

The *isSolution* function can be defined through rewards, *e.g.* a reward of +1 means that the current state is a solution. As an example of this use of CR-algorithms, CR-algorithm 15 defines the search space of LCeB, accordingly to the description above. Each state of this CR-algorithm is defined by a set of tree-structured expressions that are either input numbers or binary operations between two expressions (additions, multiplications, divisions or subtractions). Each decision step of LCeB involves the combination of two of the current expressions (lines 26–27). Final search state are either reached when one of the current expressions equals the target (lines 29–32) or when there is single remaining expression (condition line 2).

7.2.2 Feature function

In order to apply CR^{ank}_{post} to learn a BFS heuristic, we have to provide a node feature function $\phi : \mathbf{n} \rightarrow \mathbb{R}^d$. We use three kinds of features: *distance-to-target* features, *distance-to-secondary-targets* features and *product/sum distance-to-target* features. These features are illustrated in Figure 7.2 and described below:

• **Distance-to-target** These features focus on the relation between the value of an expression and the target. Let e be the value of an expression, t the target number and [.] the operator that

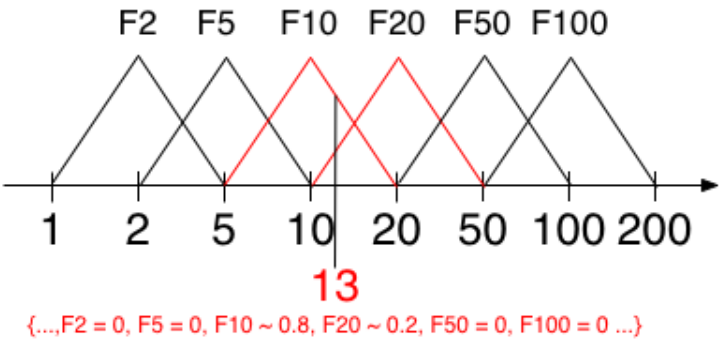


Figure 7.3: **Number features in the LCeB problem.** This figure illustrates number features. There is one feature per reference number (1, 2, 5, ...). For a given number (*e.g.* 13), only two features are non-null (*e.g.* $F_{10} \simeq 0.8$ and $F_{20} \simeq 0.2$).

returns the nearest integer of a real number. In order to compute distance-to-target features, we first compute the three following numbers for each current expression:

- The difference $t - e$. In our example, the difference values are $919 - 927 = -8$ and $919 - 8 = 911$.
- The distance-to-multiple $\frac{e}{t} - \left[\frac{e}{t}\right]$. In our example, the distance-to-multiple values are: $927/919 - [927/919] \simeq 8,705.10^{-3}$ and $8/919 - [8/919] \simeq 8,705.10^{-3}$.
- The distance-to-divisor $\frac{t}{e} - \left[\frac{t}{e}\right]$. In our example, the distance-to-divisor values are : $919/927 - [919/927] \simeq -8,63.10^{-3}$ and $919/8 - [919/8] = -0.125$.

Once the difference, distance-to-multiple and distance-to-divisor values are computed, we describe them through *number features*. Number feature are illustrated in Figure 7.3 and measure the proximity between one of the previous numbers and a reference number among the set $\{\ldots, 1.10^{-3}, 2.10^{-3}, 5.10^{-3}, 1.10^{-2}, 2.10^{-2}, 5.10^{-2}, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, \ldots\}$. Intuitively, the kind of features that are generated measure if differences are rather small, medium, large and so forth. This kind of representations for number is also motivated by the fact that we mainly deal with linear learning. Indeed, a linear function of number features can be a non-linear function of the underlying numbers.

• **Distance-to-secondary-targets** These features focus on the relation between one of the expressions and a *secondary target*. Given the value of an expression e and the target t, the secondary targets are $t + e$, $t - e$ (if $t > e$), e/t (if e is a multiple of t) and t/e (if t is a multiple of e). For each of these secondary targets, we compute the previous *distance-to-target features*.

• **Product/Sum distance-to-target** These features describe the relation between the product or the sum of all expressions and the target number. In our example, the product of the expressions is $927 \times 8 = 7416$ and their sum is: $927 + 8 = 935$.

A good heuristic for LCeB may depend on the number of computations that have already been performed in the current search node, *i.e.* the depth of the search node. In order to incorporate this information, we clone the feature described above and represented in Figure 7.3

to form depth-specific features. For example, given a feature $f^{distance-to-target}_{difference\simeq 5}(\mathbf{n}) = 0.3$, at a depth of 4, we add the following feature:

$$f^{distance-to-target}_{depth=4 \wedge difference\simeq 5}(\mathbf{n}) = 0.3$$

This cloning operation doubles the number of active features per search node and multiply the number of possible features by the maximum depth. As in all our experiments with CR-algorithms, the feature function of LCeB is sparse and high dimensional (up to 8,000 distinct features). Furthermore, the set of active features is generated directly and efficiently from the data, without enumerating the whole set of possible features.

7.2.3 Experiments

Learning-based models

We have compared various approaches to solve LCeB. CR^{ank}_{post} is the supervised approach, described in Section 7.1.3, to learn CR-algorithm 12. We compare this supervised approach with a pure reinforcement learning algorithm, namely Monte-Carlo Control [Sutton et Barto, 1998] (Mcc), applied to CR-algorithm 12. In order to evaluate the interest of modeling the BFS procedure explicitly, we also launched the Monte-Carlo Control algorithm directly on CR-algorithm 15 (Mcc-direct). We compare these methods with the recently proposed learning-for-search algorithm LaSO*.

Non-learning based models

In order to evaluate the benefit of learning, we also performed experiments with two non-learning baselines: Random and Greedy. The former corresponds to a player that chooses actions randomly while the latter corresponds to a player that chooses actions that minimize the distance to the target greedily. Formally, Greedy is a BFS procedure with the following heuristic:

$$h(\mathbf{n}) = \min_{expr \in \mathbf{n}} |expr - target|$$

where *expr* are the expressions of the current search node and *target* is the target number.

Tuning

The learning-based methods were trained on 10,000 randomly generated games. All the methods have been tested on 10,000 other generated games. CR^{ank}_{post} was configured as CR^{ank} in the previous chapters. We tried two ranking losses, most-violated-pair (*mvp*) and all-pairs (*ap*), combined with the large-margin criterion. The discount factor in Mcc and Mcc-direct was tuned manually and equal to 0.9 in both cases. The parameters of LaSO* have also been tuned manually in order to obtain the best performance. This tuning process led to a beam size of 20.

Results

The results of our experiments are given in Figure 7.4. The performance measure used here is the percentage of games that are solved with less than a given number of explored nodes. For example CR^{ank}_{post} is able to solve more than 85% of the games exploring less than 100 nodes while Mcc-direct only solves about 50% of the games and LaSO* about 30% in the same time. The results of CR^{ank}_{post} are quite satisfying: all the solvable games are solved in a maximum of 10,000 nodes, which correspond to an exploration of only 1% of the total search space. The other following conclusions can be drawn from the results:

- The *all-pairs* ranking loss seems to be more adapted here than the *most-violated-pair* ranking loss.
- The supervised approach CR^{ank} clearly outperforms the value-based reinforcement-learning algorithm Mcc.
- The interest of explicitly modeling the BFS procedure appears clearly by comparing Mcc and Mcc-direct. The main difference between these two methods concerns the number

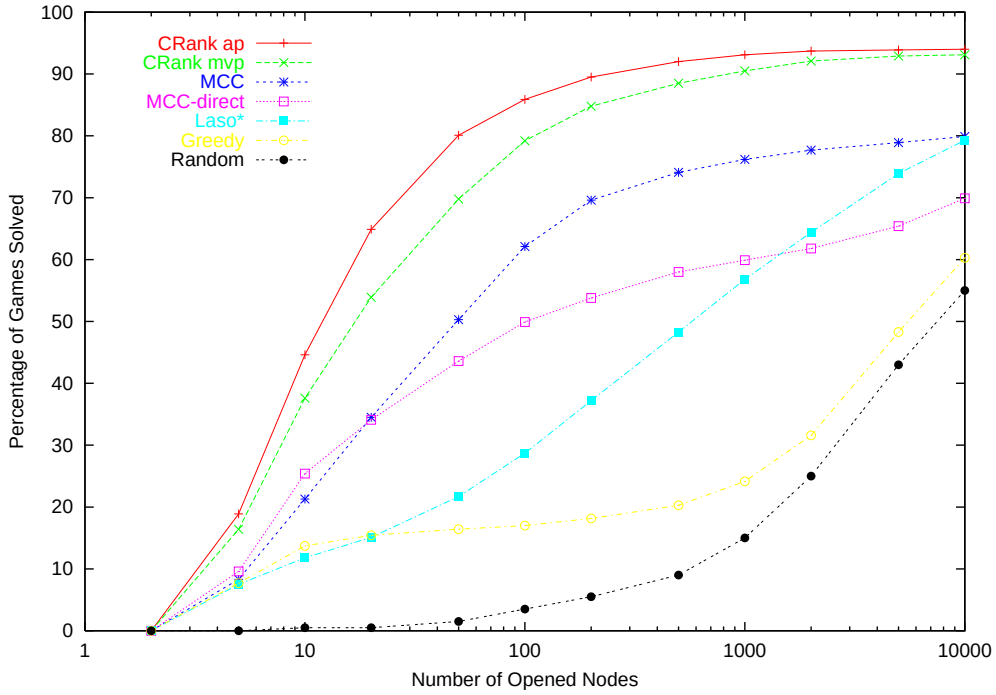


Figure 7.4: **Comparison of various search methods for LCeB.** These curves show the percentages of LCeB games solved exploring less than a certain amount of nodes. Methods from top to bottom: CR^{ank}_{post} with the all-pairs ranking loss, CR^{ank}_{post} with the most-violated-pair loss, Monte-Carlo Control on CR-algorithm 12, Monte-Carlo Control on CR-algorithm 15, LaSO*, greedy BFS and random search.

LCeB game		Humans	BFS CR-algorithm	
Numbers	Target		Mcc	CR^{ank}_{post}
5 3 6 4 100 2	659	10, 10, 28, **5**, **5**, >12, 15	69	57
10 7 8 75 3 9	584	7, 6, 10, 4, 4, 4, **3**, 6	**3**	**3**
8 9 10 3 6 7	864	24, 11, 12, 7, 33, >25, 25	16	25
4 8 7 100 6 9	844	12 ,6, 6, **3**, 5, >11, 4, **3**, 9	8	6
8 2 9 4 4 6	342	18 , 5, 25, >10, 10, >5, >33, 7	4	**3**

Table 7.2: **Comparison of machine-learning and human players on LCeB.** We give the number of explored nodes by both human players and learning-based algorithms on 5 randomly selected games. $> n$ denotes human players that stopped after n steps without finding a solution.

of search nodes explored per episode. On one side, Mcc-direct draws a single trajectory in the search space per episode and learns to maximize the probability that this unique trajectory leads to a solution. On the other side, Mcc can open any number of search nodes before finding the solution and it directly learns to minimize the expected number of required search steps to find a solution.

- LaSO* relies on a beam-search procedure and we used a beam size of 20. Thus, it requires opening at least $20 \times d$ search nodes to find a solution at depth d. This mainly explains the relative low performance of this method.
- All learning-based methods clearly out-perform the non-learning baselines on this problem.

Comparison to human players

We performed a comparison of our learning-based approach against 7 human players that are members of our research team. The results are given in Table 7.2. We gave the following instructions to human players:

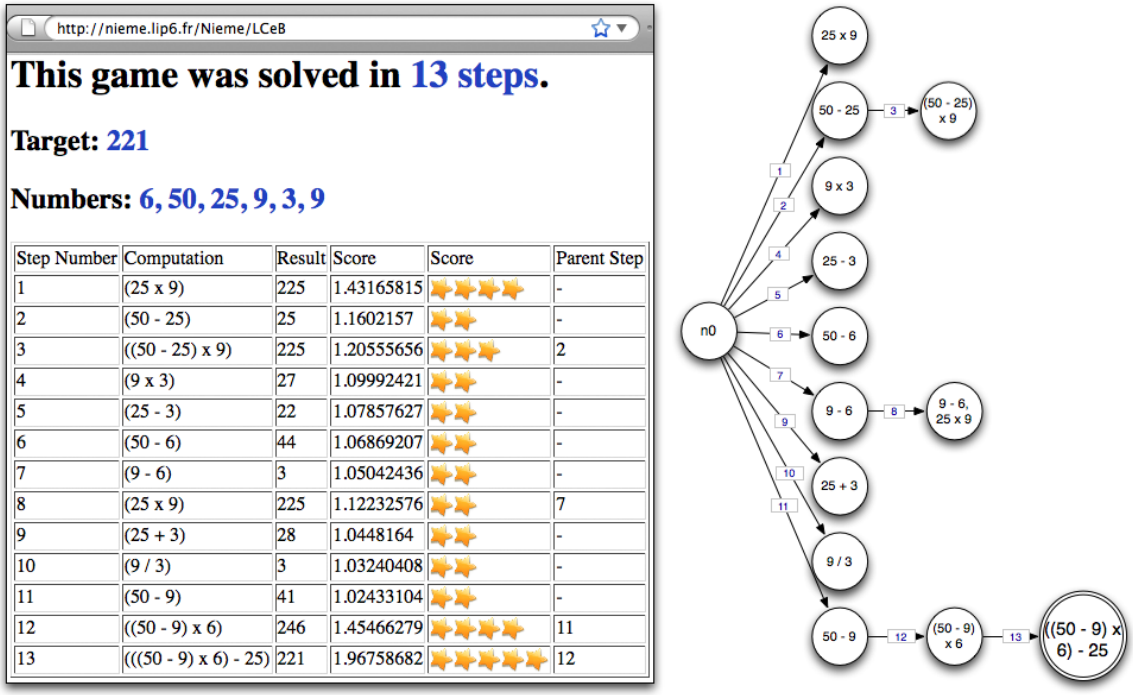
- On a paper, write one mark per performed computation,
- Count trivial computations that may have been done unconsciously, *e.g.* 6×100 in the first example of Table 7.2.
- Do not count target adjustment computations, *e.g.* "if I do 6×100, the new problem is to construct 59 with the remaining numbers". These computations are performed by the feature function ϕ, so they should not be counted for human players.

If we consider that a human player is an expert for solving this type of problem, our method behaves very well on most instances of the problem. Interestingly, the first game, where the learning-based approach is less competitive, corresponds to a less intuitive solution. Most humans start by computing 6×100 in this game and then try to construct 59 with the remaining numbers. It can then be quickly seen that this strategy does not work, which causes most human to switch on completely different solutions. Our approach to solve LCeB is state-invariant: the set of previously explored nodes does not appear in the features, which makes the kind of reasoning human players use impossible to represent. On the four other problems, learning-based approaches are competitive to human players[4].

Try it: Web Applet

The policy learned with CR^{ank}_{post} can be tested thanks to an applet, which is available at

[4]Naturally, the results concerning human players are not precise, since some computations may have been done unconsciously. However, we believe that they are still good indicators to show the competitiveness of our approach.



Step Number	Computation	Result	Score	Score	Parent Step
1	(25 x 9)	225	1.43165815	★★★★	-
2	(50 - 25)	25	1.1602157	★★	-
3	((50 - 25) x 9)	225	1.20555656	★★★	2
4	(9 x 3)	27	1.09992421	★★	-
5	(25 - 3)	22	1.07857627	★★	-
6	(50 - 6)	44	1.06869207	★★	-
7	(9 - 6)	3	1.05042436	★★	-
8	(25 x 9)	225	1.12232576	★★	7
9	(25 + 3)	28	1.0448164	★★	-
10	(9 / 3)	3	1.03240408	★★	-
11	(50 - 9)	41	1.02433104	★★	-
12	((50 - 9) x 6)	246	1.45466279	★★★★	11
13	(((50 - 9) x 6) - 25)	221	1.96758682	★★★★★	12

Figure 7.5: **Applet demonstrating our approach for learning BFS heuristics.** Left: an example BFS trajectory computed by the applet available at `http://nieme.lip6.fr/Nieme/LCeB`. Right: The part of the search space that was explored to solve the game. The double-circled node is the solution that was found.

`http://nieme.lip6.fr/Nieme/LCeB`. This applet is illustrated in Figure 7.5. Its use is simple: the user selects the six input numbers and the target number and then the applet displays the search trajectory that was performed to solve the game. For each decision step, it displays the step number, the computation that correspond to the explored node $\mathbf{n}$, the result of this computation, the heuristic score $h_\theta(\mathbf{n})$ and the parent node of $\mathbf{n}$.

7.3 Experiments with tree-edition distances

This section presents a five-months prospective work that was done in our team by Stéphane Peters during its internship. This work deals with the problem of computing edition distance between labeled trees. The tree edition distance between two trees T_A and T_B is the minimum number of nodes deleting, inserting and relabeling operations required to transform T_A into T_B. This problem has a wide range of applications in domains ranging from computational biology or structured text databases to compiler optimization, see [Bille, 2002] for an overview.

The tree-edition distance problem has been stated formally thirty years ago [Selkow, 1977, Tai, 1979] and various solutions based on dynamic programming have been proposed since [Zhang et Shasha, 1989, Klein, 1998]. Today, the best algorithms to compute tree edition distances have a complexity worst than $\mathcal{O}(n^3)$ where n is the number of nodes of the trees. This complexity is prohibitive for many applications, in particular those dealing with large trees containing thousands or millions nodes. This motivates the development of methods to compute approximate tree-edition distances with lower complexities.

7.3.1 Tree-edition CR-algorithm

The tree-edition distance problem can be formalized within the framework of CR-algorithms through a tree-edition CR-algorithm. The central idea is to learn the policy for transforming trees T_A into T_B thanks to the basic deleting, inserting and relabeling operations. Each operation is penalized by a negative reward corresponding to its cost, so that the total reward of an episode corresponds to a candidate distance between T_A and T_B. The tree-edition distance between T_A and T_B corresponds to the minimum of these candidate distances.

CR-algorithm 16 defines the tree-edition problem and works in the following way. At each time-step, we have access to the current tree T_A and the target tree T_B. While these two trees are different (line 2), we modify T_A using a basic operation (lines 3–14). We then count the cost of this operation (line 15) and give a negative reward accordingly (line 16). There are three basic operations: `delete` removes a node from T_A (line 5), `relabel` changes a label in T_A (line 7) and `create` creates a new leaf node inside T_A (line 9). As in our experiments with tree transformation (see Chapter 6), the basic operations are parameterized by the node and eventually by a label and a children position. At a given time-step, the number of basic operations is in $\mathcal{O}(n.l.c)$ where n is the number of nodes of T_A, l is the number of distinct labels in T_B and c is the maximum number of children in nodes from T_A. The complexity of inference with CR-algorithm 16 is thus in $\mathcal{O}(n.l.c.D_{max})$ where D_{max} is the maximum tree edition distance. This is quite satisfactory since the complexity is close to be linear *w.r.t.* the number of nodes of the tree.

An optimal policy π^* in CR-algorithm 16 is a policy that minimizes the total cost to transform T_A into T_B. This total cost is equal to the tree-edition distance between T_A and T_B. The values returned by the tree-edition CR-algorithm with an optimal policy are thus the exact edition distances. In practice, learning, due to the use of approximation, will not reach the optimal policy. Executing CR-algorithm 16 with a near-optimal policy returns approximated edition distances, which are over estimates of the exact edition distance. The quality of the approximation

CR-algorithm 16 Approximate tree-edition distance CR-algorithm

Input: A source tree T_A
Input: A target tree T_B
Input: A maximum distance D_{max}
Output: An approximate tree-edition distance

```
1: distance ← 0
2: while T_A ≠ T_B ∧ distance < D_max do
3:     choices ← ∅                                              ▷ Create choices
4:     for each node ∈ T_A do
5:         choices.insert((delete, node))
6:         for label ∈ labelsOfTree(T_B) do
7:             choices.insert((relabel, node, label))
8:             for each position ∈ node do
9:                 choices.insert((create, node, label, position))
10:            end for
11:        end for
12:    end for
13:    choice ← choose[T_A, T_B] choices                        ▷ Choose
14:    modify T_A with respect to choice
15:    distance ← distance + 1
16:    reward -1                                                ▷ Cost of one operation
17: end while
18: return distance
```

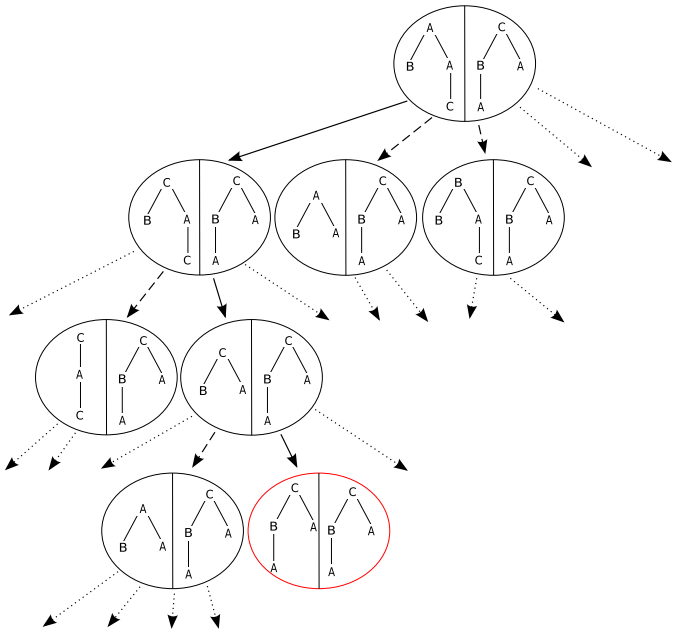


Figure 7.6: **An example MDP induced by CR-algorithm 16.** Circles are states and arrows are transitions. The circle with red edges is a final state. Each state contains a current tree (left) and the target tree (right).

depends on the quality of the policy. More formally, given a policy π, the approximated distance $\hat{d_\pi}(T_A, T_B)$ is equal to:

$$\hat{d_\pi}(T_A, T_B) = d(T_A, T_B) + \left(V^*(\mathbf{s}_1(T_A, T_B)) - V^\pi(\mathbf{s}_1(T_A, T_B))\right)$$

where $d(T_A, T_B)$ is the true edition distance between T_A and T_B, $\mathbf{s}_1(T_A, T_B)$ is the initial state of the tree-edition CR-algorithm and $V^*(\mathbf{s}_0(T_A, T_B))$ (*resp.* $V^\pi(\mathbf{s}_0(T_A, T_B))$) is total reward perceived by an optimal policy (*resp.* the policy π) when starting from the initial state. In other words, the approximation error is equal to the regret of π.

7.3.2 Features and supervision

Feature Function

We now discuss feature functions $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$ for CR-algorithm 16. Such a function jointly describes the current tree, the target tree and a basic operation. Designing features for this problem is far from being trivial. Since the main objective is to select operations that reduces the distance between T_A and T_B, we have experimented a set of features that depends on similarity functions between both trees. As in Chapter 6, we use F_1 similarity measures, which are computed in linear time *w.r.t.* the number of nodes of the trees. We have 11 such similarity functions, denoted $F_1^{(1)}, \ldots, F_1^{(11)}$, including $F_{structure}$, F_{path} and $F_{content}$. These similarities appear in three kinds of features:

• **Delta-similarities** Delta-similarity features measure the variation of similarity caused by an action $\mathbf{a}$. There is one such feature f_i^{delta} per similarity score $F_1^{(i)}$:

$$f_i^{delta}(\mathbf{s}, \mathbf{a}) = F_1^{(i)}(T_{A'}, T_B) - F_1^{(i)}(T_A, T_B)$$

where T_B is the target tree, T_A is the current tree and $T_{A'}$ is the current tree of the successor state $T(\mathbf{s}, \mathbf{a})$.

• **Delta-similarity conjunctions** We introduced delta-similarity conjunctions to enable non-linear functions *w.r.t.* delta-similarities[5]. For each pair of similarity scores $(F_1^{(i)}, F_1^{(j)})$, there is one delta-similarity conjunction defined in the following way:

$$f_{i,j}^{delta.delta}(\mathbf{s}, \mathbf{a}) = f_i^{delta}(\mathbf{s}, \mathbf{a}).f_j^{delta}(\mathbf{s}, \mathbf{a})$$

• **Similarity-delta-similarity conjunctions** These features are motivated by the idea that the current distance between the source and target trees may play an important role in the optimal tree edition behavior. Similarity-delta-similarity features are conjunctions between current similarities and delta-similarities induced by an action. For each pair of similarity scores $(F_1^{(i)}, F_1^{(j)})$, there is one such feature defined in the following way:

$$f_{i,j}^{sim.delta}(\mathbf{s}, \mathbf{a}) = F_1^{(i)}(T_A, T_B).f_j^{delta}(\mathbf{s}, \mathbf{a})$$

The feature function ϕ described above computes dense descriptions. Given that we have 11 similarity functions, the dimension of these descriptions is: $d = 11 + 11 \times 11 + 11 \times 11 = 253$.

Supervision

Supervision is the most difficult part in CR-algorithm 16. On the tree edition problem, following the random policy leads to very poor success rates. Since there are much more `create` actions than `delete` actions, the most frequent behavior of the random policy is to grow T_A, which results in states with very large current trees T_A *w.r.t.* the target trees T_B. Pure reinforcement-learning algorithms seem thus difficult to apply. How to learn CR-algorithm 16 is still a largely opened question and, for the moment, we assume the availability of a set of OLTs. Thanks to this assumption, it is possible to use CR^{ank} to perform supervised learning of the policy.

7.3.3 Experiments

Dataset

For the moment, we only experimented tree-edition distance computation on relatively small trees. We use an artificial dataset, which is composed of 22350 randomly generated tree pairs. Each tree contains between 4 and 6 nodes with three different possible labels. Furthermore, these trees have a maximal arity of 4, *i.e.* there are no nodes with more than 4 children. The distribution of tree-edition distances of the tree pairs is given in Figure 7.7. The training set is composed of 90% of the examples while the remaining examples form the test set.

CR^{ank}

In order to learn, we assume the availability of OLTs for all the training examples. This makes it possible to use CR^{ank} with the following action-cost function:

$$c(\mathbf{s}, \mathbf{a}) = \begin{cases} 0 & \text{if } T(\mathbf{s}, \mathbf{a}) \text{ belongs to the OLT} \\ 1 & \text{otherwise} \end{cases}$$

We tried two ranking losses – best-against-all and all-pairs – combined with the large-margin criterion and roughly tuned CR^{ank} manually. We limited the length of episodes to $D_{max} = 20$.

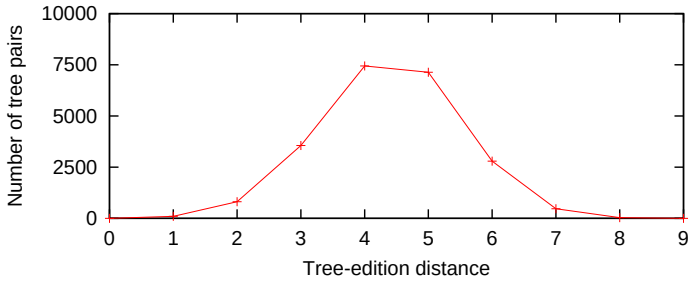


Figure 7.7: **Tree-editition distance distribution in the synthetic dataset.** This curve gives the number of tree pairs in the synthetic dataset classified by tree edition distances.

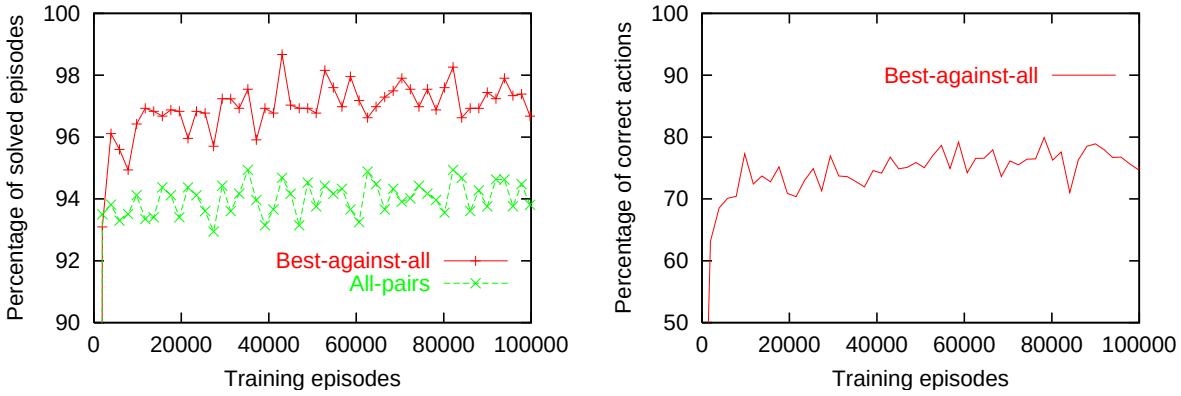


Figure 7.8: **Training behavior of CR^{ank} on the tree-edition CR-algorithm.** Left: the percentage of solved episodes in less than $D_{max} = 20$ steps in function of the number of training episodes. Right: the percentage of correctly predicted actions in function of the number of training episodes. An action $\mathbf{a}$ is correctly predicted if its cost $c(\mathbf{s}, \mathbf{a})$ is null.

Figure 7.8 illustrates the training behavior of CR^{ank} on CR-algorithm 16. It can be seen that the best-against-all loss seems here more adapted. Most of the learning with this loss is performed in 50.10^3 training episodes. We believe that this large number of training episodes can be lowered with more careful tuning of CR^{ank}. Given that on our dataset, there are about 10% correct actions per step (actions with a null cost), a random policy would predict about 10% actions correctly. It is thus quite interesting that CR^{ank} reaches a percentage of correctly predicted action of $\simeq 75\%$

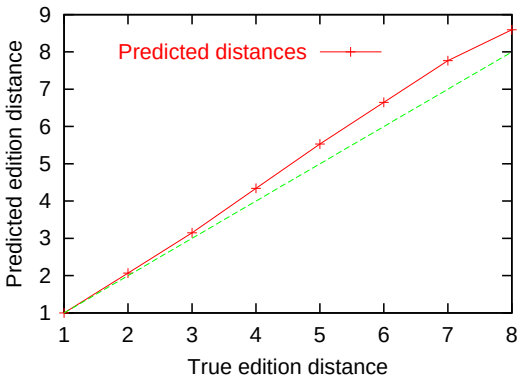


Figure 7.9: **Comparison between true edition distances and predicted edition distances.** This x-axis corresponds to correct edition distances and the y-axis corresponds to mean predicted edition distances. Ideally, the predicted edition distances should be equal to the correct edition distances (diagonal line).

Figure 7.10 gives the evaluation of CR^{ank} on the test set and Figure 7.9 gives the comparison between correct edition distances and average predicted edition distances. The error between the approximate edition distance and the correct edition distance seems to be proportional to the correct edition distance: the more trees are close, the better the estimated edition distance will be. In all cases, the mean approximation error is lower than 1, which is a satisfying result.

For the moment, the main limitation of our approach is its high computational cost. Although it has a relatively low complexity, each decision step needs much computation. In practice, the major bottleneck is the computation of the similarity-based features for each available action in each state. In order to reduce the number of available actions at each step, an alternative approach would be to completely change the CR-algorithm, to fit more closely to the ideas of existing dynamic programming algorithms for tree-edition distance computation. Briefly, this would consist in applying a heuristic-learning method such as this presented in Section 7.2 on the search-space of one of the existing dynamic programming algorithms. A promising approach here would be to develop a learning-based depth-first search algorithm that approximates the exhaustive computation performed by dynamic programming.

[5]These features are motivated by the use of learning linear. Note that a different approach here would be to use kernel-based machines with, for example, a polynomial kernel.

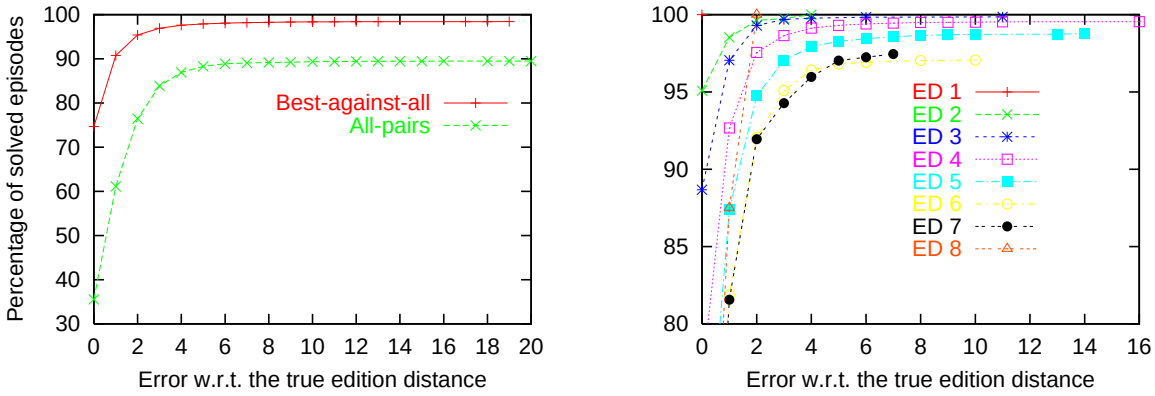


Figure 7.10: **Evaluation of the tree-edition CR-algorithm trained with CR^{ank}.** Left: the percentage of episodes where the error is less than a given threshold. Here, the error is difference between the approximate edition distance and the correct edition distance. For example, the predicted edition distance with CR^{ank} - best-against-all has an error lower than 5 for $\simeq 98\%$ of test examples. Right: the percentage of episodes where the error is less than a given threshold, sorted by correct edition distance.

7.4 Conclusion

In this chapter, we presented a domain, which is slightly beyond the main topic of this manuscript: learning-for-search. Learning-for-search relies on the intuition that machine learning techniques may be of interest to perform efficient search in the various combinatorial search problems that exists in artificial intelligence. We developed a set of *search* CR-algorithms: CR-algorithms that formalize learning-for-search problems. These CR-algorithms may be learned with reinforcement-learning techniques or supervised-learning techniques. We developed a general methodology, called post-supervision, to introduce supervision. This general methodology was applied to the problem of learning optimal heuristics for best-first search algorithms. We performed a set of experiments on a simple numbers game and obtained competitive results *w.r.t.* human players, which is quite satisfying. We also described another learning-for-search application to approximately compute tree-edition distances, which demonstrated that learning is also possible in complex search spaces with tree-structured data.

We believe that the CR-algorithms approach to learning-for-search is promising, but the work presented here is still prospective and much could be done to extend it. We envisage three major perspectives: extend the post-supervision idea to other search CR-algorithm, perform experiments on larger and more known search domains and develop a better understanding of the relations between CR-algorithms and other learning-for-search approaches.

8

Perspectives and Conclusions

Contents

In this manuscript, we introduced the CR-algorithm formalism to integrate inference procedure and associated learning problems and presented extensive experiments with this formalism and associated learning algorithms. CR-algorithms raise various perspectives with both machine-learning aspects and programming-language aspects. We give a structured overview of our research perspectives in Section 8.1. Complementary to these perspectives, the on-going project of creating a useable CR-algorithm programming language in described in Appendix **??**. The main conclusions that can be drawn from this work are given in Section 8.2.

8.1 Perspectives

We overview here the main long-term perspectives related to the CR-algorithm framework. These perspectives are structured into three directions: automating the CR-algorithm learning system (Section 8.1.1), extending the CR-algorithm formalism (Section 8.1.2) and developing new applications of CR-algorithms (Section 8.1.3).

8.1.1 Automating the learning system

One of the main goals of CR-algorithms is to form the basis for a new learning-based programming language (see Appendix **??**). Ideally, such a language should be simple enough to be accessible to non-specialists in machine learning. In particular, the learning system should be entirely automatic. Learning CR-algorithms could even be thought as a part of the compilation process leading to an executable program. Fully automatizing the learning of CR-algorithms is a long-term perspective that raises multiple challenging issues that are discussed below.

Automatic tuning of hyper-parameters All the learning methods that we introduced for CR-algorithms have some hyper-parameters that must be carefully tuned. The problem of hyper-parameter tuning is frequent in machine learning and must be solved to automatize learning. Common hyper-parameters include the *learning rate* (in all methods), the *discount* factor (in value-based reinforcement-learning methods), the β parameter (in OLPOMDP), the regularizer weight λ or the loss function (in CR^{ank}).

Learning rates

An interesting direction to tuning learning rates automatically is to make use of learning methods with adaptive learning rates [Sutton, 1992, Riedmiller et Braun, 1993, Plagianakos *et al.*, 1998, Schraudolph *et al.*, 2006]. These methods generally have one learning rate parameter α_i per parameter θ_i, which are automatically adjusted during learning. Adaptive learning rate methods still have hyper-parameter (*e.g.* a meta learning-rate parameter), but these parameters can generally be set to default values, which give satisfying results in all situations.

Discount factor

Automatically tuning the discount factor in reinforcement learning or the β parameter of OLPOMDP is a difficult problem. One possible direction is to use a general optimization algorithm, such as genetic algorithms [Cline, 2004]. Another approach, which is claimed to be biologically plausible, is based on gradient learning [Schweighofer et Doya, 2003].

Regularization

It is still not clear to what extent regularization can be helpful to improve the generalization performances of policies. In our experiments, we did not use this capability of CR^{ank}, by using a regularization parameter equal to 0. How to determine efficiently to best value for this parameter is an open question.

Stopping criterion In our work, we did not propose a stopping criterion for the learning algorithms and selected the number of training iterations manually. Usually, stopping criterion are based on performance evaluation with a validation dataset. Separating validation data from training data makes it possible to evaluate the *generalisation* ability of a learning machine. Learning should stop when this generalization measure does not increase anymore. Note that evaluation of the validation dataset may be a costly operation. Furthermore, it is not clear how often these evaluations should be done and how much time is necessary to consider that a policy will not improve anymore.

Choosing learning algorithms Depending on our experiments, we used various learning algorithms by selecting them by hand. An automated learning system for CR-algorithm should be able to automatically select which learning method is the most appropriate. Another perspective is to *chain* or *mix* different learning algorithm:

- **Chaining learning algorithms** An example of chaining is to first learn an heuristic policy by imitation and then perform reinforcement learning to improve the policy. This chaining would, for example, be particularly relevant to tree transformation (see Chapter **??**). Since most approximated reinforcement learning only optimizes the policy locally, learning to imitate an heuristic policy is a good mean to provide a good initial policy. Chaining imitation learning and reinforcement learning would have two strong advantages: making learning faster and reaching better policies.
- **Mixing learning algorithms** In CR-algorithms with multiple *choose* instructions, each *choose* may be strongly or weakly supervised. For example, the two-step order-free sequence labeling CR-algorithm(see Section 5.2.3), has a position *choose* and a label *choose*. The latter can be supervised with the OLP, while the former requires reinforcement learning. A good learning algorithm for two-step order-free labeling would thus be to mix CR^{ank} for the label *choose* with a reinforcement learning algorithm (*e.g.* OLPOMDP) for the position *choose*.

As a conclusion, the problem of automatically learning CR-algorithms is far from being solved. However, several approaches can be explored to improve this aspect of CR-algorithms.

8.1.2 Extending the CR-algorithm formalism

The CR-algorithm formalism can be extended in various directions to cover a wider range of problems.

Generality of representable MDPs All CR-algorithms implicitly define MDPs, however not all MDPs are representable by CR-algorithms with the current formalism. Here are the main possible extensions to widen the set of representable MDPs:

- **Other optimality criterions** We always assumed the total-reward criterion in our problems. Many MDPs are naturally expressed as discounted-reward or average-reward maximization problems. The definition of the optimality criterion may be part of a CR-algorithm in order to support this kind of problems.
- **Stochasticity** Reinforcement learning is generally used in stochastic environments. Most CR-algorithms of this manuscript where deterministic. However, nothing forbids to use stochastic operations between two *choose* instructions or between a *choose* and a *reward*. This makes it possible to simulate stochastic environments within CR-algorithms.
- **Continuous actions** Many control problems involve the use of continuous actions, *e.g.* $\mathcal{A} = \mathbb{R}$ or $\mathcal{A} = \mathbb{R}^d$. Extending the syntax of CR-algorithms to support such actions seems relatively simple. However, efficiently supporting such actions in the learning process is an open question.

Generality of learning problems We presented CR-algorithms as a general approach to solve structured prediction (SP) problems in a *supervised learning* setting. CR-algorithms could be extended to other learning settings:

- **Semi-supervised learning** A common situation in real-world applications of machine learning is that labeled example $(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ have a high cost, whereas it is easy to collect a large amount of unlabeled examples $\mathbf{x}^{(i)}$. Semi-supervised learning is a class of machine learning techniques that make use of both labeled and unlabeled examples. The idea of semi-supervised learning can be transposed to CR-algorithms. This would correspond to a situation where we have access to few training inputs and a large amount of normal inputs. Several approaches may be explored to provide solutions to this new problem.
- **Active learning** Another approach to tackle the labeling cost of training examples is to actively query labels to the user/teacher. Active learning methods have been developed to select among the set of unlabeled examples, those that, once labeled, may help learning the most. The idea of active learning may also be transposed to CR-algorithms. In such scenario, the learning system should actively ask for training inputs corresponding to most promising episodes.
- **Multitask learning/Transfer learning** An old question in machine learning is this of multitask learning [Caruana, 1993, Baxter, 2000]: how to exploit the structure of multiple related problems to improve learning. Transfer learning [Pan et Yang, 2008] focuses on a similar problem : how to exploit the knowledge acquired from previous problems into a new learning problem. We believe that, in the long-term, these advanced definitions of learning

may be fully relevant to CR-algorithms. In particular, we wish to study transfer between CR-algorithms. Multiple CR-algorithms of increasing complexity can be use to define solutions to the same problem. Let us give the example in sequence labeling: the simplest solution is to perform independent classification. The left-to-right approach then introduces the use of previous predicted labels to improve prediction accuracy. Multiple-pass labeling then introduces the idea to correct the first predictions in latter passes. This sequence of increasingly complex problems seems particularly well fitted to transfer learning.

CR-algorithm 17 A CR-algorithm that chooses between multiple other CR-algorithms.

Input: input parameters
Output: result

```
1: method ← choose[input parameters] {1, 2, 3, 4}          ▷ Choose the method to use
2: if method = 1 then return crAlgorithm1(inputParameters)
3: else if method = 2 then return crAlgorithm2(inputParameters)
4: else if method = 3 then return crAlgorithm3(inputParameters)
5: else if method = 4 then return crAlgorithm4(inputParameters)
6: end if
```

CR-algorithm 18 Divide-and-conquer CR-algorithm

```
1: procedure divideAndConquer(p)                          ▷ Solve the search problem p
2:     // decompose p into subproblems
3:     // this decomposition may rely on choose/reward instructions
4:     for each sp ∈ subproblems do
5:         divideAndConquer(sp)                           ▷ Recursive CR-algorithm call
6:     end for
7: end procedure
```

Miscellaneous other extensions We present here a few other possible extensions to CR-algorithms:

- **CR-algorithm calls** A very nice extension to CR-algorithms would be to support CR-algorithms calls within the body of other CR-algorithms. CR-algorithm 17 and CR-algorithm 18 illustrates two natural use cases of this extension. The former uses a *choose* to select which CR-algorithm to use (*e.g.* to choose between left-to-right labeling, order-free labeling or multiple-pass labeling). The latter illustrates a divide-and-conquer CR-algorithm based on a CR-algorithm recursive call. The key idea to support CR-algorithm calls is that the state of induced MDPs, in addition to the current parameters and variables, contains the whole stack of CR-algorithm calls at each given time step. An interesting aspect of this extension is its connection with the hierarchical reinforcement learning domain [Barto et Mahadevan, 2003], which could be a source of ideas to develop learning algorithm to support CR-algorithm calls.
- **Multiple heuristics** For the moment we only experimented CR-algorithms with a single action value (or action cost) function. In many real-world problems, there exists multiple well-known heuristics that could all be used to bias the learning process. How to exploit such knowledge is an open question.

- **Multi agent** A long-term perspective is to analyze the relevancy of CR-algorithms for multi-agent systems. A direction could be to study the behavior of CR-algorithms in multi-threaded execution schemes.

8.1.3 Developing new applications

Developing new applications of CR-algorithms will become much more easier thanks to the CR-algorithm programming language (see Appendix B). The ambition of this language is to become relevant to a wide range of applications that use some learning components. Some of the applications that could be developed include:

- **Structured prediction** We believe that CR-algorithms are powerful enough to tackle any SP problem. In particular, we are currently experimenting the behavior of CR-algorithms on the graph-labeling problem (*i.e.* prediction of a set of interdependent variables organized in a graph). The first results seem promising. We also made some experiments on the dependency-parsing task, which is a key step in natural language processing systems [Nivre, 2005].
- **Unsupervised learning** In the unsupervised learning problem, the user provides a set of examples $\{\mathbf{x}^{(i)}\}_{i\in[1,n]}$ *i.i.d.* from an unknown distribution $\mathcal{D}_{\mathcal{X}}$. Given this training set, can we use CR-algorithms to learn a policy π able to create elements sampled from $\mathcal{D}_{\mathcal{X}}$?
- **Learning-for-search** As discussed in Chapter 7, we believe that CR-algorithms may have several applications in learning-for-search.
- **Code dynamic-optimization** Another domain where CR-algorithms may be useful is code dynamic optimization. The aim here is to produce code whose running time decreases with experience. A key idea therefore is to connect rewards to the negative execution time of the code. Then, *choose* instructions can be used to tune the implementation dynamically. The learning system of CR-algorithms could then learn the mapping between situations and appropriate implementation choices, in order to optimize the running time of the code.
- **Bootstrapping** A particularly existing perspective with the previous application fields is that the learning problem of CR-algorithms raises itself several structured prediction and combinatorial search or optimization problems. This leads to the idea of *bootstrapping*[1]: the training system could be partially written with CR-algorithms, to solve other CR-algorithms more efficiently. Bootstrapping assumes that we have a initial CR-algorithm solver that could be this presented in this manuscript. This initial solver may then be used to learn the advanced CR-algorithm solver embedded into the compiler. Examples of CR-algorithms that may play a role during compilation include constraint satisfaction solvers, automatic provers or code transformation CR-algorithms.

8.2 Conclusions

We first briefly summarize the work presented in this thesis in Section 8.2.1. We then present conclusions relevant to structured prediction (Section 8.2.2), to reinforcement learning (Section **??**) and to learning-based programming (Section 8.2.4).

[1] `http://en.wikipedia.org/wiki/Bootstrapping_(compilers)`

8.2.1 Summary

In this thesis, we introduced CR-algorithms: a formalism to integrate inference procedures and associated learning problems. The inference procedure is a piece of code that solves a given problem by using machine-learning based decisions. The learning problem defines a *quality* criterion that should be maximized to find the optimal decision sequences to take during inference. CR-algorithms rely the well-established Markov decision process formalism and can be learned thanks to existing algorithms coming from domains ranging from structured prediction to reinforcement learning.

We illustrated the use of CR-algorithms on numerous different applications. We first considered sequence-labeling problems and developed three approaches for these problems: left-to-right labeling, order-free labeling and multiple-pass labeling. The resulting solutions where shown competitive with state-of-the-art models, while having lower complexities of most existing methods. We then moved on a more complex structured prediction problem: rooted ordered labeled tree transformation. In these problems, perfect supervision is rarely available. We showed the relevancy of reinforcement-learning algorithms in this context. We presented multiple solutions based on CR-algorithms, which accurately and precisely solve tree transformation problems. Furthermore, these solutions where shown to scale well with the size of data. This enabled our approaches to deal with all our real-world datasets, where previous methods failed.

Although most of this thesis was about structured prediction, we showed that the CR-algorithm formalism might also be relevant to other domains. In particular, we described CR-algorithms in the context of learning-for-search and showed promising results in this direction.

8.2.2 Structured Prediction

From the point of view of structured prediction, most of our conclusions meet those of [Daumé III, 2006].

Incremental SP

Incremental approaches to structured prediction provide efficient solutions to complex prediction problems. They do not make assumptions on feature functions nor on loss functions. This is particularly interesting, since incremental approaches can deal with non-decomposable losses and can incorporate long-term dependencies in the features. Our results on sequence labeling clearly show that long-term dependencies are of primary importance on some datasets (up to +8% improvement over dynamic-programming based models).

Work around greedy-inference

Dynamic programming is generally used to find the best compromise for a whole sequence of labels. In incremental approach, inference is greedy and may suffer from local ambiguities. We developed original approaches to sequence labeling to work around this limitation: order-free labeling and multiple-pass labeling. An important idea that should be focused is the following: although we perform greedy inference, we may be greedy in very large spaces corresponding to non purely greedy algorithms. The less greedy algorithm, multiple-pass labeling, showed very nice results (up to +11.8% improvement between left-to-right and multiple-pass).

Reinforcement-learning based SP

Our major contribution to the field of structured prediction was to introduce reinforcement learning as a concrete mean to solve hard-to-supervise tasks. On the tree-transformation task, reinforcement learning gave very satisfying results: reinforcement-learning algorithms find better strategies than the greedy behavior and succeed in learning and generalizing these strategies (up to 40% improvement). Nevertheless, reinforcement learning has two major drawbacks. Firstly, it requires huge numbers of training iterations. Secondly, if the exploration problem is too complex, reinforcement learning may not find globally optimal policies.

Chaining supervised and reinforcement learning

We believe that a very promising approach to perform weakly supervised incremental struc-

tured prediction is to chain supervised learning with reinforcement learning in a two-step approach. In the first, such an approach consists in learning to imitate a *not too bad* heuristic with supervised learning. The aim of this step is to provide an initial *not too bad* policy in few training time. The second step can then optimize this policy locally using reinforcement-learning techniques (*e.g.* a policy-gradient method).

8.2.3 Reinforcement learning

Reinforcement learning success

The Markov decision processes discussed in this thesis share some unusual properties from the point of view of reinforcement learning. They are mostly deterministic and have extremely large state spaces, the reward function may only be defined for a set of training examples and additional supervision knowledge may be available. We reported numerous successful results with general reinforcement learning algorithms on our problems. In particular, these algorithms led to competitive results with state-of-the-art domain-specific structured prediction models.

CR^{ank}: action ranking

We introduced the idea of *policy as ranking machine* with the CR^{ank} algorithm. This approach showed nice results in supervised problems and we proposed the use of rollouts to deal with weakly supervised problems. Using CR^{ank} in a pure reinforcement-learning setting with rollouts gave mitigated results. In particular, this approach seems sensible to hyper parameters, has a high computational cost and do not always lead to optimal policies. In its current version, CR^{ank} is the algorithm of choice to deal with supervised learning of CR-algorithms. An interesting aspect of CR^{ank} is that it learns policies that have the same form as those used in policy gradient methods such as OLPOMDP[2]. CR^{ank} can thus be chained with OLPOMDP to provide an efficient solution to CR-algorithms for which we know a *not too bad* heuristic.

Sparse high-dimensional features

One key of the successful results we developed is the kind of feature representation and learning machine that where used: sparse high-dimensional vectors with linear learning. Linear learning has multiple nice properties: it is simple, it leads to efficient computations and it leads to accurate models. Furthermore, since virtually anything can be put in the feature function, linear learning machine can learn highly non-linear functions of the raw data. The kind of representations we used are particularly good at describing structured or relational data. Along with the work presented in this thesis, we developed a new formalism to describe feature generators (see Appendix B and Appendix C). We hope that this framework will prove its value in the future, and help to spread to use of sparse high-dimensional descriptions combined with linear learning machines.

8.2.4 Learning-based programming

A key idea that guided the development of CR-algorithms is this of learning-based programming [Roth, 2006]. We believe that a key step in the development of machine learning toward complex learning-based tasks is to provide new programming languages that gives a central place to learning. CR-algorithms form the basis for such a learning-based programming language (see Appendix B).

Interestingly, this work shares a number of conclusions with [Roth, 2006]. In particular, both solutions put classification as the core learning-based operation and both solutions rely on feature generators and linear learning. A major contribution of our work is to explicitly describe learning-based programs as sequential decision-making problems. Another difference is that [Roth, 2006] introduces an inference engine in its learning-based programming language, whereas with CR-algorithms we propose to view inference as a trainable sequential decision-process, separated

[2]Both methods have a parameter vector θ that defines scores given action features $\langle \theta, \phi(\mathbf{s}, \mathbf{a}) \rangle$.

from the core of the language. We believe that these differences and similarities will help to get a better understanding of learning-based programming and contribute to move this new approach from academics to practitioners.

Bibliography

[Abbeel *et al.*, 2006] Pieter Abbeel, Daphne Koller, et Andrew Y. Ng. Learning factor graphs in polynomial time and sample complexity. *Journal of Machine Learning Research*, 7:1743–1788, 2006.

[Andrew et Gao, 2007] G. Andrew et J. Gao. Scalable training of L1-regularized log-linear models. In Zoubin Ghahramani, editor, *ICML 2007*, pages 33–40. Omnipress, 2007.

[Bartlett *et al.*, 2004] Peter L. Bartlett, Ben Taskar, Michael Collins, et David Mcallester. Exponentiated gradient algorithms for large-margin structured classification. In *In NIPS*, pages 113–120. MIT Press, 2004.

[Barto et Mahadevan, 2003] A. Barto et S. Mahadevan. Recent advances in hierarchical reinforcement learning, 2003.

[Baxter *et al.*, 2001] Jonathan Baxter, Peter L. Bartlett, et Lex Weaver. Experiments with infinite-horizon, policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:2001, 2001.

[Baxter et Bartlett, 2001] Jonathan Baxter et Peter L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350, 2001.

[Baxter, 2000] Jonathan Baxter. A model of inductive bias learning. *Journal of Artificial Intelligence Research*, 12:149–198, 2000.

[Berger *et al.*, 1996] Adam L. Berger, Stephen Della Pietra, et Vincent J. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[Bertsekas et Tsitsiklis, 1996] Dimitri P. Bertsekas et John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[Bertsekas, 1999] D.P Bertsekas. Rollout agorithms: an overview. In *Decision and Control*, pages 448–449, 1999.

[Bille, 2002] Philip Bille. A survey on tree edit distance and related problems. Technical report, The IT Univeristy of Copenhagen, 2002.

[Boukottaya et Vanoirbeek, 2005] Aida Boukottaya et Christine Vanoirbeek. Schema matching for transforming structured documents. In *ACM DOCENG*, pages 101–110, 2005.

[Boyan et Moore, 2001] Justin Boyan et Andrew W. Moore. Learning evaluation functions to improve optimization by local search. *Journal of Machine Learning Research*, 1:77–112, 2001.

[Boyd et Vandenberghe, 2004] Stephen Boyd et Lieven Vandenberghe. *Convex Optimization*. 2004.

[Brefeld et Scheffer, 2006] Ulf Brefeld et Tobias Scheffer. Semi-supervised learning for structured output variables. In *ICML*, pages 145–152, 2006.

[Cappé, 2001] Olivier Cappé. Ten years of hmms, March 2001. http://www.tsi.enst.fr/ cappe/-docs/hmmbib.html.

[Caruana, 1993] Richard A. Caruana. Multitask learning: A knowledge-based source of inductive bias. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 41–48. Morgan Kaufmann, 1993.

[Castano *et al.*, 2001] Silvana Castano, Valeria De Antonellis, et Sabrina De Capitani di Vimercati. Global viewing of heterogeneous data sources. *IEEE Trans. Knowl. Data Eng.*, 13(2):277–297, 2001.

[Chidlovskii et Fuselier, 2005] Boris Chidlovskii et Jérôme Fuselier. A probabilistic learning method for xml annotation of documents. In *IJCAI*, 2005.

[Chung *et al.*, 2002] Christina Yip Chung, Michael Gertz, et Neel Sundaresan. Reverse engineering for web data: From visual to semantic structure. In *ICDE*, pages 53–63, 2002.

[Cline, 2004] Ben E. Cline. Tuning q-learning parameters with a genetic algorithm, September 2004.

[Cohen *et al.*, 1998] William W. Cohen, Robert E. Schapire, et Yoram Singer. Learning to order things. In Michael I. Jordan, Michael J. Kearns, et Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[Cohen et Carvalho, 2005] William W. Cohen et Vitor R. Carvalho. Stacked sequential learning. In *Proceedings of the IJCAI 2005*, 2005.

[Collins et Roark, 2004] Michael Collins et Brian Roark. Incremental parsing with the perceptron algorithm. In *ACL'04*, pages 111–118, Barcelona, Spain, July 2004.

[Collins, 2002] Michael Collins. Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *EMNLP*, 2002.

[Crammer et Singer, 2003] Koby Crammer et Yoram Singer. A family of additive online algorithms for category ranking. *J. Mach. Learn. Res.*, 3:1025–1058, 2003.

[Cristianini et Shawe-Taylor, 2000] Nello Cristianini et John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000.

[Daumé III *et al.*, 2006] Hal Daumé III, John Langford, et Daniel Marcu. Search-based structured prediction. 2006.

[Daumé III et Marcu, 2005] Hal Daumé III et Daniel Marcu. Learning as search optimization: Approximate large margin methods for structured prediction. In *ICML*, Bonn, Germany, 2005.

[Daumé III, 2006] Hal Daumé III. *Practical Structured Learning Techniques for Natural Language Processing*. PhD thesis, University of Southern California, Los Angeles, CA, August 2006.

[Denoyer et Gallinari, 2006] Ludovic Denoyer et Patrick Gallinari. The wikipedia xml corpus. *SIGIR Forum*, 2006.

[Denoyer et Gallinari, 2007] Ludovic Denoyer et Patrick Gallinari. Report on the xml mining track at inex 2005 and inex 2006. *SIGIR Forum*, pages 79–90, 2007.

[Dietterich *et al.*, 2004] Thomas G. Dietterich, Adam Ashenfelter, et Yaroslav Bulatov. Training conditional random fields via gradient tree boosting. In *ICML*, 2004.

[Dietterich, 2002] T. G Dietterich. Machine learning for sequential data: A review. In T. Caelli (Ed.) Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Doan *et al.*, 2000] AnHai Doan, Pedro Domingos, et Alon Y. Levy. Learning source description for data integration. In *WebDB (Informal Proceedings)*, pages 81–86, 2000.

[Doan *et al.*, 2001] AnHai Doan, Pedro Domingos, et Alon Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, pages 509–520, 2001.

[Doan *et al.*, 2003] Anhai Doan, Pedro Domingos, et Alon Halevy. Learning to match the schemas of data sources: A multistrategy approach. *Maching Learning*, 50(3):279–301, 2003.

[Doan et Wong, 1997] Khanh P.V. Doan et Kit Po Wong. Shapes: A novel approach for learning search heuristics in under-constrained optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 09(5):731–746, 1997.

[Embley *et al.*, 2001] David W. Embley, David Jackman, et Li Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Workshop on Information Integration on the Web*, pages 110–117, 2001.

[Ernandes et Gori, 2004] Marco Ernandes et Marco Gori. Likely-admissible and sub-symbolic heuristics. In Ramon López de Mántaras et Lorenza Saitta, editors, *ECAI*, pages 613–617. IOS Press, 2004.

[Freund *et al.*, 2003] Yoav Freund, Raj Iyer, Robert E. Schapire, et Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, 2003.

[Fuhr *et al.*, 2002a] Norbert Fuhr, N. Govert, G. Kazai, et M. Lalmas. INEX: Initiative for the Evaluation of XML Retrieval. In *SIGIR 2002 Workshop on XML and IR*, 2002.

[Fuhr *et al.*, 2002b] Norbert Fuhr, Norbert Gövert, Gabriella Kazai, et Mounia Lalmas, editors. *Proceedings of the First Workshop of the INitiative for the Evaluation of XML Retrieval (INEX), Schloss Dagstuhl, Germany, December 9-11, 2002*, 2002.

[Garcia et Ndiaye, 1998] Frédérick Garcia et Seydina M. Ndiaye. A learning rate analysis of reinforcement learning algorithms in finite-horizon. In *ICML '98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 215–223, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[Globerson *et al.*, 2007] Amir Globerson, Xavier Carreras, Terry Koo, et Michael Collins. Exponentiated gradient algorithms for log-linear structured prediction. In *ICML'07*, 2007.

[Hastie *et al.*, 2001] T. Hastie, R. Tibshirani, et J. H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[Hoffmann et Nebel, 2001] Jörg Hoffmann et Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)*, 14:253–302, 2001.

[Hou *et al.*, 2002] Guiwen Hou, Jingyue Zhang, et Jiahong Zhou. Mixture of experts of ann and knn on the problem of puzzle 8, 2002.

[Howard, 1960] Ronald A. Howard. *Dynamic Programming and Markov Processes*. Technology Press-Wiley, Cambridge, Massachusetts, 1960.

[J. Si et II, 2004] P. W. B J. Si, A. G. Barto et D. W. II. *Handbook of Learning and Approximate Dynamic Programming*. Wiley&Sons, INC., Publications, 2004.

[Johnson, 1998] Mark Johnson. Pcfg models of linguistic tree representations. *Comput. Linguist.*, 24(4):613–632, 1998.

[Jousse *et al.*, 2006] Florent Jousse, Remi Gilleron, Isabelle Tellier, et Marc Tommasi. Conditional random fields for xml trees. In *ECML Workshop on Mining and Learning in Graphs*, 2006.

[Kakade et Langford, 2002] Sham Kakade et John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, pages 267–274, 2002.

[Kassel, 1995] Robert Howard Kassel. *A comparison of approaches to on-line handwritten character recognition*. PhD thesis, Cambridge, MA, USA, 1995.

[Klein, 1998] Philip Klein. Computing the edit-distance between unrooted ordered trees. *Proceedings of 6th European Symposium on Algorithms*, pages 91–102, 1998.

[Kulesza et Pereira, 2008] Alex Kulesza et Fernando Pereira. Structured learning with approximate inference. In J.C. Platt, D. Koller, Y. Singer, et S. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 785–792. MIT Press, Cambridge, MA, 2008.

[Lafferty *et al.*, 2001] John Lafferty, Andrew McCallum, et Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML'01*, 2001.

[Lafferty *et al.*, 2004] John Lafferty, Xiaojin Zhu, et Yan Liu. Kernel conditional random fields: representation and clique selection. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 64, New York, NY, USA, 2004. ACM Press.

[Lagoudakis et Parr, 2003] Michail G. Lagoudakis et Ronald Parr. Reinforcement learning as classification: Leveraging modern classifiers. In *ICML*, pages 424–431, 2003.

[Langford et Zadrozny, 2005] John Langford et Bianca Zadrozny. Relating reinforcement learning performance to classification performance. In *ICML*, pages 473–480, 2005.

[Langley, 1983] Pat Langley. Learning effective search heuristics. In *IJCAI*, pages 419–421, 1983.

[LeCun *et al.*, 2006] Yann LeCun, Sumit Chopra, Raia Hadsell, Ranzato Marc'Aurelio, et Fu-Jie Huang. A tutorial on energy-based learning. In *Predicting Structured Data*. MIT Press, 2006.

[Leinonen, 2003] Paula Leinonen. Automating xml document structure transformations. In *ACM DOCENG*, pages 26–28, 2003.

[Li et Clifton, 2000] Wen-Syan Li et Chris Clifton. Semint: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl. Eng.*, 33(1):49–84, 2000.

[Liao *et al.*, 2007] Lin Liao, Tanzeem Choudhury, Dieter Fox, et Henry A. Kautz. Training conditional random fields using virtual evidence boosting. In *IJCAI*, pages 2530–2535, 2007.

[Lio et Nocedal, 1989] D. C. Lio et J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3):503–528, 1989.

[Mayoraz et Alpaydin, 1998] Eddy Mayoraz et Ethem Alpaydin. Support vector machine for multiclass classification. IDIAP-RR 06, IDIAP, 1998. Submitted for publication.

[Nguyen et Guo, 2007] Nam Nguyen et Yunsong Guo. Comparisons of sequence labeling algorithms and extensions. In *ICML*, pages 681–688, 2007.

[Nivre, 2005] Joakim Nivre. Dependency grammar and dependency parsing. Technical report, Växjö University, 2005.

[Palopoli *et al.*, 1998] Luigi Palopoli, Domenico Saccà, et Domenico Ursino. Semi-automatic semantic discovery of properties from database schemas. In *IDEAS*, pages 244–253, 1998.

[Pan et Yang, 2008] Sinno Jialin Pan et Qiang Yang. A survey on transfer learning. Technical Report HKUST-CS08-08, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China, November 2008.

[Petrovic *et al.*, 2007] Smiljana Petrovic, Susan L. Epstein, et Richard J. Wallace. Learning a mixture of search heuristics, 2007.

[Phan et Nguyen, 2005] Xuan-Hieu Phan et Le-Minh Nguyen. Flexcrfs: Flexible conditional random field toolkit, 2005. http://flexcrfs.sourceforge.net.

[Plagianakos *et al.*, 1998] V. P. Plagianakos, D. G. Sotiropoulos, et M. N. Vrahatis. An improved backpropagation method with adaptive learning rate, to be appear in. In *Procceeding of the 2nd Intern. Confer. on: Circuits, Systems and Computers*, 1998.

[Platt, 1998] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical report, Advances in Kernel Methods - Support Vector Learning, 1998.

[Platt, 1999] John C. Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[Rabiner, 1990] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Readings in speech recognition*, pages 267–296, 1990.

[Rahm et Bernstein, 2001] Erhard Rahm et Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[Ramshaw et Marcus, 1995] Lance Ramshaw et Mitch Marcus. Text chunking using transformation-based learning. In David Yarovsky et Kenneth Church, editors, *Proceedings of the Third Workshop on Very Large Corpora*, pages 82–94, Somerset, New Jersey, 1995. ACL.

[Ratliff *et al.*, 2006a] Nathan Ratliff, Andrew J. Bagnell, et Martin Zinkevich. Subgradient methods for maximum margin structured learning. In *Workshop on Learning in Structured Output Spaces at ICML*, 2006.

[Ratliff *et al.*, 2006b] Nathan Ratliff, James (Drew) Bagnell, et Martin Zinkevich. Maximum margin planning. In *International Conference on Machine Learning*, July 2006.

[Riedmiller et Braun, 1993] Martin Riedmiller et Heinrich Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *ICNN*, pages 586–591, San Francisco, CA, 1993.

[Rosenblatt, 1958] F. Rosenblatt. The Perceptron : probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[Roth, 2006] Dan Roth. Learning based programming. In *Innovations in Machine Learning*, pages 73–95, 2006.

[S. et A, 1985] Guiasu S. et Shenitzer A. The principle of maximum entropy. *The Mathematical Intelligencer*, 7, 1985.

[Samadi *et al.*, 2008] Mehdi Samadi, Ariel Felner, et Jonathan Schaeffer. Learning from multiple heuristics. In Dieter Fox et Carla P. Gomes, editors, *AAAI*, pages 357–362. AAAI Press, 2008.

[Schapire, 1990] Robert E. Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2):197–227, 1990.

[Schraudolph *et al.*, 2006] Nicol N. Schraudolph, Jin Yu, et Douglas Aberdeen. Fast online policy gradient learning with smd gain vector adaptation. In *Advances in Neural Information Processing Systems 18*, pages 110–119. MIT Press, 2006.

[Schraudolph et Graepel, 2003] N. Schraudolph et T. Graepel. Combining conjugate direction methods with stochastic approximation of gradients. In Christopher M. Bishop et Brendan Frey, editors, *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, AISTATS*, 2003.

[Schweighofer et Doya, 2003] Nicolas Schweighofer et Kenji Doya. Meta-learning in reinforcement learning. *Neural Netw.*, 16(1):5–9, 2003.

[Selkow, 1977] Stanley Selkow. The tree-to-tree editing problem. *Information Processing Letters*, 1977.

[Sha et Pereira, 2003] Fei Sha et Fernando Pereira. Shallow parsing with conditional random fields. In *Proceedings of Human Language Technology-NAACL 2003*, Edmonton, Canada, 2003.

[Shalev-Shwartz *et al.*, 2007] S. Shalev-Shwartz, Y. Singer, et N. Srebro. Pegasos: primal estimated sub-gradient solver for SVM. In Zoubin Ghahramani, editor, *ICML 2007*, pages 807–814. Omnipress, 2007.

[Shvaiko et Euzenat, 2005] Pavel Shvaiko et Jérôme Euzenat. A survey of schema-based matching approaches. pages 146–171, 2005.

[Su *et al.*, 2001] Hong Su, Harumi A. Kuno, et Elke A. Rundensteiner. Automating the transformation of xml documents. In *WIDM*, pages 68–75, 2001.

[Sutton et Barto, 1998] R. Sutton et A.G. Barto. *Reinforcement learning: an introduction*. MIT Press, 1998.

[Sutton, 1992] Richard S. Sutton. Adapting bias by gradient descent: an incremental version of delta-bar-delta. In *In Proceeding of Tenth National Conference on Artificial Intelligence AAAI-92*, pages 171–176. MIT Press, 1992.

[Tai, 1979] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM*, pages 422–433, 1979.

[Taskar *et al.*, 2003] Benjamin Taskar, Carlos Guestrin, et Daphne Koller. Max-margin markov networks. In *NIPS*, 2003.

[Taskar *et al.*, 2005] Benjamin Taskar, Simon Lacoste-Julien, et Michael Jordan. Structured prediction via the extragradient method. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005*, 2005.

[Taskar *et al.*, 2006] Ben Taskar, Simon Lacoste-Julien, et Michael I. Jordan. Structured prediction, dual extragradient and bregman projections. *Journal of Machine Learning Research*, 2006.

[Titov et Henderson, 2007] Ivan Titov et James Henderson. Incremental bayesian networks for structure prediction. In *ICML*, pages 887–894, 2007.

[Tsochantaridis *et al.*, 2004] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, et Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML'04*, 2004.

[Tsochantaridis *et al.*, 2005] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, et Yasemin Altun. Large margin methods for structured and interdependent output variables. *J. Mach. Learn. Res.*, 6:1453–1484, 2005.

[van Rijsbergen, 2001] C. J. van Rijsbergen. Getting into information retrieval. pages 1–20, 2001.

[Vapnik, 1995] Vladimir N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Vapnik, 1999] V. N. Vapnik. An overview of statistical learning theory. *Neural Networks, IEEE Transactions on*, 10(5):988–999, 1999.

[Veloso *et al.*, 1995] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, et Jim Blythe. Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:81–120, 1995.

[Vishwanathan *et al.*, 2006] S. V. N. Vishwanathan, Nicol N. Schraudolph, Mark W. Schmidt, et Kevin P. Murphy. Accelerated training of conditional random fields with stochastic gradient methods. In *ICML*, pages 969–976, 2006.

[Wainwright, 2006] Martin J. Wainwright. Estimating the "wrong" graphical model: Benefits in the computation-limited setting. *Journal of Machine Learning Research*, 7:1829–1859, 2006.

[Wallach, 2002] Hanna Wallach. Efficient training of conditional random fields. Master's thesis, University of Edinburgh, 2002.

[Wallach, 2004] Hanna M. Wallach. Conditional random fields: An introduction. Technical report, 2004.

[Watkins, 1989] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[Weston *et al.*, 2002] Jason Weston, Olivier Chapelle, André Elisseeff, Bernhard Schölkopf, et Vladimir Vapnik. Kernel dependency estimation. In Suzanna Becker, Sebastian Thrun, et Klaus Obermayer, editors, *NIPS*, pages 873–880. MIT Press, 2002.

[Wisniewski *et al.*, 2007] Guillaume Wisniewski, Ludovic Denoyer, Maes Francis, et Patrick Gallinari. Probabilistic model for structured document mapping. In *5th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM'07)*, Germany, 2007.

[Xu *et al.*, 2007] Yuehua Xu, Alan Fern, et Sung Wook Yoon. Discriminative learning of beam-search heuristics for planning. In *IJCAI*, pages 2041–2046, 2007.

[Xu et Fern, 2007] Yuehua Xu et Alan Fern. On learning linear ranking functions for beam search. In *ICML*, pages 1047–1054, 2007.

[Yoon *et al.*, 2006] Sung Wook Yoon, Alan Fern, et Robert Givan. Learning heuristic functions from relaxed plans. In *ICAPS*, pages 162–171, 2006.

[Zhang et Shasha, 1989] Kaizhong Zhang et Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 1989.

A

Nieme toolkit

All the experiments described in this manuscript where performed with our open-source machine-learning toolkit. This toolkit provides an experimental environment particularly tailored to the needs of our framework. In particular, it provides support for supervised learning and large decision processes. It also implements all the learning algorithms experimented in this manuscript. Since NIEME has reached a certain level of maturity we spent some time to release to source code.

NIEME is released under the GPL license. It is efficiently implemented in C++, it works on Linux, Mac OS X and Windows and provides interfaces for C++, Java and Python. The NIEME website includes a quick-start guide with compilation instructions and tutorials to get started with NIEME. Furthermore, it includes a complete reference documentation of the interface. All functions of NIEME's interface are unit-tested within the *python unittest* framework. NIEME is published in the Journal of Machine Learning Reseach in the special issue on Machine Learning Open Source Software (MLOSS) [XX cite].

In this appendix, we first describe the general architecture of NIEME (Section A.1) and then focus on the two domains convered by NIEME: supervised learning (Section A.2) and learning in decision processes (Section A.3).

A.1 Architecture of Nieme

Core implementation

Figure A.1 illustrates the general architecture of NIEME. The core of NIEME is implemented in C++ and we gave a special care to its design, which is entirely object-oriented and makes use of several design patterns. The core of NIEME contains code for features, vectors, tables, datasets,

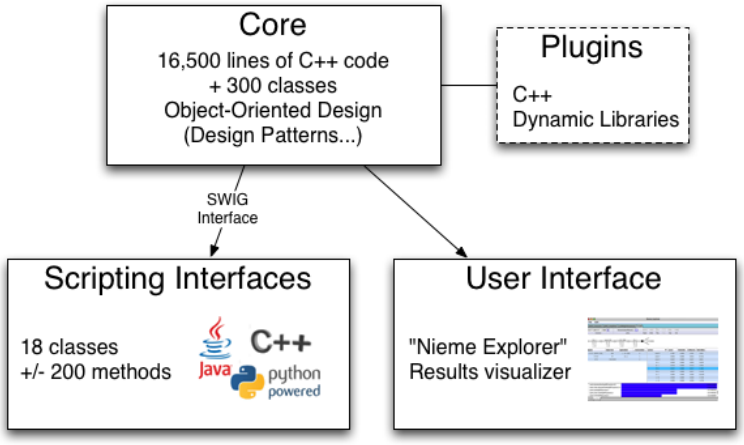


Figure A.1: **Architecture of the Nieme toolkit.** Each box corresponds to one software component of NIEME. Top: implementation in C++. Bottom: NIEME entry points: the scripting interfaces and the graphical user interface.

learning machines, learning algorithms, decision processes, value functions, policies and policy learning algorithms. NIEME can be extended through a mechanism of plugins, which are written in C++. We usually develop such a plugin for each new application (*e.g.* sequence labeling, tree transformation or best-first-search heuristic learning).

Entry points

NIEME provides two entry points that are illustrated by Figure A.2: a set of scripting interfaces for C++, Java and Python and a graphical user interface, called "NIEME explorer". The former ease the work of developing machine-learning experiments while the latter is a visualization tool for demonstration and debugging purpose. NIEME relies on a generic mechanism of introspection[1], which enables generic operations such as loading, saving, comparing and cloning objects or visualizing objects with the user interface. Thanks to this mechanism all objects in NIEME can be created and serialized from the scripting interfaces. Any saved object can then be opened with NIEME explorer. The explorer can be extended through plugins, which makes it possible to develop custom graphical components to display or manipulate objects.

A.2 Supervised learning

Energy-based models

Most learning machines in NIEME rely on the unified framework of energy-based models introduced by [LeCun *et al.*, 2006]. In this framework, illustrated in Figure A.3, a learning machine can be interpreted as a combination of an architecture $\mathcal{A}$ with parameters θ, a per-example loss Δ, a set of regularizers $\{\Omega_1, \ldots, \Omega_R\}$ and a learner $\mathcal{L}$. Given the architecture, the per-example loss, the regularizers and a set of training examples $D = \{e^{(1)}, \ldots, e^{(n)}\}$, we can define the learning energy, which is a generalized form of regularized empirical risk (see Section 2.1.3):

$$\hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \Delta(e^{(i)}, \mathcal{A}, \theta) + \sum_{i=1}^{R} \Omega_i(\theta)$$

[1] `http://nieme.lip6.fr/Nieme/IntrospectiveFeature`

Figure A.2: **Nieme entry points: explorer and scripting interfaces.** Top: NIEME explorer. Any object in NIEME can be opened in the explorer for visualization and debugging purpose. We give here examples for parameter vectors, experimental result tables, decision processes and images. Bottom: scripting interfaces. We give the same program in three languages (Python, Java and C++). The program loads a classification data set, trains a maximum-entropy classifier and saves the resulting model in a file called "example.model".

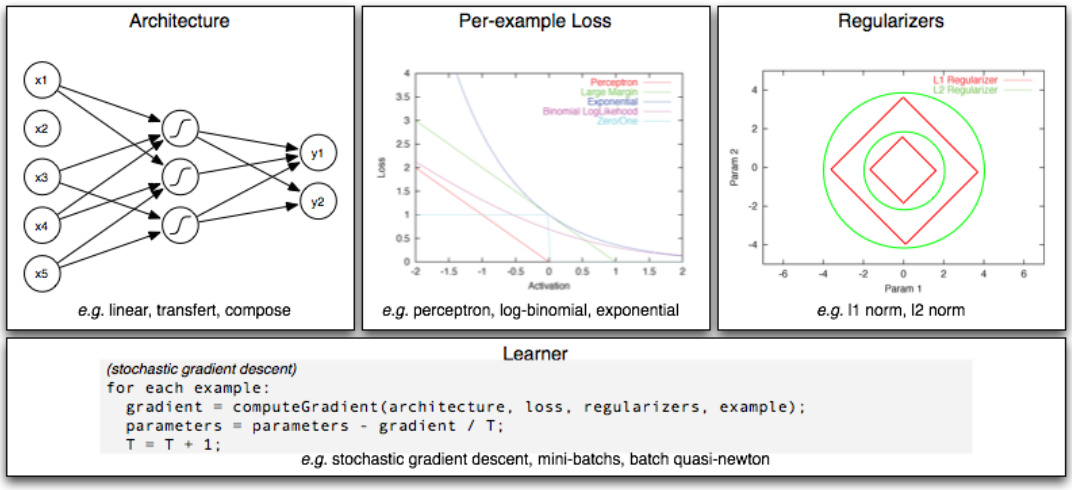


Figure A.3: **Energy based models framework.** Top: the three components that define the learning energy. Bottom: the *learner* component that performs energy minimization.

The aim of the learner is then to find parameters θ^* that minimize the regularized empirical risk given the training set D:

$$\theta^* = \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \hat{R}(\theta)$$

We now describe each component of the energy-based models framework:

• **Architecture** The architecture is a parameterized function that computes predictions given input vectors. A simple example is the linear architecture, which computes a single output as a scalar product between the input vector and the parameter vector. NIEME supports elementary architectures (linear, multi-class linear, neural network transfer function), as well as a *composition* operation, which allows users to create new architectures by chaining existing ones.

• **Per-example Loss** The per-example loss quantifies *how bad* an architecture and its parameters perform on a given learning example. The precise nature of the examples $e^{(i)}$ has only to be known by the per-example loss component in our framework. NIEME implements loss functions for classification examples, for regression examples and for ranking examples. Support a new kind of learning example (*i.e.* multi-dimensional regression examples) is simply a matter of writing an appropriate loss function. Discriminative losses supported by NIEME are the Perceptron loss, the hinge loss, the log-binomial loss and the exponential loss. NIEME also provides two regression losses – the squared loss and the absolute loss – and ranking losses, which are combinations of discriminative losses and decomposition strategies, such as all-pairs, most-violated-pair or best-against-all.

• **Regularizers** Regularizers are functions that measure the *complexity* of an architecture and its parameters. Up to now, NIEME includes the two most commonly used regularizers: the l1-norm and l2-norm of the parameters.

Model	Architecture	Loss	Regularizers	Learner
Perceptron	linear	perceptron	none	stochastic descent
Logistic regression	linear	log-binomial	none	batch quasi-newton
Pegasos linear SVM	linear	hinge loss	l2	pegasos learner
Multilayer perceptron	linear ∘ transfer ∘ linear	perceptron	none	stochastic descent
L1-maxent classifier	multi-class linear	log-binomial	l1	batch quasi-newton
Pegasos multi-class SVM	multi-class linear	hinge loss	l2	pegasos learner
Least-square regression	linear	squared loss	none	batch quasi-newton
Custom	linear ∘ transfer	absolute loss	l1 + l2	batch rprop
Many others	...	...	...	...

Table A.1: **Examples of energy-based models in Nieme.** Each model is defined by its architecture, per-example loss, regularizers and learner. The ∘ symbols denotes architecture composition.

• **Learner** The learner is the algorithmic component that optimizes the parameters θ. NIEME implements three batch learners: the large-scale limited-memory quasi-Newton method of [Lio et Nocedal, 1989], the recently proposed method of [Andrew et Gao, 2007] for minimizing large-scale l1-regularized models and the r-prop method of [Riedmiller et Braun, 1993]. If batch learning is not possible for a given problem, NIEME proposes classical online methods such as stochastic gradient descent. Finally, NIEME also offers mini-batch methods including the recent SVM solver of [Shalev-Shwartz *et al.*, 2007].

As illustrated in Table A.1, many combinations are possible in our energy-based framework. Some correspond to well-known learning machines, others to more original approaches. This modular design of learning machines of course induces an execution cost when compared to hard-coded algorithms, as for example a Perceptron made of 20 lines of C. Nevertheless, we believe that this cost is moderated and does not prevent an efficient implementation.

Composite vectors Vectorial computations are at the core of machine-learning library such as NIEME. It is thus of primary importance to provide an efficient data-structure to deal with vectors and vector computations. Instead of storing all the vector components in a flat way (as do most computing libraries), NIEME introduces an original data structure called *composite vectors*. A composite vector is either a flat vector, either the union of multiple sub-vectors. Composite vectors, which are illustrated in Figure A.4, are advantageous in numerous situations such as the ones below:

- **Generic architecture compositions** Multi-layer learning machines have two sets of parameters: one set for the first layer, another set for the second layer. Composite vectors make it possible to view the parameters as a single vector, while keeping layer-specific parameters in separated sub-vectors.
- **Feature Descriptions.** Often, input objects are described with features coming from different families (e.g. bigrams, trigrams, content features or structural features). With composite vectors, features can be grouped in a tree-structured way. See Appendix C for a longer discussion on this subject.
- **Sub-vector sharing.** It is often the case that multiple objects share a set of common features. Thanks to composite vectors, it is easy to share the common corresponding sub-vectors.

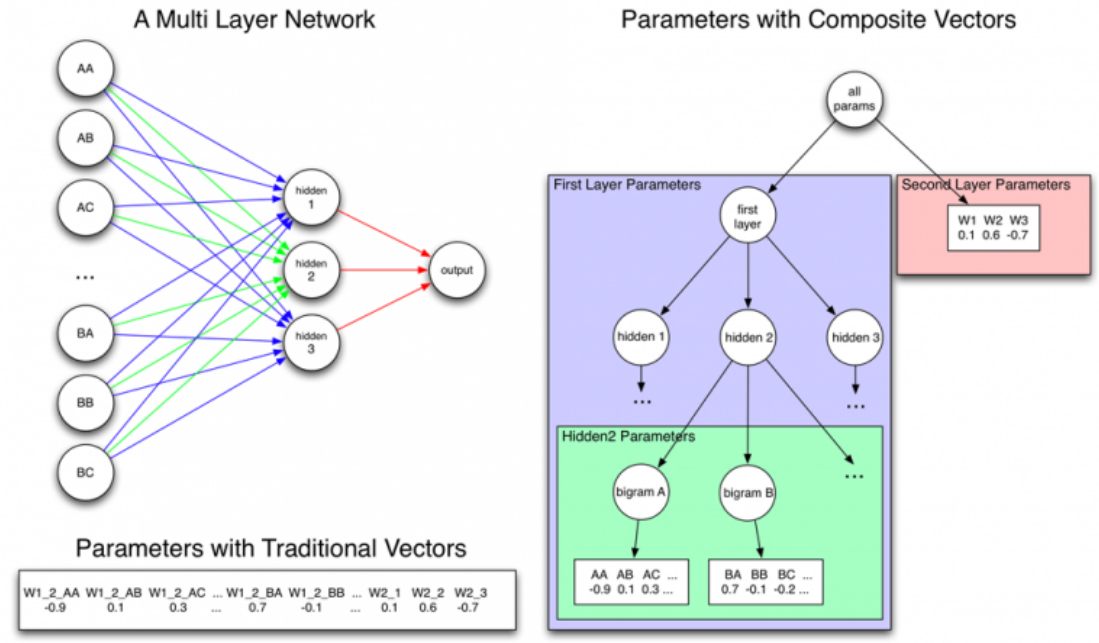


Figure A.4: **Nieme composite vectors.** This figures illustrates a use case of composite vectors. Left: a multi-layer network, where each edge correspond to an element of the parameter vector. Left-bottom: the classical way to represent vectors – all the parameters are grouped into a flat structure. Right: a composite vector that represents the parameters in a tree-structured way. The root node corresponds to the parameters of the multi-layer network. These parameters are composed of two sub-vectors: parameters for the first layer and parameters for the second layer. The first layer is itself decomposed into one sub-vector per hidden node. Even the input features can be represented as composite vectors. Here we have two such groups of features: "bigram A" and "bigram B".

- **Sub-linear dot products.** The last but not the least: when performing dot products between two composite vectors, computations can be pruned each time a sub-vector appears on one side but not on the other. Furthermore, in the case of sub-vector sharing, sub dot-product results may be cached and reused in multiple computations.

A.3 Decision Processes

NIEME provides a set of base classes in C++ to implement large discrete MDPs, such as those induced by CR-algorithms. An MDP is implemented by inheriting these classes to define states, actions, reward, transitions, action values/costs and feature generators. Once this work has been done, NIEME provides the graphical user interface and the scripting environment to manipulate the MDPs. Figure A.5 illustrates the user interface, which features interactive navigation in the MDPs. Figure A.6 illustrates the scripting environment, which gives access to all the learning methods developed in this manuscript.

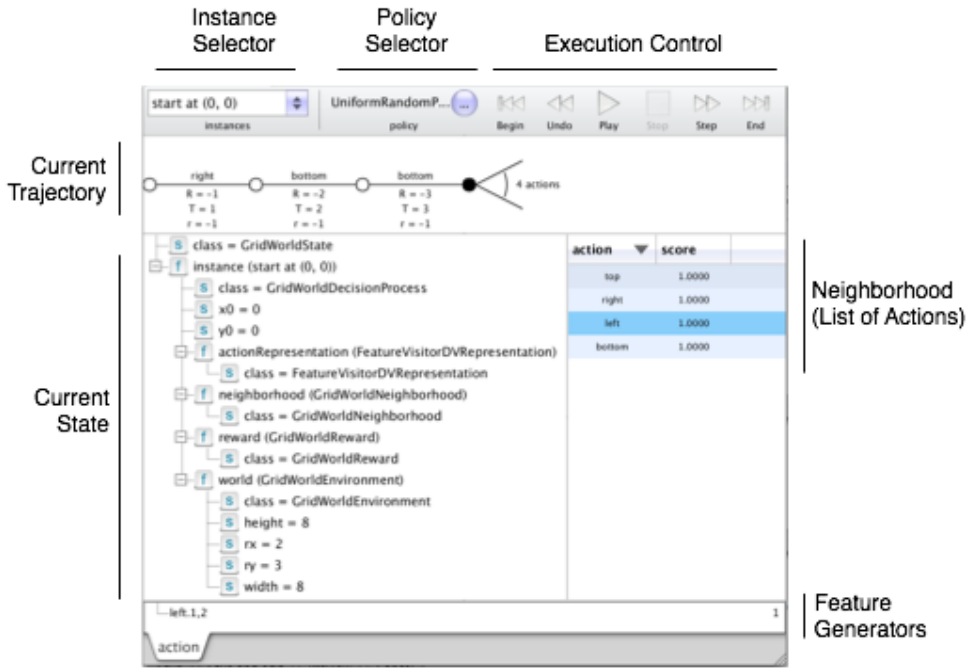


Figure A.5: **Screenshots of Nieme's explorer.** Nieme explorer is the graphical user-interface to visualize and to manipulate objects manipulated by Nieme. Top: use cases of the explorer. Bottom: the user interface to interactively move into MDPs.

```
import nieme
from nieme import Graphs

graphs = Graphs.loadGraphData("graph.nodes", "graph.edges")
decisionProcesses = Graphs.createDecisionProcesses(graphs)

# run a random policy
policy = Policy.createRandom()
policy.run(decisionProcesses)

# verbose output
policy.addConsoleLogger(3).run(decisionProcesses)

# store information into a Table
table = Table.create()
policy.addTableLogger(table).run(decisionProcesses)

# retrieve mean and stddev of reward
averageReward = table.statistics("reward").getDouble("mean")
stddevReward = table.statistics("reward").getDouble("stddev")

# create the QLearning algorithm with a discount of 0.9
machine = EnergyBasedMachine.create(
    Architecture.createLinear(),
    Loss.createSquare(),
    EBMLearner.createStochasticDescent())

policy = ActionValue.createMachineRealPrediction(machine).greedyPolicy()
qlearning = Policy.createQLearning(policy.epsilonGreedy(0.1), machine, 0.9)
qlearning.run(decisionProcesses)
```

Figure A.6: **Examples of Python commands to manipulate Nieme's MDPs.** The example script give some examples of common manipulation of MDPs.

In the future, the decision-process related features of NIEME may be highly influence by the development of the CR-algorithm programming language. This language is described in the next appendix.

B

CR-algorithm programming language

In order to perform our experiments, we have manually implemented the MDPs corresponding to our CR-algorithms within the Nieme toolbox, described in the previous appendix. This process has several disadvantages: writing an MDP is much less intuitive than writing a CR-algorithm, it requires a huge implementation effort and advanced programming skills and it is error-prone. Now that CR-algorithms reach a certain level of maturity, a natural perspective is to create a programming language to directly implement CR-algorithms. A CR-algorithm programming language should incorporate all the information necessary to generate the corresponding MDPs: *choose* and *reward* instructions, and constructions to define action values/costs and feature generators. We describe in this appendix one of our on-going work, which is a development of such a language.

We first described technical choices on which our prototype relies (Section B.1). We then detail a full example (Section B.2) and discuss the automatic transformation of CR-algorithms into MDP definitions (Section B.3).

B.1 Technical choices

Extension of an existing language

Programming languages can be created in two ways: developing a full compiler from scratch, or extending an existing language. Clearly, the second option leads to a much smaller development effort. Furthermore, extending an existing language is interesting for three main reasons. Firstly, existing languages generally provide advanced standard libraries (*e.g.* data structures, input/output functions or string manipulation libraries), which may relevant for CR-algorithms. Secondly, extending an existing language eases the task of integration between classical code and CR-algorithm code. It is thus easier to develop a CR-algorithm as an independent software block that can be integrated in larger software. Lastly, choosing a well-known base language can make the learning phase of CR-algorithms easier for new users.

Source-to-source C++ transformation

Multiple languages may be good candidates as base languages for CR-algorithms. Let us cite OCaml[1] that can directly be extended thanks the Camlp4 tool, Python[2] for its simplicity, and the mainstream compiled languages Java, C# and C++ for their huge user communities. For our prototype, we adopted a source-to-source C++ transformation approach for the following practical issues:

[1] `http://caml.inria.fr/ocaml/`
[2] `http://www.python.org/`

- Many programmers know C++ and have existing code written in C++.
- Since MDPs in NIEME are written in C++, using C++ as base-language makes it easier to connect CR-algorithms to the existing architecture of NIEME.
- C++ makes it possible to produce very efficient code. In particular, its capability to inline source code can be exploited in order to produce efficient combinations of CR-algorithms and corresponding policies.
- A clear drawback of C++ is that it is a complex language, which is known to be hard to parse, to manipulate or to extend. We thus overviewed several existing projects focusing on these problems. One of the most advanced and useable such project is called Synopsis[3]. This projects includes a preprocessor and a parser of C++, a set of classes to represent abstract syntax trees, a visitor mechanism to implement operations on these abstract syntax trees and symbol/scope analysis tools. All these components can be extended to extend the C++ language.

Our prototype consists in three parts:

1. A CR-algorithm Synopsis-based compiler that transforms CR-algorithm programs written in extended C++ into classical C++. In addition to those available in classical C++, the CR-algorithm language provides a set of constructions to deal with CR-algorithms definitions, *choose* and *reward* instructions, action-values definitions and feature-generators definitions.
2. Glue code between the C++ code generated by the compiler and NIEME. Since we want to use our existing architecture NIEME, we developed *adaptator* code to define NIEME MDPs given the output of the compiler.
3. The last part of the prototype is the existing code from NIEME on which the experiments described this manuscript relies.

B.2 An example

We consider the left-to-right sequence labeling CR-algorithm, given in CR-algorithm 3, Section 5.1. The code of to the whole CR-algorithm program, including the feature generator and the action value function, is given in Figure B.1. We now describe each component of this program.

• **crAlgorithm (line 1)** The **crAlgorithm** keyword declares a new CR-algorithm. A CR-algorithm contains classical C++ code augmented with *choose* and *reward* instructions, feature generators and action values. A **crAlgorithm** definition closely resembles to a classical function definition: it has a return type, a set of parameters and a body. In our example, labels are represented with the $std::string$ type, sequences of labels have the $std::vector < std::string >$ type, the set of possible labels is a set of strings ($std::set < std::string >$) and the input sequence is a vector of feature vectors ($std::vector < DoubleVectorPtr >$). Our CR-algorithm returns a sequence of labels ($std::vector < std::string >$). Since the correct sequence *ycorrect* can either be known (training examples) or unknown (other examples), this parameter is optional.

[3] http://synopsis.fresco.org

```
crAlgorithm std::vector<std::string> leftRightLabeling(
                const std::vector<DoubleVectorPtr>& x,
                const std::set<std::string>& labels,
                const std::vector<std::string>* ycorrect = NULL, /* training examples */
                size_t contextSize = 1)
{
  std::vector<std::string> ypred(x.size());
  size_t t;

  featureGenerator leftRightFeatures(const std::string& choice) {
    featureScope (choice) {
      featureScope ("content") {
        featureSense (x[t]); /* content features */
      }
      featureScope ("structural") {
        for (size_t delta = 1; delta <= contextSize; ++delta)
          featureScope (delta) {
            if (t >= delta)
              featureSense (ypred[t - delta]); /* previous label feature */
            else
              featureSense ("N/A");  /* N/A label feature */
          }
      }
    }
  }

  actionValue optimalValues(const std::string& choice) {
    return ycorrect && choice == (*ycorrect)[t] ? 1.0 : 0.0;
  }

  for (t = 0; t < x.size(); ++t)
  {
    ypred[t] = choose <std::string> (labels, optimalValues, leftRightFeatures);
    if (ycorrect && ypred[t] == (*ycorrect)[t])
      reward 1.0;
  }
  return ypred;
}
```

Figure B.1: **An example CR-algorithm program supported by our prototype.** This example implements the left-to-right sequence labeling presented in Section 5.1. All the keyword of the CR-algorithm programming language are shown in blue.

• **Core of the CR-algorithm (lines 31–37)** The core of the CR-algorithm performs the left-to-right labeling loop. At each iteration of the loop, it chooses a label for *ypred*[*t*] and eventually gives a corresponding reward.

• **choose (line 33)** The **choose** keyword denotes a *choose* expression. Its syntax is the following:

$$\textbf{choose} < choose\_type > (choices, param_1, \ldots, param_n)$$

where *choices* is the set of possible choices to choose among. *choices* is any C++ container[4] (such as *std::vector* or *std::set*). *choose_type* is the type of the elements of *choices*. $param_1, \ldots, param_n$ is a set of parameters that are either feature functions or action values. In our example, we choose among the set of *labels*, with the *optimalValues* and the *leftRightFeatures*. The value of a **choose** expression is one of the element of *choices*. It can thus be directly be assigned to *ypred[t]* in our example.

• **reward (lines 34-35)** If the correct sequence *ycorrect* is known, we can compute a reward corresponding to the previously made choice. In our example, we give a reward of +1 for each correctly predicted label.

• **actionValue (lines 27–29)** The **actionValue** keyword declares a new state-action value function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Such a function can compute any scalar number given the current state and one of the possible choices. The current state is composed of all the parameters and variables of the current **crAlgorithm**. The selected choice is given as a parameter of the **actionValue** function. By default, an action-value function should be maximized. When using CR^{ank} , action values are inverted and normalized in order to define the action cost function:

$$c(\mathbf{s}, \mathbf{a}) = \left(\operatorname*{argmax}_{\mathbf{a}' \in \mathcal{A}_\mathbf{s}} Q(\mathbf{s}, \mathbf{a}') \right) - Q(\mathbf{s}, \mathbf{a})$$

In our example, the *optimalValues* function returns +1 for good labels and 0 for bad labels[5].

• **featureGenerator (line 10–25)** Feature generators are functions that generate a set of active features given a state-action pair. Formally, feature generators define functions $\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$. Finding an efficient way to express feature generators is key challenge. The solution we propose here is the result of several design and implementation experiments that are described in Appendix C.

Similarly to action value functions, feature generators depend both on the current state (all the parameters and the variables of the CR-algorithm) and on a possible choice, which is given as a parameter. Features are created and organized in a tree-structured way through two constructs called **featureScope** and **featureSense**. **featureScope** defines a group of features and its syntax is the following:

$$\textbf{featureScope}(groupIdentifier)\ \{\ body\ \}$$

[4]More precisely, the choices can be representation by any class defining an *iterator* type and *begin()* and *end()* functions to create the iterations.

[5]Note that the action-value function and the reward may seem redundant here. Remember that action-values are only hints that are given to the learning system, while rewards specify the learning goal. In left-to-right sequence-labeling, we have access to the OLP and translate this knowledge into the action-value function. This is a very special case, since the OLP is unknown in many CR-algorithms. Action-values can then be *not-too-bad* heuristic to help the learning process that maximizes the rewards.

Where *groupIdentifier* is an identifier (a string or an integer) and *body* is feature generation code. **featureSense** declares an active feature and its has multiple possible syntaxes:

featureSense($featureIdentifier$) or
featureSense($featureIdentifier, featureValue$) or
featureSense($aVectorOfFeatures$)

where $featureIdentifier$ is the name of the feature and $featureValue$ is its value. If no value is specified, a value of 1 is assumed by default. The last syntax declares the whole set of features contained in the vector $aVectorOfFeatures$.

In our left-to-right sequence labeling example, we use content features and structural features. Content features where defined in the following way:

$$f_{l,\ldots}(\mathbf{s},\mathbf{a}) = \begin{cases} 1 & \text{if chosen label} = l \wedge \text{input features in } \mathbf{x}_t \\ 0 & \text{otherwise} \end{cases}$$

Structural features where defined in the following way:

$$f_{l,l',\delta}(\mathbf{s},\mathbf{a}) = \begin{cases} 1 & \text{if chosen label} = l \wedge \hat{\mathbf{y}}_{t-\delta} = l' \\ 0 & \text{otherwise} \end{cases}$$

A first remark on these features, is that they all share a "if chosen label = l" part. For a given state-action pair, all the active features thus share the same l value, *i.e.* they all belong to a group of features "if chosen label = l". This group is created thanks to the **featureScope** instruction line 11. Since we have two kinds of features, we create feature subgroups for content features (line 12–14) and structural features (line 15–24). Line 13 assumes that the input features are already stored in the vector $x[t]$ and declares all the features contained by this vector with the **featureSense** keyword. A key property the **featureSense** and **featureScope** constructs is that generated features are both identified by their name and by the scope they belong to. Thanks to this mechanism, line 13 can be seen as transforming the set of features $x[t]$ into a set of content features from the group "if chosen label = l". The structural features are grouped per δ value (line 17). For each possible value of δ, we create a feature for the corresponding $\hat{\mathbf{y}}_{t-\delta}$ label (line 19) or we create the *N/A* special feature to denote elements that are before the beginning of the sequence (line 21).

B.3 CR-algorithms transformation

This section describes the core of our prototype: the CR-algorithm compiler. The compiler takes a CR-algorithm program as input and transforms it into classical C++ code. CR-algorithm programs are written in extended C++, whose grammar is defined formally in Figure B.2. The main task of the compiler is to convert each input CR-algorithm into a C++ class. Each parameter and local variable of the CR-algorithm becomes a member of this generated class. Furthermore, the generated class provides the following member methods:

- a constructor to initialize the CR-algorithm with a given set of parameters (*e.g.* x, labels, ycorrect and contextSize in our example).
- a copy-constructor that can be used to clone the current state of a CR-algorithm.

```
definition ::= classical-C++-definition
             | cralgo-definition

statement ::= classical-C++-statement
             | reward-statement
             | action-value-statement
             | feature-generator-statement
             | feature-scope-statement
             | feature-sense-statement

expression ::= classical-C++-expression
             | choose-expression

cralgo-definition ::= crAlgorithm identifier ( function-parameter ) function-body

reward-statement ::= reward expression

action-value-statement ::= actionValue local-state-action-function
feature-generator-statement ::= featureGenerator local-state-action-function
local-state-action-function ::= identifier ( function-parameter ) function-body

feature-scope-statement ::= featureScope ( function-arguments ) function-body
feature-sense-statement ::= featureSense ( function-arguments )

choose-expression ::= choose < type_id >  ( function-arguments )
```

Figure B.2: **Extended grammar for the CR-algorithm programming language.** This figure gives the set of extensions that are implemented in our prototype language for CR-algorithms. The rules are classical BNF grammar rules. Terminal symbols are shown in bold.

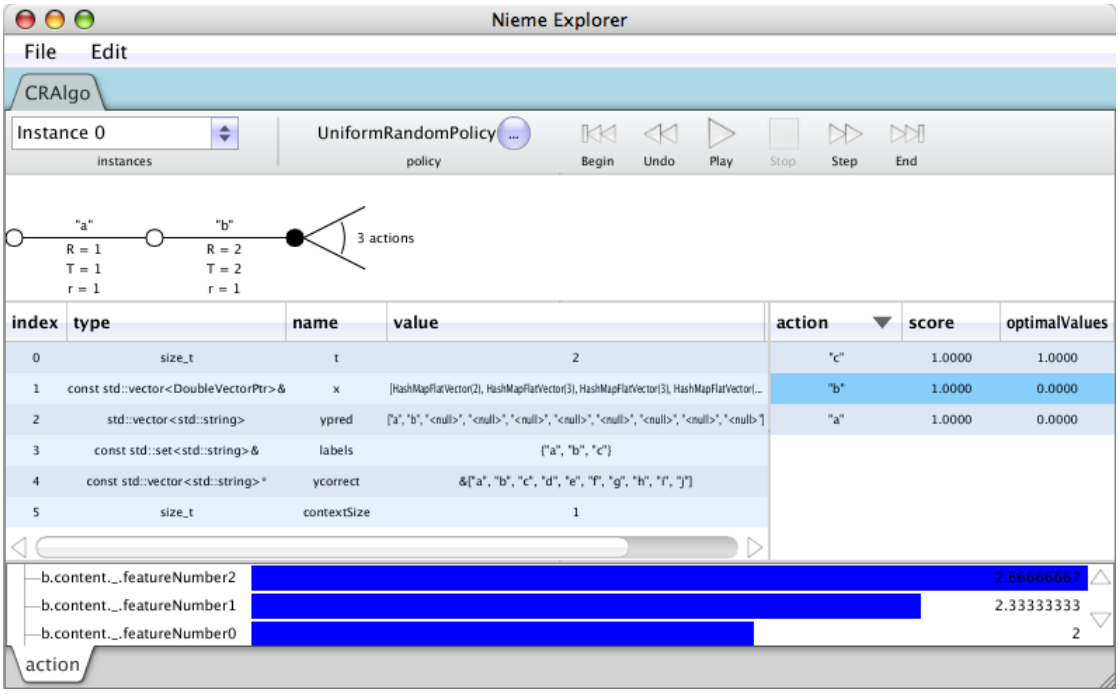


Figure B.3: **Nieme explorer on an automatically generated MDP.** This screenshot shows the MDP manipulation tool of Nieme on an MDP, which was generated automatically from the CR-algorithm program given in Figure B.1.

- introspection methods to enumerate the set of variables composing the state of the CR-algorithm. These functions are used to display the state of CR-algorithms in the graphical user interface.
- one method per feature generator and one method per action-value function
- a *run* method to transmit a choice to the CR-algorithm and execute it until the next *choose* occurs. In order to generate the *run* method, the CR-algorithm must be split into pieces of code separated by *choose* instructions. We found a surprisingly simple solution to this problem, which is based on *label* and *goto* statements.

This generated class makes it possible to define an MDP in Nieme and to navigate freely into this MDP. In particular, states may be cloned, which makes it possible to try multiple possible executions, to backtrack and so forth. Figure B.3 shows the user interface of Nieme on the automatically generated MDP from the CR-algorithm program given in Figure B.1.

For the moment, our prototype is far from being complete. Several important features have not been implemented yet, such as error detection and reporting, type checking or support for variables defined in local scopes. Continuing the compiler until reaching a fully useable programming language is one of our main short-term perspectives.

C

Overview of our work on feature generators design

Finding an efficient way to express feature generators is key challenge. We performed several design and implementation experiments on this question. This appendix briefly summaries this work, which directly motivates the **featureScope**/**featureSense** solution presented in Section **??**. Here are the various solutions to express feature generations that we identified:

• **Function that returns a dense vector.** This is probably the most classical approach, which consists in storing the feature vector $\mathbf{x} \in \mathbb{R}^d$ into a dense vector. This approach has a major drawback: each feature must be associated to a manually selected index (dimension number) and the total number of features must also be computed by hand. Furthermore, dense vectors are not efficient *w.r.t.* sparse descriptions which are common in text processing, structure processing or CR-algorithms.

• **Function that returns a sparse vector indexed with integers.** This is a common approach to deal with sparse features. Instead of storing all the features in $\mathbf{x} \in \mathbb{R}^d$, only the active features are stored into a sparse data-structure. Such data-structure may be a vector of (index, value) pairs or a hash-table mapping indices to values. Similarly to the previous approach, each feature must be associated to a manually selected index in this approach.

• **Function that returns a sparse vector indexed with strings.** Instead of indexing features with integer indices, an alternative consists in using feature identifiers represented by strings. In this approach, it is not necessary to manually define an arbitrary mapping from features to indices. Instead, this mapping is performed implicitly. Furthermore, representing features with strings helps to construct understandable debugging or visualization tools for features and learning machines.

• **Function that returns a composite vector.** In many applications, the set of possible features has some hierarchical structure. For example, there may be different feature groups (*e.g.* content features and structural features) and each feature group may itself be decomposed into sub-groups. Composite vector is an original idea implemented in Nieme, which makes it possible to explicit this structure. A composite vector is either a classical vector (a mapping from strings to scalar values) or a vector composed of sub-vectors. Formally, composite vectors can still be seen as elements of $\mathbb{R}^d$ and all the classical operations on vectors can be implemented on composite vectors. Composite vectors provide a wide range of advantages over classical vectors

as discussed on the website of NIEME[1]. Notably, they ease the task of debugging and visualizing features and learning machines, they enable sub-vector sharing and feature-generation code reuse. This last point is particularly interesting. Feature groups may be nested as much as wanted. This makes it possible to decompose the feature-generation task into several different functions, where each functions only focuses on a particular kind of features. It is thus possible to write a set of base feature-generators that can be then reused in several different places.

• **Function that transmit features to a feature visitor.** One the main drawbacks of the composite vector solution is the cost induced by complex data-structures. Practice with NIEME showed that, when dealing with CR-algorithm, a huge amount of running time ($\approx 20\%$ of cpu-time) was spent in allocating and freeing these structures, while manipulating $\phi(\mathbf{s}, \mathbf{a})$ values. A central idea to avoid dealing with complex data-structures is to store feature values in a vector only if its really necessary. Indeed, in many cases, feature vectors are only used for a single operation that does not require to create the feature vector in memory. For example, a dot-product $\langle \theta, \phi(\mathbf{s}, \mathbf{a}) \rangle$ can be computed directly while enumerating the features, without creating the full vector in memory.

We introduced an original design pattern for feature generators in NIEME, called *feature visitors*[2]. Instead of considering feature generators as functions that compute a data-structure storing feature values, we propose to view feature generators as factory of features. The *visitor* represents a client, which is interested by the features generated by the feature generator. Visitors are passed as arguments when executing a feature generation. For each generated feature, the feature generator calls a method of the visitor to transmit the name and the value of the feature. Visitor can then use the generated features in many different ways. Here are some examples of feature visitors implemented in NIEME:

- Dot-product visitor (parameterized by θ, returns a scalar): $d \leftarrow \langle \theta, \phi(\mathbf{s}, \mathbf{a}) \rangle$
- Add-weighted visitor (parameterized by θ and α): $\theta \leftarrow \theta + \alpha\phi(\mathbf{s}, \mathbf{a})$
- Store visitor (returns a composite vector) $\mathbf{x} \leftarrow \phi(\mathbf{s}, \mathbf{a})$
- Save to file, print to screen, display in the user-interface, ...

Each of these feature visitors, corresponds to operations that can be done with $\phi(\mathbf{s}, \mathbf{a})$ without creating the data-structure representing $\phi(\mathbf{s}, \mathbf{a})$ in memory. In our experiments with NIEME, implementing the feature-visitor design pattern has led to a major speed-up ($\approx \times 2$ to $\approx \times 50$) on all our learning algorithms.

The **featureScope** and **featureSense** constructions proposed in Section **??** are directly linked to the idea of feature visitors. When translating these constructions into classical C++, **featureScope** and **featureSense** become calls of member functions of a feature-visitor.

[1] `http://nieme.lip6.fr/Nieme/CompositeVectorFeature`

[2] The idea of the **featureSense** instruction come from the work of [XX cite dan roth]. Dealing with tree-structured feature vectors and **featureScope** constructions is original in our work. The idea of feature-visitors is original.

D

Summary of Notations

Supervised Learning

Inputs	$\mathcal{X}$	the input space
	$\mathbf{x}$	an input in $\mathcal{X}$
	$\mathbf{x}_j$	the j-th component of a sequential input
Outputs	$\mathcal{Y}$	the output space
	$\mathbf{y}$	an output in $\mathcal{Y}$
	$\mathbf{y}_j$	the j-th component of sequential output
	$\hat{\mathbf{y}}$	a predicted output
	$\hat{\mathbf{y}}_j$	the j-th component of a predicted sequential output
	$\bar{\mathbf{y}}$	a partial output
	ϵ	an empty partial output
Examples	$\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$	a learning problem: a distribution over input-ouput pairs
	$D = (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})_{i\in[1,n]}$	a training dataset: a set of *i.i.d.* samples from $\mathcal{D}_{\mathcal{X}\times\mathcal{Y}}$
	$\mathbf{x}^{(i)}$	the i-th training input
	$\mathbf{y}^{(i)}$	the i-th training output
Training	$\phi : \mathcal{X} \to \mathcal{R}^d$	an input description function
	$\phi : \mathcal{X} \times \mathcal{Y} \to \mathcal{R}^d$	a joint input-output description function
	$\mathbb{R}^d$	the parameters space
	θ	the parameters being learned
	$\mathbf{c}$	a vector of costs associated to ranking alternatives
	$\mathbf{c}_j$	the j-th component of a vector of costs
	α	the learning rate parameter in $\mathcal{R}^+$

Sequential Decision Making

States	$\mathcal{S}$	the state space
	$\mathbf{s}$	a state in $\mathcal{S}$
	$\mathbf{s}_t$	the state at time step t

Actions	$\mathcal{A}$	the action space
	$\mathcal{A}_{\mathbf{s}}$	the set of actions available in state $\mathbf{s}$
	$\mathbf{a}$	an action in $\mathcal{A}$
	$\mathbf{a}_t$	the action at time step t
Transition, Reward	T	the transition function
	r	a reward value
	r_t	the reward perceived at time step t
	γ	the discount factor in $[0, 1]$
Policy	$\pi : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$	a stochastic policy
	$\pi : \mathcal{S} \to \mathcal{A}$	a deterministic policy
	$V^{\pi} : \mathcal{S} \to \mathcal{R}$	the state value function of π
	$Q^{\pi} : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$	the action value function of π
	π^*	an optimal policy
	V^*	the optimal state value
	Q^*	the optimal action value

CR-algorithms

Inputs	$\mathcal{I}_{\mathcal{X}}$	the space of normal inputs of a CR-algorithm
	$\mathbf{i}_{\mathbf{x}}$	normal inputs values in $\mathcal{I}_{\mathcal{X}}$
	$\mathcal{I}_{\mathcal{Y}}$	the space of training inputs of a CR-algorithm
	$\mathbf{i}_{\mathbf{y}}$	training inputs values in $\mathcal{I}_{\mathcal{Y}}$
	$\mathbf{i}^{(i)}$	the i-th training input
MDP	$\mathcal{P}$	a CR-algorithm
	$MDP(\mathcal{P}, \mathbf{i})$	the MDP corresponding to CR-algorithm $\mathcal{P}$ with input param
	$\mathbf{s}^{initial}(\mathcal{P}, \mathbf{i})$	the initial state of $MDP(\mathcal{P}, \mathbf{i})$

Math Operators

$\mathbb{1}\{b\}$	the indicator function whose value is 1 if b and 0 otherwise
$\langle \mathbf{a}, \mathbf{b} \rangle$	the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$
$\mathsf{E}_B\{A\}$	the expectation of A *w.r.t.* B