UNIVERSITÉ LIBRE DE BRUXELLES
Faculty of Sciences
Department of Computer Science

Reinforcement Learning in Complex Environments: Evaluating Algorithms on Image Classification

Denis Steckelmacher

Supervisor :
Dr. Peter Vrancx
Promotor :
Prof. Tom Lenaerts

Mémoire présenté en vue
de l'obtention du grade de
Master en Sciences Informatiques

Academic year 2015 - 2016

Thanks

Thanks to Dr. Peter Vrancx for his advice, support, ideas and comments, and for having proofread this thesis so many times. His invaluable feedback allowed me to define the subject of this thesis and he often pointed me to the solution I was looking for.

Thanks to Prof. Dr. Ann Nowé and the whole COMO lab at the Vrije Universiteit Brussel for their support and their general kindness. Doing my internship at the COMO lab allowed me to discover the world of research and to participate in a scientific conference. Very interesting discussions with Pieter Libin, Felipe Gomez Marulanda, Anna Harutyunyan, Tim Brys and many other researchers allowed me to greatly improve this thesis and the experiments it describes.

Thanks to all the members of the jury, Dr. Peter Vrancx, Prof. Gianluca Bontempi, Prof. Joel Goossens and Prof. Tom Lenaerts, for reviewing and evaluating this master thesis.

I would also like to thank the Université Libre de Bruxelles and its Computer Science department for the 5 beautiful years of my studies, diverse and interesting lectures and the general organization of the Computer Science curriculum.

Thanks to Maryka Peetroons, student secretary of the Computer Science department, for having perfectly well handled all sorts of documents, forms and reports, organized the exams, forwarded important messages to the students and having accomplished many other tasks without which being a student at the Computer Science department would have been much less pleasant.

I also thank all my friends with whom I worked on projects, studied, followed the lectures and enjoyed some free time during my studies. In random order: Daniele, Hadrien, Florian, Jonathan, Nicolas, Arnaud, Jean, Ségolène, Philippe, Thomas, Robert, Simon, Loïs, Kamil, Antoine, Steven, Marien. Thanks to non-CS students who made these 5 years even more joyful: Antoine, Florina, Pierre, Anaïs, Laura, Nathalie, Amin, Sharon, Madli, Corentin, Simon.

Thanks to my family for having provided me all the love and comfort I need, and for having allowed me to work on a computer for 14 years.

Finally, thanks to Hélène Plisnier for her emotional and scientific support. Our discussions allowed me to design some of the experiments described in this thesis, and to find several solutions to my problems.

Introduction

Reinforcement Learning is used whenever an agent has to learn which action to perform in any situation in order to maximize a reward. This framework, used to learn control policies, has applications in robotics [MPR+02, DN10], industrial machines [WO12], computer game playing [MKS+15, SHM+16] and automatic algorithm development [ZS15], among others.

In recent years, Reinforcement Learning has been applied to increasingly complex and challenging problems. This complexity appears at different levels, as the environment in which the agent behaves can be in any of possibly millions distinct states, each state is described using a large number of variables and features, and action sequences (from initial state to goal) are becoming increasingly long, if not infinite in *continuous* or *life-long* Reinforcement Learning.

Reinforcement Learning algorithms have largely improved over the years and are now able to tackle complex problems. They also have been able to leverage advances in Supervised Learning, for instance Deep Learning [MKS+15], in order to allow larger environments and more complex state descriptions.

The overall goal of this master thesis is to explore alternatives to current algorithms and to evaluate their applicability to Reinforcement Learning in complex environments. The algorithms being evaluated have been chosen for the promising results obtained by their authors and their mathematical properties. They cover three *layers* of Reinforcement Learning agents: the environment, the learning algorithm itself, and the function approximation used for knowledge representation.

Problem statement

As outlined in the previous paragraphs, modern Reinforcement Learning faces three challenges:

1. Environments are very large and very complex. Discovering their properties and exploring them is difficult. Agents must also be able to maintain their knowledge over long time periods in order not to forget information about rarely-visited regions of the environment.

2. The tasks to be accomplished by Reinforcement Learning agents are very complex, with low success rates (only a small set of action sequences lead to a reward). A large number of actions may be required before reaching a goal state.

3. Complex environments and advanced learning algorithms need substantial computing resources. Training agents can take several days even on multiple computers [MKS+15].

A fourth challenge, which is outside the scope of this master thesis, is partially observable environments. In most applications, the agent is not able to completely observe its environment. For instance, it cannot see behind itself, or through walls. This prevents the agent from choosing actions based on a complete knowledge of its environment and forces it to keep track of large amount of data and/or guess some information. Several results regarding partially observable environments are presented in Section 1.5, but Reinforcement Learning in these environments is still considered a difficult and unsolved problem.

This master thesis studies the three challenges listed above and evaluates how novel or uncommon algorithms allow to tackle them. The problem to be solved is an image classification task, simple enough to allow reasonably quick experimentation[1] and make analysis of the results possible, and complex enough to allow measuring how different algorithms scale. Classification is not a problem commonly solved by Reinforcement Learning, but is easy to evaluate using standard measures of accuracy and is easy to express. The fact that it is not a traditional Reinforcement Learning problem also makes it more challenging and interesting to study.

The algorithms being evaluated in this master thesis are the following ones:

Visual Attention (Challenge 1)

Recent research has explored the concept of attention. Instead of trying to handle all the complexity of the data at once, for instance by looking at all the pixels of an image, agents learn how to focus their attention on specific parts of their input depending on their needs. This allows to reduce the amount of data to be processed at once, but requires several iterations per classification (e.g. looking at a succession of glimpses in an image). While most work consists of attending at specific parts of images, as the human eye would do [MHGK14, Ran14, BMK14, SFR14, BVB+96], the method is also applicable to voice recognition [CBS+15] and general algorithmic tasks [GWD14, ZS15].

Research on attention currently focuses on classification tasks (mostly of images) and are built on recurrent neural networks. The classifier learns to focus its attention by progressively becoming better at that, using standard neural network training algorithms. This master thesis explores to which extend Reinforcement Learning can be used to learn how to focus attention. A Reinforcement Learning formulation of attention-based classification is given in Chapter 5.

Gaussian Mixture Models (Challenge 1, long-term memory)

All Reinforcement Learning agents, except the simplest or problem-specific ones, use function approximation to represent their knowledge. Good results and convergence proofs have been obtained for linear regression [TV97], but learning based on neural networks is known not to always be possible [BM95].

[1] Not requiring weeks of computations on a supercomputer each time a parameter is changed.

Ormoneit suggests that neural networks and linear regression may not be suited to Reinforcement Learning, and advocates the use of kernel-based function approximation [Orm02]. In his work, Gaussian kernels are used and evaluated. Comparably, Heinen proposes a Gaussian Mixture Model applied to Reinforcement Learning and provides promising results [HE10]. Chapter 4 extends this algorithm and Chapter 7 confirms its applicability to Reinforcement Learning and shows that it performs better than neural networks in our experiments. Gaussian Mixture Models also have the nice property of maintaining their knowledge even in rarely-visited parts of the environment, whereas neural networks tend to progressively forget information in these parts[2] [Fre91].

Distributed Reinforcement Learning (Challenges 2 and 3)
The algorithms presented in Chapter 6 are more diverse than the ones outlined here-above. They have been mostly built by assembling a set of state-of-the-art Reinforcement Learning algorithms targeted at better exploration in complex environments, improved learning based on limited experience, and overall more successful learning. This chapter also proposes to distribute Reinforcement Learning algorithms across agents and suggests ways of exchanging data between the agents so that each of them learns faster than if it was alone.

Structure of this document

This master thesis is divided in three parts. The first one presents some background knowledge in Supervised Learning, Reinforcement Learning and Attention Models. The second part presents the contributions of this master thesis, original or improved algorithms developed in order to tackle the three challenges listed in the previous section. Finally, the last part concludes this thesis with general results (assembling and mixing the three contributions) and a discussion.

Chapter 1 gives some background in Reinforcement Learning and presents the algorithms on which this master thesis builds. Chapter 2 introduces some Supervised Learning concepts and notations, then compares three function approximation models. Chapter 3 presents Attention Models, along with their current training algorithms.

Chapter 4 presents a function approximation algorithm specifically designed for Reinforcement Learning, the Incremental Gaussian Mixture Model, based on [HE10]. Chapter 5 proposes a Reinforcement Learning environment built on attention, and Chapter 6 describes distributed and scalable Reinforcement Learning algorithms.

Finally, Chapter 7 assembles and evaluates the three contributions, and Chapter 8 discusses the results of this thesis.

[2] Most Reinforcement Learning algorithms train their model incrementally.

Contents

Part I

Background

Chapter 1

Reinforcement Learning

Reinforcement Learning allows an agent to learn to control a dynamic environment, and has close ties with control theory. In discrete-time Reinforcement Learning, as studied in this master thesis, the agent receives at each time-step an observation s_t of its environment and chooses an action a_t to be executed. Once the action is executed, the agent receives a reward r_{t+1} and its next observation, s_{t+1}. Figure 1.1 provides a graphical view of Reinforcement Learning.

Compared to Supervised Learning, Reinforcement Learning has much less information to work with. Instead of complete input-output samples, only the reward signal is available [MDG09]. Moreover, delayed rewards are often used, which means that the agent may have to wait some time before knowing if its action is good or bad (by receiving a +1 reward after having solved a complete puzzle without having received any reward before that).

The actions executed by the agents are drawn according a policy π, that gives at each time-step the probability of executing any of the available actions (using the current state observation or an history of these observations, for instance). The goal of the agent is to learn the optimal policy $\pi*$, that allows the agent to maximize its discounted cumulative reward $R = \sum_{t=0}^{\infty} \gamma^t r_t$, with $0 \leq \gamma \leq 1$ a scalar that allows to give more weights to rewards obtained early.

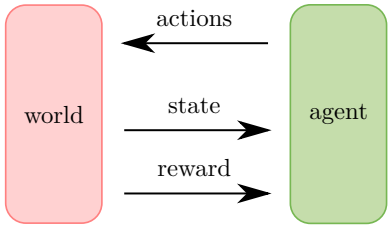

Figure 1.1: The agent performs an action in the environment, which changes its state and provides a reward.

1.1 Discrete Markov Decision Processes

Discrete Markov decision processes are the most well-studied Reinforcement Learning problems, but also the ones that make the most assumptions and are the most restrictive.

In Markov decision processes (MDPs), the agent makes the hypothesis that the reward it gets after taking an action solely depends on the action taken and the state it was in when it took the action. Discrete MDPs also require that the state space is discrete and that the action space is finite. More formally, a discrete MDP is modeled as follows [SB98, WO12]:

- S is the finite set of states in which the agent can be. The agent is able to sense in which state it is.
- A is the finite set of actions that can be taken. The hypothesis is made that all the actions can be taken in all the states. Actions which should not have been possible can either give a very negative reward or do nothing.
- $T(s, a)$ is the *transition function* that returns a probability distribution over the next states in which the agent may end up when it takes action a in state s.
- $R(s, a, s')$ is the *reward function* that gives the expected reward obtained by the agent when performing action a in state s.

The $T(s, a)$ and $R(s, a)$ functions are used to formally describe a Reinforcement Learning problem but are part of the environment and unknown to the agent. The only way the agent can discover how these functions behave is by taking actions and observing its reward and new state.

Finding the optimal policy can be done using a variety of exact or stochastic algorithms. Because the agent usually interacts with its environment for a small amount of time compared to the amount of information required to build an exact optimal policy, stochastic methods are nearly always preferred [WO12]. The next sub-sections explain how simple stochastic Reinforcement Learning methods work.

1.1.1 Online and batch learning

Different strategies can be adopted when solving MDPs in a Reinforcement Learning context. If the agent learns a policy while interacting with its environment, learning is considered *on-line*. If the agent interacts with its environment, stores the sequence of actions-observations it has done, and then learns from them, learning is considered *off-line* or *batch*.

Batch learning is generally considered easier, because the learning algorithm has complete histories of states and actions to work with. It can iterate over them several times, each time improving its understanding of the system. This allows quick learning without having to interact too many times with the environment, which is handy when interactions with the environment are expensive, represent some risk or take too much time.

Online learning forces the learning algorithm to use each $(s_t, a_t, r_{t+1}, s_{t+1})$ experience tuple at most once. Because any single tuple provides only a small amount of information about the environment and the task to be realized, online learning requires a large amount of interactions with the environment.

Actual applications of Reinforcement Learning usually lie in-between online and offline learning. For instance, experience replay adds small library of past experience tuples to an online Reinforcement Learning algorithm. This library is then used to periodically re-learn from past experience, thus giving more tuples to the learning algorithm without requiring more interactions with the environment [ABB12].

1.1.2 On-policy and off-policy learning

In a Reinforcement Learning context, two policies can be distinguished. The *behavior* policy is the one used by the agent in its environment in order to choose actions. The *learned* policy is the policy that the agent is learning by observing its environment.

If those two policies are kept separate, learning is considered to be off-policy. Usually, the agent will learn the optimal policy while following an exploratory policy (a policy in which it tries some random actions, or actions it has seldom used, in order to discover parts of the environment it has not yet visited). Horde is a Reinforcement Learning framework based on a collection of agents each learning a small and specific part of the problem. This framework takes off-policy learning one step further as it allows any agent to follow the policy of any other agent, while learning its own value function [SMD+10].

If the behavior and learned policies are the same, learning is considered to be on-policy. This allows the agent to immediately make use of what it has learned in order to guide its exploration, and produces policies that take risk and exploration into account, as explained in Section 1.2.2.

1.2 Value Iteration

An agent receives a reward after each action taken, but this reward can be sparse, with the agent receiving no reward most of the time. Furthermore, the agent has to optimize its behavior in order to obtain the maximum cumulative reward, like a child exploring a garden trying to get as much Easter eggs as possible. Sparse reward and optimization for cumulative reward require that the agent plans its actions in order to achieve its goal. This planning is done by storing, for each state, the expected discounted cumulative reward obtainable from this state and following the current policy, what is called its V-Value [WO12]. This is how "promising" a state is.

$$V^{\pi}(s) = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s\right] \tag{1.1}$$

Figure 1.2 shows a grid-world containing a simple maze. The agent receives a reward of -1 for each action that does not lead to the goal, and a reward of $+10$ when reaching the goal. The figure shows how the values of states decrease as they become further from the goal.

1	2	3	4	5	4
10	9	8	7	6	5
1	2	3	4	5	4
0	1	2	3	4	3

Figure 1.2: Simple maze showing the V value of all the states when following the optimal policy (going as fast as possible to the goal). The agent receives a reward of -1 per time step, and $+10$ when reaching the goal. No discounting used ($\gamma = 1$)

Learning the optimal policy mainly consists of discovering which are the values of the states when the optimal policy is followed. Once the agent knows the "best-possible" value of all the states, it can simply try to move from state to state by always going to the state with the highest value.

The problem is that the agent usually does not know to which state an action will take it, so it cannot choose to go to a specific state. This problem is solved using Q-values: values associated with (s, a) state-action pairs and that represent the expected value of taking action a in state s (based on the states in which the agent may end up after having taken the action). The following sections explain how to approximate the Q-values of the states and how to use them for Reinforcement Learning.

1.2.1 Q-Learning

The best-known algorithm used to solve discrete MDPs is Q-Learning [WD92]. The idea is to compute the optimal Q-value of every state-action pair, the expected discounted reward obtained from a given state when taking a given action. The optimal policy is then found by choosing at each state the action with the largest Q-value.

This algorithm approximates the Q-values of the state-action pairs by applying the Bellman equation [Bel56] every time the agent takes an action a_t in state s_t and observes the new state s_{t+1} and the reward r_{t+1} :

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)) \tag{1.2}$$

With α a learning factor (between 0 and 1) and γ the discount factor.

1.2.2 SARSA

Equation 1.2 uses a maximum over the Q-values of the actions in state s_{t+1}, and is an off-policy algorithm [WD92] (it computes the Q values of the optimal policy due to the max, while the agent can follow any policy). An on-policy variation of Q-learning is SARSA, which works in the exact same way but replaces the maximum in Equation 1.2

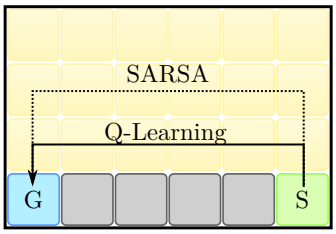

Figure 1.3: Two paths from start to goal taken according to two policies. The plain line shows a short but risky path discovered using Q-Learning. The dotted line corresponds to a longer but safer path. The gray squares represent a pit. The agent receives a very negative reward if it falls into this pit.

with the Q-value of the next state-action pair, and replaces k with t in order to express that the updated Q-values are immediately used at the next time-step:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha(r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)) \tag{1.3}$$

The main difference between policies learned using SARSA and Q-Learning is that SARSA policies are aware of the risk of the exploratory actions. Instead of computing Q-values assuming that the optimal policy is followed, SARSA computes Q-values that correspond to the exact policy followed by the agent, even if random actions are sometimes performed. Regularly choosing random actions allows the agent to explore its environment, discover new states and potentially find new opportunities and improve its policy.

In Figure 1.3, the agent has to go from a start state to a goal state, and must absolutely avoid falling into a pit (gray cells). The agent is slightly stochastic and does not always perform the action dictated by its policy. Moreover, the policy may not be optimal until learning is finished. These two points make the shortest path, found by Q-Learning, very dangerous as it is very close to the pit. The policy found by SARSA is not the optimal one but limits the risk of the agent slipping into the pit, even if some random actions are taken.

1.3 Policy Iteration

Section 1.2 presents algorithm that compute the value of states or state-action pairs and use them in order to select actions. Those algorithms are part of the Value Iteration family as they incrementally refine their approximation of the values they compute.

Another way of learning a policy is to learn it directly. Instead of learning values and using them to choose actions, the agent directly learns a mapping from state to action. A variety of policy iteration algorithms exist, but this master thesis will only introduce two of them: policy evaluation and improvement, and rollout-based policy iteration.

1.3.1 Policy Evaluation and Improvement

This first class of algorithms has nice mathematical properties, but has the disadvantage of requiring temporary value functions in addition to the policy. Learning happens by repeating two steps until convergence: an existing policy is evaluated, which produces a value function, that is then used to build an improved policy. When the improvement from a policy to the next drops below a threshold, learning stops.

Evaluating a policy can be done in a variety of ways. For instance, the policy can be executed in the environment of the agent (if performing actions in the environment is cheap), can be simulated against a model of the environment, or can be evaluated using samples of past trajectories of the agent. The first method uses the real environment and therefore makes the least mistakes, but cannot be used when performing actions is expensive. The second method requires a model of the environment, and inaccuracies in this model introduce errors in the evaluation of the policy. The last method, used in [Lag03], does not require too many interactions with the environment nor a model and performs very well.

Regardless of the exact algorithm used to evaluate a policy, the goal is to compute the empirical Q-value function associated with the policy being evaluated:

$$Q^{\pi}(s,a) = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \mid s_t = s, a_t = a\right] \tag{1.4}$$

Once this Q function is known, a new policy can be inferred:

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \tag{1.5}$$

After each evaluation-improvement cycle, the policy π' is at least as good as π [KB99]. When π' becomes close enough to π, the iteration can stop as a local optimum has been found.

1.3.2 Rollout-based Policy Iteration

Algorithms in this class are much less numerous than the ones described in Section 1.3.1. Instead of evaluating a policy by computing its Q-values, and then using these Q-values in order to build an improved policy, the Q-values are obtained as needed from the environment [LP03] or from a model of it [Hes12] by performing rollouts.

A rollout consists of performing a sequence of actions starting from state s_0 according to a policy π while recording the succession of rewards obtained after each action. The rollout ends either after a fixed number of steps (around 50 in [Hes12]), or when a final state has been reached. The list of rewards obtained at each time step can be used to compute the discounted cumulative reward obtained starting from s_0. By forcing several rollouts to start with action a_0, it is possible to compute an empirical estimation of $Q(s_0, a_0)$ (see Equation 1.6).

$$Q(s_t, a) = \frac{1}{N} \sum_{i=0}^{N} \text{rollout}_\pi(s_t, a) \tag{1.6}$$

$$\equiv \frac{1}{N} \sum_{i=0}^{N} \sum_{k=0}^{\infty} \gamma^k r^i_{t+k} \mid (s^i_{t+k}, a^i_{t+k}, r^i_{t+k+1}, s^i_{t+k+1}, a^i_{t+k+1}, r^i_{t+k+2}, \ldots) \tag{1.7}$$

$$\pi'(s_t) = \underset{a}{\text{argmax}}\, Q(s_t, a) \tag{1.8}$$

The learning algorithm described in Section 6.2 uses rollout-based policy iteration. It is inspired from [LP03] and trains a classifier that maps states to actions.

1.4 Reinforcement Learning with function approximation

When the state-space is large or continuous, storing the Q-values of all the state-action pairs becomes very difficult if not impossible. Furthermore, Q-Learning finds the optimal policy only if the agent visits all the state-action pairs infinitely often. Even visiting millions of state-action pairs once is unfeasible in most real-world problems.

The biggest problem of Q-Learning and derived methods is that they need to compute one value per state or state-action pair. One solution to this problem is to use some form of function approximation in order to generalize Q-values across states.

Function approximation algorithms are described in Chapter 2. In a Reinforcement Learning context, function approximation is used to build the $\hat{Q}(s, a)$ function, which approximates the real $Q(s, a)$. A simple algorithm for doing so is a variation of Neural Fitted Q-Learning [Rie05] and is shown in Algorithm 1. While function approximation allows Reinforcement Learning to scale to large and/or continuous environments, most learning algorithms are not guaranteed to converge if function approximation is used[1].

Algorithm 1 On-line Reinforcement Learning with Q-Learning and function approximation

$\hat{Q}(s, a)$ is a function approximator
$t \leftarrow 0$
Observe s_0
loop
  $a_t \leftarrow \text{argmax}_a\, \hat{Q}(s_t, a)$
  Perform a_t, observe s_{t+1} and r_{t+1}
  $\delta \leftarrow r_{t+1} + \max_a \hat{Q}(s_{t+1}, a) - \hat{Q}(s_t, a_t)$
  $\tilde{Q} \leftarrow \hat{Q}(s_t, a_t) + \alpha\delta$
  Train $\hat{Q}(s_t, a_t)$ towards $\tilde{Q}$
  $t \leftarrow t + 1$
end loop

In Reinforcement Learning, it is often important that the function approximator can be trained in an incremental fashion. For instance, online Reinforcement Learning

[1] Q-Learning with linear regression has been proven to converge [SS92]

algorithms have to update their model after each observation, and having to re-train a regression using all the previous observations would be intractable. Incremental update algorithms exist for regression trees [VK14, LCG11, PS05], Gaussian Mixture Models [HE10, Hei11] and neural networks [BP92].

1.5 Partially observable environments

The version of Q-Learning presented in the last sections makes the hypothesis that the agent is able to observe the state s_t of the environment. This allows it to associate a Q-value to each state-action pair, based on the V-value of the different states.

However, in most situations, the agent is not able to distinguish all the possible states. For instance, it can have a camera that cannot see through walls. Two completely different states, requiring completely different actions, may therefore produce the same observation.

More formally, an environment is considered partially observable as soon as the reward function R cannot be expressed anymore as $R = f(s_t, a_t, s_{t+1})$, with s the observation of the agent. This happens when R depends on some hidden state or on parts of the history of observations.

Several methods tackle the problem of partially observable environments. If the hidden, non-observable state-space has a known structure, the agent can maintain a belief over the states in which it thinks to be [CKL94]. This solution has been well studied and has nice properties, but scales poorly to large hidden state spaces. If there are $|S|$ hidden states, the belief space is $\Re^{|S|}$.

Memory bits consist of adding actions and observations that allow the agent to set (and observe) bits of memory. This allows it to learn when to set a specific bit, and when to use the value of this bit to disambiguate states [PMK01]. This solution has the advantage of being intuitive and to require a very small amount of memory compared to belief spaces, but limits the abilities of the agent if the amount of memory bits is too small. However, if too many bits is available, the Reinforcement Learning problems becomes too complex and the agents does not manage to learn which bit to use in which context.

The most widely used method these days is based on sequences of observations. In order for the agent to use as much information as possible, Q-values are not associated to state-action pairs, but to complete histories of observations. Only a couple of function approximators are able to associate values to sequences of arbitrary length, such as recurrent neural networks presented in Section 2.3. Recurrent neural networks provide state-of-the-art results in partially observable environments as they are able to store useful information for long periods of time without keeping useless information [Bak01, Bak07, ZS15].

Chapter 2

Supervised Learning

Supervised Learning consists of learning a mapping from an input to an output. The input is generally represented as a vector of real values, x, and the output can be of two types: a class or a vector of real values.

When the output is a class, the problem is a *classification*. Classification is used in tasks where a label (from a small, finite set of labels) has to be associated with something, for instance when a program has to recognize hand-written digits (there are only 10 digits).

When the output is one or several real values, the problem is a *regression*. Regression is used for simplifying data, for instance by fitting a curve through a set of (possibly noisy) points so that new points can be predicted. Regression can also be considered as function approximation because $\hat{f}(x)$ is approximated by fitting a model with $(x, f(x))$ samples. Function approximation is often used in Reinforcement Learning as explained in Section 1.4.

Several Supervised Learning algorithms exist, and most of them can be used for classification or regression without any change. The next section presents some of the properties that classifiers and regressors have, and justifies why only a subset of regression algorithms are explored in this master thesis. Then, specific regression techniques are detailed. Section 2.2 presents feed-forward neural networks, Section 2.3 shows how recurrent neural networks compare with feed-forward neural networks, Section 2.4 introduces classification and regression trees, and Section 2.5 presents Gaussian Mixture Models, that have interesting properties compared to neural networks and regression trees.

2.1 Theoretical background

This section presents some simple and high-level background on classification and regression. Inspired from lecture notes [Bon14], it provides some insight to the properties of classifiers, when they are useful, and what they imply.

Classification and regression tasks map an input $x \in \Re^D$ to an output $y \in \Re^N$ (to be general, y is allowed to be a vector of reals, not only a scalar). For regression tasks, y is an output vector that is directly used to solve a specific problem. For instance, y can be a tuple representing the force of wind in two dimensions, while x is a tuple representing

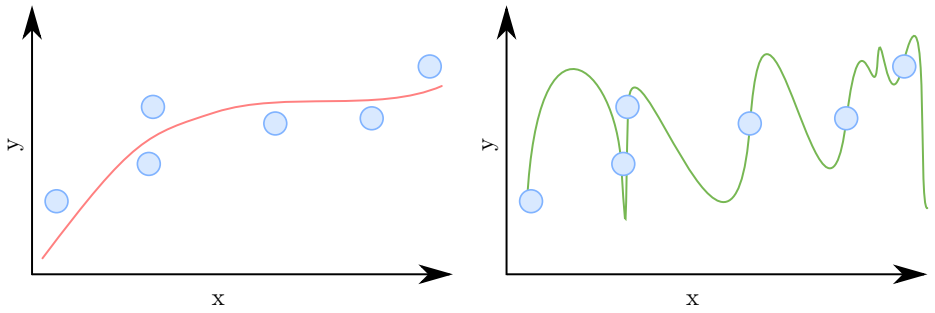

Figure 2.1: On the left, a model fits training points with some inaccuracies, but is probably a good representation of the data. On the right, a model overfits training points. It makes no error at all on the training points but is likely very inaccurate on testing points.

geographical coordinates. For classification tasks, y usually has one element per class, and its i-th element gives the probability that the input belongs to the i-th class.

2.1.1 Training and testing

Directly after creation, a classification or regression model is not able to make any useful prediction. For instance, it could map any x to a null or random y vector. In order to become useful, the model has to be trained on input-output tuples (the training set) with a specific training algorithm.

The quality of a model is computed using different measures of accuracy. The most commonly used one is the mean integrated squared error (MISE), which can be expressed in a simplified form as $E[(f(x) - \hat{f}(x))^2]$. The idea is to compute the average squared error made by the model on previously unseen input-output samples (the testing set). This quantity is always positive. The smaller it is, the better the model is able to predict outputs associated with previously unseen inputs.

2.1.2 Properties of models

Different classification and regression models exist, and they each have different properties, which makes them useful in different contexts. Here is a list of properties that have an impact on how models can be used for Reinforcement Learning.

Global/Local

A global model adjusts its entire output response for any input-output sample presented to it. For instance, training a linear regression model will update its linear parameters, which will change the values it predicts even for inputs far away from the training input. A local model does not exhibit this behavior and can be trained on input-output samples without altering its general behavior.

Aggressive/Non-aggressive

Aggressiveness represents the overall speed at which a model adapts to new training

samples [HE10]. Neural networks, for instance, are not aggressive as they need to perform several gradient descent steps (possibly using several training samples) before adjusting their parameters. On the other hand, a regression tree may decide to create a new leaf for a new training sample, thus immediately adjusting its predictions to it.

Iterative/Immediate
An iterative model has to see the input-output samples several times in order to learn the input-output mapping, while a single-run model learns an useful mapping even when seeing each input-output sample once. This property is somewhat related to aggressiveness as a model that learns slowly can be trained several time on each sample in order to learn faster, and an aggressive model can usually learn with very few presentations of the input.

Incremental/Batch
An incremental model can be updated as new data arrives, without having to be re-trained on the entire training set.

Overfitting
A model is used to predict the output associated with previously-unseen input samples, which makes its accuracy on training samples irrelevant in practice. However, some models tend to overfit their data: they have a very high accuracy on training samples, but very low on unseen samples. Figure 2.1 illustrates this behavior.

2.1.3 Properties of datasets

The data itself has several properties that influence the kind of models that can be used.

Dimensionality
The bigger the input dimension is, the harder training the model becomes. This comes from a sampling problem (high-dimension spaces are "larger" and need more samples to be covered) and a computational problem (more input or output variables usually require more computation and more memory space). Several dimensionality reduction techniques exist, so that high-dimensional inputs can be mapped to a lower-dimensional representation before being passed to the model.

Dataset size
Incremental models can learn from as many input-output samples as needed, but going over the samples takes time. Non-incremental models, or models that store the training set for later use, have problems when the dataset becomes too large.

Concept drift
A non-stationary data distribution means that the input-output mapping changes over time. For instance, the force and direction of the wind can change while measures are taken. This typically means that the model should forget old input-output samples as new ones arrive, so that it remains up-to-date with the latest data.

Independent samples
: Most regression models expect the input-output samples to be independent from each other. This basically means that input points must be presented in a random order, and are obtained by randomly sampling $f(x)$. In Reinforcement Learning, x is usually a representation of the state. Because the agent moves from state to state according to the environment dynamics, values of x are closely related and clustered along a path.

Sample distribution
: Most models tend to be more accurate in parts of the input space where more input-output samples are available. Having only a very small part of the input space covered by samples (as is the case in Reinforcement Learning when the environment can only be in a limited amount of states) makes learning difficult for some learning algorithms. In relation to the previous property, having the region covered by input-output samples moving over time can cause problems. This happens when a Reinforcement Learning agent interacts deeply with a part of its environment then moves to a new, previously unvisited region.

These model and dataset properties are discussed for each Supervised Learning model presented in this master thesis, with reference to what Reinforcement Learning requires from a model.

2.2 Neural networks

Neural networks have originally been inspired from how neurons work in the human brain. A neuron receives activation and inhibition signals from other neurons. If the sum of activation signals is higher than the sum of inhibition signals, the neuron fires and emits an activation signal. If inhibition is stronger than activation, the neuron emits an inhibition signal [MP43].

This activation-inhibition scheme is formalized by defining a neuron that has a set of inputs and one output. Each input has a real value from $-\infty$ to $+\infty$. The output of the neuron is computed as $y = f(\sum_i x_i)$, with f an activation function. The original neuron described in [MP43] can be emulated by using a step activation function, that returns 1 for positive inputs and -1 for negative ones. Nowadays, more advanced activation functions are used, as detailed in the following subsection.

2.2.1 Layered feed-forward networks

In very early work, neurons are assembled in networks that can have any shape [MP43]. The connections between neurons are not weighted (the output of a neuron directly reaches all the neurons it is connected to), and the neural network is designed by a human. These networks allow to formalize how the brain computes values, but is not easily trainable.

The perceptron is a neural architecture in which neurons are assembled in two layers, one representing the input of the network, the other the output [Ros58]. The multilayer perceptron extends this architecture by allowing any number of "hidden layers" to be

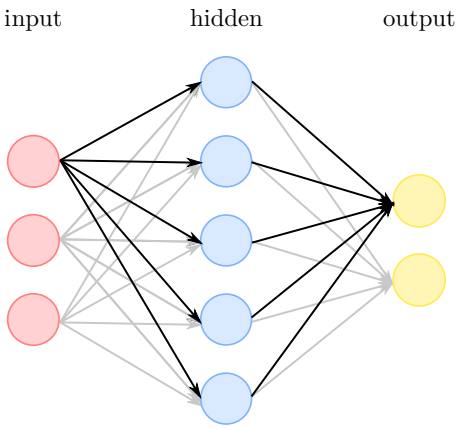

Figure 2.2: Perceptron with a single hidden layer. Each input neuron takes the value of one input variable, and the value of output neurons are copied in the output of the model. More hidden layers can be added, and the number of neurons in the hidden layers can vary.

stacked between the input and output layers, which allows the network to approximate any function [RHW85] (not just linear ones).

In a multilayer perceptron, the output of each neuron of layer i is connected to the input of all the neurons of layer $i+1$. This means that each neuron of layer $i+1$ has one input per neuron of layer i, as depicted in Figure 2.2. In this architecture, there is one neuron per input variable in the input layer, and one neuron per output variable in the output layer. The connections are weighted, so that neurons can be more or less sensitive to their different inputs.

Other neural architectures are possible, as long as there are no connections going backwards, from layer i to layer $i-k$ (such connections are explored in Section 2.3). For instance, convolutional neural networks use two-dimensional layers, stacked one on top of the other, and each neuron of layer $i+1$ receives as input the output of *some* neurons of layer i, usually arranged in a square [LBBH98] (see Figure 2.3).

2.2.2 Prediction

Each layer of a layered neural network can be seen as a function having I inputs and O outputs. In the case of the perceptron shown in Figure 2.2, the value of each output neuron is the weighted sum of all the input neurons, on which an activation function is applied:

$$y_j = f(\sum_i w_{ij} x_i) \tag{2.1}$$

$$= f(W_j \cdot X) \tag{2.2}$$

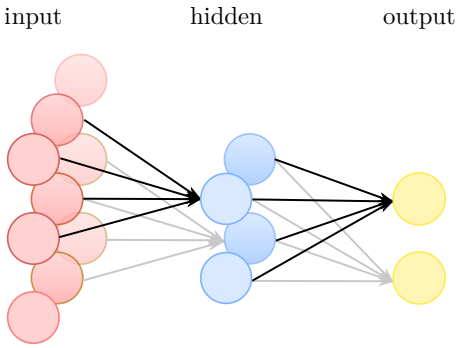

Figure 2.3: Convolution neural networks. Each neuron in the hidden layer is connected to 4 neurons in the input layer, arranged in a 2×2 square. Only some connections are shown for clarity.

Where y_j is the value of the j-th output neuron, x_i is the value of the i-th input neuron, and w_{ij} is the weight of the connection between i and j. Equation 2.2 rewrites the weighted sum as a dot product. When all the output neurons are considered, this allows to see that a neural network layer can be implemented using a simple matrix multiplication:

$$Y = f(WX) \tag{2.3}$$

The activation function can be any piecewise differentiable function. Most of the time, one of these functions is used (see Figure 2.4):

1. $\sigma(x) = \frac{1}{1+e^{-x}}$, the sigmoid activation function with domain in $\Re$ and image in $[0, 1]$.
2. $\tanh(x) = 2\sigma(2x) - 1$, a scaling of the sigmoid activation that has an image in $[-1, 1]$, which is useful in problems where predictions done by the network sometimes have to be negative.
3. $\max(0, x)$, the rectified linear unit (this function is piecewise differentiable)

Predicting a value is as easy as computing the output of each layer, from input to output. The prediction is also called *forward propagation* as values flow from the input neurons to the output ones.

2.2.3 Training

Training a neural network consists of setting the weights of the connections to values that allow the neural network to minimize its prediction error.

When a neural network is created, its weights are initialized to small random values. At this point, the neural network predicts random outputs for any input vector. Training

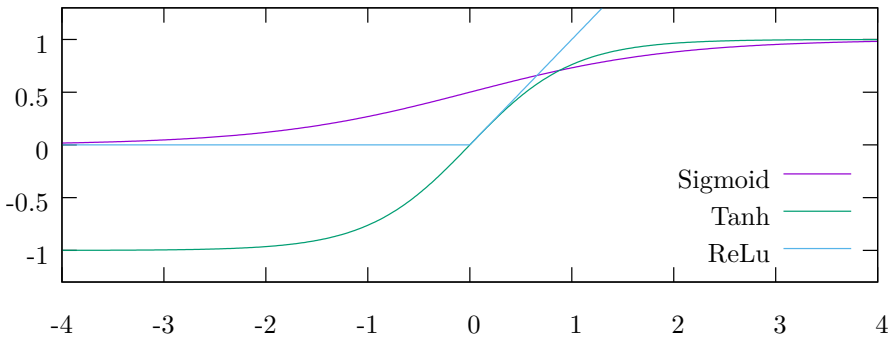

Figure 2.4: The three most common activation functions: sigmoid, tanh and the rectified linear unit.

is done by slowly adjusting the weights using gradient descent. Let's express a neural network as a parameterized function (a function that depends on its input and some parameters):

$$Y = f(X, W) \tag{2.4}$$

With X the input vector, Y the output vector, and W the parameters of the function. For neural networks, the parameters are the set of weight matrices for all the layers.

The performance of the neural network is evaluated using an error function J, usually the Mean Integrated Squared Error of the network, that has to be minimized. This minimization is done by computing the gradient of J relative to all the different weights, which gives the direction of maximum increase of error. In order to minimize the error, the weights are simply moved in the opposite direction, with α a small learning factor that prevents overshooting (see [BP92] for a more detailed explanation of the process):

$$e_i = \frac{\partial J}{\partial w_i} \tag{2.5}$$

$$w_i = w_i - \alpha e_i \tag{2.6}$$

Computing the J error function is difficult for neural networks, that have to be optimized for many input-output pairs, have many weights and sometimes have a very complex structure. Fortunately, the computation of the error function and the update of the weights can be done in a single backwards pass through the network using the backpropagation algorithm [Wer74]. The idea is to compute an error at the output neurons, then to apply the network in reverse (using the derivatives of the activation functions), from the outputs to the inputs. The values that pass over the connections are used as error signals for their weights, that are updated using Equation 2.6.

The backpropagation algorithm finds weights that represent a local minimum of the error function. This means that different neural networks starting with different random weights, but trained on the same data, will approximate the function differently and

will not have the same performance. More advanced variants of the backpropagation algorithm allow to find better local minimums by improving how the error signal is computed (see [DCB15] for a review of some of these techniques).

2.2.4 Properties

Here is a review of the properties of neural networks using current learning algorithms:

Global
: Each training sample causes all the weights of the neural network to be updated, hence altering its global behavior.

Not aggressive
: Learning as performed by Equation 2.6 uses a very small learning factor, which means that the equation must be applied a large number of times before the network fully adjusts to a new training sample. In practice, less than a couple gradient updates are performed by training sample in order to limit risks of catastrophic forgetting [Fre91]. Catastrophic forgetting happens when a network adjusts its weight to become very accurate in a region of its input space, but dramatically reduces its accuracy everywhere else.

Iterative
: The gradient of the error function must be re-computed after each weight update, and several weight updates are required for the network to learn the input-output mapping.

Incremental
: When some care is taken in order to limit catastrophic forgetting, new data can be used for training while discarding old data. However, the error of the network will slowly increase in the areas of the input space corresponding to old training data if its knowledge of these regions is not refreshed in some way.

The learning algorithms used for neural networks also work best when the training input-output tuples are independent from each other.

2.3 Recurrent neural networks

Feed-forward neural networks as described in Section 2.2 are able to represent any function from a multi-dimensional input to a multi-dimensional output. For instance, they are used to associate labels to images, sequences of words, time-series data, etc. However, the dimensions of the input data is fixed because each input variable corresponds to a single input neuron. All the images processed must therefore have the same size, all sequences of words must contain the same amount of words, etc.

Feed-forward neural networks can also be seen as functions approximators having no memory. Every time a new input is presented to the network, all the neurons are recomputed layer by layer. Adding memory to the network allows it to remember past

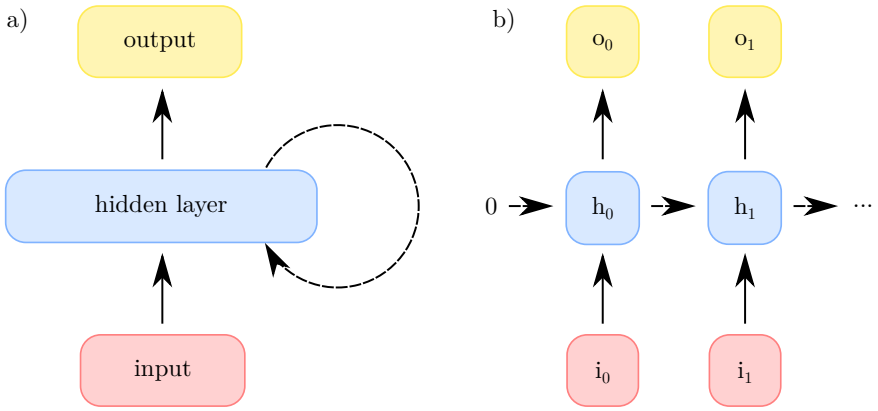

Figure 2.5: Simple recurrent neural network. (a) The dashed line indicates a connection from all the outputs of the hidden layer at time $t-1$ to all the inputs of the hidden layer at time t. (b) Unrolled view of the network, showing more explicitly the flow of information from time $t-1$ to t.

information (in the form of an internal state). This is done by adding recurrent connections, that connect the output of some neurons at time $t-1$ to the input of other (or the same) neurons at time t, as shown in Figure 2.5. The neural network therefore approximates a function of the current input and all its past inputs. This allows it to map arbitrary-length sequences of inputs to sequences of outputs of the same length [HS97].

2.3.1 Prediction

Predicting values with recurrent neural networks is comparable to prediction using feed-forward neural networks, except that the input and output are vectors of value. For each input-output sample, N is the number of values making the sample, X_t with $1 \leq t \leq N$ is the t-th input vector and Y_t is the t-th output vector.

Input values are presented from $t = 1$ to N, and each presentation allows the network to produce Y_t. At each time-step, the network also stores some information in the recurrent connections, that will be used at the next time-step. When the first input is presented, the recurrent connections are considered to contain null values:

$$Y_t, S_t = f(X_t, S_{t-1}) \tag{2.7}$$
$$S_0 = 0 \tag{2.8}$$

Another way of seeing this step-by-step prediction is to consider that the recurrent neural network can be dynamically unrolled (see Figure 2.5). For each input-output sample, the neural network is unrolled N times, then the N input values are used to predict the N output values. All the connections, including the recurrent ones, are duplicated for each time step. The information flowing from $t-1$ to t represents the internal state S_t of the network.

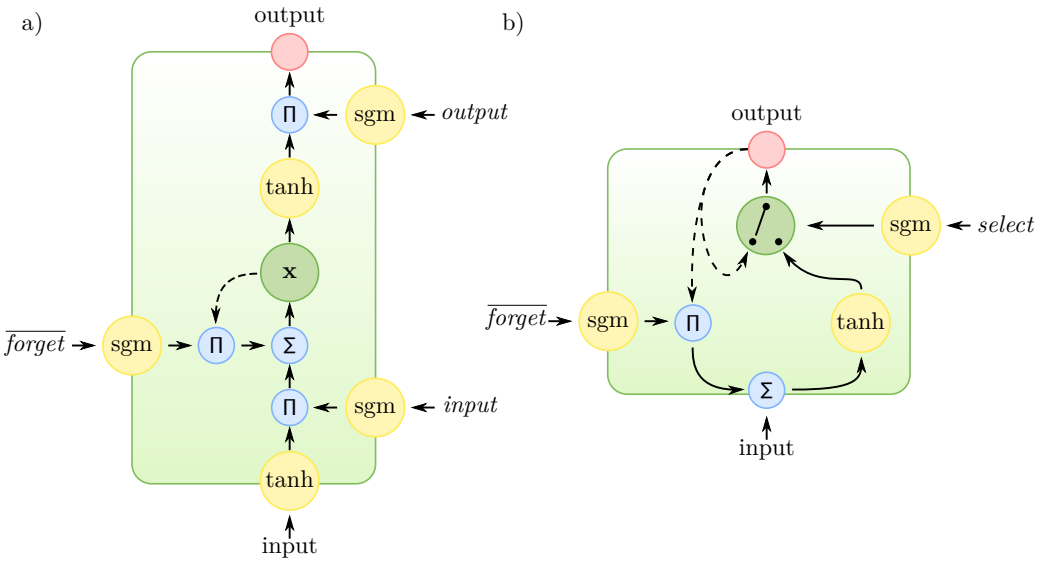

Figure 2.6: Long Short-Term Memory cell and Gated Recurrent Unit. The dotted lines indicate a recurrent connection. Gates are written in italic. (a) LSTM. The core of the LSTM cell is a memory cell to which the input is continuously added, which can be reduced to an integrator. (b) GRU. The core of the GRU cell is a mixer between the old and the candidate output. When the mixer is completely set to its "candidate output" position, GRU also behaves like an integrator.

2.3.2 Training

Recurrent neural networks are trained using the backpropagation through time algorithm [VT07]. This algorithm consists of unrolling the network as shown in Figure 2.5, then to apply standard backpropagation from Y_N to X_N and back through the recurrent connections, then from Y_{N-1} to X_{N-1}, down until X_1 is reached. In Figure 2.5, the error flow would go from the upper right to the lower left, taking each connection in reverse.

Training simple recurrent neural networks as shown in Figure 2.5 is difficult because, for long sequences, the unfolded network behaves like a very deep neural network. This causes problems for the error signal, that has a tendency of blowing up or disappearing as the depth increases. More advanced recurrent architectures have been proposed in order to avoid this problem.

The best-known is Long Short-Term Memory [HS97, GC99, GSK+15], that replaces the hidden layer with a layer of *cells* as depicted in Figure 2.6, a. The core of this cell is a memory element. At each time-step, the input is added to the discounted value of the element at the previous time-step. When the forget gate has a value close to zero, the cell quickly forgets its past values. The forget gate, thanks to its sigmoid activation, never has a value greater than 1 or smaller than -1. This prevents the error flowing back through time from getting larger as time passes, thus preventing error blowup. The

input and output gates allow the cell to ignore some of its input and to disconnect its output from the rest of the network.

Another recurrent neural architecture is the Gated Recurrent Unit [CvMBB14]. Instead of using a memory cell in a structure that looks like an integrator, the Gated Recurrent Unit computes its output at time t as a mix between its output at time $t-1$ and a candidate output computed from the input (see Figure 2.6, b). The mixing factor between the old and the candidate input is given by the select gate. Despite having a design completely different from LSTM, the GRU cell performs comparably to it in different tasks [CGCB14, SV15].

2.3.3 Properties

The most interesting property of recurrent neural networks is that they are one of the very few function approximators that are able to map arbitrary-length inputs to outputs. Being comparable to classic feed-forward neural networks, they have the same properties, as listed in Section 2.2.4.

2.4 Decision trees

A decision tree predicts the output value corresponding to an input by performing a succession of tests on the input. It is built as a tree in which each node corresponds to a test and each leaf is associated with a value. Usually, binary tests are used, such that the tests are either true or false, therefore reducing the decision tree to a binary tree.

2.4.1 Prediction

Predicting a value is done by traversing the tree starting from the root node. At each node, the input is matched against the test it contains ($X_3 \geq 2.13$ for instance). If the test is satisfied, exploration goes to the left child of the current node. If the test is not satisfied, the right child is explored. When a leaf is reached, the value it contains is returned as prediction.

Figure 2.7 shows a simple decision tree used for classification. Figure 2.8 shows that extensions are possible: here, the leaves don't contain a single value but a linear regression that allows the tree to be more efficient at regression tasks [VK14, LCG11, PS05].

2.4.2 Training

Training a decision tree consists of adding nodes, choosing tests and storing the proper values in the leaves. The goal of the training is to find the tree achieving the best prediction performance, while also being shallow and well-equilibrated. This multi-criteria optimization problem is very hard and exact solutions are impossible to compute.

Several learning algorithms instead build the tree incrementally [PS05]. The tree starts by being a single leaf associated with one value. For each input-output sample, the tree is traversed as in prediction, and the leaf associated with the input is updated according to the expected output. Leaves usually contain various statistics about their

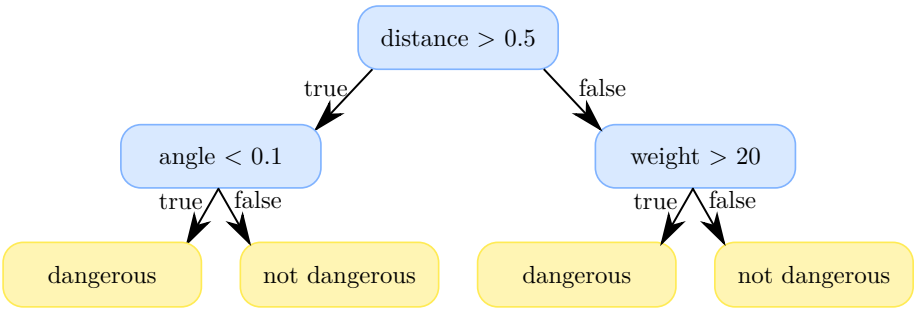

Figure 2.7: Simple decision tree giving the class of an object depending on some of its attributes

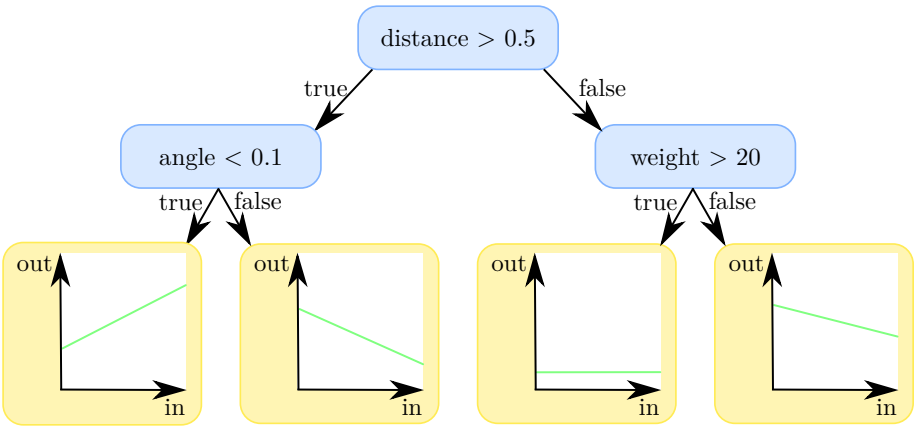

Figure 2.8: Regression tree using a linear regression to produce the output associated with an input

values (mean and variance for instance) so that the learning algorithm can detect inaccurate leaves that should be transformed to nodes (this operation is named "split"), or groups of leaves that can be merged together. The value or linear regression parameters of the leaves also have to be updated.

Other algorithm work on the global dataset and try to iteratively partition it to piecewise linear or piecewise constant functions of increasing accuracy.

2.4.3 Properties

Decision trees have a set of properties completely different from neural networks, which makes them very useful in some situations.

Local

Most of the time, adding a training sample to a decision tree will only update one leaf (or one leaf and a couple of nodes directly above it), which keeps the changes close to the input sample.

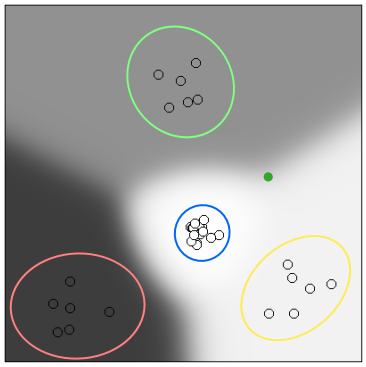

Figure 2.9: Ellipses represent an approximation of cluster centers and covariances. The value predicted at the green point is a mix of the values of the green, blue and orange clusters.

Aggressive
Each input-output sample is directly used to update a leaf. However, some algorithms avoid overfitting or growing overly large trees by allowing some small errors in the leaves, which translates to reduced aggressiveness.

Incremental or Batch
While most tree-learning algorithms are batch, several incremental algorithms exist (for instance [PS05]), even if they are very complex because they have to constantly and quickly adjust the tree and keep it balanced.

Direct
Batch tree-learning algorithms perform many operations on the dataset and cannot therefore be considered direct algorithms. However, [PS05] proposes an incremental algorithm that reads each input sample only once when incorporating it into the tree, which shows that direct tree-construction algorithms do exist.

Decision trees are also robust to several dataset properties, except that they usually don't perform well when facing concept drift (they need to discover which leaves or nodes become irrelevant and can be removed).

Another big drawback of decision trees is that they act as a piecewise constant or piecewise linear function. The function they predict presents discontinuities at each leaf boundary, which may be problematic for some applications.

2.5 Gaussian Mixture Models

Gaussian Mixture Models present set of properties that nicely complement neural networks and decision trees. They are built on sound probabilistic theorems and are easily implemented. For those reasons, this master thesis uses a Gaussian Mixture Model, described in depth in Chapter 4. This section gives an overview of Gaussian Mixture Models in general.

A Gaussian Mixture Model identifies clusters of input samples whose outputs are similar. It then associates a value [HE10] or a linear regression [Hei11] to each of these clusters (the average output of samples belonging to the cluster for instance), and mixes them using Gaussian kernels: the closer a cluster is to the query point, the more it influences it. Figure 2.9 gives a visual example of a function approximated using a Gaussian Mixture Model.

2.5.1 Prediction

Predictions of Gaussian Mixture Models are based on the Bayes rule. Given an input vector, the probability that this input belongs to any of the clusters is computed. Each cluster is represented as a normal distribution with its own mean and covariance matrix. The Bayes rule is then used to compute the probability that the cluster corresponds to the input:

$$p(x|i) = \mathcal{N}(x|\mu_i, \sigma_i) \tag{2.9}$$

$$p(i|x) = \frac{p(x|i)p(i)}{\sum_j p(x|j)p(j)} \tag{2.10}$$

$$\hat{f}(x) = \sum_i p(i|x)v_i(x) \tag{2.11}$$

With x the input vector and i a cluster. The output of the model is the output $v_i(x)$ of each cluster weighted by the probability that the input belongs to that cluster. $v_i(x)$ can be a constant, a linear regression of x, or any other function.

2.5.2 Training

Training a Gaussian Mixture Model consists of two operations: adding or removing clusters as needed, and updating existing clusters. A new cluster is added if the new input is too far from the already-existing ones ($\max p(x|i) < \varepsilon$ for instance). A cluster is removed if it is deemed irrelevant. This criterion is more complicated to measure and this master thesis uses an algorithm that is different from other works (compare Section 4.3.2 and [HE10]). A cluster is updated by updating its empirical mean and variance.

The main advantage of Gaussian Mixture Models is that they are able to create and remove clusters as required for representing the input-output mapping, thus automatically adjusting their complexity to the task at hand.

2.5.3 Properties

Gaussian Mixture Models have some properties in common with regression trees (for instance, both are local function approximators), but have the advantage of offering a smooth function approximation. The learning algorithm is also simpler as it simply consists of adding or removing clusters, whereas efficient learning algorithms for decision trees are quite complex.

Local
Each training sample leads to the update of all the clusters, but the weight of the update depends on the distance between the input sample and the cluster. This allows clusters distant from the input to be nearly untouched, which makes the update operation local.

Aggressive
If a cluster has to be created for an input sample, it will immediately have the output value associated with this sample. Cluster updates use a learning factor that is much larger than the ones used by neural networks, which allows the model to quickly learn new values (at the expense of older ones that get overwritten over time).

Direct
The algorithm proposed in this master thesis and the original algorithms [HE10, Hei11] only need to see each sample once for learning the model.

Incremental
Gaussian Mixture Model algorithms are incremental.

Chapter 3

Attention Models

Most classification, regression or prediction tasks can be performed using a large amount of input data. Image classification, for instance, is usually performed by feeding whole images to deep convolutional neural networks. The convolutional part of the network (pixels from the image are combined in overlapping patches) allows it to recognize objects or features wherever they are, and sometimes even independently of their size.

The problem is that this approach is more like "look everywhere at once" than "look where the object actually is". It works but requires a large amount of computing power (see [KSH12] for example), and convolutional neural networks tend to loose spatial information in their higher layers (they recognize features, but cannot tell where they

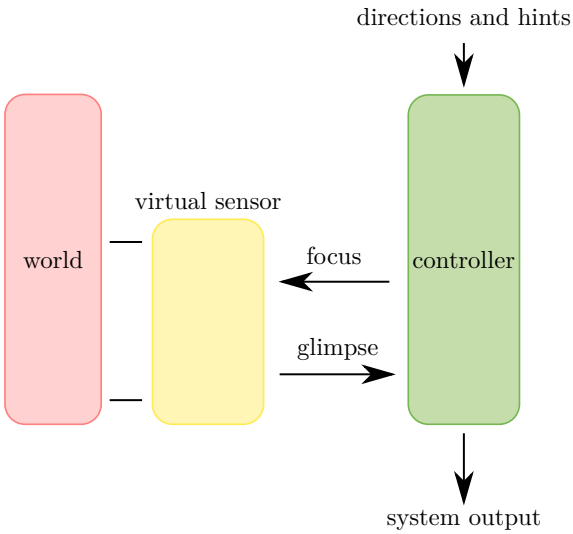

Figure 3.1: General architecture of an agent that learns to move a virtual sensor over the input space. The agent is given hints and directions (the expected class, a reward, or both), and learns to focus the virtual sensor on the input. The succession of glimpses is used to produce the final output of the system.

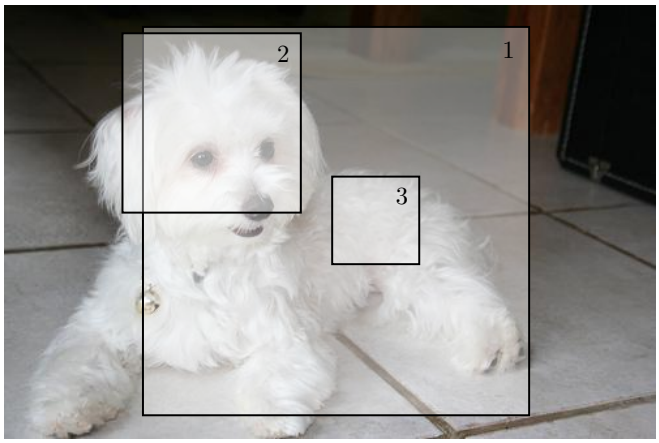

Figure 3.2: Example of how three glimpses could be used to locate a Stanford Dog [KJYFF11] in the picture, look at the shape of its head, and finally the texture of its coat. This succession of glimpses allows to recognize the breed of the dog

are). The aim of attention models is to allow a classifier to focus its attention to some parts of the input data, temporarily ignoring the rest, as if it observes the data through a small virtual sensor (see Figure 3.1). This allows the classifier to recognize features regardless of their location.

Instead of classifying a complete view of the input data, the classifier learns how to move the virtual sensor in a way that allows it to take a succession of meaningful glimpses, which are used to perform the final classification. This can be compared to how humans identify objects by looking at several simple parts of them, the recognizing which object is made of all the observed parts.

3.1 Visual Attention

Attention models are naturally applied to images, that can be seen as very high dimensional inputs ($W \times H \times 3$ for RGB images for instance) easily segmented in regions or objects. Visual attention consists of extending a classifier with the ability to take partial observations of the image being classified, called glimpses. The glimpses are usually small square regions of the image. Their size can be adjusted by the learning agent, and the image area they cover is presented to the agent in a down-scaled form. Figure 3.2 shows an example of how features of a dog can be used to classify it.

3.1.1 Foveal vision

Foveal vision tries to mimic how the animal eye works. Instead of producing a square glimpse of uniform pixel density, the idea is to allocate more pixels to the central region of the glimpse, and less pixels to the peripheral region. This allows the agent to have a detailed view of the object it is looking at, while also having a fairly large amount of

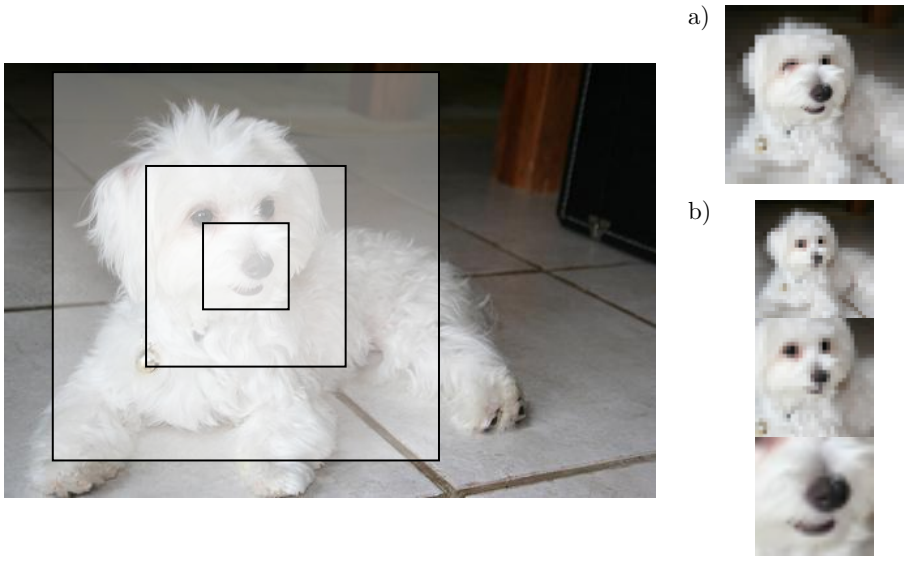

Figure 3.3: A multi-resolution foveal glimpse. a) Combined glimpse that shows how the image is distorted in order to contain more information near the center. b) Composite glimpse as usually implemented in the literature.

peripheral data that could allow it to approximately locate the next object to which to attend [BS89]. Figure 3.3 illustrates this concept.

The most common implementation of foveal vision [BS89, MHGK14, SFR14] consists of extracting several square patches out of the image, each centered around to current focus point and having a different size. Each patch is then downscaled to a size that allows efficient computation (depending on the experiments, patch size goes from 8×8 to 96×96). Finally, all the patches are combined as shown in Figure 3.3, b. This provides complete context to the learning agent and is easy to implement.

Some other articles use a different approach as the glimpses taken by the agent are not foveal (they are made of a single square patch of uniform resolution), but the agent receives as first glimpse a scaled-down version of the image [Ran14, BMK14, BGSF15]. No comparison between these two approaches exist, but foveal vision as implemented by Mnih et al. achieves a testing error of 1.20% on a translated MNIST dataset [MHGK14] while a non-foveal implementation by Ba et al. only achieves a testing error of 1.62% on a scaled and translated MNIST dataset [BGSF15]. The two experiments are not completely comparable though.

Learning how to focus the glimpse sensor is a very difficult problem, as the agent needs to be able to recognize glimpses in order to move the sensor, and also needs to be able to move the sensor in order to recognize glimpses.

3.1.2 Q-Learning

[BVB+96] builds on the foveal sensor descrived in [BS89], but features are extracted from each multi-resolution glimpse before being processed by the Reinforcement Learning agent. Instead of observing a large number of pixels, the agent therefore only observes probabilities that features are part of the glimpse. Features appearing near the edge of the glimpse have a lower probability as they lie in a low-resolution part of the sensor.

Moving the glimpse sensor is done by querying for features. Instead of learning direct (x, y) movements, the Reinforcement Learning agent learns to query specific features using the Q-Learning algorithm. Each query is processed by the environment that finds the shortest move allowing the specified feature to be detected. After each glimpse, the features just observed are merged with all the features that have already been seen for the current image and are used to make a prediction. The reward obtained by the agent is the reduction in entropy that a glimpses permits. The final prediction is made when the entropy goes below a predefined threshold.

This algorithm provided good results on the experiments carried out by its authors and has the nice property of taking a varying number of glimpses depending on the need. Some classes are easier to identify and therefore need less glimpses. It is also interesting to note that the policy learned by the algorithm (querying for a feature and observing the features actually observed) is comparable to how decision tree works, and is actually called an "Action Tree" [BVB+96].

3.1.3 Neural networks

Most modern attention models are built around neural networks, that receive glimpses and glimpse coordinates as input and produce new glimpse coordinates as output.

Neural networks are trained using the backpropagation algorithm presented in Section 2.2, which allows to minimize the error of the network but requires any operation done by the network to be differentiable. However, producing glimpses consists of cropping the input image, which is not differentiable.

This problem is solved in [Ran14] by learning the attention model in two steps and circumventing the fact that cropping is not a differentiable operation. The part of the network that predicts glimpses position is learned by randomly moving the glimpse sensor until coordinates that maximize the score given to the correct class of the image is found. These coordinates are used to train the "glimpse network". After this training step, the rest of the network is trained to give a higher score to the correct class.

This two-step learning avoids the cyclic dependency between how the glimpse sensor is moved and how glimpses it produces are classified. Perturbating the location of the glimpse sensor allows to find a local optimum as backpropagation would have done, even if the cropping operation is not differentiable.

The biggest problem of the algorithm described in [Ran14] is that it takes a fixed number of glimpses before making a decision. This does not allow it to skip glimpses for simple classes, or take more of them when ambiguous data is presented to it.

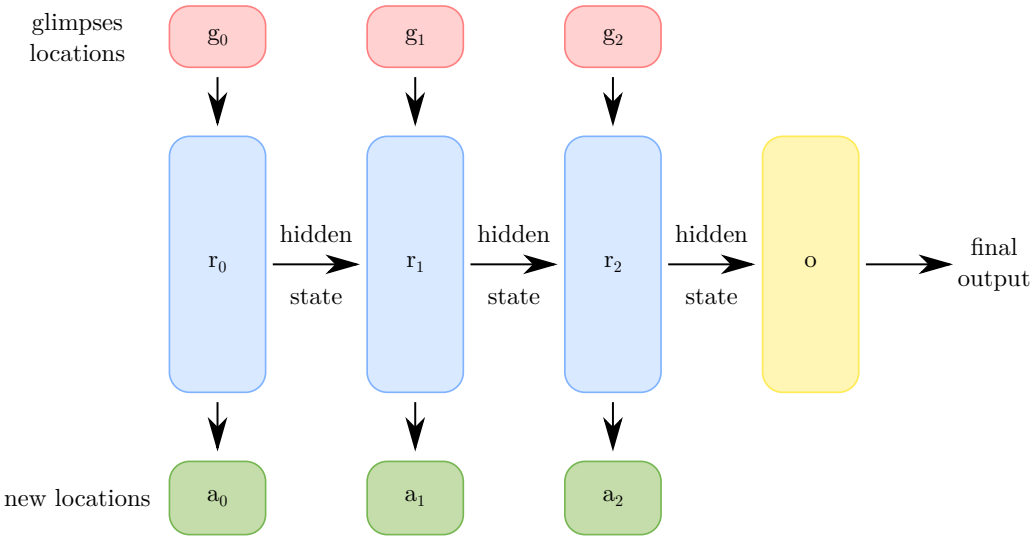

Figure 3.4: High-level neural architecture used by most visual attention models. At each time step (denoted in subscript in this figure), the location of the virtual sensor and the glimpse it has taken (red) is fed to a recurrent neural network (blue), that produces the next location to which to attend (green). After a possibly variable number of time-steps, the network is required to make a decision about the whole image it has explored (yellow).

3.1.4 Recurrent neural networks

Even if [Ran14] uses simple feed-forward neural networks, most other attention models are based on recurrent neural networks so that they can take a variable number of glimpses. They all share an architecture comparable to the one depicted in Figure 3.4, where a recurrent neural network receives as input a glimpse with its location and produces as output the coordinates of the next glimpse [MHGK14, SFR14, BMK14, BGSF15]. The hidden state of the network, remembered from time-step to time-step, acts as a memory allowing the network to produce a final output that depends on the complete succession of glimpses. The neural network is trained using backpropagation through time [VT07], which optimizes the recognition and focusing parts of the neural network.

Learning how to move the glimpse sensor is still a challenge because recurrent neural network also require all operations to be differentiable. The agent receives a reward that depends on how well it has classified an input image (either 1/0 or a measure of the accuracy of the agent), and trains the recurrent neural network to maximize this reward using Policy Iteration in Partially Observable Environments [MHGK14]. Basically, increases the probability of performing actions that lead to a good reward, and decreases the probability of performing actions that lead to a bad reward.

[MHGK14] extends Figure 3.4 in order to produce actions in addition to new glimpse locations. In their experiments, actions are used to classify the input (and end the

experiment). They obtain state-of-the-art results on MNIST even if the numbers are translated or cluttered.

[BMK14] allows the network to predict a sequence of classes. This is applied to the recognition of multi-digits numbers (based on the MNIST dataset, or using Google Street View house numbers).

[SFR14] uses the very powerful GoogLeNet deep convolutional network as the glimpse network (red boxes of Figure 3.4). This allows the network to process 96×96 glimpses.

[BGSF15] extends the previous works by adding a "teacher" network that learns to predict next glimpse locations based on the current glimpse and the class to predict. Knowing the expected class allows it to quickly learn a good policy for moving the virtual sensor. This policy is used as expert knowledge in order to speed-up learning of the network that cannot observe the class to be predicted.

Finally, [Kav15] extends the virtual sensor so that it can rotate on itself (in addition to translation and scaling). This allows it to recognize objects even when they are rotated in the image.

3.2 Acoustic attention

Attention models are not limited to image processing, they have also been applied to speech recognition [CBS+15]. Instead of focusing a square sensor on a two-dimensional image, attention-based speech recognition splits an input stream of voice into overlapping phonemes for which features are computed. Then, a recurrent neural network following the general architecture of Figure 3.4 is used to select which phonemes to attend. Their features are taken as glimpses and processed by the neural network, that produces phoneme labels. Experiments show that this architecture is able to properly recognize phonemes in a phrase, and emit them in the right order.

3.3 Incremental solution construction

Incremental solution construction is a supervised task where the agent produces a solution step by step. Some of the algorithms presented in the previous sub-sections are part of this class of algorithms, for instance the incremental recognition of speech [CBS+15] or digit-by-digit recognition of house numbers [BMK14].

[MDG09] implements a more general form of those algorithms, and is based on standard Q-Learning without requiring any specific function approximator (such as recurrent neural networks). In addition to the input, the agent is also able to sense its partial output, so that it can incrementally refine it. Examples include the incremental recognition of handwritten words (each letter is recognized, and the agent can also use neighboring letters to recover unclear information), and the processing of trees (parsing for instance).

[MDG09] does not explicitly mentions virtual sensors, but actions are designed to focus on specific portions of the candidate output. The input is either fed in a specific order to the agent, or more actions can be used to focus on specific parts of it.

One of the most interesting properties of incremental solution construction is that it provides results comparable (in accuracy) to advanced non-incremental Supervised

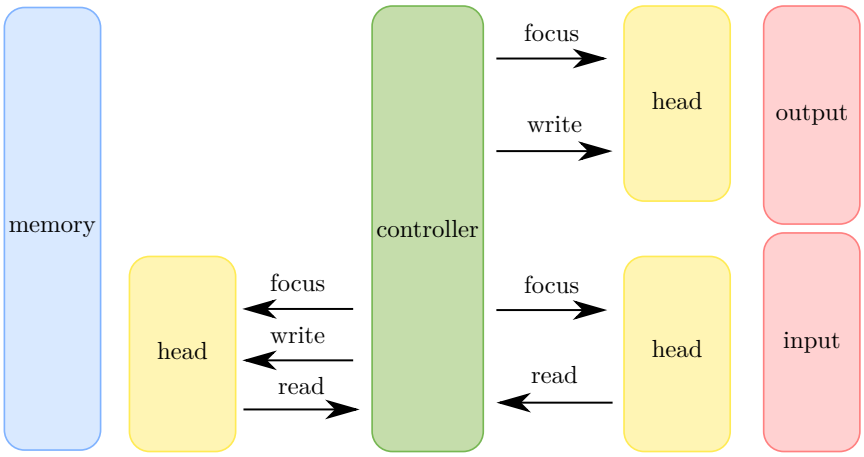

Figure 3.5: Neural Turing Machine (generalization of [GWD14] and [ZS15]), able to read an input tape, write to an output tape, and manage some internal memory.

Learning algorithms [MDG09]. Unfortunately, training is much slower, because a Reinforcement Learning agent only senses a scalar reward while Supervised Learning agents are provided much more information when learning.

3.4 Neural Turing machines

Neural Turing machines are a generalization of attention models and incremental solution construction[1]. On a very high level, a Turing machine is composed of two main elements:

1. A semi-infinite memory tape, made of an infinite number of memory cells $M_0 ... M_\infty$. Each memory cell contains a symbol from a finite alphabet. Unused memory cells usually contain the ϵ (empty) symbol, part of the alphabet.

2. A controller that, at each time step, reads the memory tape at a precise position using a *head*, write a new value at that position and finally move the head one position to the left or to the right.

In its original description, the controller of a Turing machine consists of a deterministic finite automaton[Tur38] that emits actions every time it changes state. Because observing an input (reading the memory tape) and choosing an action (writing to memory or moving the head) fits the Reinforcement Learning framework, Reinforcement Learning agents have been used as the controller of Turing machines [GWD14, ZS15].

[GWD14] implements a Neural Turing Machine using a neural network as controller. The neural network is able to read the memory tape, to produce new values to be stored on the tape, and to focus its attention on specific parts of it (equivalent to moving the head). The neural network also receives an external input and has to produce an

[1] And also a generalization of everything else computable[Tur38]

external output. The tape is therefore only used as scratch memory and is not exposed to the outside world. Figure 3.5 illustrates the general concept behind Neural Turing Machines.

The Neural Turing Machine is trained using standard backpropagation (or backpropagation through time if a recurrent neural network is used as controller). Unfortunately, backpropagation requires the complete system to be differentiable. [GWD14] solves that by replacing the discrete head by a continuous weighting of memory locations. Read and write operations therefore take into account all the memory cells, with each cell having a specific weight. Several addressing modes (by position or by content) have been implemented. The Neural Turing Machine is able to perform moderately complex tasks, like sorting lists and learning key-value mappings. Experiments using a feed-forward neural network achieved about the same results as the ones using recurrent neural networks, but learned more slowly.

Another version of the Neural Turing Machine, based on Reinforcement Learning and using discrete operations (instead of fully-differentiable ones) has been introduced recently [ZS15]. This version uses a memory tape, an input tape and an output tape. The controller is a recurrent neural network that takes as input the content of the memory cell under the memory head (and only that, no mix of other cells), the content of the input cell under the input head, and a history of actions taken previously. The output of the network consists of a value to write to the memory and ternary predictions (backwards, forwards, none) for the memory and input heads. The network also outputs a prediction of the next output symbol, and whether or not this prediction is ready to be written on the output tape.

It is interesting to note that the Neural Turing Machines cannot directly observe their output, and can only focus their attention to specific parts of their internal memory tape. However, a Turing machine can build its output in-memory, then copy it to the output once it is completely finished. This therefore still generalizes incremental solution construction, though not in a straightforward way. Chapter 5 presents an architecture based on Reinforcement Learning where the agent is able to observe and focus its attention on its input, memory and output.

Part II

Contributions

Chapter 4

Incremental Gaussian Mixture Model

Chapter 2 presented three classes of function approximation models: neural networks, decision trees and Gaussian Mixture Models. This chapter presents a Gaussian Mixture Model that has been designed for maximum performance in a Reinforcement Learning context. The model presented in this chapter is based on work by Heinen [HE10, Hei11], and improves it in areas important for use in Reinforcement Learning.

Section 4.1 introduces the context in which the model has to work and gives properties of the dataset that are relevant to how it is implemented. Section 4.2 details the architecture of the Gaussian Mixture Model and gives the prediction algorithm. Section 4.3 presents the learning algorithm used for training the model. Finally, Section 4.5 experimentally compares the Gaussian Mixture Model presented in this thesis with a neural network. In high dimensions, the Gaussian Mixture Model of Heinen creates so many Gaussian clusters than experiments become impossible to run, hence the absence of comparison between this version of GMM and Heinen's.

4.1 Introduction

Reinforcement Learning agents need to compute value functions (see Section 1.2) or direct approximations of the policy to follow (see Section 1.3). These values and policies are usually used to train a function approximator in order to cope with high-dimensional state spaces and to allow good generalization of the agent (see Section 1.4).

Any function approximation model can be used for Reinforcement Learning, but the input-output samples generated by a Reinforcement Learning agent have properties that make some models perform worse than expected. Using the notation introduced in Section 2.1.3, datasets produced in a Reinforcement Learning context (Q-values or state-action pairs for instance) have the following properties:

High dimensionality
: Even if simple Reinforcement Learning problems are usually of small dimensionality ((x, y) coordinates in a grid for instance), real-world problems are of much larger dimensionality. For instance, the Reinforcement Learning problem described

in Chapter 5 uses an image sensor that produces $N \times N \times 3$ variables, with N a number of pixels from around 3 to 20. Even small sensor sizes produce hundreds of variables.

Large dataset size
Value Iteration algorithms produce at least one input-output sample per time-step, and usually several of them if eligibility traces or experience replay is used. Policy Iteration algorithms also produce around one input-output sample per time-step. If the agent learns for several time-steps per episodes and for a large number of episodes, the amount of data produced can quickly reach millions of input-output samples.

Concept drift
Reinforcement Learning agents progressively and iteratively learn their policies and Q-values, which means that the function to be approximated by a model is non-stationary. Even the distribution of input samples is non-stationary as the states visited by the agent change as learning progresses.

Samples on trajectories
Even stochastic environments are not completely random, which means that the agent usually follows a "trajectory" in the environment. Input-output samples on which a function approximation model is trained are therefore highly correlated as one input follows the previous one.

Sparse sample distribution
It is impossible for a Reinforcement Learning to explore is complete environment. Instead, it quickly learns to focus its attention on promising states by using some sort of guided exploration. This means that most of the input space of the function to be approximated is unexplored and contains no input-output sample, while other parts are much more densely covered.

A Reinforcement Learning agent facing a difficult task in a large environment is also expected not to forget what it has learned in rarely-explored parts of the environment. Combined with sparse sample distribution, this hints at the use of local models instead of global ones.

The Gaussian Mixture Model presented in this chapter inherits all the properties of the model described in [Hei11] (local, incremental, aggressive, see Section 2.5), while also avoiding forgetting old but accurate values, and better coping with high-dimensional inputs.

4.2 Prediction

A Gaussian Mixture Model makes the hypothesis that the input data follows a distribution that can be approximated using a finite number of multivariate gaussian distributions, each of them having a center, a covariance matrix, and an average output (its value). One possible way of predicting an output would be to select the gaussian cluster

having the highest probability of corresponding to the input, and then using its average output as prediction. However, this would introduce discontinuities in the model (when the best cluster of one input is not the same as the one of another input) and reduces its accuracy. Predictions are therefore made using a weighted sum:

$$\hat{f}(x) = \sum_c P(c|x)\hat{f}_c(x) \tag{4.1}$$

With x a D-dimensional input vector, and $\hat{f}(x)$ a possibly-multidimensional output vector. $\hat{f}_c(x)$ represents the value predicted by cluster c (its average output for instance), and $P(c|x)$ is the probability that the input x belongs to cluster c. This probability is computed as follows:

$$\mathcal{N}(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^D|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \tag{4.2}$$

$$P(x|c) = \mathcal{N}(x|\mu_c, \Sigma_c) \tag{4.3}$$

$$P(c|x) = \frac{P(x|c)P(c)}{\sum_{c'} P(x|c')P(c')} \tag{4.4}$$

With μ the D-dimensional center of a gaussian cluster, Σ its covariance matrix, $|\Sigma|$ the determinant of Σ and $P(c)$ the prior probability of cluster c, estimated in [HE10] by counting how many input points belong to it (very simplified view of the algorithm). All vectors are row-vectors.

Equation 4.4 uses the mathematically-correct Bayes rule in order to compute $P(c|x)$ given $P(x|c)$ and $P(c)$, as do [HE10, Hei11]. However, experiments have shown that assuming an uniform distribution on the priors, thus removing $P(c)$ from the equations, gives better results, because clusters for which a small amount of training samples are available (and thus having a small prior probability) are not particularly irrelevant as they may contain valuable information stored for a long time in rarely-visited areas of the input space. Equation 4.4 is therefore changed to:

$$P(c|x) \approx \frac{P(x|c)}{\sum_{c'} P(x|c')} \tag{4.5}$$

This approximated value of $P(c|x)$ still sums to one over all c and can therefore still be used as weight.

Each cluster maintains an average output $\bar{y}$ and an input-output covariance matrix Σ_{xy}. Those two values are used to compute $\hat{f}_c(x)$ as a local linear regression:

$$\hat{f}_c(x) = \bar{y} + \Sigma_{xy}^T\Sigma^{-1}(x-\mu) \tag{4.6}$$

4.3 Learning

Learning a Gaussian Mixture Model consists of solving two problems: creating or removing gaussian clusters, and updating input and output statistics of existing clusters. This section first explains how existing clusters are updated, then how useless ones are removed and new ones are added.

4.3.1 Update

When a (x, y) training sample is presented to the Gaussian Mixture Model, every existing cluster is updated towards this sample using a weight that depends on the posterior probability of this cluster, $P(c|x)$. This operation updates the center μ of the cluster, its input covariance matrix Σ, its average output $\bar{y}$ and the input-output covariance matrix Σ_{xy} used by Equation 4.6. All those values are updated using a weighted moving average, so that they follow an empirical estimate of their true value, biased towards recent input-output samples in order to address concept drift:

$$w_c = \alpha P(c|x) \tag{4.7}$$
$$\Sigma = (1 - w_c)\Sigma + w_c \left((x - \mu)(x - \mu)^T + \epsilon I\right) \tag{4.8}$$
$$\Sigma_{xy} = (1 - w_c)\Sigma_{xy} + w_c(x - \mu)(y - \bar{y})^T \tag{4.9}$$
$$\mu = (1 - w_c)\mu + w_c x \tag{4.10}$$
$$\bar{y} = (1 - w_c)\bar{y} + w_c y \tag{4.11}$$

With α a learning rate (0.1 in the experiments) and ϵ a small value (around 1×10^{-4}) that prevents Σ from becoming non-invertible.

Equations 4.8 and 4.9 differ from the ones given in [HE10, Hei11], but are simpler and are justified by the non-stationary nature of the function being approximated. Equation 4.9 can be developed as:

$$\begin{aligned}
\Sigma_{xy} &= (1 - w_c)\Sigma_{xy} + w_c(x - \mu)(y - \bar{y})^T \\
&\approx E_{(x,y)}[(x - \mu)(y - \bar{y})^T] \\
\Sigma_{xy}(i, j) &\approx E_{(x,y)}[(x_i - \mu_i)(y_j - \bar{y}_j)]
\end{aligned}$$

Which corresponds to the definition of the covariance matrix of two vectors of random variables. Equation 4.8 can be developed in the same way, except that ϵI biases the estimate of the mean in order to prevent Σ from becoming non-invertible.

4.3.2 Removal

In order to detect useless clusters, a score s is associated with each cluster. This score is based on the average posterior probability of the cluster. This allows to detect clusters that don't contribute much to the output value of the mixture model even when the input sits very near their center. These clusters usually reside in regions of the input

space where there are far too much clusters (hence each fighting for contribution to the output value). These clusters also tend to have very similar values of $\hat{y}$, thus being redundant.

This filtering by score is different from the algorithm proposed by [HE10], that removes clusters based on their probability $P(c)$[1]. When a cluster has a probability that falls below a given threshold, it is removed. This has the unfortunate effect of removing rarely-visited clusters that may contain valuable long-term information. The algorithm proposed in this thesis does not exhibit this behavior.

The score of a cluster is updated for every input-output sample presented using those formulas:

$$\mathcal{D}_{\mathcal{M}}(x|\mu,\Sigma)^2 = (x-\mu)^T\Sigma^{-1}(x-\mu) \tag{4.12}$$

$$w_c = \alpha \exp\left(-\beta \mathcal{D}_{\mathcal{M}}(x|\mu,\Sigma)^2\right) \tag{4.13}$$

$$s_c = (1-w_c)s_c + w_c P(c|x) \tag{4.14}$$

With α a learning rate (0.05 in the experiments), β a value around 4 that makes w_c decrease faster as x becomes further away from μ (so that the score is more biased towards points close to μ), and $\mathcal{D}_{\mathcal{M}}(x|\mu,\Sigma)^2$ the squared Mahalanobis distance between x and the center of the cluster. This value is also used for the computation of $\mathcal{N}(x|\mu,\Sigma)$ (see Equation 4.2) and does not need to be recomputed. Using the above formulas, s_c is an estimator of $E[P(c|x)]$ strongly biased towards x values close to the center of the cluster.

Once all the scores have been computed, all the clusters having a score under s_{min} (0.05 in the experiments) can be removed. If the bias of w_c is lowered (by decreasing β), overall scores become lower and the Gaussian Mixture Model starts to remove rarely-visited by important clusters.

4.3.3 Addition

The addition of new clusters follows more closely the algorithm given in [HE10, Hei11]. The idea is to add a new cluster whenever the existing ones do not allow the model to predict an accurate enough value. When an (x, y) input-output sample is presented to the model, the predicted output $\hat{y} = \hat{f}(x)$ is computed. The error between y and $\hat{y}$ (both are vectors) is computed as the maximum difference between corresponding elements of y and $\hat{y}$:

$$y_{max} = \max(y, y_{max}) \tag{4.15}$$

$$y_{min} = \min(y, y_{min}) \tag{4.16}$$

$$e = \|(y-\hat{y}) \div (y_{max} - y_{min})\|_\infty \tag{4.17}$$

[1] The exact score used by [HE10] is $\sum_t P_t(c)$, but the behavior is the same as simply using an instantaneous value of $P(c)$

With y_{min} and y_{max} used to compute the range of all the elements of y, and $\div$ the element-wise division operation. The minimum distance between x and any of the clusters is also computed (see Equation 4.12):

$$d = \min_c \mathcal{D}_{\mathcal{M}}(x|\mu_c, \Sigma_c) \tag{4.18}$$

A new cluster is created if e is above a threshold ε and d is above δ. This means that a cluster is created only when the model is inaccurate and updating an existing cluster (close to x) would not improve its accuracy. A intuitive value for δ can be computed by merging Equation 4.2 (the normal density probability function) and Equation 4.12 (the squared Mahalanobis distance). This allows to approximate the relation between δ, a distance, and p, the probability of x belonging to the closest cluster:

$$\hat{p} = \exp(\frac{-1}{2}\delta^2) \tag{4.19}$$

$$\log(\hat{p}) = \frac{-1}{2}\delta^2 \tag{4.20}$$

$$\delta^2 = -2\log(\hat{p}) \tag{4.21}$$

$$\delta = \sqrt{-2\log(\hat{p})} \tag{4.22}$$

By fixing an acceptable probability of x being part of an already-existing cluster (0.5 in the experiments), Equation 4.22 can be used to compute δ (1.386 in the experiments).

The new cluster is centered at the x input, has a default diagonal covariance matrix (which corresponds to a circular gaussian cluster), and has a score that starts at 1 in order to avoid newly-created clusters being removed immediately due to a low score:

$$\begin{aligned} \mu &= x \\ \Sigma &= \sigma I \\ \bar{y} &= y \\ \Sigma_{xy} &= 0 \\ s &= 1 \end{aligned}$$

With σ a parameter that gives the initial variance (or fuzziness) of the clusters. σ and ε are the two most important parameters of the Gaussian Mixture Model, as they directly influence its bias and variance.

4.4 Performance optimization and numerical stability

Another challenge of Reinforcement Learning is that the function approximation used must be computationally efficient. A prediction must be made at each time-step, the model is trained once per Q-value update (which can happen several times per time-step if experience replay is used, see Section 1.1.1), and the agent needs a large number of

episodes and time-steps before learning the task. Moreover, online learning does not lead to easy parallelization or distribution across compute nodes.

The first optimization that can be done is to avoid computing expensive values too often. For instance, Equation 4.2 uses Σ^{-1}, the inverse of Σ that is updated in Equation 4.8. However, prediction (and hence the computation of cluster probabilities using Equation 4.2) occurs much more often than cluster updates. When a cluster is updated, the inverse of Σ is therefore pre-computed and stored, which speeds prediction by more than 8 times.

The fraction that appears in Equation 4.2 can also be pre-computed (this is a normalization factor that depends on Σ, but not x), so that the determinant of Σ does not have to be repeatedly computed. This second optimization provides an additional speed-boost of almost 5 times (for both learning and prediction).

Even with these optimizations, the matrix multiplications of Equation 4.6 are quite compute-intensive (two matrix-matrix and one matrix-vector multiplications). The efficiency of the Gaussian Mixture Model can be improved by not computing $f_c(x)$ for clusters having a probability $P(c|x)$ that is below a threshold (1×10^{-5} in the experiments). This allows most of the clusters to be entirely skipped in parts of the input space where they don't really contribute to the output, and speeds prediction by a factor of 2.

The same trick can be applied to the update of clusters. Clusters for which $\mathcal{D}_{\mathcal{M}}(x|\mu, \Sigma)$ (Equation 4.12) is too large (above 9.21 in the experiments, corresponding to $P(x|c) \approx 0.01$ per Equation 4.22) are simply not updated. This avoids having clusters influenced by points too far from their centers, and saves the expensive computation of Σ^{-1} and $|\Sigma|$ for most of the clusters, speeding-up learning by more than 6 times.

Finally, computations are done using single-precision floating point values, which are accurate enough for general use by the model, but lack precision when computing large exponentials. The computation of Equation 4.2 is particularly problematic as a potentially large exponentiated value is divided by a very large value ($(2\pi)^D$ becomes huge as D increases). All the computations are therefore done in logarithmic space, with the exponentiation done only at the very end:

$$\begin{aligned}
\mathcal{N}(x|\mu, \Sigma) &= \frac{1}{\sqrt{(2\pi)^D|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right) \\
&= \exp\left\{\log\left[\frac{1}{\sqrt{(2\pi)^D|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)\right]\right\} \\
&= \exp\left\{\log\left[\frac{1}{\sqrt{(2\pi)^D|\Sigma|}}\right] - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \\
&= \exp\left\{-\log\left[\sqrt{(2\pi)^D|\Sigma|}\right] - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}
\end{aligned}$$

In the above equations, $(2\pi)^D|\Sigma|$ is still a very large value, that becomes of reasonable size only once it passes through the logarithm. However, if the LL^T decomposition of Σ is computed (which can be done because Σ is a covariance matrix biased by Equation

4.8, hence positive definite), its determinant can be computed easily as the product of the diagonal of L (a $D \times D$ matrix):

$$\Sigma = LL^T \tag{4.23}$$

$$|\Sigma| = \prod_i L_{ii} \tag{4.24}$$

This allows to continue the development of $\mathcal{N}(x|\mu, \Sigma)$:

$$\begin{aligned}
\mathcal{N}(x|\mu,\Sigma) &= \exp\left\{-\log\left[\sqrt{(2\pi)^D|\Sigma|}\right] - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \\
&= \exp\left\{-\log\left[\sqrt{\prod_{i=1}^{D} 2\pi L_{ii}}\right] - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \\
&= \exp\left\{-\log\left[\prod_{i=1}^{D} \sqrt{2\pi L_{ii}}\right] - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\} \\
&= \exp\left\{-\sum_{i=1}^{D} \log(\sqrt{2\pi L_{ii}}) - \frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right\}
\end{aligned}$$

Which now performs enough operations in the log-space in order to avoid overflows.

4.5 Experiments

Simple experiments have been carried out in order to assert the quality of the model and to see how it compares to neural networks, that are often used in Reinforcement Learning for approximating Q-values or policies. The testing set is based on the function shown in Equation 4.25, that generates the i-th output component for any input value x (see Figure 4.1 for a 2-inputs 1-output view of this function). The training set simply adds noise to this function, as shown in Equation 4.26:

$$f_{test}(x)_i = cos((i+1)\|x\| + \delta_t) \tag{4.25}$$

$$f_{train}(x)_i = f_{test}(x) + \mathcal{N}(0, \sigma) \tag{4.26}$$

With x a D-dimensional input vector, δ_t a time-dependent value that allows to implement some concept drift, $\|x\|$ the norm of x, allowing the function to be computed for an arbitrary D, and σ a noise factor varied through the experiments.

In order to evaluate a model, a succession of training x_u and testing x_v values are produced, with u and v counters that are incremented every time one of their corresponding x is generated. Two training points are generated and used for (online) training before one testing point is generated and used for evaluation, which means that u increases twice as fast as v.

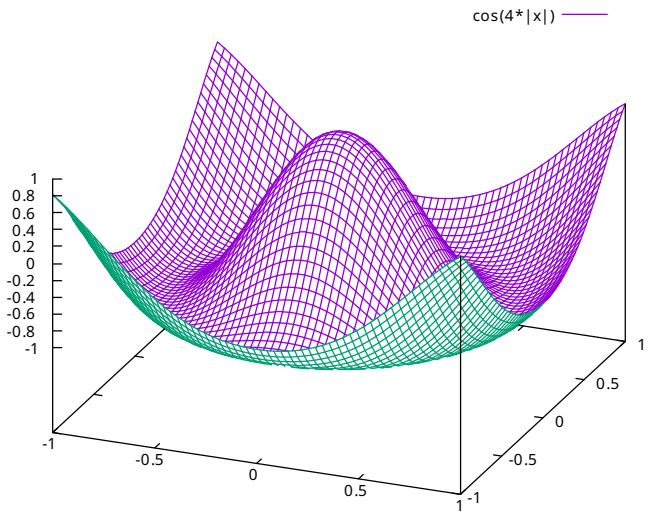

Figure 4.1: Shape of the function described in Equation 4.25

The x values themselves can be drawn from two distributions: uniformly across a $[-1, 1]$ D-dimensional hypercube, or along a u- or v-periodic path that covers all the D dimensions (see Figure 4.2):

$$\begin{aligned} f_j(x) &= \sin(x) \text{ if } j \text{ is odd, } \cos(x) \text{ otherwise} \\ x_j^{u/v} &= f_j(0.001 \times {}^u\!/_v \times (j \bmod 4 + 1)) \end{aligned} \tag{4.27}$$

With j identifying the j-th input dimension. When x is drawn uniformly, training and testing x values are completely uncorrelated. However, training and testing x values sampled on a path are correlated, but their distance varies during the experiment as u and v increase at their own paces, and therefore place them at different points on the path. This distribution approximates the behavior of a Reinforcement Learning agent, that tends to update Q-values associated with states close to the ones for which it queries the model.

4.5.1 Uniform sampling

The first experiment uses an output space of 3 dimensions and an uniformly sampled 20-dimensional input space. This is the sampling method usually assumed by regression models as it has the nicest mathematical properties: input points are drawn independently from each other and the entire input space is covered uniformly. The models are trained incrementally using Equation 4.26 with a noise factor of 0.5. The models are tested after every two training point so that their accuracy over time can be

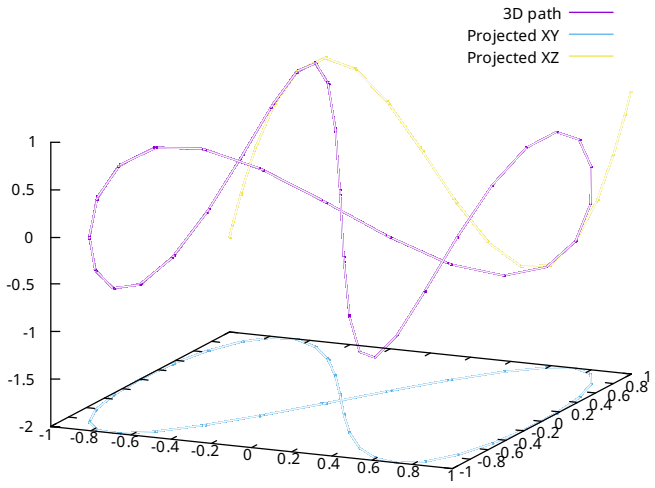

Figure 4.2: Sample of a path in three dimensions. The path is cyclic and produces highly correlated and sparsely-distributed values of x

precisely measured. All the parameters used during the experiments have been found experimentally to maximize the performance of the models on the specific task.

Figure 4.3 shows the performance of a Gaussian Mixture Model with an initial variance σ of 0.5 and a maximum error of 0.25. The error decreases quickly at the beginning of training, but soon reaches a plateau. The large initial variance allows the model to quickly fill the 20-dimensional input space with clusters, but smooths-out its approximation, therefore reducing its accuracy. Allowing a maximum error of 0.25 reduces the accuracy of the model but prevents it from creating too many clusters. Even with a large error, the model creates more than 12 000 clusters and becomes more than 100 times slower at predicting values than the neural network described in the following paragraph.

On the other hand, Figure 4.3 shows that a neural network with 200 neurons in the hidden layer can approximate the function very well. However, neural networks need much more training samples than Gaussian Mixture Models before learning the function, which is problematic for Reinforcement Learning and prevents the network from adjusting to concept drift.

4.5.2 Path sampling

In a Reinforcement Learning context, the most important aspect of a function approximation model is its behavior regarding input vectors highly correlated in time. In this experiment, input vectors are drawn according to Equation 4.27. A testing sample is drawn after every two training samples, which means that training points “advance”

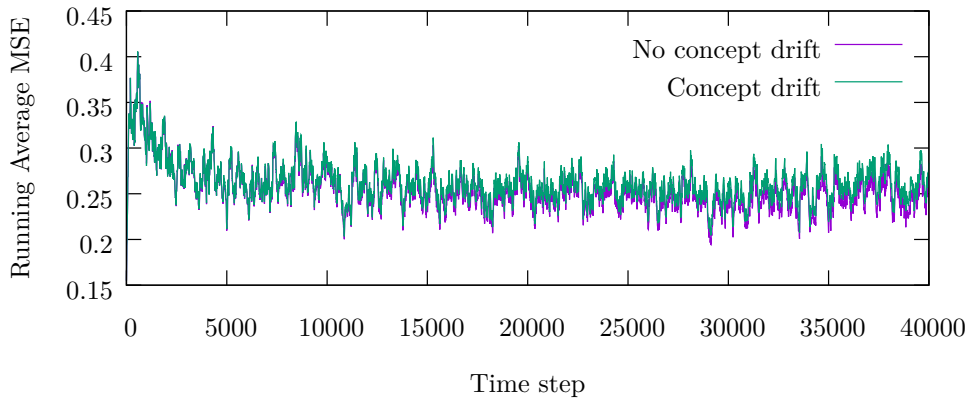

Figure 4.3: Running average of the MSE for a Gaussian Mixture Model with $\sigma = 0.5$ and $\varepsilon = 0.25$. Concept drift has no real impact on the model, but its accuracy remains low. Input points drawn uniformly using the same random seed for both runs.

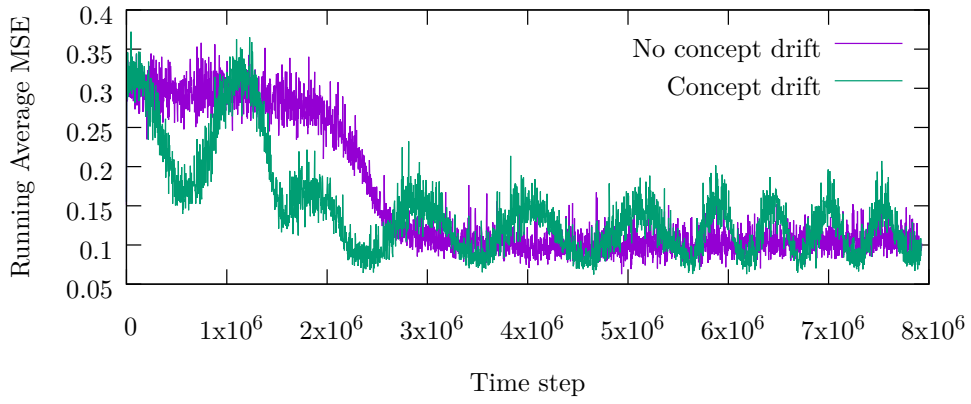

Figure 4.4: Running average of the MSE for a Neural Network with one hidden layer of 200 neurons and a learning rate of 1×10^{-5}. The neural network takes time to learn the function but reaches a MISE of 0.10 (0.26 for Gaussian Mixture Models), except when concept drift is introduced, where its performance starts to oscillate.

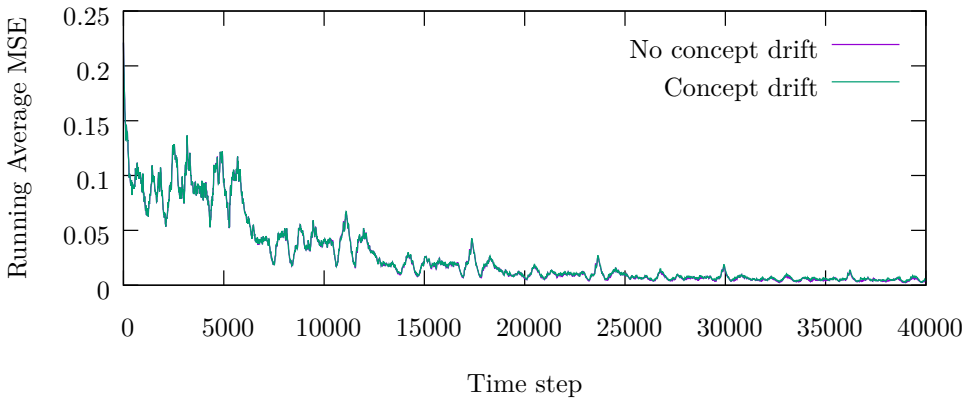

Figure 4.5: Running average of the MSE for a Gaussian Mixture Model with $\sigma = 1 \times 10^{-4}$ and $\varepsilon = 0.01$. Concept drift has no real impact on the model, and its accuracy is now very high (MISE of 0.005).

faster than testing points, and are therefore sometimes close to them, sometimes further away.

Figure 4.5 shows that the Gaussian Mixture Model performs very well when input points are drawn along a trajectory. The input space is only sparsely covered, which allows the model to concentrate small and accurate clusters around the path without having to handle too many of them (at the end of the experiment, the model uses 3141 clusters). The step-wise nature of Figure 4.5 comes from the fact that the training samples are noisy. Every time the trajectory loops over, new training samples become available for points that have already been visited. Because the samples have a zero-mean noise, each new sample allows the model to build a better approximation of the function.

Figure 4.6 shows that neural networks have a hard time learning a function based on time-dependent input-output samples. Even after an extensive tuning of the parameters of the network, no satisfactory learning is possible. The experiment has not been run with concept drift enabled.

4.5.3 Conclusion

These experiments show that neural networks perform better than Gaussian Mixture Models in large dimensions with an uniform sampling and coverage of the input space, which is the most common way of sampling training data. However, Gaussian Mixture Models performed much better than neural networks in experiments where input samples are not independent from each other and follow a path in the input space, as is the case in Reinforcement Learning. Section 7.2 obtains the same results on a more real-world experiment.

These differences come mainly from the fact that Gaussian Mixture Models are local function approximators that are able to store input-output mappings for an indefinite

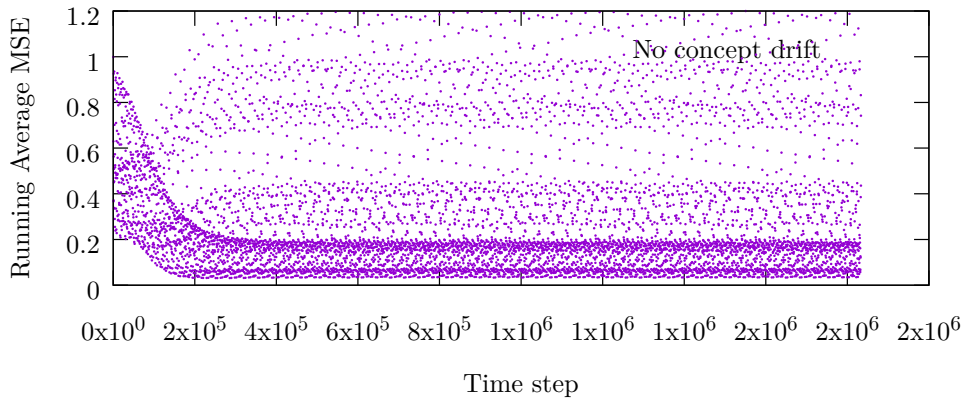

Figure 4.6: Running average of the MSE for a Neural Network with one hidden layer of 20 neurons and a learning rate of 1×10^{-7}. The network learns a function that is acceptable (but not great) most of the time, with some very large errors.

Sampling	Gaussian Mixture Model	Neural Network
Uniform	0.2529	**0.1100**
Uniform Concept Drift	0.2594	**0.1451**
Path	**0.0035**	0.2853
Path Concept Drift	**0.0054**	N/A (> 0.28)

Table 4.1: Summary of the Mean Integrated Squared Error of Gausian Mixture Models and neural networks once converged.

amount of time. Intense training in one part of the input space therefore does not impair the prediction accuracy of the model in another part. Moreover, the model does not need to create any cluster or to store any information related to unexplored areas of the input space.

Neural networks are global function approximators, which are able to generalize their knowledge very well, as shown in the first experiment. Their ability to learn does not depend on the dimensionality of the input, but on the general complexity (frequency, non-linearity) of the function being learned.

Chapter 5

Virtual Sensors

This chapter presents a Reinforcement Learning architecture comparable to Turing machines (see Section 3.4), that allows an agent to focus its attention to specific parts of its input while ignoring the rest. This complexifies the learning problem, as the agent has to learn which parts of its input requires attention in addition to the overall task to be accomplished, but allows a drastic reduction of input-space dimensionality.

The architecture presented in this chapter is close to Attention Models, but replaces the use of backpropagation with a general Reinforcement Learning formulation. This removes several limitations of backpropagation algorithms, that require all the operations performed to be differentiable (no hard cropping of images allowed, for instance), and is subject to local optima.

The architecture proposed in this master thesis builds on the concept of *virtual sensors*. A virtual sensor is a piece of software that produces observations for a Reinforcement Learning agent. The best example, taken from the attention models literature, is a glimpse sensor that focuses its attention on specific parts of an image and allows the

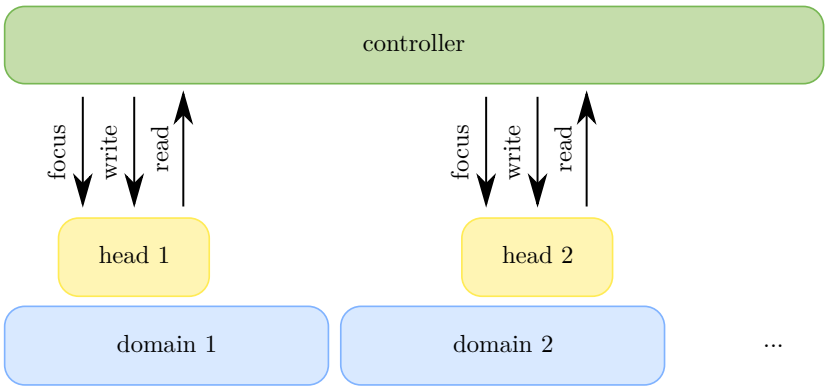

Figure 5.1: General architecture proposed by this thesis. This is a generalization of attention models and neural Turing machines. There can be as many virtual sensors as needed, and the domain of the sensors is problem-specific (images, memory, counters, robot effectors, etc)

Reinforcement Learning agent to observe glimpses. Virtual sensors can be moved (or focused) by the RL agent using actions. This architecture, outlined in Figure 5.1, has a number of nice properties:

- Virtual sensors are moved over a state-space, as done with attention models. This simplifies the learning problem and reduces its dimensionality by allowing the agent to only consider a part of its state-space at a time. Depending on how the virtual sensors are implemented, this can also make learning scale-, translation- and rotation-invariant.
- Several virtual sensors are used, each of them having their own domain. This is inspired from work in neural Turing machines, that usually are multi-tape Turing machines (input, output, internal memory).
- The controller, that learns to move the virtual sensors and to output symbols, is trained using Reinforcement Learning instead of backpropagation or gradient descent. Virtual sensors therefore do not need to be differentiable (they can take square glimpses of images, increment discrete locations of memory, etc).

Figure 5.1 shows a graphical representation of the architecture proposed by this master thesis. A Reinforcement Learning agent (the controller) learns to focus virtual sensors, that provide state information (sensor readings) and accept write commands (toggling values, moving graph nodes, etc).

This master thesis focuses on image recognition using a glimpse sensor. Even if a large variety of problem-specific sensors can be implemented, only two are described here: an image sensor that produces glimpses, and a memory sensor that allows the agent to select which class an image belongs to. Figure 5.2 depicts these two virtual sensors and shows what observations they produce.

5.1 Image sensor

The image sensor can be used to focus the attention of an agent in a 2D color image. The domain of this sensor is a grid of pixels of arbitrary width and height. At the beginning of each episode, an image is given to the sensor, that produces an initial glimpse of it (based on a low-resolution version of the image). The agent can zoom in or out the image and move the sensor in 4 directions. At each time-step, the sensor produces an observation based on the image pixels currently under the glimpse sensor.

5.1.1 Observations

The image sensor provides two kinds of observations: coordinates and actual observations. Coordinates represent the location of the sensor, and other information independent from the pixels under the sensor.

$$zoom = \sqrt{\frac{S^2}{W_i \times H_i}} \tag{5.1}$$

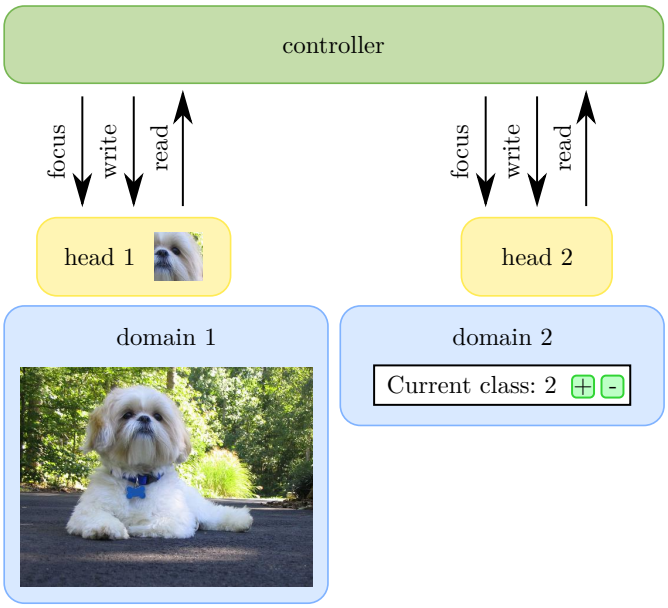

Figure 5.2: Two virtual sensors are used in this master thesis: an image sensor focuses on specific parts of an image and produces glimpses, and a memory sensor (consisting of only one cell in this example) allows the agent to choose to which class the image belongs.

x, y 2D coordinates of the center of the sensor. These coordinates are resolution-independent and normalized between $(0, 0)$ (top-left corner) and $(1, 1)$ (bottom-right corner). The glimpses produced by the sensor are always square and respect the original aspect-ratio of the image.

zoom Percentage of the image area covered by the sensor, which allows the agent to know the scale of things it observes. Equation 5.1 shows how this coordinate is computed based on the ratio between the area covered by the sensor and the total area of the image.

remaining glimpses In most cases, the designer wants to limit the number of glimpses the agent takes before making a prediction. This observation allows the agent to know how many glimpses are left before it must absolutely make a prediction.

visited map 5×5 real numbers representing how much the agent has looked at the image. The map starts completely white (the 25 values are equal to 1), and portions covered by the glimpse sensor are slowly decreased so that they tend towards zero. Portions having a value close to zero therefore correspond to regions of the image that have been regularly observed. This allows the agent to know that it has already looked at some parts of the image and that it may now be interesting to look elsewhere.

Figure 5.3: Random glimpses are extracted from an image. The glimpse classifier is trained to associate all these glimpses to the class of the complete image.

The image sensor also produces a *candidate class* observation. In addition to the coordinates described above, the sensor takes the image pixels it currently covers, converts them to the HSV color space so that color information is more easily extracted, downscales them to an $N \times N$ glimpse (6×6 in the experiments) and feeds them to a neural network classifier. The classifier produces as output a Softmax distribution over the classes the image may belong to.

The neural network used by the image sensor maps glimpses to candidate classes, is easy to train and runs quickly. It is a simple and small feed-forward network, having 3 hidden layers of 100 neurons in the experiments. The goal of the neural network is to reduce the dimensionality of the glimpses. Instead of observing $N \times N$ color pixels, the Reinforcement Learning agent only observes C real values, with C the number of classes in the problem. Moreover, neural networks generalize well across glimpses while Reinforcement Learning algorithms have more difficulties[1]

The neural network, also called the *glimpse classifier*, must be pre-trained before the image sensor can be used. Pre-training the glimpse classifier is the subject of the next sub-section.

5.1.2 Pre-training

Pre-training the glimpse classifier is a very simple operation, summarized in Figure 5.3. Random images are selected in the training set, and a dozen glimpses located at random locations (and zoom levels) are extracted from it. The glimpse classifier is then trained to classify each of these random glimpses in the class to which the complete image belongs.

By continuously training the classifier on random training images, its classification accuracy slowly increases. More interestingly, its accuracy becomes very high on very

[1] In experiments run early in the writing of this thesis, no glimpse classifier was used and the RL agent was able to directly observe pixels. It achieved very high accuracies on the training set, but very poor on the testing set. Even if training and testing images are drawn from the same distribution, the RL agent sees the testing images as different *environments* and fails to transfer its knowledge to them. Transfer Reinforcement Learning is still an actively-researched field.

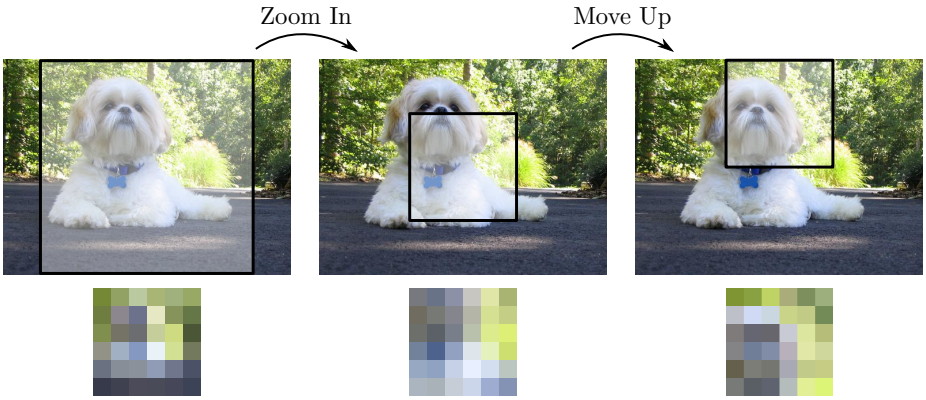

Figure 5.4: Initial glimpse sensor location (left), followed by its location after two actions are taken. 6×6 glimpses produced by the sensor (before being converted to HSV and classified) are shown at the bottom of the images.

specific glimpses (that are highly correlated with a class, for instance the eye of a dog), while it remains low on useless glimpses (grass or foliage appear in all the images regardless of the dog they represent).

Because the glimpse classifier produces a Softmax distribution over the classes, it is possible to measure its confidence in its prediction. The goal of the Reinforcement Learning agent, once the glimpse classifier is trained, is to move the image sensor to parts of the image where the glimpse classifier is highly confident in its prediction.

5.1.3 Actions

At the beginning of a run, the glimpse is located at the center of the image and is completely zoomed-out. The image sensor can be focused on its domain using 6 different actions. The first 4 ones move the sensor (top, left, bottom, right), the two other ones control the zoom factor (zoom in, zoom out). Zooming in divides S by two, while zooming out multiplies S by two, with the image sensor covering an $S \times S$ square in the image. After the operation, S is clamped between $\frac{1}{16}$ and 1.

Moving the sensor consists of moving its center in the direction corresponding to the action that has been performed, for a distance equal to half its size. For instance, moving the sensor to the top is done by computing $(x_{t+1}, y_{t+1}) = (x_t, y_t - \frac{S}{2})$. Figure 5.4 shows short sequence of actions and their effect on the glimpse sensor and the glimpses it produces.

5.1.4 Performance optimization

In order to speed-up learning and prediction, performing actions and producing observations must be as fast as possible. For the image sensor, the bulk of the processing concerns the extraction of an HSV glimpse from the image. In order to speed up this operation, as much as possible is performed early and kept in a cache.

When an image is first loaded and associated with the sensor, it is scaled-down to the minimum resolution that still allows the sensor to reach its maximum precision. Because the zoom level of the sensor is limited, the dimension S of the sensor will never be smaller than $\min(\frac{W_i}{16}, \frac{H_i}{16})$. Because the glimpse taken by the sensor has a size of $N \times N$ pixels, the image does not need to be bigger than $16N \times 16N$. For a small N (usually 4 to 8), this allows a fair reduction of image size (128×128 instead of full-resolution). In order to avoid aliasing issues, the image is scaled-down to two times its minimum size as computed by the above formulas.

Once the image has been scaled-down, it is converted to HSV, stored in-memory and serialized to a binary cache. Training a classifier requires many runs on the same image, so images will always be used several times. Serializing them to a file also allows repeated experiments to run faster, without having to re-read, scale and convert all the images in the dataset every time they are needed.

5.2 Memory sensor

Reinforcement Learning agents can only interact with their environment by performing actions. If the agent has to produce classes, it is possible to have one action per class. Triggering the action selects the class and ends the episode. This solution is sometimes used in incremental Reinforcement Learning in order to build solution components [MDG09], but cannot be extended to continuous output values. Moreover, having a Reinforcement Learning agent learn to perform one action, the best one, then stop, reduces it to a simple Supervised Learning agent (but with some Reinforcement Learning overhead). It is also possible to have one agent per class. Each agent is run on the input, explores it, and the agent corresponding to the class of the input recognizes it and triggers a specific "it's me" action [WHPS11].

The solution proposed in this master thesis builds on incremental solution construction, attention models and neural Turing machines. Instead of producing a value (class or anything else), the Reinforcement Learning agent writes it in some memory location. The memory sensor can be focused by the agent to a specific memory cell, that can be incremented or decremented. The agent senses the index of the cell currently under the sensor (the coordinates of the sensor) and the value it contains (the observation of the sensor). In the experiments, the memory sensor contains only one memory cell, but generalization to several memory cells is easy (this allows some sort of scratch memory, or multiple classification as in [BMK14] where the agent selects the classes corresponding to different parts of an image).

At the beginning of an episode, the memory sensor is focused on the first cell, numbered 0, that contains 0. If the agent tries to move the sensor before the first cell or after the last one, nothing happens. Each memory cell also has a minimum or maximum value, which allows the designer of a problem to choose whether cells represent bits or integer values in an interval (a class number for instance). Incrementing or decrementing a cell past its range does nothing. The agent receives a reward of 0 at any time-step, except the last one of an episode.

$$r_i = \begin{cases} 0 & : M_i \neq T_i \\ 1 & : M_i = T_i \end{cases} \tag{5.2}$$

$$r = 1 + \sum_i r_i \tag{5.3}$$

After a predefined number of time-steps, the episode is stopped and the agent gets a positive reward. For each memory cell, the agent gets a reward of +1 if the contents of the cell matches its target value (training class of an image for instance) or +0 if the contents of the cell don't match its target value. A one is added to the reward to make it always strictly positive (see Equation 5.3).

If the image sensor ends the episode because too many glimpses have been taken, before the memory sensor ends the episode itself, the agent receives a reward of -2 (given by the image sensor), plus the reward of 0 given by the memory sensor at time-steps during which it does not end the episode itself. This means that the agent receives a reward of -2 at the end of the episode if it takes too many glimpses, 1 if it does not exceed the maximum number of glimpses but fails to set any memory cell to its expected value, and more than 1 if it properly sets memory cells.

5.3 Using several sensors at once

The Reinforcement Learning agent uses several sensors at once. For instance, it focuses and observes an image sensor and several memory sensors. As each sensor provides its own observations and exposes its own actions, a combined view of them must be defined.

The observation given to the Reinforcement Learning agent is the concatenation of all the coordinates and observations produced by the sensors. The reward is the sum of the rewards provided by all the sensors:

$$s_t = \bigcup_i C_t^i \cup O_t^i \tag{5.4}$$

$$r_t = \sum_i r_t^i \tag{5.5}$$

$$f_t = \bigwedge_t f_t^i \tag{5.6}$$

With s_t the observation made at time t, r_t the reward obtained at time t, and C_t^i, O_t^i and r_t^i the coordinates, observation and reward exposed by the i-th sensor at time t. The episode finishes when f_t is true, that is when any of the i sensors wants the episode to finish.

Combining sensors this way allows the system to be very general and to be able to accommodate any number of sensor. However, this requires that all the sensors produce observations, rewards and finished signals that are independent from each other, which prevents the memory sensor from changing its reward function based on information from the image sensor, for instance. This limitation of the framework can be circumvented

by allowing some sensors to know each other. For instance, two memory sensors could have references to each other and use them to manage a common reward function.

5.4 Summary

Here is a short summary of the sensors that are available to the Reinforcement Learning agent. Each sensor exposes a set of actions, an observation and coordinates. For the Reinforcement Learning agent, observations and coordinates are merged as a single large observation.

	Image	**Memory**
Domain	Image to classify	Memory cells
Observations	Candidate classes	Current cell's value
Coordinates	$(x, y, zoom, map)$	Current cell index
Actions	4 moves, zoom in/out	2 moves, Inc./dec. current cell
Reward	-2 after 5 actions	At the end: 1 + num. correct cells

Chapter 6

Distributed Reinforcement Learning Algorithms

Modern computers contain from 2 to a dozen CPU cores, which can be used to execute more than one instruction stream at once. Due to current limitations in manufacturing techniques, the absolute speed of a single CPU only increases marginally each year and is reaching a plateau [TP06]. Speeding up computers is now only possible by miniaturizing CPU cores and putting more of them in a single chip [OH05].

Graphical Processing Units (GPUs) push parallelization even further as they contain several thousand cores per die[1]. A very high number of cores and fast memory allow GPUs to perform massively parallel operations, like matrix multiplications or neural network passes, much faster than CPU cores [BBB+10], while also being much more power efficient.

The main drawback of multiple CPU cores and GPUs is that leveraging their parallelization abilities requires substantial work from the developer of an algorithm. Moreover, CPU cores and GPUs must be installed in the same physical computer, which limits their total performance as computers have limited power draw, space, storage solutions and memory.

Another trend in high-performance computing is the use of many low-power computers. Instead of developing an algorithm that can run on a GPU and several CPU cores, it is designed to run on several computers at a time, each one communicating with the others over the network. This is known as distributed computing and allows to scale to thousands if not millions of interconnected machines.

Many algorithms related to Reinforcement Learning, mainly neural networks, have been implemented on GPUs and multi-core CPU systems [BBB+10, Col11, KSH12]. However, higher-level algorithms like Q-Learning or variants of Policy Iteration are much less commonly parallelized, even if work in that direction exist [MBM+16, NSB+15].

This chapter proposes two distributed Reinforcement Learning algorithms, based on Q-Learning and Policy Iteration. The algorithms make hypotheses based on the fact that they are used in environments comparable to what Chapter 5 proposes:

[1] The nVidia GTX Titan Z contains 3072 CUDA cores [Smi15b], the AMD Radeon Fury X contains 4096 "stream processors" [Smi15a]

- Interaction with the environment is fast and cheap (no robotic arm needs to move).
- The environment is easily reset to a previous state.
- The environment has no physical reality and can be instantiated as much as needed. Every instance has the exact same dynamics as the others.

6.1 Value Iteration

Value Iteration algorithms, such as Q-Learning and presented in depth in Section 1.2, iteratively compute the value of each action in each state. The value of a (s, a) state-action pair is the expected cumulative reward obtainable from state s by taking action a and then following the policy π. Q-Learning uses $(s_t, a_t, r_{t+1}, s_{t+1})$ interactions with the environment to update estimates of state-action values by repeatedly applying the Bellman equation [Bel56]:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t)) \tag{6.1}$$

When learning is finished, the optimal policy is obtained by selecting the action with the maximum value in the current state. When learning, the agent typically selects a "good" action based on their values, then slightly randomizes its behavior in order to explore its environment.

$$a_t = \operatorname*{argmax}_a Q(s_t, a) \tag{6.2}$$

A more detailed description of Q-Learning, its variants, how exploration works and how actions are selected is presented in Sections 1.2 and 1.4. This section builds on basic Q-Learning and presents extensions that allow it to be more efficient at learning.

6.1.1 Experience replay

Experience replay consists of storing experiences in a buffer, and to reuse them later on to speed up learning. Instead of updating a single Q-value after each interaction with the environment, by applying Equation 6.1 once, the agent updates several Q-values using its last interaction with the environment and N experiences from the buffer [Lin92]. Usually, experiences that are replayed are selected randomly, but [ABB12] suggests that selecting sequential experiences may prove useful in some contexts. In this master thesis, experiences are selected randomly.

Algorithm 2 shows how the agent collects experiences, and Algorithm 3 shows how the agent learns from them.

Experience replay provides the same benefits as eligibility traces, and are a bit more efficient than them in complex environments [ABB12].

Algorithm 2 Every interaction with the environment is stored in a buffer as an experience. In practice, the size of the buffer is limited to a couple thousand experiences and old ones are removed periodically.

for $t = 0..\infty$ **do**
  Observe s_t
  if $t \neq 0$ **then**
    $E \leftarrow E \cup \{(s_{t-1}, a_{t-1}, r_t, s_t)\}$
    Learn using N random experiences from E
  end if
  Choose $a_t = \text{argmax}_a\, Q(s_t, a)$
  Execute a_t
  Obtain reward r_{t+1}
end for

Algorithm 3 Learning from experiences, based on Q-Learning. This algorithm ensures that the last experience is always used for learning, so that new data is used as quickly as possible.

Require: Experiences E
for $i = 0..N$ **do**
  if $i = 0$ **then**
    $(o_t, a_t, r_{t+1}, o_{t+1}) \leftarrow$ last element of E
  else
    $(o_t, a_t, r_{t+1}, o_{t+1}) \leftarrow$ random element from E
  end if
  $\delta = r_{t+1} + \gamma \max_a \hat{Q}(o_{t+1}, a) - \hat{Q}(o_t, a_t)$
  Train $\hat{Q}(o_t, a_t)$ towards $\hat{Q}(o_t, a_t) + \alpha\delta$
end for
Remove old experiences from E in order to keep its size smaller than 5000

6.1.2 Exploration and exploitation

The performance of a Reinforcement Learning agent depends greatly on how it balances exploration and exploitation. If the agent always takes actions it thinks are optimal, it never discovers new parts of the environment and does not learn anything. If the agent always takes random actions, it never uses what it learns. A good balance between exploration and exploitation is needed, and usually takes one of these forms:

1. Select a random action with probability ε, and the optimal one with probability $1 - \varepsilon$.

2. Compute a Softmax distribution over the actions, which allows actions that are much better than the others to be selected with great probability (see Equation 6.3)

3. Adjust ε or the Softmax temperature over time, so that the agent exploits more and more as it discovers the dynamics of the environment.

$$p(s,a) = \frac{\exp(\frac{Q(s,a)}{\tau})}{\sum_{a'} \exp(\frac{Q(s,a')}{\tau})} \tag{6.3}$$

In the environment described in Chapter 5, careful balance between exploration and exploitation is required. The environment is large and complex, the reward is sparse and can be obtained only if a very precise sequence of actions has been taken. This requires much exploration early in learning, but the exploration must go down quickly afterwards so that the agent starts to exploit sufficiently to reach a good solution often enough to learn useful skills.

The algorithm proposed by this thesis is based on the Softmax action selection scheme, and slowly decreases the temperature τ over time using an algorithm inspired by [Tok10]. The idea is to set τ based on a running average of the temporal difference error. When the Q function is badly approximated, the temperature is high and the agent explores much (bad approximations often indicate that the agent has not properly learned the Q-values). When the Q function is more accurate, the temperature decreases and the agent starts to exploit its knowledge:

$$\bar{\delta}_t = (1-\alpha)\bar{\delta}_{t-1} + \alpha * |\delta_t| \tag{6.4}$$

$$\tau_t = T\bar{\delta}_t \tag{6.5}$$

With δ_t the temporal-difference error made at time t (the difference between the predicted and update Q-value of the action that has been taken [Sut88]), α a small constant that defines the time-scale of the running average (0.005 works well in the experiments), T a constant that allows to define the magnitude of the temperature (between 0.2 and 1.0 in the experiments), and τ_t the Softmax temperature at time t.

Using those formulas, the temperature is low at the very beginning of the experiment, because most of the Q-values are still equal to zero (the default value of the model), and hence produce only small errors when the agent does not receive any reward. At this time, exploration comes from the Softmax distribution that gives comparable probabilities to all the actions. Once the agent starts to obtain some rewards, the TD-error quickly increases and the temperature reaches values near 1. After some time, the $\hat{Q}(s,a)$ model becomes more accurate and the temperature goes down. This low temperature combined to the great difference between the value of good and bad actions allows the Softmax action selection policy to nearly always select the best action.

Figure 6.1 compares two fixed temperatures (τ_t = constant) with the adaptive temperature proposed in this thesis. Results are taken from the experiments run in Section 6.3 and show that using an adaptive temperature provides a significant advantage over fixed temperatures both in learning speed and performance.

A very interesting behavior of this variable temperature is that it does not always have to decrease. For instance, the agent may learn good Q-values in a specific part of the environment, decrease its temperature, and start to exploit its knowledge, which allows it to reach a new part of the environment. The Q-values in this new part are

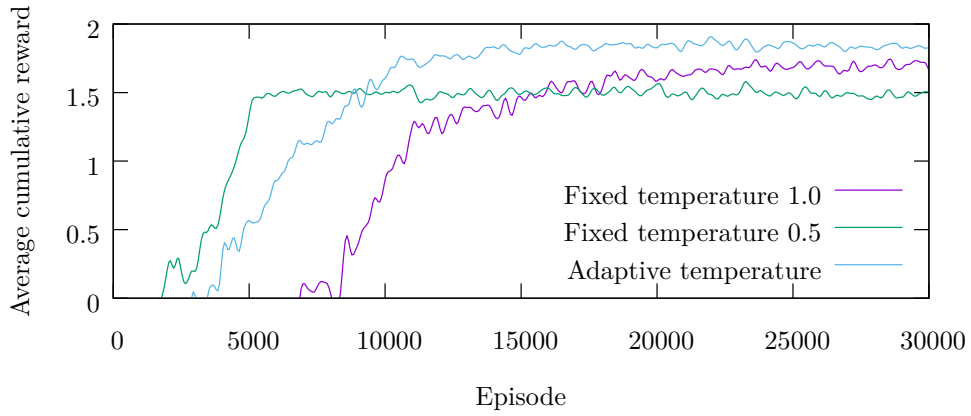

Figure 6.1: Average cumulative reward over episodes for different Softmax temperatures (experiments in the environment described in Section 6.3). A high temperature allows the agent to learn reasonably well but prevents it from exploiting sufficiently to achieve a high performance. A low temperature discourages exploration and leads to a sub-optimal policy in which the agent becomes stuck. Adaptive temperature allows the agent to learn quickly without being stuck in sub-optimal policies, and allows it to exploit once learning is complete. No experience replay nor parallelism used, initial variance is 0.01.

quite inaccurate, which causes the agent to raise the temperature and start to explore a bit more. The cycle starts over several times until the agent manages to learn and exploit its knowledge of the complete environment.

6.1.3 Double Q-Learning

Double Q-Learning consists of maintaining two Q-function models that stabilize each other. Double Q-Learning also prevents Q-Learning from over-estimating Q-values [HGW10]. The Q update rule of Equation 6.1 is modified so that one model, Q^A, uses as target the other model, Q^B:

$$Q^A_{k+1}(s_t, a_t) = Q^A_k(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q^B_k(s_{t+1}, a) - Q^A_k(s_t, a_t)) \tag{6.6}$$

In this master thesis, Q^A and Q^B are Gaussian Mixture Models as described in Chapter 4. When learning, Q^A is modified but Q^B remains fixed. Every 4 episodes, Q^A is copied into Q^B so that both models become identical (this updates Q^B to the latest information available). This differs from what [HGW10] proposes (updating Q^A using Q^B as target or Q^B using Q^A as target randomly at each time-step) but provides better results in the experiments.

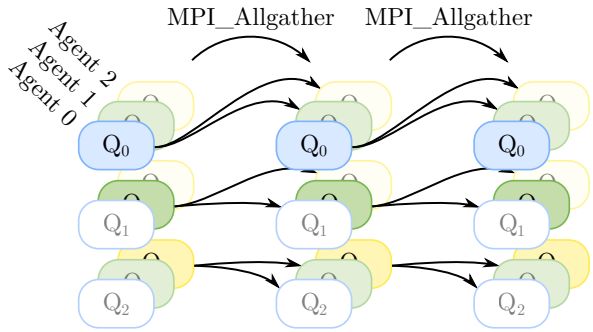

Figure 6.2: Every agent (one color per agent) updates its own Q function (bright colors) and stores copies of the other agent's Q functions (light colors). Every few episode, all the agents send their Q function to all the other ones, that use this information to update their copies of the Q functions. This allows to parallelize learning (each agent learns only one Q function) but duplicates predictions across all the agents.

6.1.4 MPI implementation

Reinforcement Learning algorithms are inherently sequential because the agent must take a sequence of actions, and has to wait for observations between actions. It is for instance impossible to simulate N successive time-steps in parallel, as the action chosen at time t depends on the outcome of what happened at time $t-1$.

However, Q-Learning algorithms can be parallelized with some tricks. In [MBM+16], for instance, several agents interact with the environment, each agent in its own instance so that they don't interfere with each other. [MBM+16] makes the the hypothesis that the Q function is approximated by a neural network, and allows each agent to compute partial gradient updates based on their experiences. The partial gradient updates are periodically combined and sent to all the agents, that all update their Q function at once.

This approach works very well when interactions with the environment are much slower than Q updates, as interactions are run in parallel by each agent. However, all the agents need to train the neural network, which duplicates work without any speed gain if this operation represents most of the compute time of the algorithm. Other algorithms exist and try to be clever about how the agents communicate and what they learn themselves [NSB+15].

The algorithm proposed in this section makes no hypothesis about how the Q function is stored, and assumes that training the function approximator is much slower than making a prediction. As in [MBM+16], several agents are executed in parallel in different instances of the environment. Each agent maintains its own Q function and never learns from other agents, which allows learning to be fully parallelized and avoids duplicating work between the agents.

Each agent also possesses a copy of the Q function of all the other ones. At each time-step, the Q-values at the current state are computed by mixing the predictions of all the available Q functions (the one of the agent, and the copies from other agents),

Algorithm 4 Every agent learns its own Q function (Q_self) but uses the Q estimates from all the other agents when choosing an action.

Let Q_i be the Q function model of agent i (self included)
for $t = 0..\infty$ **do**
  Observe s_t
  Compute $Q(s_t, a) = \text{merge}_i\ Q_i(s_t, a)$
  Choose $a_t = \text{argmax}_a\, Q(s_t, a)$ and execute it
  Obtain reward r_{t+1}
  Compute $\delta_t = r_{t+1} + \gamma \max_a Q_{self}(s_{t+1}, a) - Q_{self}(s_t, a_t)$
  Train $Q_{self}(s_t, a_t)$ towards $Q_{self}(s_t, a_t) + \alpha\delta_t$
  if some time passed **then**
    Send Q_{self} to all the agents
    Update Q_i with $i \neq self$ using models received from other agents
  end if
end for

much like mixtures of experts work. After every couple episodes, all the agents send their updated Q function model to all the other ones so that copies of these models remain relatively up to date. The more often this synchronization is done, the more accurate the Q-value estimates are, but too many synchronizations take time and prevent the algorithm from scaling to large number of compute nodes.

Algorithm 4 illustrates how agents compute Q-values using their Q function approximator and copies of the Q functions of the other agents. Figure 6.2 depicts all the communications that happen over the network, along with the MPI functions actually used by the implementation.

In Algorithm 4, Q-values obtained from all the agents need to be merged. This merge can take three forms, one of which is statistically correct (the average, which is used in this master thesis), the three other ones providing interesting results that may require more research:

average Average the Q-values obtained by all the agents. This strictly corresponds to how mixtures of experts work, and gives good results regardless of how the reward function is computed.

max Take the maximum Q-value returned by the agents. This performs better than *average* when the Q-function is initialized pessimistically and the environment is deterministic.

min Take the minimum Q-value returned by the agents. This provides good results when the Q-function is initialized optimistically and the environment is deterministic.

largest magnitude If the Q-value having the largest absolute value is positive, use *max*. Use *min* otherwise. This merge operation requires more theoretical and practical research as it seems to outperform *average*, *min* and *max* regardless of

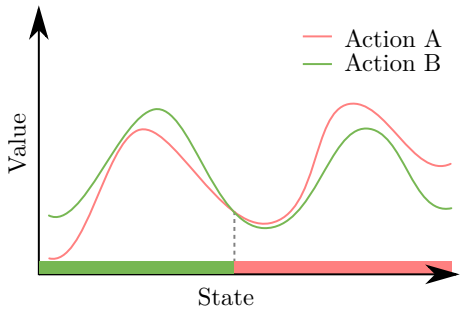

Figure 6.3: A complex highly non-linear value function leads to a very simple policy (greedy policy depicted at the bottom of the graph).

how the Q function is initialized, but may subtly depend on the shape of the reward function. It also likely requires a deterministic environment.

6.2 Rollout-Based Policy Iteration

While Value Iteration algorithms update a value function and use it to choose actions, Policy Iteration algorithms directly learn a policy. The main advantage of these algorithms is that policies are generally much simpler than value functions, as illustrated on Figure 6.3.

The algorithm presented in this section is inspired by [LP03], and consists of computing an empirical estimate of the Q function whenever an action must be chosen. Q-values are estimated by averaging the discounted cumulative reward obtained by N rollouts (sequences of actions) that all start from the same initial state s_t, take a as first action, and stop at a terminal state:

$$Q_\pi(s,a) = \frac{1}{N}\sum_N \text{rollout}_\pi(s,a) \tag{6.7}$$

$$= E_\pi\left[\sum_{k=0}^{\infty}\gamma^k r_{t+k} \mid s_t = s, a_t = a\right] \tag{6.8}$$

By comparing Equations 6.8 and 1.4, we see that performing rollouts allows to compute an unbiased estimate of the Q-value of a state-action pair. The outcome of rollouts is used to greedily choose the action to execute, by selecting the action with the greatest Q-value.

Performing rollouts requires that a policy π, used to choose actions in the rollouts, exists. Several algorithms use random policies [Hes12], but using learned non-random policies allows the rollouts to be more informative, biased towards good trajectories and states. Moreover, performing rollouts using a policy π instead of a random one leads to an estimate E_π instead of E_{rand} in Equation 6.8, which matches what Equation 1.4 expects.

Algorithm 5 Using rollouts to compute Q-values, used to define a policy represented by a classifier. Note that using rollouts assumes that the environment can be reset to any previous state when a rollout finishes (so that the next one can start).

function POLICYITERATION
  for $t = 0..\infty$ **do**
    Observe s_t
    Compute $Q(s_t, \cdot) = \frac{1}{N} \sum_N \text{rollout}_\pi(s_t, \cdot)$
    Choose $a_t = \text{argmax}_a Q(s_t, a)$
    Execute a_t
    Obtain reward r_{t+1}
    Train the π classifier with (s_t, a_t)
  end for
end function
function ROLLOUT(s_t, a_t)
  outcome $\leftarrow 0$
  for $k = 0$..terminal state reached **do**
    Observe s_{t+k}
    Choose $a_{t+k} = a_t$ if $k = 0$, $\pi(s_{t+k})$ if $k > 0$
    Execute a_{t+k}
    Obtain reward r_{t+k+1}
    outcome $\leftarrow$ *outcome* $+ \gamma^k r_{t+k+1}$
  end for
  Reset environment to state s_t
  return *outcome*
end function

[LP03] proposes to start with a random policy that is progressively trained using the outcome of rollouts. At each time-step, the agent performs rollouts, which allows it to compute Q-values. Then, it chooses the optimal action a_t^* in state s_t, executes it and trains a policy classifier mapping states to actions on (s_t, a_t^*). This classifier can be used by future rollouts as policy, and should tend towards the optimal policy as its accuracy increases [LP03]. Algorithm 5 describes this algorithm in more detail.

6.2.1 MPI implementation

Algorithm 5 can be efficiently parallelized at two places:

- At each time-step, N rollouts can be executed in parallel. This requires the environment to be distributable across threads or compute nodes, as each rollout execution needs to be independent from the others. More precisely, independent instances of the environment must be available, or some sort of thread safety must be implemented in the environment.

- After each episode, the agent has produced a series of (s, a^*) tuples on which π must be trained. Several agents can execute in parallel and periodically exchange

Algorithm 6 Each agent interacts with its own instance of the environment, performs rollouts and produces (s, a^*) tuples. The agents periodically exchange these tuples and use them to train their own π classifier, used to perform rollouts.

```
function AGENT
    T ← ∅
    for i = 0..∞ do
        T ← T ∪ Episode()
        if i is a multiple of K then
            Send T to all the other agents
            Receive T_a from all the other agents a (self included)
            T ← ⋃_a T_a
            Train policy π on the (s, a*) tuples found in T
            T ← ∅
        end if
    end for
end function
function EPISODE(s_t, a_t)
    T ← ∅
    for k = 0..∞ do
        Observe s_t, choose a_t using rollouts
        T ← T ∪ {(s_t, a_t)}
        Execute a_t and obtain reward r_{t+1}
    end for
    return T
end function
```

their (s, a^*) tuples, so that every agent learns from what the others have done.

The first solution does not degrade the quality of π, as performing rollouts in parallel then averaging their outcomes corresponds exactly to how the sequential algorithm works. However, the scalability of this solution is limited by N, as it is impossible to parallelize N rollout executions to more than N compute nodes.

The second solution is interesting because it allows any number of agents to interact in their own instance of the environment. However, the agents must wait some time (at least an episode) before receiving the (s, a^*) tuples of the other agents, which may slightly slow learning down. This is nevertheless the solution that has been implemented in this master thesis, and it provided very good results in practice. Algorithm 6 describes how the agents produce $(s, a*)$ tuples, send them to other agents and learn from them.

6.3 Experiments

The algorithms presented in the previous sections have several parameters that must be tuned and that affect the speed and quality of learning. This section presents experiments that highlight the impact of different parameters and allow to compare the algorithms.

Because the algorithms presented in this master thesis are compared against themselves, reduced and non-standard training and testing set can be used. By reducing the size of the dataset, experiments run faster, which allows more of them to be run. A complete evaluation of how the work proposed in this chapter scales to larger datasets is carried out in Chapter 7.3. Chapter 7.1 compares Reinforcement Learning as implemented in this thesis with other classification approaches.

The experiments in this section consists of running a Reinforcement Learning agent as described in Chapter 5 on a reduced version of the MNIST hand-digit dataset [LCB98] that only considers ones and twos[2]. An image sensor observes 6×6 glimpses of the 28×28 MNIST images. The glimpse classifier consists of a feed-forward neural network of $6 \times 6 \times 3$ inputs, 3 hidden layers of 100 neurons each, and two outputs (one per class). A memory sensor of one binary memory cell allows the agent to select whether the digit it observes is a one or a two. The agent has actions to move the glimpse sensor (4 directions, zoom in, zoom out), and can increment or decrement the contents of the memory cell. There is also one action that does nothing, which allows the agent to remain idle once it is satisfied of its choice of class, before the episode ends (episodes run for 9 time-steps).

The Reinforcement Learning agent uses the distributed Q-Learning or Policy Iteration algorithms, depending on the experiment. The Q function or policy is approximated using a Gaussian Mixture Model as described in Chapter 4. The parameters of the model are set to values described in Chapter 4, except the maximum error before creating a new cluster that is set to a very low value (0.01), and the initial variance (initial size of the clusters) that varies from experiment to experiment.

The experiments consist of observing the effect of different parameters on the speed and quality of learning. These parameters are as follows:

Initial Variance Initial variance of Gaussian clusters, roughly translates to the smoothness of the Gaussian Mixture Model. The lower this value is, the most accurate but prone to overfitting the model is. High values produce a smoother approximations of the function.

Parallel instances Number of compute nodes used to run the experiment. This parameter allows to measure how parallelization affects learning, for instance by reducing the variance of Q estimates as more models are averaged.

Experience Replay (Value Iteration) Number of experiences that are replayed at each time-step. Influences learning speed but tends to make the agent fall in local optima.

Softmax temperature (Value Iteration) T parameter of Equation 6.5. This parameter influences how the agent explores the environment. Increasing it tends to slow down learning but makes the agent fall in local optima, except if a high number of parallel instances are used.

[2]Images are extracted from the training data available on [LCB98]. Half of these images are randomly assigned to the *training set*, the remaining ones are part of the *testing set*.

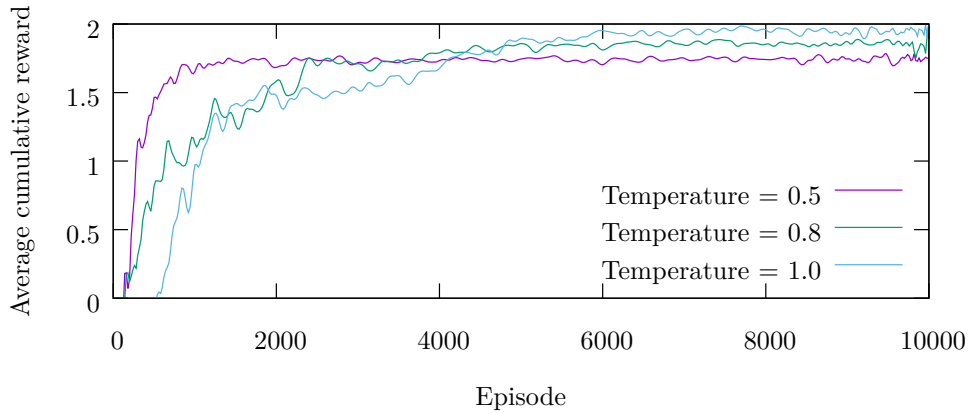

Figure 6.4: Higher Softmax temperatures slow down learning but increase the performance of the agent once it has learned. Initial variance is 0.010, experience replay is 2 and parallelization is disabled.

Rollouts (Policy Iteration) Number of rollouts performed at each time-step. More rollouts require more CPU time but allow a more precise estimate of the Q-value of a state-action pair.

The following sub-sections present results obtained by fixing some parameters and varying others. Each data line presented results from 4 runs using a specific set of parameters. The figures show which parameters have been changed between each data line, while their captions describe at what value the other parameters were fixed.

If several agents are run in parallel, episodes of each of them are counted (episode 200 is reached when every agent has performed $\frac{200}{N}$ episodes, for instance). This allows to compare how data-efficient several agents are, but does not show the total wall-clock time needed to reach a given episode. For small numbers of agents (below 16) and a fast network connection[3], the number of episodes run per second scales linearly with the number of parallel agents.

6.3.1 Temperature and Parallelism (Value Iteration)

Two parameters that have a great influence on learning speed and quality are the Softmax temperature and how many compute nodes are used to run the experiments. These two parameters influence the exploration/exploitation trade-off and have an inter-dependent effect.

At low temperatures, the agent exploits too much and fails to explore the environment. It quickly learns not to take too many glimpses, as this behavior is easy to learn (the agent observes the number of glimpses it can still take, but it often fails to correctly

[3]The algorithms scale better across the network than on a single computer, as each agent is strongly memory-starved. Running 4 agents on the same computer makes them fight for RAM bandwidth, while this problem does not occur over the network

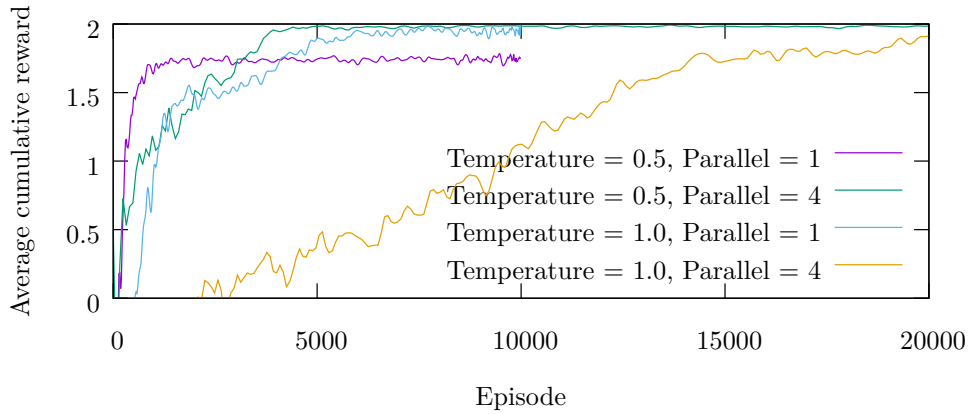

Figure 6.5: Lower temperatures become usable when parallelism is increased, as more parallelism leads to more "experts" in the mixture of experts used in Algorithm 4. At high temperatures, adding compute nodes does not provide any speed-up. This figure shows that the best performance is achieved by using a low temperature and several compute nodes. Initial variance is 0.010 and experience replay is 2.

classify the images and gets an average cumulative reward of 1.5 (half the classifications are correct, thus yielding a reward of 2, while the others are incorrect, thus yielding a reward of 1).

When the temperature increases, learning is much slower but the agent finally manages to learn the task and achieves a very good score. It is interesting to note that a high temperature does not prevent the agent from successfully classifying images, as Equation 6.5 multiplies it with the average TD-error made by the agent. This means that the actual temperature goes to 0 as the agent becomes more and more precise. High temperatures slow down learning but does not reduce the maximum accuracy achievable. Figure 6.4 illustrates the impact of temperature on learning speed and accuracy.

The impact of parallelism on the results is more subtle and more complex. Moreover, it depends on the temperature. At high temperatures, increasing the number of compute nodes does not speed up learning. If the agent runs on N cores, it runs N times more episode per second but needs about N times more episodes before learning the task, thus losing all the benefit of parallelization. However, more parallelism allows lower temperatures to still work, which at the end makes learning faster (see Figure 6.5, green line).

6.3.2 Initial Variance (Value Iteration)

Initial variance allows to configure the smoothness of the Gaussian Mixture Model used to approximate the Q function. The higher this value is, the smoother the function becomes. High values allow more generalization between states but also introduce errors. Figure 6.6 shows that increasing the initial variance speeds learning up (as more generalization between states allows the agent to learn a decent behavior more quickly), but decreases

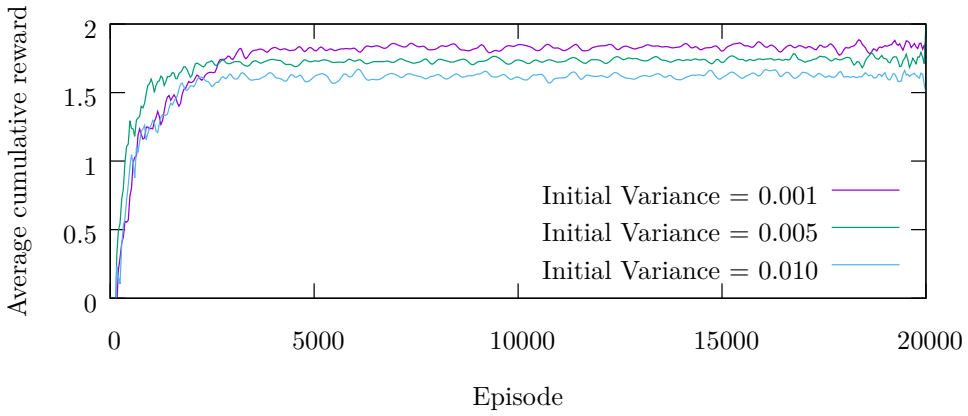

Figure 6.6: Lower initial variance slows down learning but allows higher performance once learning is done. Experience replay is 2, temperature is 0.5 and parallelization is 2. Using other parameters, the effect is still there but much less visible.

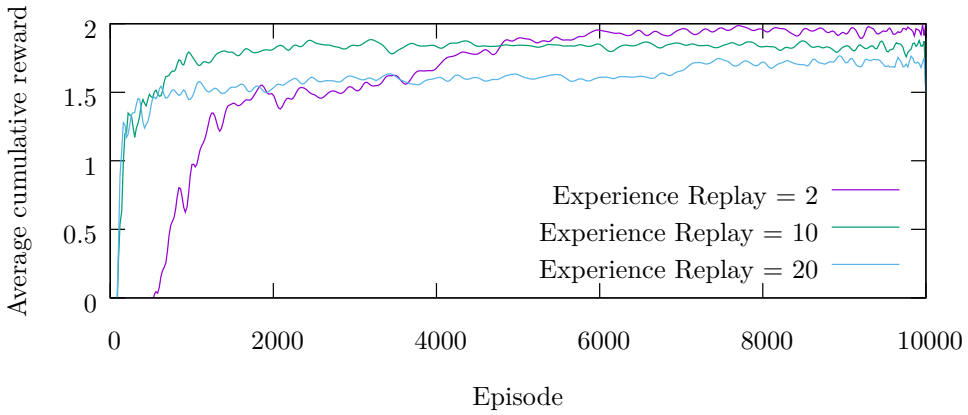

Figure 6.7: More experience replay speeds up learning at the early stages, but may prevent the agent from achieving good results. Initial variance is 0.010, temperature is 1.0 and parallelization is disabled.

the maximum performance achieved by the agent, as a smoother approximation of the Q function prevents the agent from learning precise and accurate policies. However, the overall impact of initial variance is quite small for the values used in the experiments, and is measurable only for specific sets of parameters. Usually, a large initial variance (0.010) allows fast learning without decreasing accuracy too much.

6.3.3 Experience Replay (Value Iteration)

The agent maintains a list of its latest 100 experiences, from which some of them are replayed at each time-step. The number of replayed experiences has an interesting

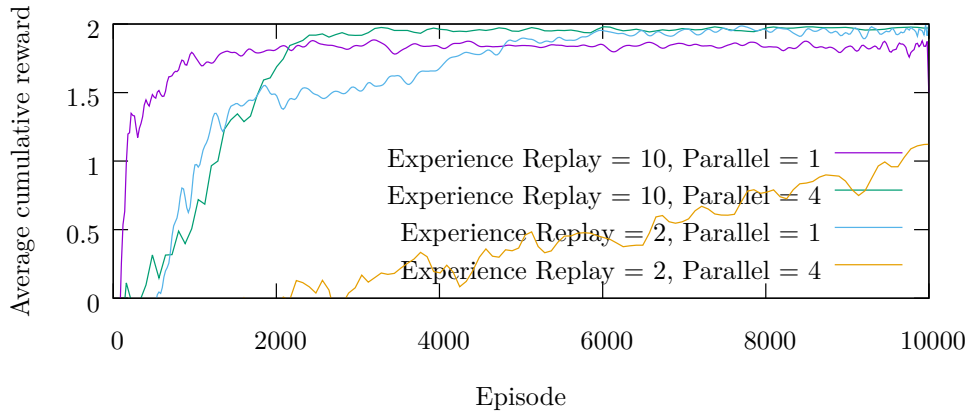

Figure 6.8: Parallelism allows more experience replay to be used without the agent being stuck at a sub-optimal policy. When a high degree of parallelism and experience replay is used, the agent quickly learns a very good policy that steadily improves afterwards. Initial variance is 0.010 and temperature is 1.0.

impact on the performance of the agent. The higher this number is, the faster the agent learns. However, increasing the experience replay factor also biases learning towards old experiences and sometimes prevents the agent from achieving the task, as shown in Figure 6.7 where the curve corresponding to 4 experiences replayed at each time-step sits lower than the other ones. This behavior is is even more present when the experience replay buffer is longer, or if experiences are randomly removed from the buffer instead of based on their age.

As could be seen in previous results, increasing parallelism allows to overcome exploration related problems and decreases the probability that the agent becomes stuck at a sub-optimal policy. When increasing parallelism, more experience replay can be used while still allowing learning, as shown in Figure 6.8. This could mean that increasing experience replay reduces exploration, and that increasing parallelism or temperature allows to compensate for that.

6.3.4 Policy Iteration

Policy Iteration has much less parameters than Value Iteration algorithms, as there are no notion of temperature, balance between exploration and exploitation or experience replay. Two parameters that have an non-negligible impact on the performance of Policy Iteration is the number of CPU nodes used by the agent and how many rollouts are used each time an action must be chosen.

Figure 6.9 shows that increasing the number of rollouts used to compute each Q-value allows the agent to learn faster and to achieve slightly better results. More rollouts reduce the variance of the estimate of Equation 6.8 by the law of the large numbers, which reduces the chance that the agent performs a bad action based on a wrong Q-value estimate. It is interesting to note that even doing one rollout per action allows

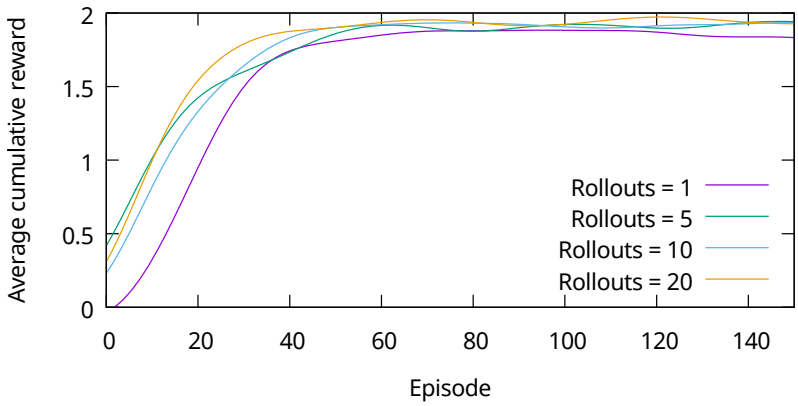

Figure 6.9: More rollouts per time-step and action allow the agent to build a more accurate estimate of the Q-values at the current state, which leads to better and faster learning. Initial variance is 0.01 and parallelization is disabled.

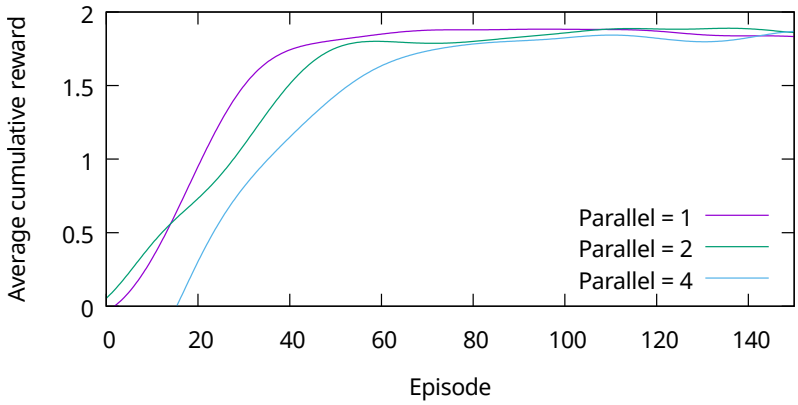

Figure 6.10: Doubling the number of CPU cores used by the agent does not double its time before learning, which indicates that added parallelism is beneficial to the overall wall-clock learning time. This shows that the algorithm proposed in Section 6.2.1, while slowing down learning a bit due to some approximations, has a manageable effect on learning speed. Initial variance is 0.01 and 10 rollouts are used.

the agent to learn reasonably fast and well. More rollouts do not seem strictly necessary and require more CPU time.

Figure 6.10 shows that increasing parallelism slows down learning, but less than linearly. This means that adding more CPU cores makes the agent require more episodes but less wall-clock time before learning.

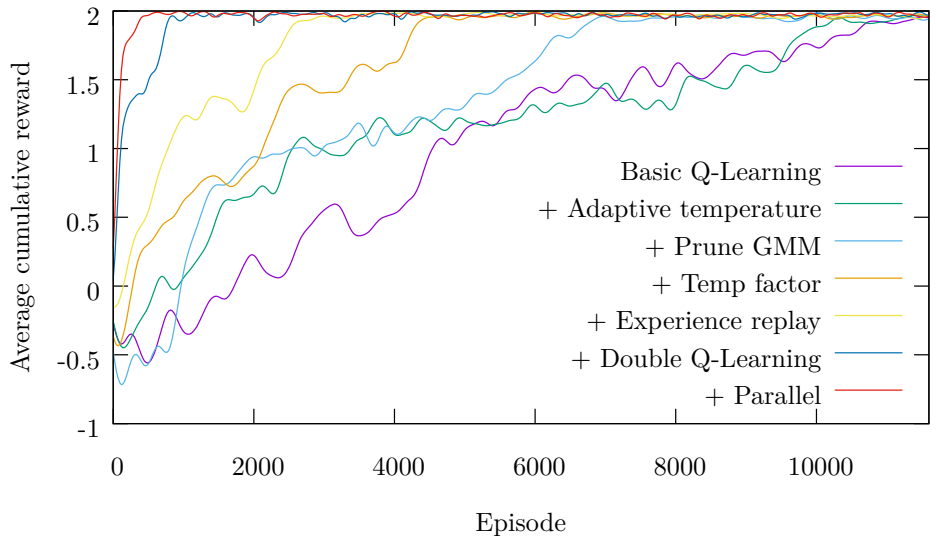

Figure 6.11: Learning curves for different agents. For the parallel agent, only the episodes of the first agent are shown (this allows to compare wall-clock learning speeds across curves as the four parallel agents execute concurrently). All the agents learn the same 2-digits MNIST classification task, but features and improvements described in Section 6.1 are incrementally enabled. The complete distributed agent using Double Q-Learning, Experience Replay and Adaptive Temperature learns nearly 23 times faster than the simple Q-Learning agent. See text for details about the experimental setup.

6.3.5 Value Iteration performance

Section 6.1 presents several extensions to Q-Learning that allow the agent to learn more efficiently. This section presents results of different experiments that have been run by selectively enabling and disabling extensions. This allows to measure the impact of each extension on learning speed.

The experiments start with the most basic variant of Q-Learning, then enable features. Each feature has an impact on the overall learning behavior of the agent. For instance, Double Q-Learning makes learning more stable and allows lower temperatures, which in turn speeds learning up. Here is a list of all the experiments run and their parameters. The task to be performed by the agent is the usual 2-digits MNIST classification used throughout this chapter.

1. Simple Q-Learning with Softmax action selection and a fixed temperature of 0.5. A Gaussian Mixture Model with an initial variance of 0.10 is used. In this experiment, the Gaussian Mixture Model does not remove useless clusters and therefore behaves like the original implementation of Heinen [HE10].

2. Use Adaptive Temperature (see Section 6.1.2), with a temperature factor (T in Equation 6.5) of 1.0 as used in [Tok10]. No other parameter changed for this

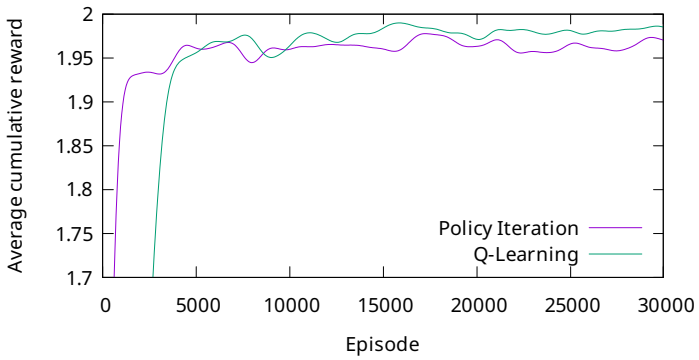

Figure 6.12: A Policy Iteration run and a Value Iteration run, both achieving the highest top performance of their class. Policy Iteration allows faster learning but only achieves an accuracy of 96%, while Value Iteration learns slower but achieves an accuracy of 99% after 100000 episodes (not shown in this figure). Accuracies are measured on the testing set. Policy Iteration uses 20 rollouts, an initial variance of 0.001 and 2 CPU cores. Value Iteration uses no experience replay, a temperature of 0.5, an initial variance of 0.005 and 4 CPU cores.

experiment.

3. Allow the Gaussian Mixture Model to remove useless clusters.

4. Introduce the T factor in Equation 6.5 and set it to 0.8.

5. Add Experience Replay (see Section 6.1.1), replay 2 experiences per time-step. Replaying more experiences sometimes prevents the agent from learning the task, as explained in Section 6.3.3.

6. Use Double Q-Learning. This makes learning more stable and allows to lower the temperature factor to 0.5. Experience replay remains at 2.

7. Use 4 parallel agents. More agents increase exploration and allow to replay 4 experiences per episode (instead of 2). The temperature factor is set to 0.6.

Figure 6.11 shows that the successive improvements presented in this chapter allow the agent to learn much faster than simple Q-Learning. Interestingly, the largest improvement is provided by Double Q-Learning (3 times faster), that prevents the agent from being stuck in sub-optimal policies and allows lower temperatures, thus increased exploitation. Adaptive Temperature does not seem to provide any benefit in these experiments, but it indirectly affects the performance of the agent as disabling Adaptive Temperature makes the fully-featured agent unable to learn anything (see Section 6.1.2).

6.4 Conclusion

This chapter presented two Reinforcement Learning algorithms, one based on Q-Learning and one on rollout-based Policy Iteration. Both algorithms have been fine-tuned to the environment described in Chapter 5. Several experiments validate their performance along with the applicability of the model described in Chapter 4 to Reinforcement Learning (either as a Q function or policy approximator). The results presented in this chapter show that Reinforcement Learning allows an agent to learn how to focus and use a glimpse sensor and a memory cell for classification.

Reinforcement Learning has also been compared with a fixed policy consisting of taking random glimpses then selecting the class based on the average prediction of the glimpses. The Reinforcement Learning agents achieve a top accuracy much larger than 90% (99% for the best Q-Learning configuration, 96% for Policy Iteration), while the hard-coded policy only achieves 89% accuracy. This shows that the policies learning by the agents a more clever than simply taking random glimpses, which is encouraging.

Policy Iteration, as implemented here, has the main advantage of being much faster than Q-Learning. Policy Iteration agents achieve their top performance much quicker than Q-Learning ones, after less than 100 episodes compared to a couple thousand. However, the graphs shown here do not allow to clearly see that the performance achieved by Q-Learning agents (around 99%) is much higher than what Policy Iteration agents manage to reach (96% at best). Figure 6.12 shows the best Value Iteration run against the best Policy Iteration run.

These last results allow us to conclude that the architecture proposed in Chapter 5 allows a Reinforcement Learning agent to classify images more accurately than simply taking random glimpses and averaging their candidate class predictions. Moreover, Q-Learning provides higher accuracy once learning is finished, even though this algorithm is much slower to learn. Chapter 7 compares Q-Learning against state-of-the-art image classification techniques on the complete MNIST dataset and some variants of it.

Part III

Evaluation

Chapter 7

Evaluation on Image Classification

This master thesis proposes three main contributions, an incremental Gaussian Mixture Model, a Reinforcement Learning formulation of Attention Models, and advanced distributed Reinforcement Learning algorithms tailored to complex environments. While the experiments in Chapter 6 assemble these three contributions, the resulting agent is only compared with itself, by varying several parameters. This chapter compares the contributions of this master thesis with well-known algorithms and evaluates their benefits and drawbacks.

The experiments of this chapter consist of classifying MNIST digits [LCB98]. See Section 6.3 for a detailed description of this task. The two main advantages of this evaluation method are that classification is a well-known problem with well-defined quality metrics (accuracy, MISE), and classifying images is a natural application of visual attention.

Moreover, selecting a class using a memory cell, moving a sensor and recognizing glimpses make a challenging Reinforcement Learning problem with many possible actions and long episodes. A mix of continuous (class probabilities, coordinates) and discrete (current class selected) variables, each with different ranges and resolutions, stresses the modeling capabilities of the Gaussian Mixture Model described in Chapter 4.

The agent uses virtual sensors as described in Chapter 5 and interacts with an environment summarized as:

Observations

$(x, y, zoom)$ coordinates of a 6×6 glimpse (taken out of a 28×28 MNIST image randomly selected at the beginning of each episode), number of glimpses remaining before reaching the maximum number of glimpses, currently-selected class, and a vector of C real values representing a probability distribution over the C classes as output by the glimpse classifier (see Section 5.1).

Actions

Move the glimpse sensor up, down, left or right, zoom in and zoom out, increment or decrement current class.

Episode length
The episode ends when the agent tries to take more than 5 glimpses or after $2 \times C$ time-steps.

Reward
0 at any time except after the last time-step. When an episode ends, the agent receives a reward of -2 if it excessed the maximum number of glimpses, 2 if it properly classified the input, and 1 if it improperly classified the input.

The glimpse classifier consists of a feed-forward neural network of $6 \times 6 \times 3$ inputs[1], 3 hidden layers of 200 neurons each, and C outputs (one per class).

A memory sensor of one memory cell allows the agent to select which class the digit it observes belongs to. At the beginning of each episode, the memory sensor is initialized to a random value in $[0, C[$, following an uniform distribution. The agent selects the class to which the image belongs by incrementing or decrementing this counter.

The Q-values learned by the agent are stored in a Gaussian Mixture Model for which hyper-parameters are given for each experiment. Most experiments described in this chapter consist of classifying MNIST images of ones and twos. Section 7.3 presents results on larger subsets of MNIST.

7.1 Reinforcement Learning for image classification

Section 6.3 shows that a Reinforcement Learning agent is able to achieve good accuracy on a simple classification task (99% on two classes), but does not study whether Reinforcement Learning actually provides any benefit over other algorithms. This section compares the agent of Section 6.3 with two other classification techniques:

1. An agent based on exactly the same observations, actions and glimpse classifier, but that follows a predefined and fixed policy. The policy consists of taking 5 random glimpses (by randomly performing 5 glimpse-moving actions), then using their average prediction to select a class.

2. No agent at all: a feed-forward neural network of 28×28 inputs, 2 layers of 150 neurons and two outputs directly maps input images to classes.

The techniques are compared on their classification accuracy, wall-clock training time before reaching their close-to-maximum performance, and wall-clock time per prediction. This allows to identify the most accurate technique along with the fastest one depending on the needs of a specific application. Experiments are run on an Intel Core i5-3230M processor (dual core, 2.6 Ghz, 3.2 Ghz turbo, 3M cache) with 6 GB of 1600 Mhz DDR3 memory. All images are stored in memory.

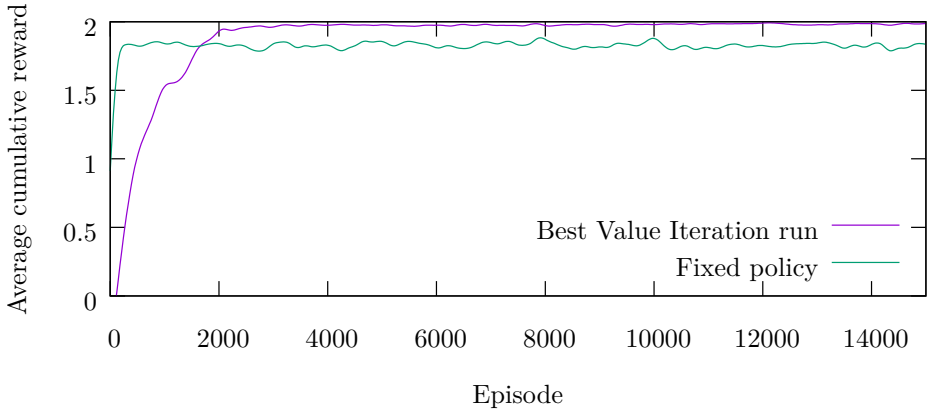

Figure 7.1: Best Value Iteration run (initial variance is 0.005, experience replay is 2, temperature is 0.5, 4 CPU cores used). The agent reaches an average cumulative reward of 1.99, corresponding to 99% accuracy. This is much higher than what the fixed policy agent achieves (using candidate classes and no learning).

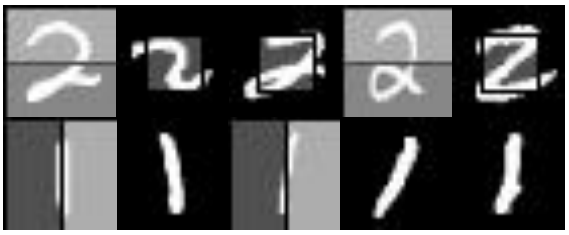

Figure 7.2: Example of images classified by the Reinforcement Learning agent. Every classification is correct (once learning is finished, the accuracy of the agent is above 99%). Semi-transparent squares indicate glimpse locations and sizes (they may be partly outside an image). The first glimpse, covering the entire image, is not represented by any square. The agent takes one, two or three glimpses, never more, depending on how informative the previous ones were. Once the agent zooms in (as shown on three images of the top row), it does not move the sensor and remains focused on the center of the image. The agent never zooms in on a one.

7.1.1 Fixed policy agent

Figure 7.1 shows that the average cumulative reward obtained over time by the best Value Iteration Reinforcement Learning agent is much higher than what a naive, hard-coded agent obtains. The hard-coded agent takes 5 random glimpses, averages their candidate classes, then selects the one with the highest probability. This shows that Reinforcement Learning provides a real benefit when using visual attention.

Closer study of the glimpses taken by the Reinforcement Learning agent shows that two main rules are learned by the agent (see Figure 7.2 for a sample of heatmaps):

- A class is chosen only when the glimpse classifier is confident enough in the current glimpse (one of the class has a sufficiently high value).
- If the current glimpse does not provide any useful information, another one is taken mostly at random, but preferring glimpses near the center and not too zoomed-in (which makes sense as a $8\times$ zoom is unneeded on small 28×28 pixels images).

The Reinforcement Learning agent reaches its close-to-maximum accuracy after 64 seconds (standard deviation on 10 runs: 26 seconds). Predicting a class takes 0.006 second per image (standard deviation: 0.0012 second). The fixed-policy agent has a much lower accuracy but is able to predict one class every 0.00019 second, which is more than 30 times faster than the Reinforcement Learning agent. Note that both the RL and the fixed-policy agent require a pre-trained glimpse classifier to work, which adds about one hour and a half to their total training time.

7.1.2 Neural network

Neural networks are very often used for image classification and provide very good results. Depending on the specific implementation and shape of the network, accuracies as high as 99.6 % on the complete MNIST set are achievable. When convolutional neural networks are used (see Section 2.2), accuracies of 99.77 % are reported[2].

In this section, a much simpler neural network is compared against the Reinforcement Learning agent described in this master thesis. It consists of 28×28 inputs and two hidden layers of 100 neurons. A reduced version of MNIST is used, containing only ones and twos, as in the previous sections. The neural network implementation is `nnetcpp`[3], a highly-optimized even though purely CPU-based C++ feed-forward and recurrent neural network library.

Neural networks excel at classification tasks. Figure 7.3 shows that the neural network described above has an accuracy comparable to the best Value Iteration Reinforcement Learning agent (99%). The neural network achieves its maximum performance after 10.9 seconds (standard deviation of 1.19 seconds, about 6 times faster than the Reinforcement Learning agent) and is able to predict a class in 0.00058 seconds, which

[1] The glimpse sensor is designed for HSV color images. MNIST being black and white, the neural network ignores the hue and saturation channels and only considers the value.

[2] Results obtained from [LCB98], in a table on `http://yann.lecun.com/exdb/mnist/`

[3] `https://github.com/steckdenis/nnetcpp`

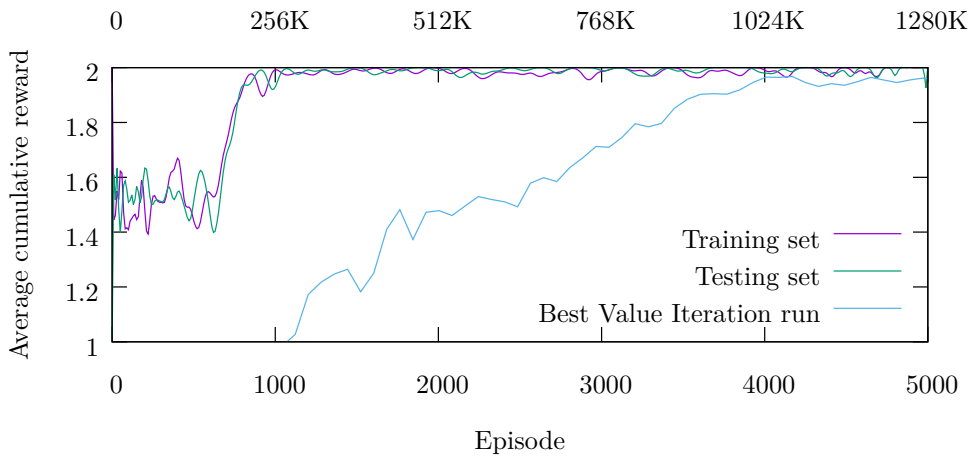

Figure 7.3: Testing and training scores of a neural network, using the same reward scheme as the Reinforcement Learning agent (shown in light blue). The neural network uses two hidden layers of 100 neurons each and has been trained using a learning rate of 1×10^{-4}. Note that the neural network uses a different X scale and learns after about 200000 episodes.

is 10 times faster than the Reinforcement Learning agent but 3 times slower than the fixed-policy agent.

7.1.3 Summary

The table below contains a small summary of the three classification algorithms studied in this section. The *Reinforcement Learning* agent is the one described in this master thesis, *Fixed Policy* corresponds to a non-learning agent that takes random glimpses, and *Neural Network* represents the feed-forward neural network that directly classifies complete MNIST images.

Agent	**Top accuracy**	**Learning time**	**Prediction time**
Reinforcement Learning	**99%**	64s (1h30 pretrain)	0.006s
Fixed Policy	80%	(1h30 pretrain)	**0.00019s**
Neural Network	**99%**	**10.9s** (no pretrain)	0.00058

These results show that even if Reinforcement Learning can be used for classification tasks and provides results that are not too much worse than other approaches, dedicated Supervised Learning algorithms perform much better and are much more efficient. However, the goal of this master thesis is not to use Reinforcement Learning for classification, but to improve Reinforcement Learning algorithms so that they can be used for very challenging tasks. These results show that classification is indeed very challenging for Reinforcement Learning agents, and that the algorithms proposed in this thesis allow them to remain comparable in speed and accuracy with dedicated algorithms.

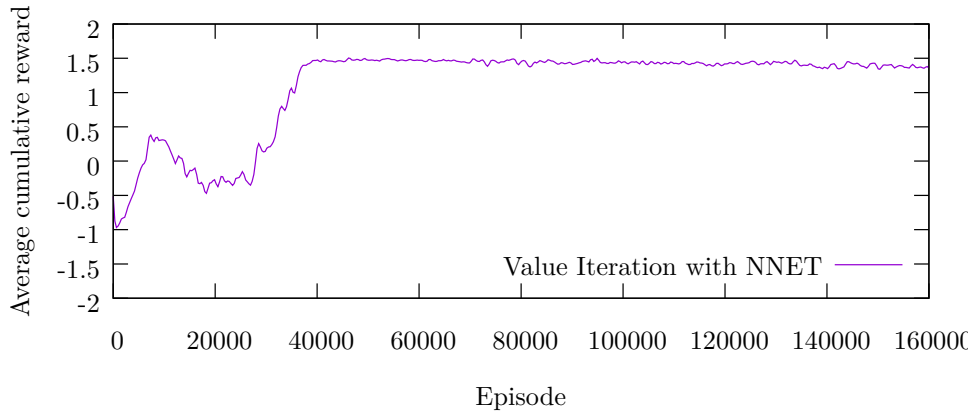

Figure 7.4: Average cumulative reward obtained by an agent on MNIST (2 digits) using Q-Learning. The Q function is approximated by a feed-forward neural network of one hidden layer of 1000 neurons. The neural network learning rate is 1×10^{-6}.

7.2 Gaussian Mixture Models and Neural Networks

Chapter 4 argues that local function approximators like Gaussian Mixture Models are more suited for use with Reinforcement Learning, as they avoid catastrophic forgetting and long-range interferences between training points, and cope better with correlated training points. Chapter 4 also presents results that indicate that Gaussian Mixture Models actually learn better and faster when training points are clustered along a line instead of uniformly scattered throughout the input-space, as happens in Reinforcement Learning when an agent slowly explores a large environment without being able to visit all the states.

All the experiments run in this thesis, in particular this chapter and Chapter 6, use Gaussian Mixture Models to store Q-values or policies. In this section, some experiments are re-run, and use neural networks as function approximators.

7.2.1 Experiments

Figure 7.4 shows the results obtained by a Value Iteration agent that stores Q-values in a neural network. Instead of a steady increase, the cumulative reward obtained by the agent increases, then quickly decreases to finally recover.

The agent finally manages to learn half the task (it submits a classification without exceeding the maximum number of glimpses), but is not able to complete its learning and does not exceed the accuracy of a random classifier.

Figure 7.4 represents the best run out of several dozens. Numerous neural network configurations (layers, hidden neurons and learning rates) have been tried but none allowed cumulative rewards above 1.5. This shows that the problem faced by the agent is very hard and requires special attention to the function approximator used.

Figure 7.5 shows results obtained by a Policy Iteration agent. As detailed in Section

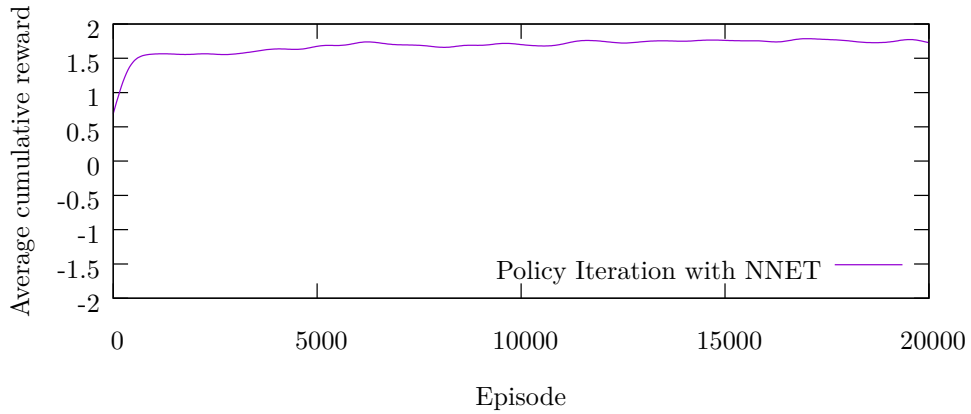

Figure 7.5: Average cumulative reward obtained by an agent on MNIST (2 digits) using Policy Iteration. The Q function is approximated by a feed-forward neural network of one hidden layer of 3000 neurons. The neural network learning rate is 1×10^{-6}.

6.2, the agent uses rollouts to compute empirical Q-values from which an action is selected. This action is used to train a neural network representing policy π [LP03]. Unlike Value Iteration, this learning algorithm still provides good enough results using a neural network, and reaches 80% accuracy after 100 000 episodes. However, obtaining this result requires careful configuration of the neural network (a single layer of 3000 neurons), and this accuracy is still much lower than what the Gaussian Mixture Model allows. Training the neural network also requires many more episodes (100000 vs 4000).

7.2.2 Summary

This section shows that the Gaussian Mixture Model presented in Chapter 4 provides much better results than neural networks in the Reinforcement Learning classification task used throughout this thesis. This extends the results of Chapter 4, obtained with very synthetic experiments, to a "real-world" and large-scale use of the model.

Gaussian Mixture Models are also much easier to configure, as their only large-impact parameter is the initial variance, that allows to adjust the bias/variance trade-off of the model. Moreover, Gaussian Mixture Models are quite robust regarding this parameter, as shown in Section 6.3. Neural networks, however, need much more care when configured as their shape and learning rate can deeply affect their accuracy.

7.3 Scaling to larger datasets

The previous experiments used a reduced version of the MNIST training dataset, using only images of zeros and ones. This section presents results when more digits are kept in the dataset. Each experiment consists of classifying MNIST images belonging to classes $0, 1, \ldots, C$, with C the number of classes used in the experiment. Varying how classes

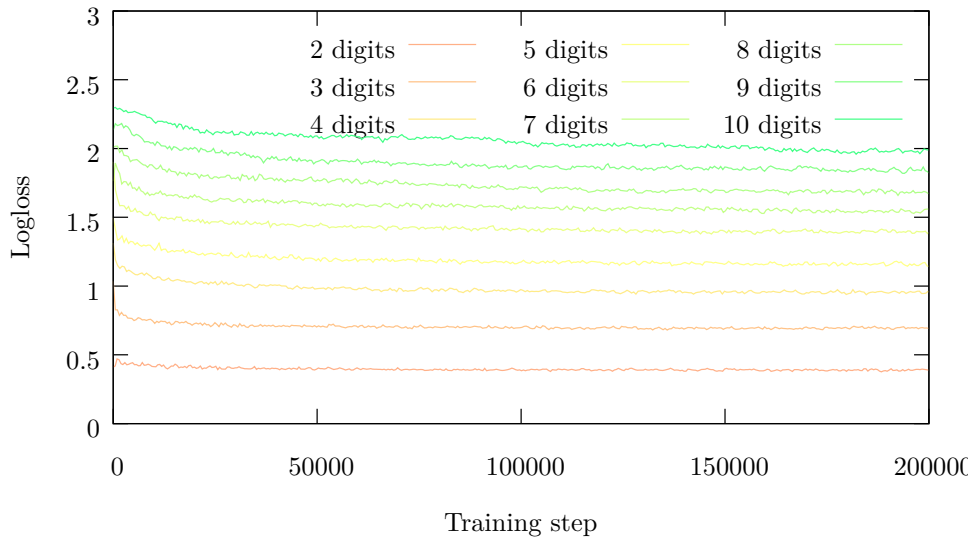

Figure 7.6: Logloss obtained by the glimpse classifier on MNIST digits 0 to $C-1$, with C varying from 2 to 10. As more digits are added to the dataset, the logloss increases (the accuracy decreases), which reduces the amount of information available to the Reinforcement Learning agent. When 10 digits are used, a logloss of 2 indicates an accuracy of 13%, barely above the 10% obtained by random classification.

are selected (for instance, using 3 and 5 instead of 0 and 1) does not change the results in any significant way.

Adding more classes to a classification problem makes it more difficult. This is especially true for the Reinforcement Learning agent described in this thesis, that is sensitive to the number of classes at many different levels:

1. The glimpse classifier produces a probability distribution over possible classes. If an additional class is added, one more observation is given to the agent. Adding classes therefore increases the dimensionality of the Reinforcement Learning problem.

2. The agent must select the proper class using the memory sensor (by increasing and decreasing a class selector). The class selector is initialized at random at the beginning of each episode, which means that the agent needs on average $\frac{C}{2}$ memory operations in order to select the proper class. Adding a class therefore makes episodes longer.

3. The glimpse classifier still takes 6×6 glimpses and uses three layers of 200 hidden neurons. Even if the classification problem becomes more complex, the model is not adjusted, which leads to decreasing accuracy as classes are added. Figure 7.6 shows that the logloss achieved by the glimpse classifier goes up as classes are added, which forces the Reinforcement Learning agent to classify images using less accurate glimpse information.

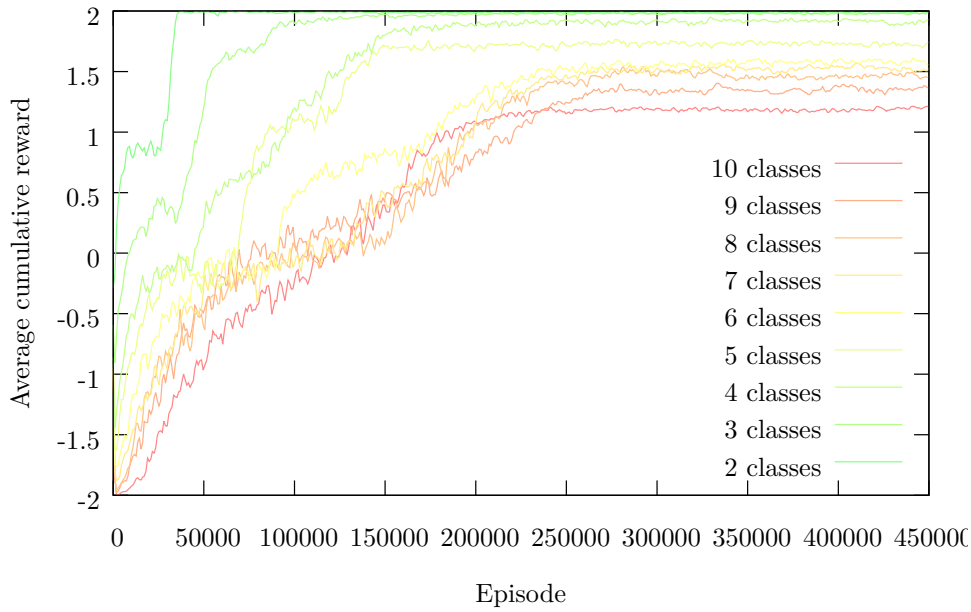

Figure 7.7: Average cumulative reward obtained by the Reinforcement Learning agent on 9 digit sets. Performance remains good below 4 digits, then progressively decreases. Experience replay is disabled, 4 CPU cores are used, temperature is 1.2 and initial variance is 1.0.

4. The glimpses themselves become more ambiguous as classes are added, which makes the Reinforcement Learning problem much less Markovian, hence much less suited to Reinforcement Learning algorithms not specifically designed for partially observable environments. This also contributes to the decrease in accuracy of the glimpse classifier: even if more neurons are added, accuracy does not improve because the glimpses don't contain enough information.

When the above points are combined, we see that adding a class quickly makes the Reinforcement Learning problem much more complex, with a higher dimensionality, longer episodes and less informative glimpses.

Figure 7.7 shows that the average cumulative reward obtained by the Reinforcement Learning agent decreases as classes are added, and quickly becomes unusable (above 4 digits, accuracy drops below 80%). However, two important points are observed:

1. The accuracy of the agent is always much higher than random selection (when run on 8 classes, the agent has an accuracy of 50% instead of 12.5%, and manages to still reach an accuracy of 20% instead of 10% when 10 classes are used).

2. The accuracy of the agent is higher than the accuracy of the glimpse classifier. For instance, when 10 classes are used, the glimpse classifier has a logloss of 2 (probability of the correct class is 13.5% on average). The Reinforcement Learning

agent manages to combine glimpses and select the correct class with a probability of 20%.

Even if these results clearly show that the architecture proposed in Chapter 5 cannot be used for classification and does not replace state-of-the-art attention models (see Chapter 3), the Reinforcement Learning agent itself, using algorithms and function approximators described in this thesis, still manages to do its best with the available information.

Chapter 8

Discussion

This master thesis presents three complementary approaches that allow Reinforcement Learning agents to efficiently learn complex tasks in large environments:

Attention
: Instead of observing its complete state, the Reinforcement Learning agent voluntarily ignores some parts of it and focuses its attention on specific state variables. Chapter 5 presents an attention-based architecture tailored for image classification.

Distributed Reinforcement Learning
: Advanced Reinforcement Learning, using function approximation, double Q-Learning, experience replay and adaptive temperatures are presented in Chapter 6. These algorithms have been parallelized and implemented in C++ over MPI. Distributed Reinforcement Learning algorithms allow to increase the amount of computing power available for learning, which is required in large and complex environments if training times have to remain reasonable.

Gaussian Mixture Model
: Chapter 4 presents a Gaussian Mixture Model suited to Reinforcement Learning tasks. Even if neural networks are better adapted to large state-spaces (they classify large images without problems), they do not work so well with data produced by a Reinforcement Learning agent. The Gaussian Mixture Model presented in this thesis is as suited to Reinforcement Learning as [HE10], but also manages high dimensions and large datasets more efficiently.

These three contributions are combined to form a Reinforcement Learning agent, that is evaluated on an image classification task. This task is very complex and not often solved using Reinforcement Learning, but close to real-world applications and easy to evaluate using standard accuracy metrics.

8.1 Results

Chapter 7 presents experiments that compare the Reinforcement Learning agent described in this thesis with other algorithms. Its results are summarized as follows:

Attention

On image classification, using Reinforcement Learning to "learn where to look" does not provide conclusive results. The agent has an accuracy higher than random classification, but is still far below state-of-the-art results and much less efficient than neural networks.

Distributed Reinforcement Learning

The distributed Value Iteration and Policy Iteration presented in Chapter 6 provide amazing results. The agent is always able to learn good policies even in very complex environments and does so rapidly. Section 6.3.5 shows that the improvements over simple Q-Learning proposed in this thesis speed learning up by almost 23 times. Combined with optimizations in the Gaussian Mixture Model (presented in Section 4.4), the Reinforcement Learning agent described in this thesis is nearly 100 times more efficient than a naive implementation, and is therefore well suited to complex Reinforcement Learning problems.

Gaussian Mixture Model

The Gaussian Mixture Model compares favorably to neural networks in synthetic (Chapter 4) and real-world Reinforcement Learning contexts (Section 7.2).

8.2 Future research

The promising results obtained by the Gaussian Mixture Model and distributed Reinforcement Learning algorithms will be applied to other, fully Markovian environments. Planned applications of those algorithms are Reinforcement Learning for disease prevention, hierarchical Reinforcement Learning and Reinforcement Learning on robots, where interactions with the environments are more costly. In addition to the general interest of the scientific community for distributed Reinforcement Learning algorithms [NSB+15, MBM+16], several researchers have expressed their interest in the algorithms presented in Chapter 6, and suggest to have them applied to other problems and published. A more thorough study of the properties of the Gaussian Mixture Model is also planned for publication.

Other ways of combining Attention Models and Reinforcement Learning may also be possible. Even if Reinforcement Learning cannot replace gradient descent as the training method of those models, the idea of learning where to look may still be applicable at different places. For instance, Attention Models could learn how to focus their attention separately from a Reinforcement Learning agent that learns the task to accomplish (but not where to look), hence using both algorithms where they best perform.

Another research direction is the use of Reinforcement Learning as a boosting method: depending on an input vector, an RL agent chooses which classifiers to use and how to combine them in order to achieve higher accuracy than any classifier alone. Preliminary results have already been obtained for this approach [DS02], but not on classification tasks.

8.3 Source code

The complete source code used in this master thesis is available at `http://public.steckdenis.be/ulb/thesis.zip`. This code depends on Qt5, OpenMPI, Eigen3, FFTW3 and *libnnet*, a neural network library available at `https://github.com/steckdenis/nnetcpp`.

Bibliography

[ABB12] Sander Adam, Lucian Buşoniu, and Robert Babuška. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 42(2):201–212, 2012.

[Bak01] Bram Bakker. Reinforcement Learning with Long Short-Term Memory. *Nips 2001*, 2001.

[Bak07] Bram Bakker. Reinforcement learning by backpropagation through an LSTM model/critic. *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007*, pages 127–134, 2007.

[BBB+10] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

[Bel56] R Bellman. Dynamic Programming and Lagrange Multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767–769, 1956.

[BGSF15] Jimmy Ba, Roger Grosse, Ruslan Salakhutdinov, and Brendan Frey. Learning Wake-Sleep Recurrent Attention Models. pages 1–9, 2015.

[BM95] Justin a Boyan and Andrew W Moore. Generalization in Reinforcement Learning: Safely Approximating the Value Function. *Advances in Neural Information Processing Systems 7*, pages 369–376, 1995.

[BMK14] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple Object Recognition with Visual Attention. pages 1–10, 2014.

[Bon14] Gianluca Bontempi. *Statistical Foundations of Machine Learning*. Machine Learning Group, Computer Science Department, ULb, 2014.

[BP92] N. Benvenuto and F. Piazza. The backpropagation algorithm. *IEEE Transactions on Signal Processing*, 40(4):967–969, 1992.

[BS89] C. Bandera and P.D. Scott. Foveal machine vision systems. In *Conference Proceedings., IEEE International Conference on Systems, Man and Cybernetics*, pages 596–599. IEEE, 1989.

[BVB+96] Cesar Bandera, Francisco J Vico, Jose M Bravo, Mance E Harmon, and U S A F Academy. Residual Q-Learning Applied to Visual Attention. *Convergence*, (July):3–6, 1996.

[CBS+15] Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. Attention-Based Models for Speech Recognition. pages 1–19, 2015.

[CGCB14] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR*, abs/1412.3, 2014.

[CKL94] A R Cassandra, L P Kaelbling, and M L L B CassandraETAL.AAAI1994 Littman. Acting optimaly in partially observable stochastic domains. *Aaai*, (April), 1994.

[Col11] Ronan Collobert. Torch7: A matlab-like environment for machine learning. *BigLearn, NIPS Workshop*, pages 1–6, 2011.

[CvMBB14] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation : Encoder – Decoder Approaches. *Ssst-2014*, pages 103–111, 2014.

[DCB15] Yann N Dauphin, Junyoung Chung, and Yoshua Bengio BENGIOY. RMSProp and equilibrated adaptive learning rates for non-convex optimization. *arXiv preprint arXiv:1502.04390*, 2015.

[DN10] B Doroodgar and G Nejat. A hierarchical reinforcement learning based control architecture for semi-autonomous rescue robots in cluttered environments. *Automation Science and Engineering CASE 2010 IEEE Conference on*, pages 948–953, 2010.

[DS02] Kenji Doya and Kazuyuki Samejima. Multiple Model-based Reinforcement Learning. *Neural computation*, 14(6):1347–1369, 2002.

[Fre91] R M French. Using Semi-Distributed Representations to Overcome Catastrophic Forgetting in Connectionist Networks. *Proceedings of the 13th Annual Cognitive Science Society Conference*, pages 173–178, 1991.

[GC99] Felix a Gers and Fred Cummins. Learning To Forget, 1999.

[GSK+15] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. LSTM: A Search Space Odyssey. *arXiv*, page 10, 2015.

[GWD14] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *Arxiv*, pages 1–26, 2014.

[HE10] Mr Heinen and Pm Engel. An incremental probabilistic neural network for regression and reinforcement learning tasks. *Icann*, pages 170–179, 2010.

[Hei11] Milton Roberto Heinen. A Connectionist Approach for Incremental Function Approximation and On-line Tasks. (March), 2011.

[Hes12] T Hester. *TEXPLORE: Temporal Difference Reinforcement Learning for Robots and Time-Constrained Domains*. 2012.

[HGW10] Hado Van Hasselt, Adaptive Computation Group, and Centrum Wiskunde. Double Q-learning. *Nips*, pages 1–9, 2010.

[HS97] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1–32, 1997.

[Kav15] Max Jaderberg Karen Simonyan Andrew Zisserman Koray Kavukcuoglu. Spatial Transformer Networks. *Nips'15*, pages 1–14, 2015.

[KB99] Vijaymohan R. Konda and Vivek S. Borkar. Actor-Critic–Type Learning Algorithms for Markov Decision Processes. *SIAM Journal on Control and Optimization*, 38(1):94–123, jan 1999.

[KJYFF11] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel Dataset for Fine-Grained Image Categorization. In *First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition*, Colorado Springs, CO, 2011.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances In Neural Information Processing Systems*, pages 1–9, 2012.

[Lag03] Mg Lagoudakis. Least-squares policy iteration. *The Journal of Machine Learning Research*, 4:1107–1149, 2003.

[LBBH98] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher Burges. The MNIST Database of Handwritten Digits, 1998.

[LCG11] Andre Lemos, Walmir Caminhas, and Fernando Gomide. Fuzzy evolving linear regression trees. *Evolving Systems*, 2(1):1–14, 2011.

[Lin92] Long J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3-4):293–321, 1992.

[LP03] M. Lagoudakis and R. Parr. Reinforcement learning as classification: Leveraging modern classifiers. *In Proceedings of the Twentieth International Conference on Machine Learning*, 20:424, 2003.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. pages 1–28, 2016.

[MDG09] Francis Maes, Ludovic Denoyer, and Patrick Gallinari. Structured prediction with reinforcement learning. *Machine Learning*, 77(2-3):271–301, 2009.

[MHGK14] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent Models of Visual Attention. *Nips*, pages 1–12, 2014.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei a Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943.

[MPR+02] Michael Montemerlo, Joelle Pineau, Nicholas Roy, Sebastian Thrun, and Vandi Verma. Experiences with a mobile robotic guide for the elderly. In *Proceedings of the 18th National Conference on Artificial intelligence*, AAAI '02, pages 587–592. AAAI, 2002.

[NSB+15] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv:1507.04296*, page 14, 2015.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Stanford University Queue*, page 9, 2005.

[Orm02] Dirk Ormoneit. Kernel-Based Reinforcement Learning. *Machine Learning*, 49:161–178, 2002.

[PMK01] Leonid Peshkin, Nicolas Meuleau, and Leslie Kaelbling. Learning Policies with External Memory. *Sixteenth International Conference on Machine Learning*, page 8, 2001.

[PS05] Duncan Potts and Claude Sammut. *Incremental Learning of Linear Model Trees*, volume 61. 2005.

[Ran14] Marc'Aurelio Ranzato. On Learning Where To Look. pages 1–14, 2014.

[RHW85] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Internal Representations by Error Propagation. sep 1985.

[Rie05] Martin Riedmiller. Neural fitted Q iteration - First experiences with a data efficient neural Reinforcement Learning method. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3720 LNAI:317–328, 2005.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. 1958.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 1998.

[SFR14] Pierre Sermanet, Andrea Frome, and Esteban Real. Attention for Fine-Grained Categorization. (2014):1–11, 2014.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[SMD+10] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A Scalable Real-time Architecture for Learning Knowledge from Unsupervised Sensorimotor Interaction. (1972):761–768, 2010.

[Smi15a] Ryan Smith. The AMD Radeon R9 Fury X Review: Aiming For the Top, 2015.

[Smi15b] Ryan Smith. The NVIDIA GeForce GTX 980 Ti Review, 2015.

[SS92] Richard S Sutton and Csaba Szepesv. A Convergent O(n) Temporal-difference Algorithm for Off-policy Learning with Linear Function Approximation. *Computing*, pages 1–8, 1992.

[Sut88] Richard S Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3(1):9–44, 1988.

[SV15] Denis Steckelmacher and Peter Vrancx. An Empirical Comparison of Neural Architectures for Reinforcement Learning in Partially Observable Environments. In *27th Benelux Conference on Artificial Intelligence*, Hasselt, 2015.

[Tok10] Michel Tokic. Adaptive ??-greedy exploration in reinforcement learning based on value differences. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6359 LNAI:203–210, 2010.

[TP06] Scott E. Thompson and Srivatsan Parthasarathy. Moore's law: the future of Si microelectronics. *Materials Today*, 9(6):20–25, 2006.

[Tur38] Alan Turing. On Computable Numbers, with an Application to the Entscheidungsproblem - A Correction. *Proceedings of the London Mathematical Society*, 2(1):544–546, 1938.

[TV97] John Tsitsiklis and Benjamin Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[VK14] N Denizcan Vanli and Suleyman Serdar Kozat. A Comprehensive Approach to Universal Piecewise Nonlinear Regression Based on Trees. *Ieee Transactions on Signal Processing*, 62(20), 2014.

[VT07] Mario Ventresca and Hamid R. Tizhoosh. Opposite transfer functions and backpropagation through time. *Proceedings of the 2007 IEEE Symposium on Foundations of Computational Intelligence, FOCI 2007*, (Foci):570–577, 2007.

[WD92] Christopher J C H Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Wer74] P Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. 1974.

[WHPS11] Marco a. Wiering, Hado Van Hasselt, Auke-Dirk Pietersma, and Lambert Schomaker. Reinforcement learning algorithms for solving classification problems. *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 91–96, 2011.

[WO12] Marco Wiering and Martijn Van Otterlo. *Reinforcement Learning State-of-the-Art*. 2012.

[ZS15] Wojciech Zaremba and Ilya Sutskever. Reinforcement Learning Neural Turing Machines. *Arxiv*, 2015.