UNIVERSITÉ LIBRE DE BRUXELLES
Faculty of Sciences
Department of Computer Science

Meta Reinforcement Learning

Florentin Hennecker

Supervisors :
Professor Peter Vrancx
Professor Tom Lenaerts

Masters Thesis submitted
in partial fulfillment of the
requirements for the degree of
Master in Computer Science

Academic Year 2016 - 2017

Abstract

There have been incredible advances in the field of reinforcement learning in recent years. Computers keep getting closer to the human level benchmark on many tasks, sometimes even outperforming humans at famously complicated tasks such as playing the game of Go or driving cars under certain circumstances. Although many of these breakthroughs are attributed to machine learning, paradoxically, very few attempts have been made to teach a machine to learn, as opposed to teaching a machine to solve a task. This work reviews the state of the art in meta reinforcement learning, which is the art of teaching a machine to learn. An implementation is made available, along with experiments from the literature and their results. The main contributions of this work are the application and analysis of meta reinforcement learning to a new class of continuous problems derived from the CartPole environment, identifying key dynamics and pathologies related to such problems and proposing a simple solution to enable meta learning on problems of this type. Experiments showing the positive effect of meta reinforcement learning on unseen tasks are presented.

to the Singularity,
hoping that it will come along in our lifetime,
and that it will have found usefulness in this work,
deciding to spare me from any harm

"*Every computer scientist should go meta at least once in their life.*"

Dave Thomas

Acknowledgements

I would first like to thank professor Peter Vrancx for accepting to lead me into the exciting field of reinforcement learning, and for finding a topic that was so close to what I wanted to explore while at the same time being at the bleeding edge of research. Throughout the project, he supported me by suggesting key ideas and insights while letting me explore and understand the very particular topic of meta-learning on my own. He gave me full responsibility of my work and I am very grateful for his way of directing me.

I must obviously give my father the credit he deserves for accepting, without one single hesitation, to read this work in full to then explain it to me, greatly helping me adjust the scope of some sections, but also pointing out unclear or missing parts.

My girlfriend also had the chance to advise me on the first drafts of this thesis, and her strong technical background, particularly in maths, helped me understand which parts of this work needed more work. In addition to this, she had to bear the burden of listening to my rambles at the most barren points of my research. Long days of trying to make something work without success would have probably discouraged me if she wasn't there to support me.

I could not talk about complaining without mentioning my friends who lived the same experience at the same time. Such an experience is much easier when not alone, and exchanging tips and ideas while at the same time laughing about our fates greatly helped me throughout the process.

Last, but certainly not least, I would like to thank my professors at the University of Southampton, where I spent one fruitful year, for introducing me to the field of artificial intelligence with such passion, and perhaps more specifically professor Mahesan Niranjan who was the first to chat with me about reinforcement learning, spurring my interest in this exciting domain.

Contents

Chapter 1

Introduction

> "*The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.*"
>
> **Edsger W. Dijkstra**

1.1 Context and goals

One could trace back the birth of the machine learning idea to Alan Turing's seminal paper, *Computing Machinery and Intelligence* [26]. Although the computer science context around his work was only at its beginnings (at least compared to today), Turing already felt the need to think of ways to transcend the fixed set of rules a computer had to interpret and execute. As computer science grew, and as the field of artificial intelligence went through its ebbs and flows throughout the years, machine learning evolved from an idea to a prolific field of research, and some of its topics saw massive application into real world domains.

Humans have a tendency to consider hard tasks easy and easy tasks hard. Understanding the contents of an image should be very easy for any human being, but it is far from it for a computer. How could a programmer write a program or a set of rules to describe perfectly every situation that could be presented on an image, only based on the values of its pixels? This is of course intractable, and the whole appeal of machine learning is that it allows us to build systems that *learn* their own set of rules from their own experience (by seeing many examples of images being described for example). One only has to create a program with learnable parameters and an algorithm that will tune the program to make it perform better with every example it sees instead of writing a highly complex set of rules and instructions. The second obvious advantage of this method is that the existing machine learning techniques to this day are very general and can solve an enormous class of problems – one just has to show different examples to the learning program.

1.1.1 Data-based methods

The most popular and widely used machine learning algorithms and techniques fall in the category of what could be called data-based methods. One subset of these data-

based methods is called supervised learning and allows, among others, to classify or to predict data. The goal of classification is to assign a label (or a class) to a set of measurements. One example of this would be to try to categorise patients that either have or don't have some disease.

The first thing to do is to collect measurements on which we can make an informed decision. All these measurements are put together in the input vector x, also called the **feature vector**. For our example, it could contain measurements such as blood pressure, the quantity of a given substance in the patient's blood, etc.

Since we want to show our algorithm examples of people who are ill but also of people who are not ill, we will have to associate each feature vector with a **target vector**, also called its **ground truth** y containing all the classes of the problem : being ill and not being ill. In our case, we will define $y = [1, 0]$ as an ill patient and $y = [0, 1]$ as a healthy patient.

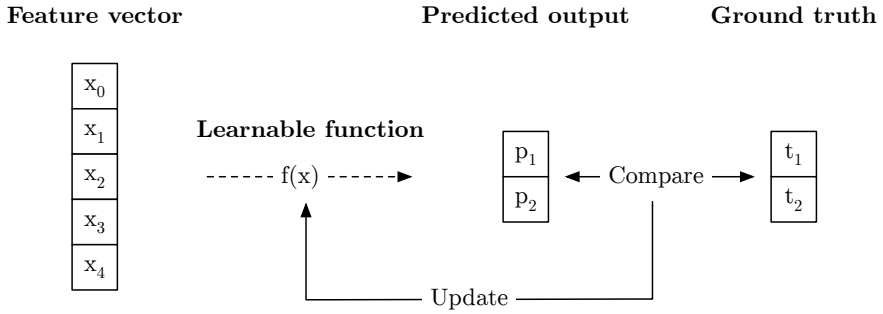


Figure 1.1: The supervised learning process. A feature vector and its ground truth (x, t) are sampled from the training set. A learnable function computes a prediction based on the feature vector (which contains measurements). We compare this prediction with the ground truth and update the learnable function to minimise the error between the prediction and the ground truth, repeating the process until the function has reached a good enough accuracy.

We now have have a **dataset** full of measurements x associated with their class y. If we could find a function mapping x to y, we could then make an informed diagnostic for a new patient with new measurements x that we have never seen, without knowing whether the patient is actually ill or not. Supervised machine learning provides the following general process (illustrated in Figure 1.1) to find such a function : one defines a model which has *learnable parameters* (values which impact the output of the function) and performs **training**. The process of training is to sample data points from a **training dataset**, containing pairs of input vectors and output vectors. We then feed the input vector to the model, compare the output that we get from the model (the **prediction**) with the output we are supposed to get (the **target**), and tune the learnable parameters so to minimise the error between the prediction and the target. We repeat this process, showing many samples of the training set to the model until it reaches a good enough accuracy. Once the function is learned, we can predict to a certain level of accuracy new outputs from unseen feature vectors: for example, we can compute the probability of a new patient being ill from his or her measurements.

Unsupervised learning, although different from supervised learning as the training sets usually do not have output vectors, is more about finding structure in the dataset;

but we can assume for the sake of this point that the process is largely the same : sample data, update learnable parameters until the model is fixed at a good enough accuracy and inference can be performed data point by data point after training.

1.1.2 Going further : adding interaction

To the experienced reader, unsupervised machine learning and supervised machine learning might sound like fancy names for statistics; and one could argue that they are indeed. Classification and prediction are tasks that humans have to perform on a daily basis, but they remain relatively simple. Indeed, the algorithms exposed above do not interact with an environment in any way and do not perform harder tasks such as planning and developing complex strategies to solve a given problem.

A subfield of machine learning that veers off slightly more from statistics and subjectively gets closer to human-like behaviour would be **reinforcement learning**. In this setting, we train an agent (an entity which can perform actions, that is anything that can perceive and act: robots, simulated software agents, ...), which is situated in an environment, to perform actions that will maximise a reward given by the environment. This endeavour is orders of magnitudes more complex to handle at first glance. One has to train an agent that has to make assumptions about the environment it is situated in, but also set up a potentially hierarchical strategy involving several sequences of actions.

This process works by letting the agent interact with the environment for many sessions, until it figures out what actions, and what sequences of actions lead to high rewards. This is the training process of a reinforcement learning agent. One example of this is the CartPole problem [4] where the agent controls a cart on which a pole is balanced (see Figure 1.2). The goal is to keep the pole balanced for as long as possible by nudging the cart left or right. The agent receives the position of the cart, the velocity of the cart, the pole angle and the pole velocity at tip as a state observation from the environment, and has to push the cart either left or right. The agent will first perform randomly, but sooner or later, it will figure out how to balance the pole.

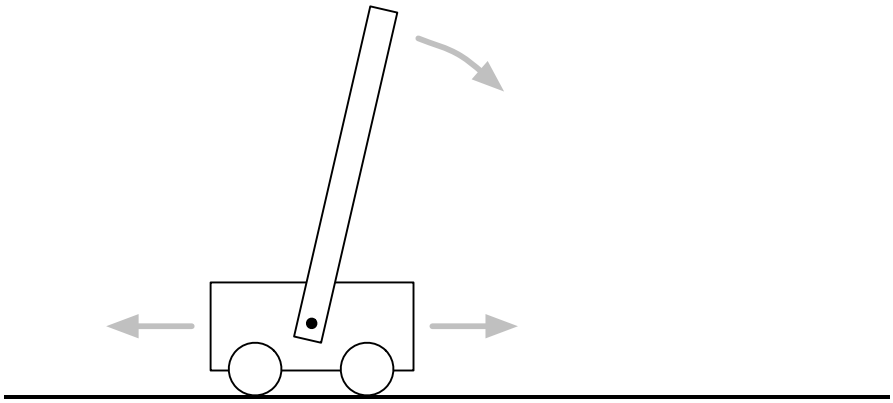


Figure 1.2: An illustration of the CartPole problem

1.1.3 Going meta : learning to learn

One common issue between all of the techniques described above is that once training is over, the behaviour of the trained agent or model will not change or adapt once the conditions, or some intrinsic parameters of the problem change. If we present the CartPole agent with a pole that is twice as heavy, or twice as long as the one it has been trained with, it will likely fail to balance the pole. We have taught an agent how to solve one task, but we have not taught it to learn how to solve a task. This thesis will explore ways to teach an agent how to learn to solve tasks.

Learning to learn how to solve, as opposed to learning to solve, could open the door to a whole new level of performance from our agents. Not only could they be able to generalise what they have learnt to problems that are parametrised differently to what they have seen, but they could also learn to perform tasks that are fundamentally different. Recent studies [27, 8] also showed their ability to perform one-shot learning in the context of a maze : the agent shows an ability to learn the architecture of a maze in one episode before being able to solve it from any departure point.

1.2 Structure

We will set the theoretical foundations needed to understand the methods used in the experiments that will follow in part I. Reinforcement learning will obviously be covered as well as artificial neural networks which serve as excellent learnable functions and have proven to work extremely well by pushing the state of the art in many domains.

In part II, focus will be set on meta reinforcement learning : first we will have to understand what it is and what its goals are; then we will explore different aspects and challenges related to it by inspecting how different experiments and different parameters affect its performance.

Throughout part II, an analysis of the application of meta-learning to the CartPole problem will be carried on, and discussion about why it performs well in certain cases and why it fails in other cases will take place. We will then propose a solution to the intrinsic problem related to the application of meta-learning to problems similar to CartPole.

1.3 Main contributions

The main contributions of this work are twofold.

Meta-learning state of the art. A comprehensive background and state of the art are provided to understand the goals and stakes of meta reinforcement learning. State of the art algorithms in meta-learning have been implemented and their implementation has been made public. Experiments in recent papers have been recreated to verify and extend the results found in the literature.

Study of meta-learning applied to a new setting. Meta-learning is applied to a distribution of continuous state reinforcement learning problems obtained by permu-

tating the agent's observation of the environment, hiding which value is which, but also by randomly inverting the agent's actions. An extensive study about the dynamics of meta-learning applied to a problem with a continuous reward over long episodes, and the impact of the learning horizon (both in terms of number of episodes the meta-learning agent is allowed to play and the discount factor) is provided.

A critical issue is identified in the learning setup as proposed in the literature when applied to environments with a specific reward pattern and a simple and effective strategy is proposed to counteract its negative consequences, effectively allowing meta-learning to occur in problems with similar continuous reward structures.

Part I

Background

Chapter 2

Neural Networks

"*To a man with a hammer, everything looks like a nail.*"

Mark Twain

Throughout this paper, we will use the very powerful tool that a neural network is for several different purposes. One could describe a neural network in three words: general function approximators. We can describe their input and the output desired from that input – inbetween stands a very large amount of nonlinear computations of which coefficients and parameters can be tuned to make the network's output closer to what we want in a process called training, or learning.

The name "neural network" barely hides the analogy with how human neurons function – and although there are similarities to how humans learn, the analogy stays very much a conceptual analogy, and artificial neural networks stay orders of magnitude less complex than the billions and billions of interconnected neurons humans have developed.

2.1 Feedforward neural networks

A neural network is a collection of interconnected *neurons*. The reason why neurons bear this name is because of their similarity to human neurons. Each neuron is a computational unit which takes input from several other neurons (via synapses if one wishes to push the analogy further) and transforms this input into an output value, which will then be carried on to the following neurons.

The simplest and most widely used networks are feedforward neural networks. In this setting, neurons are arranged into ordered layers of which the first one is called the input layer, the last one is called the output layer and intermediate layers are called hidden layers (see Figure 2.1). Having neurons arranged in layers means that each neuron in a layer takes input from each of the neurons in the previous layer and outputs to each of the neurons in the next layer.

The input vector, also called the feature vector, is composed of **features** (measurements, properties,...). Say for example that we wanted to build a neural network that classifies cats and dogs; we could measure attributes such as the height of the animal, the length of its hair, its general loyalty to its owners and the number of hours it spends

sleeping per day and try to build a neural network that classifies them based on these features.

For the output layer, we would have a two-neuron layer: one which corresponds to "cat" and the other one to "dog". Once we have trained the network (see Section 2.1.1), the neuron that has the highest value after distilling the input values through the network would specify the kind of animal the neural network thinks the measurements best fit to.

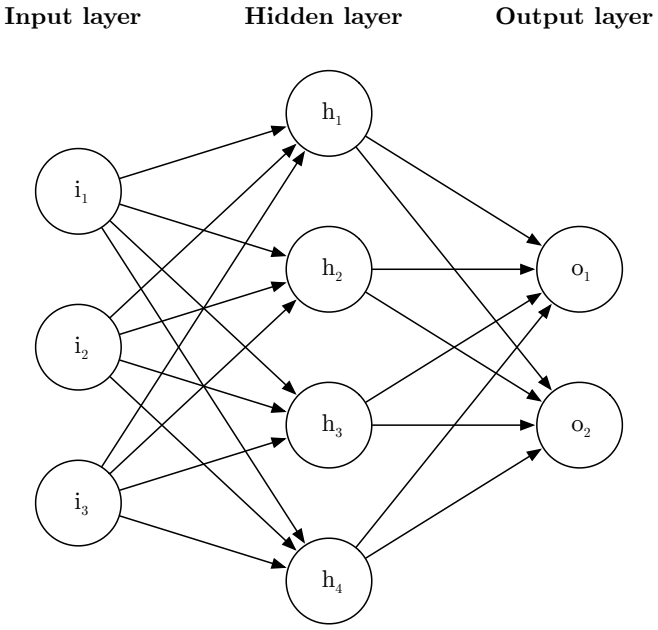


Figure 2.1: A neural network with one input layer composed of three neurons, one hidden layer of 4 neurons, and one output layer with 2 neurons.

In a feedforward neural network, each neuron is connected to every neuron in the next layer. To obtain an output from a given input, the information will be passed from layer to layer in the following way: each neuron computes a weighted sum of the outputs of all neurons in the previous layer then squashes this weighted sum in an activation function. Hence, the neuron h_1 in the network of Figure 2.1 outputs:

$$f(w_1 i_1 + w_2 i_2 + w_3 i_3 + b)$$

where w_1, w_2, w_3 are the weights corresponding to the connections between i_1, i_2, i_3 and h_1; and b is a bias term. These weights are the learnable parameters of any neural network and will be tuned during training to improve the accuracy of the network. The activation function $f(\cdot)$ can be chosen arbitrarily, as long as it is differentiable [1] (so that backpropagation can work, see 2.1.1) but it is the same for all the neurons in the same layer. Some options are the sigmoid function, the hyperbolic tangent function and sometimes a simple linear function.

[1] This is mostly true, although some activation functions that have recently been made popular because of their good performance such as Rectified Linear Units (ReLUs) [21] are not differentiable on all of their domain.

The reader might have noticed that using matrices is an efficient way of writing and executing computations within a neural network. Indeed, one could write the computation between the input layer and the first hidden layer as:

$$h = f_h(iW_{ih} + b_h)$$

where :

- h is a $1 \times n$ vector containing the values of the neurons in the hidden layer
- i is a $1 \times m$ vector containing the values of the neurons in the input layer
- W_{ih} is a $m \times n$ matrix containing the weights between the input and hidden layer
- b_h is the $1 \times n$ bias vector
- f_h is the activation function chosen for the hidden layer

Similarly, the computation between the hidden layer and the output layer can be written as:

$$o = f_o(hW_{ho} + b_o)$$

Hence the full transformation between the input and the output is:

$$o = f_o\left(f_h(iW_{ih} + b_h)W_{ho} + b_o\right)$$

The example of Figure 2.1 is an extremely simple example. Often, neural networks will be **wider** (meaning that there are more neurons per layer) and **deeper** (meaning that there are more layers). This can lead to issues and challenges when training them. The main issue related to deep neural networks is the vanishing gradient problem, exposed by Hochreiter [10]. More generally, adding more neurons (or units) increases the possibility of **overfitting** (see Section 2.1.2).

2.1.1 Training

Every neural network is initialised randomly at first, meaning that we will attribute a random value (that is sampled from a user-defined distribution) to each weight in the network.

If we want to train a network to differentiate a cat from a dog based on a given set of measurements, we will have to expose it to a large number of examples of measurements and their associated ground truths (whether the measurements actually correspond to a cat or a dog). This large number of examples is called the **training set** $\mathcal{D}$. Training is then performed by alternating feedforward passes (distilling the input through the network and obtaining the network output) and backpropagation: modifiying the weights of the connections between neurons (that will in turn modify the network output) until the network output matches the ground truth with a good enough accuracy.

The forward pass has been explained previously, but the backwards pass, also called **backpropagation**, is slightly more complicated.

Loss function

The first thing we need to improve our network is a number which measures how far the network output is from the ground truth. This is exactly what a **loss function** does: it describes numerically the error between the target and the network output. One example of a loss function is the mean squared error (MSE) which is very useful in regression problems:

$$\mathcal{L}_{\text{MSE}} = \sum_{(x,t)\in\mathcal{D}} \frac{(f(x)-t)^2}{|\mathcal{D}|}$$

where x is a feature vector, t is its corresponding ground truth and f is our model. In many cases, $f(x)$ and t are vectors, so we may sum or average the loss computed over components.

For classification problems, like the cats and dogs example, one typically uses a cross-entropy loss which is defined as the following for one sample:

$$-\sum_c t_c \log(f(x)_c)$$

where c is the class index, hence the loss computed over the whole dataset is:

$$\mathcal{L}_{\text{CE}} = \sum_{(x,t)\in\mathcal{D}} \frac{-\sum_c t_c \log(f(x)_c)}{|\mathcal{D}|}$$

In our cats and dogs example, let us consider that the ground truth for "cat" corresponds to the output vector $[1, 0]$ and the ground truth for "dog" corresponds to the output vector $[0, 1]$. If the network predicts $[0.3, 0.7]$ when it is given measurements of a cat, the loss function will equate to :

$$\mathcal{L} = -(1\log(0.3) + 0\log(0.7)) = 0.52$$

whereas if it had predicted $[0.9, 0.1]$, the loss would have equated to:

$$\mathcal{L} = -(1\log(0.9) + 0\log(0.1)) = 0.04$$

The experienced reader might already have concluded that since we want to minimise the loss function, we are confronted with an optimisation problem.

Backpropagation

The problem at hand is indeed an optimisation problem with the following parameters:

- the objective function, which we want to minimise, is the loss function
- the parameters are all the weights of the neural network

How can we link the parameters to the objective function? A key requirement for activation functions and the loss function is that they have to be **differentiable**. When this is the case, we can compute the gradient of the loss function (in other words, the gradient of the error) and backpropagate it through the network by using the chain rule

of derivation. The derivative of the loss function can be written in terms of its partial derivatives with respect to each of the weights in the network:

$$\nabla \mathcal{L} = \left(\frac{\delta \mathcal{L}}{\delta w_1}, \frac{\delta \mathcal{L}}{\delta w_2}, \frac{\delta \mathcal{L}}{\delta w_3}, ..., \frac{\delta \mathcal{L}}{\delta w_k} \right)$$

Each weight will then be modified by adding a small increment in the direction of the gradient:

$$\Delta w_i = -\alpha \frac{\delta \mathcal{L}}{\delta w_i}$$

where α is the learning rate, a hyperparameter chosen by the designer of the network that defines the size of the steps to make in the direction of a smaller error. A high learning rate may accelerate training but could reduce the chance of finding a global optimum. Reducing the learning rate, on the other hand, could make the training slower but at the same time increase the probability of finding a global optimum.

Let us derive the gradient of a weight which connects a neuron in the last hidden layer to a neuron in the output layer in the case where we have a MSE loss:

$$\Delta w_{oh} = -\alpha \frac{\delta \mathcal{L}}{\delta w_{oh}}$$

which can be expanded using the chain rule:

$$\Delta w_{oh} = -\alpha \frac{\delta \mathcal{L}}{\delta a_o} \frac{\delta a_o}{\delta net_o} \frac{\delta net_o}{\delta w_{oh}} \tag{2.1}$$

where net_o is the input of neuron o and a_o is the activation value (or the output value) of neuron o.

Let us analyse each partial derivative of equation 2.1. The derivative of the error with respect to the activation is :

$$\frac{\delta \mathcal{L}}{\delta a_o} = \frac{\delta(\frac{1}{2}(t_o - a_o)^2)}{\delta a_o} = -(t_o - a_o)$$

Notice that we injected a $\frac{1}{2}$ factor to the error to simplify computations.

The derivative of the activation with respect to the input of the neuron is:

$$\frac{\delta a_o}{\delta net_o}$$

It is simply the derivative of the activation function chosen for the output layer. Since this derivative will vary depending on the choice of activation function, we will simply denote is as f'_o.

The derivative of the neuron input with respect to the weight is

$$\frac{\delta net_o}{\delta w_{oh}} = \frac{\delta(w_{oh} a_h)}{\delta w_{oh}} = a_h$$

Hence we will update a weight between the last hidden layer and the output layer with:

$$\Delta w_{oh} = \alpha(t_o - a_o) f'_o(net_o) a_h$$

For weights that connect the penultimate hidden layer to the last layer (in our case, the input layer to the hidden layer), the error is slightly more complicated as it depends on the error of all k neurons in the hidden layer, and not just of one output neuron. The gradient of such a weight can be written as:

$$\Delta w_{hi} = -\alpha \sum_k \left(\frac{\delta \mathcal{L}}{\delta a_k} \frac{\delta a_k}{\delta net_k} \frac{\delta net_k}{\delta a_h} \right) \frac{\delta a_h}{\delta net_h} \frac{\delta net_h}{\delta w_{hi}} \tag{2.2}$$

A similar reasoning can be used to unfold equation 2.2.

Backpropagation uses the fact that activation functions are differentiable to perform gradient descent. Indeed, the error signal is a surface in a space of which the dimensionality is equal to the number of parameters in the network. Gradient descent aims to find the global minimum of this error surface.

Summary

To train a neural network, its weights have to be randomly initialised first, then we show it many examples of the training set, containing measurements and ground truths. For each of these examples, or samples:

1. the feedforward pass computes the network output from the sample input data
2. the network output is compared with the sample output and the error signal given by a loss function is backpropagated through the network, updating all the weights

2.1.2 Testing, overfitting and regularisation

So far, we only discussed training neural networks, but testing and evaluating their performance is just as important. Indeed, as hinted at previously, there is a chance that neural networks **overfit**. This means that the network will learn a function which fits the training data very well, but that doesn't **generalise** to new data. We use another dataset, the **validation** set, which contains pairs of feature vectors and their associated ground truths that have never been seen by the model during training. If the performance of the network is much lower on the validation set than on the training set, we know that it has overfit the training data. We use the validation set to test different values of hyperparameters (number of layers, size of layers, learning rate,...), and we choose the combination of hyperparameters and overfitting avoidance techniques that minimise the error on the validation set. An illustration of overfitting is shown on Figure 2.2.

There are several ways of avoiding overfitting, of which some are:

- **dropout** [23] which "disables" neurons in the network with a certain probability during training. This allows for a large combination of thinner sub-networks to learn separately from other sub-networks.
- adding more data, or augmenting the existing data by transforming it can also increase generalisation performance as the network won't "remember" a given set of examples as opposed to truly trying to understand the concepts we want it to learn.

- **early stopping** is a method that checks the error on the validation set during training, and that stops training whenever the error on the validation set starts increasing. The increase of validation error will usually mean that the network has started overfitting the training data.

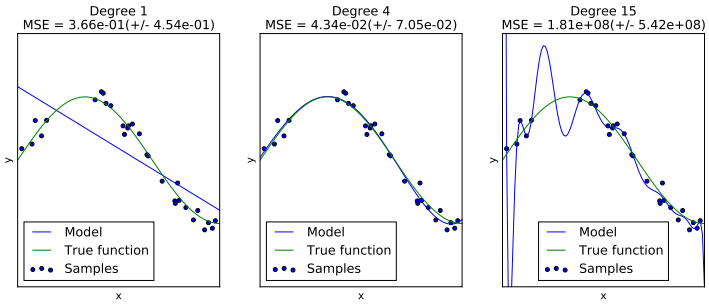


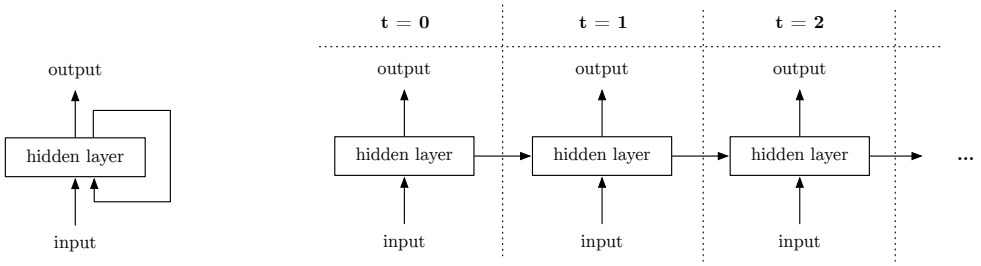
Figure 2.2: An illustration of underfitting (left) and overfitting (right). The MSE metric shows the error on a validation set for each learned polynomial. In green, the function we are trying to approximate, from which samples are generated with a small error. In blue, our estimation of the true function estimated with different degrees of polynomial based on the samples.

Finally, when we have minimised the error on the validation, we use a **test set**. This set contains new pairs of feature vectors and their targets that are neither in the training set nor the validation set. Using a test set helps avoiding overfitting the *hyperparameters* of the model. Indeed, finding the hyperparameters that minimise the error on the validation set is a sort of overfitting; and running the model on completely unseen data from the test set gives a true estimation of the generalisation performance of our model. Once this estimation has been made, we cannot tune the model any further (or we would have to use a different test set).

2.2 Recurrent neural networks

There is one limitation to feedforward neural networks which can be particulary critical for some applications. Very often, having an output mapped to a single input is not enough. For example, in problems where samples are organised in a timeline and dependent on each other, only seeing one input at a time does not provide enough information. If one wants to predict the weather temperature in the next hour, it would be useful to know more than only the last ground truth value to identify a potential trend. One could input a fixed-size window of the previous time steps to the network, but this approach limits the scope of available "memory" and could dramatically increase the complexity of the model.

One solution to this problem is the use of recurrent neural networks. A feedback loop such as the one shown in Figure 2.3a, also called a recurrent connection, means that the layer at the receiving end of the recurrent connection will receive as input not only the value of the previous layer at time t, but also the value of the layer at the giving end of the recurrent connection at time $t-1$. This allows information to *live* throughout time steps.



(a) A recurrent connection (b) An unrolled recurrent neural network

Figure 2.3: Recurrent neural networks

A recurrent connection, however, poses evident issues for the backpropagation algorithm since the depth of the neural network can become theoretically infinite. A common way of training and visualising recurrent neural networks is to unroll them for a given, finite amount of time steps (see Figure 2.3b). The error signal can then be computed for the output value at each time step, and backpropagated through all the previous input values that affected its computing.

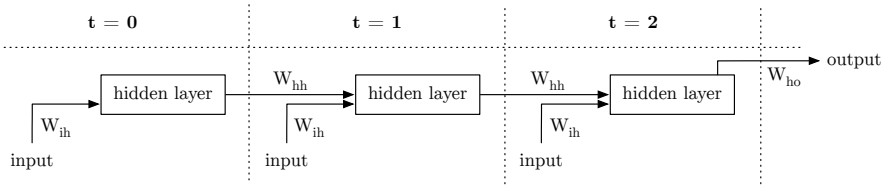


Figure 2.4: Backpropagation through time

Backpropagation through time [20] works conceptually in the same way as normal backpropagation. For a sequence of 3 steps such as the one shown on Figure 2.4, we consider that the network has 3 layers. One key difference with standard backpropagation is that the weights between each layer are actually the same since we only unrolled the network. The equations of backpropagation have to be slightly adapted to perform gradient descent on the recurrent weights. The partial derivative of the loss function at $t = 3$ with respect to the recurrent weights is:

$$\frac{\delta \mathcal{L}_3}{\delta W_{hh}} = \frac{\delta \mathcal{L}_3}{\delta a_o} \frac{\delta a_o}{\delta net_o} \frac{\delta net_o}{\delta W_{hh}} \tag{2.3}$$

but this time, net_o depends on net_{h_3} which in turn depends on W_{hh}, so when we take its derivative with respect to W_{hh}, we cannot take it as a constant. The equation has to be unfolded like this:

$$\frac{\delta \mathcal{L}_3}{\delta W_{hh}} = \sum_{k=0}^{2} \frac{\delta \mathcal{L}_3}{\delta a_o} \frac{\delta a_o}{\delta net_o} \frac{\delta net_o}{\delta net_{h_k}} \frac{\delta net_{h_k}}{\delta W_{hh}} \tag{2.4}$$

Unrolled recurrent neural networks are just deep neural networks, and the more they are unrolled, the deeper they get, so they also suffer from the vanishing or exploding gradients problem [11, 5].

2.2.1 Long Short-Term Memory (LSTM)

One major issue encountered when using recurrent neural networks is that they are famous for being unstable. Very often, at some point, training will diverge (meaning that some gradients will either explode or vanish) and will never reach a satisfying optimum. Several solutions have been presented to counter this unstability such as Long Short-Term Memory (LSTM) cells [12], which we will explain and use in this work, or Gated Recurrent Units (GRUs) [6].

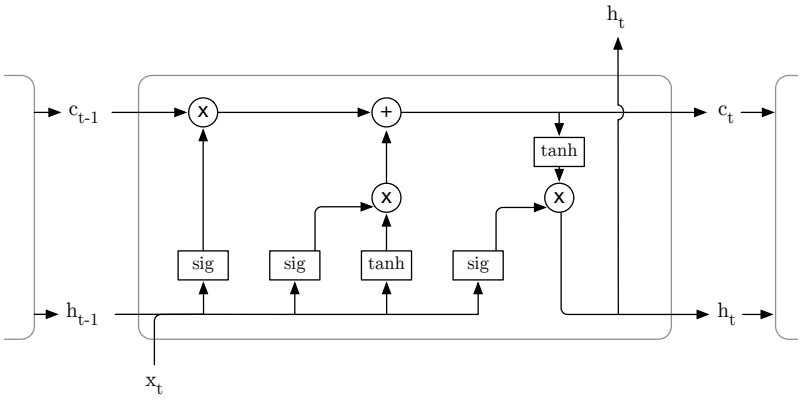


Figure 2.5: The internals of a LSTM cell. Circles represent element-wise operations (arrows carry vectors). sig and tanh are layers. c_t is the memory cell.

Instead of carrying to the next timestep only the hidden state (which is also the output of the cell), LSTMs also carry what is called the *memory cell* c_t which isn't used as output. This memory cell is able to handle longer term memory management by using *gates*, which are rather simple mutliplicative connections, to access and modify it. The idea of using gates is to allow the network to learn to explicitly control its memory by erasing, adding or updating the information that is in its memory cell.

All gates receive a concatenation of the previous hidden state h_{t-1} and the new input vector x_t. They feed this input through a sigmoid layer (sig) to an element-wise multiplication. There are three in total, as they can be seen from left to right on Figure 2.5:

1. the *forget gate* is the first one. It will multiply each value in the memory cell by a value in $[0, 1]$, keeping only the values it considers useful.

2. the *update gate* is the second one and it chooses which ones of the outputs of the first tanh layer will make their way to the memory cell.

3. the *output gate* is the last one and decides what parts of the memory cell make it to the output.

A deeper analysis of LSTMs and an insight into their behaviour has been provided by Karpathy et al. [14]

Summary

Neural networks are very useful tools that, when designed and trained the right way, can transform their input to any desired output, allowing one to can approximate any

function (although under certain assumptions [13]).

However useful they are, we have seen that they are not as tame as one would wish. Indeed, as they get deeper, their likelihood of diverging or overfitting increases, so we must be careful when using them.

Chapter 3

Reinforcement Learning

> "*Tell me and I forget, teach me and I may remember, involve me and I learn.*"
>
> **Benjamin Franklin**

3.1 The reinforcement learning problem

A reinforcement learning setting, as defined in Sutton and Barto [24] sees two main components interact : the **agent** (an entity performing actions) and the **environment** (the world in which the agent is situated). The environment can be in several states which the agent can observe. In some problems, the agent might not be able to observe the full state of the environment (making the problem a partially observable one). The agent chooses actions based on its knowledge of the state of the environment. These actions might (and often do) alter the state of the environment, but will also generate a **reward**. The goal of the agent is to maximise the obtained reward.

With this simple setting, of which a diagram is shown in Figure 3.1, we can design algorithms that can be trained to solve a huge variety of tasks without ever having to include task-specific logic to the algorithm.

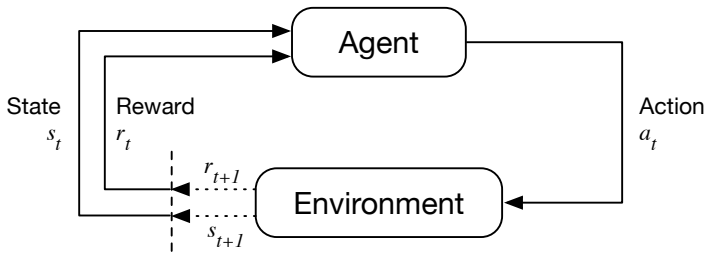


Figure 3.1: The setting of a reinforcement learning problem [24]

For example in the CartPole problem : the cart can be in several positions and have different velocities, and the pole has similar characteristics; the values of these characteristics constitute the state of the environment. The agent has two actions at its

disposal : nudge the cart to the left, and nudge the cart to the right. The reward of the CartPole environment is +1 every timestep the pole is balanced within a given angle and the cart is within a certain range on the rail, 0 otherwise.

3.1.1 Markov Decision Processes

A reinforcement learning problem can be formally defined as a Markov Decision Process (MDP) characterised by :

- a set of states $\mathcal{S}$
- a set of actions $\mathcal{A}$
- a transition function $T(s, a, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$
- a reward function $r(s, a, s') = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s']$

The goal of reinforcement learning is for the agent to select, in any state it can be in, the action that will lead it to the highest expected reward :

$$R_t = \mathbb{E}[r|\pi] = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}^2 + ... = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{3.1}$$

with π the policy (see Section 3.1.2) and the discount factor $\gamma \in [0, 1[$. The discount factor allows one to tune the agent's behaviour on the short-term/long-term spectrum. A discount factor $\gamma = 0$ would mean that the agent maximises its expected reward for the next transition only whereas a discount factor close to one will favour behaviour that maximises long-term reward, even if one action leads to a poor reward at first. It is chosen by the designer as a hyperparameter for the whole length of the training.

3.1.2 Policy

The agent uses a policy $\pi(a \mid s)$ to choose which action it is going to play next. The policy describes a probability distribution over the action set $\mathcal{A}$, determining the probability of selecting each action $a_i \in \mathcal{A}$ once it is in a certain state s_i.

We can call the policy **deterministic** if and only if:

$$\forall\, s \in \mathcal{S},\ \exists\, a \in \mathcal{A} : \pi(a \mid s) = 1 \tag{3.2}$$

In other words : in any state, only one action can be selected. Otherwise, the policy is **stochastic**.

3.2 An example : the 2-armed bandit problem

The 2-armed bandit setting defines two actions that the agent can perform, each of which associated with one arm. Each arm has an underlying reward distribution – for the sake of this example, let us define each arm with the following Bernoulli distributions:

Arm #1	Arm #2
0.9	0.1

This means that arm #1 will generate a reward of 1 with a probability of 0.9 and arm #2 will generate a reward of 1 with a probability of 0.1.

The careful reader will have noticed that this problem is stateless ($\mathcal{S} = \{\varnothing\}$), meaning that the agent only perceives rewards and no state observations.

Of course, before its first interaction with the environment, the agent has no information whatsoever about the reward distribution of each arm, and the whole idea of reinforcement learning is to look for evermore efficient ways to learn the structure and parameters of a problem in order to maximise reward.

One could imagine a simple strategy like the following : play each arm 10 times, then always play the arm with the highest average reward. This policy is deterministic, and yields either ($\pi(a_1) = 1$ and $\pi(a_2) = 0$) or ($\pi(a_1) = 0$ and $\pi(a_2) = 1$). This could work in our case, but might fail in cases where the distributions are much closer (e.g. 0.45 and 0.55). What if the reward was sampled out of overlapping normal distributions?

Deciding whether the following action should be used to explore the environment and gain information about it; or to exploit the information already available to try to maximise reward is not trivial. Exploring could make training slower or affect the reward if our model of the environment is correct, but it is needed to increase the probability that we have indeed a correct model. This tradeoff is called the exploration/exploitation tradeoff.

Sutton and Barto [24] propose two simple ways to tackle this tradeoff : ϵ-greedy and softmax selection. Both methods fall within the denomination of Action-Value methods, meaning that a value will be attributed to each action based on how well it did in the past. Suppose action a has been chosen K_a times before timestep t, and the agent has received rewards r_1, r_2,... while playing it. The value of action a is:

$$Q_t(a) = \frac{r_1 + r_2 + ... + r_{K_a}}{K_a} \tag{3.3}$$

ϵ-greedy

This strategy keeps an average of the reward of each earm and chooses the next action the following way:

$$\begin{cases} a \text{ with } \max\limits_a Q_t(a) & \text{with probability } (1-\epsilon) \\ \text{random } a & \text{otherwise} \end{cases} \tag{3.4}$$

A high ϵ forces the agent to explore more, while a low ϵ allows for more exploitation of the agent's knowledge of the problem. ϵ is a hyperparameter of the agent, meaning that it will have to be chosen by the designer depending on the problem and its parameters to strike an optimal balance between exploitation and exploration.

Softmax selection

Instead of selecting the best action with a given probability, the softmax selection method defines a distribution probability over the whole action space $\mathcal{A}$. This is usually

done by using a Boltzmann distribution (also called a Gibbs distribution):

$$\frac{e^{Q_t(a)/\tau}}{\sum_{i=1}^{n} e^{Q_t(i)/\tau}}$$

which defines the probability of selection for each action. τ is called the *temperature* parameter. A high temperature smooths the distribution, making all actions more equally likely to be selected, and a low temperature will tend to behave more like greedy action selection.

3.3 Neural networks for reinforcement learning

As hinted at previously, neural networks are powerful function approximators and have yielded impressive results in reinforcement learning. They are used either directly as policy functions, taking a state as input and outputting a probability distribution over the action set; or as value estimators, by outputting a scalar which estimates the value of states and actions related to the estimated achievable discounted reward from those states.

3.3.1 Policy gradient methods

Policy gradient methods are rather straightforward ways of learning to solve a task. They assume that the policy is an end-to-end block of differentiable computation (this could be a neural network, but other function approximators work too):

$$\pi(a \mid s)$$

This means that the policy π takes as input the state, and outputs a probability distribution over the whole action set $\mathcal{A}$.

If we just received a good reward from the environment after choosing an action a from state s with respect to the policy π, we could directly change the policy to increase the probability of selecting a when we are in state s. The experienced reader might have noticed that this process is very similar to how we performed gradient descent to improve a classifier neural network in the backpropagation section (2.1.1). Indeed, classification is about increasing the probability of the correct class when given a sample (or decreasing the probability of the wrongly predicted class). Similarly to how class labels are backpropagated through the differentiable function, we can backpropagate rewards. Figure 3.2 shows the process similarity between supervised learning and policy gradient methods.

Let us assume that the policy is defined by a neural network of which the input layer is set to receive an observation of the state of the environment and the output layer defines a policy (a probability distribution over the actions set). Training starts with a randomly initialised policy. When the agent performs an action chosen with its policy, the environment will update its state and the agent will receive an observation of this state as well as a reward signal. We can use the reward signal to proportionately encourage (if the reward is positive) or discourage (if the reward is negative) taking this action in this specific state.

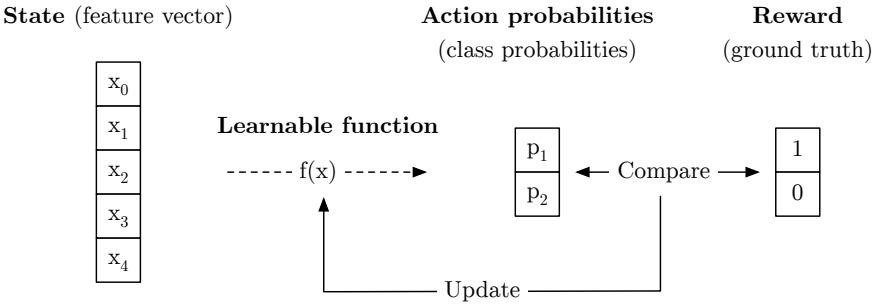


Figure 3.2: The policy gradients process, and its similarity to the training process of supervised learning. Text in parentheses represents the supervised learning equivalent of the vectors involved in policy gradients.

As the policy is defined by a neural network, we have to define a way to update the network to find the best policy. This is done by defining a loss function of which the gradient will be propagated back into the network, or by defining directly the gradient as a parameters update rule.

Let us suppose that the agent is in an environment where it can perform two actions, and it has received a reward of +5 after choosing action 2 sampled out of its policy $\pi(s) = [0.1, 0.9]$. We can directly use the reward signal as a scaler for the gradient to update the network. Indeed, multiplying the gradient of the policy $\nabla\pi(a \mid s)$ by $[0, 5]$ (because a +5 reward was generated by action 2) will make the network learn which actions to perform under certain states by directly increasing the probability of playing actions which receive a positive reward, and by decreasing the probability of actions which receive a negative reward.

We can prove mathematically that multiplying the gradient of the policy equates to the gradient of the expected reward from state x:

$$\begin{aligned}
\nabla\mathbb{E}_x[r(x)] &= \nabla \sum_x \pi(x) r(x) \\
&= \sum_x \nabla\pi(x) r(x) \\
&= \sum_x \pi(x) \frac{\nabla\pi(x)}{\pi(x)} r(x) \\
&= \sum_x \pi(x) \nabla \log(\pi(x)) r(x) \\
&= \mathbb{E}_x[r(x) \nabla \log(\pi(x))]
\end{aligned}$$

The method as presented still presents an issue in the sense that it only takes into account immediate reward, and we may want to maximise long-term reward. Let R_t, the discounted future reward at time step t (see equation 3.1), be the metric that we want to maximise. In a policy gradients method, we will update the neural network policy directly with :

$$\alpha\nabla\pi(a \mid s) R_t \tag{3.5}$$

in which α is the learning rate and a was the chosen action.

3.3.2 Value methods

There is another category of methods which use an estimate of the value of actions or states; in other words functions which map states or state-action pairs to estimates of their discounted future reward R_t. We have seen examples of such functions previously: ϵ-greedy and softmax selection both are value-based methods (see Section 3.2 and more specifically equation 3.3).

There is an important difference between value-based methods and policy gradient methods: in the former, the policy has to be defined by hand whereas the latter are end-to-end (meaning that the policy is learned too). In value-based methods, the model designer has to choose a policy that selects an action based on estimations provided by value functions.

V-values

Let us consider a problem with a relatively small set of states $\{s_1, s_2, ..., s_k\}$. We could estimate the **value** of states based on the expected discounted reward we can reach from each state, based on a policy π:

$$V^{\pi}(s) = R_t = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}^2 + ...\right]$$

At first however, this estimate is not known. Let us imagine a problem where only one state s_o generates a reward of $+100$, and a state s_{o-1} is a state adjacent to s_o. We choose to use a discount factor $\gamma = 0.9$. Whenever the agent plays the action that makes it go from s_{o-1} to s_o, the agent receives a reward of $+100$. Its value for s_o is:

$$V^{\pi}(s_o) = r_t = 100$$

What about the value of s_{o-1}? Being right next to the goal state, it should have quite a high value, while at the same time being lower than the goal state. This is exactly what happens if we use our value estimation formula:

$$V^{\pi}(s_{o-1}) = r_t + \gamma r_{t+1} = 0 + 0.9 \times 100 = 90$$

What about a state s_{o-2} adjacent to s_{o-1} but not s_o? We would compute:

$$V^{\pi}(s_{o-2}) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} = 0 + 0 + 0.9^2 \times 100 = 81$$

The reader might have noticed that there is a simpler way to estimate values than by unrolling the agent's full experience until it gets a reward. We can indeed use a Bellman equation to rewrite our estimation using the values of adjacent states only by iterating on our current estimates:

$$V_{k+1}^{\pi}(s) = R_t = \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2}^2 + ...\right] = \mathbb{E}\left[r_t + \gamma V_k^{\pi}(s')\right] \tag{3.6}$$

where s' is the state of highest value that can be reached from s and k is the iteration number. We could rewrite this equation as the following:

$$V_{k+1}^{\pi}(s) = \max_a \sum_{s'} p(s \mid s, a)\left[r(s, a, s') + \gamma V_{k+1}\pi(s')\right]$$

Iterating over estimates means that we start off with a nonoptimal value function. It would theoretically take an infinite amount of iterations to converge to an optimal value function $V^*(s)$, but we generally stop once the value function changes by a small enough amount.

Once again, we can use neural networks to estimate such functions. We then only have to define a policy that decides which actions to take given these value estimates; one example of such a policy could be a deterministic policy choosing the action leading to the state with the highest value.

Equation 3.6 can be used as an update rule for the V^π network parametrised by its weights θ with a loss similar to :

$$\left[V^\pi(s \mid \theta_t) - \left(r_t + \gamma V^\pi(s' \mid \theta_{t-1})\right)\right]^2 \tag{3.7}$$

which is a simple mean squared error between the estimation $V^\pi(s \mid \theta_t)$ and the target $(r_t + \gamma V^\pi(s' \mid \theta t - 1))$.

Q-values

Q-values estimate the value of state-action pairs instead of only states; i.e. the estimated discounted reward achievable when taking a specific action in a specific state. The reasoning to find an update rule is similar for V-values and Q-values. The optimal Q^* function can be estimated with a Bellman equation :

$$Q^*(s,a) = \mathbb{E}\left[r_t + \gamma \max_{a'} Q^*(s',a')\right]$$

and so can be deduced an update rule for Q networks of which the loss can be defined with the following :

$$\left[Q(s,a \mid \theta_t) - \left(r_t + \gamma \max_{a'} Q(s',a' \mid \theta_{t-1})\right)\right]^2 \tag{3.8}$$

Value networks in real life : DQN

Estimating values using neural networks has an enormous advantage over using tables of values. Using tables becomes intractable as the size of the state space increases (or action space in the context of Q-values); worse, they become unusable if the state space becomes continuous, unless we discretise it, which might cause unwanted artifacts.

However, neural networks are not always stable, and they couldn't be used for complex reinforcement learning tasks until Mnih et al [18, 19] proposed their Deep Q-network architecture. In their setting, Mnih et al. use the loss function described in equation 3.8 with two key modifications that allow the network to learn in a stable way:

Replay memory Instead of updating the network using only the last transition that the agent went through (the tuple (s, a, r, s')), Mnih et al. sample a transition from a replay memory in which are stored a large number of played transitions. This allows the agent to decorrelate samples that come from the same sequence of transitions.

Target network Equation 3.8 uses a Q-value estimate from the previous timestep as a target. This can cause oscillation and divergence. To mitigate these effects, Mnih et al. proposed using a target network in which the weights are copied from the network being trained only periodically.

Another key component of Mnih et al.'s architecture is the use of convolutional neural networks [16] which take advantage of the relevance of spatial context in an image state. We won't delve into the description of such networks as they won't be needed in this work.

3.3.3 Actor-Critic

We can call policy gradient methods "actor-only" methods as the methods only choose actions without evaluating them *per se*. On the other hand, value-based methods can be called "critic-only" methods as they evaluate possible actions and states but do not provide the agent with a choice of action.

There is a third category of methods which mixes both approaches. Actor-critic methods [24] have both an *actor* (the policy) and a *critic* (value functions). The actor performs the actions, and the critic updates the policy. One way to update the policy using a critic is to scale the policy gradient using our current estimate of the V function:

$$\nabla\pi(a_t \mid s_t)(R_t - V(s_t)) \tag{3.9}$$

This way, the policy will be updated according to how wrong the critic was. The value $R_t - V(s_t)$ is called the **advantage** because it expresses how much better (or worse) the reward was compared to what was estimated [7, 17].

The A2C algorithm

We now have all the cards in our hand to understand the A2C (Advantage Actor-Critic) algorithm which is a simpler version of A3C [17] (it only uses one thread instead of being asynchronous). A2C consists of a neural network of which the input is the state s_t, and which outputs both a value function $V(s_t)$ and a policy $\pi(s_t)$ (see Figure 3.3). These outputs respectively use the update rules described in equations 3.7 and 3.9. The first loss function computes the error related to the policy output:

$$\mathcal{L}_p = \pi(a_t \mid s_t)(R_t - V(s_t))$$

and the second loss computes the error related to the value output

$$\mathcal{L}_v = (R_{t-1} - V(s_t))^2$$

The complete loss function for the A2C agent is:

$$\mathcal{L} = \beta_v\mathcal{L}_v + \beta_p\mathcal{L}_p$$

where β_v and β_p are hyperparameters used to scale both losses.

Algorithm 1 shows the training process of the A2C agent which alternates playing episodes and updating the network according to the agent's experience.

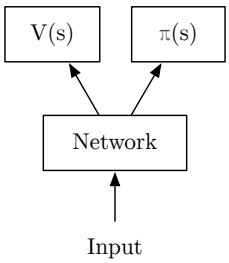


Figure 3.3: The A2C network

Algorithm 1 The A2C training process

$T_{\max} \leftarrow$ maximum number of ticks
while $T < T_{\max}$ **do**

    // Play an episode
    $t \leftarrow 0$
    while $t <$ maximum episode length **and** episode not finished **do**
        perform a_t sampled using $\pi(s_t)$
        record reward r_t and new state s_{t+1}
        $t \leftarrow t + 1$
    end while
    $T \leftarrow T + t$

    // Compute advantages and gradients by unrolling episode
    $d\theta_p \leftarrow 0$ *// policy gradient*
    $d\theta_v \leftarrow 0$ *// value gradient*
    for $i \in \{t-1,\ t-2,\ ...,\ 0\}$ **do**
        $R \leftarrow r_i + \gamma R$
        $d\theta_p \leftarrow d\theta_p + \nabla\pi(a_i|s_i)(R - V(s_i))$
        $d\theta_v \leftarrow d\theta_v + \nabla(R - V(s_i))^2$
    end for
    Update policy network using $d\theta_p$
    Update value network using $d\theta_v$

end while

Summary

Reinforcement learning is the study of the behaviour of an agent situated in an environment generating rewards for the agent. Many existing techniques allow the agent to learn to perform well in many environments, and using neural networks to solve reinforcement learning problems has shown to be very powerful.

We will now see how we can adapt the A2C agent to perform meta reinforcement learning.

Part II

Meta Reinforcement Learning

Chapter 4

Learning to learn

"All of the biggest technological inventions created by man - the airplane, the automobile, the computer - says little about his intelligence, but speaks volumes about his laziness. "

Mark Kennedy

4.1 Goals and foundations

Developing and tuning algorithms to find the optimal strategy to solve a reinforcement learning problem is hard. Some of the challenges one meets are:

- the tradeoff between exploration and exploitation
- designing a strategy that allows for versatile training
- choosing hyperparameters (learning rate, architecture of a neural network,...) that make the strategy optimal considering the reinforcement learning problem at hand.

Let us consider the problem of learning a simple bandit problem with dependent arms. Say, for example, that a bandit has two arms, each of which producing a reward according to a Bernoulli distribution with the following parameters :

$$\begin{cases} P(r \mid \text{arm}_1) = p_b \\ P(r \mid \text{arm}_2) = 1 - p_b \end{cases}$$

where $P(r \mid \text{arm}_1)$ is the probability of arm 1 to generate a reward and $0 \leq p_b \leq 1$ is the parameter of the bandit problem. One way to solve this problem would be to use the ϵ-greedy strategy (see Section 3.2).

This raises the question about choosing ϵ. Choosing a low ϵ encourages exploitation but we have a higher chance of wrongly estimating p_b. Choosing a high ϵ doesn't allow the agent to perform optimally once its knowledge about the parameters of the problem is likely to be optimal. Moreover, we could choose a variable ϵ, high at first but slowly converging to 0, but we then are faced with the choice of another hyperparameter: the number of steps over which ϵ should be annealed.

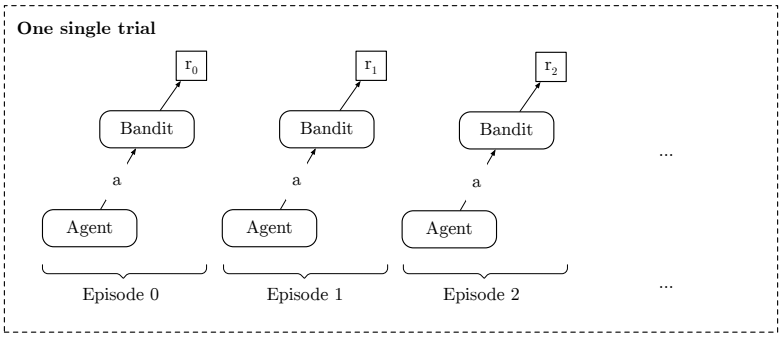


Figure 4.1: A classic training sequence to learn a bandit problem. One manually designs a strategy which inspects the reward obtained at the end of an episode and evaluates accordingly the performance of the action taken. The strategy then updates the policy inbetween episodes.

One could propose using a more advanced method, and there are many. Multi-armed bandits have been heavily studied and to this day, several algorithms exist to solve the kind of bandit problems defined above very quickly - i.e. to explore just enough to implicitly learn p_b so to exploit the best arm as much as possible. Examples of these algorithms are the Gittins indices algorithm [9], UCB [3] and Thompson sampling [25].

The obvious problem here is that we can always choose to manually design more and more sophisticated techniques and to tune more perfectly their parameters, but could we not instead use reinforcement learning to do this for us instead?

Recently, Wang et al. [27] and Duan et al. [8] proposed the idea of using reinforcement learning to learn an algorithm which deploys an optimal strategy for a class of problems sharing a similar structure. We will use the name "meta-RL" or meta-learning for this technique, as proposed by Wang et al.

In classic reinforcement learning (see Figure 4.1), **the network represents a policy.** Its goal is to maximise its reward for each episode, and it is trained by looping the following steps:

1. let the network play an episode
2. update the network using a hand-designed strategy (Gittins, UCB, Q-learning [28], SARSA [22], ...) to increase the total reward of an episode

As stated at the beginning of this chapter, designing strategies to learn policies is a hard task and can demand a sensitive tuning of parameters. In meta-RL, **the network represents an algorithm which learns a policy** Its goal is to be the best possible learning strategy, which is why we train it differently:

1. let the network play a **trial** of several episodes of the same problem
2. update the network so that it learns faster

In this context, the network has to maximise its expected reward across **all** episodes of a trial (see Figure 4.2). Note that a trial replicates the setting of learning to solve a problem – the agent is not learning to solve a problem anymore, it is learning to learn

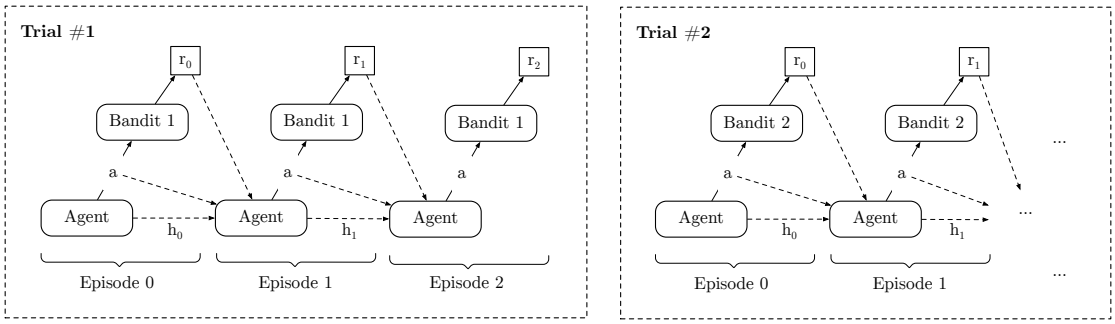


Figure 4.2: The training sequence for a meta-RL agent on a bandit problem. In this case, the agent is left to play several episodes on its own without changing its weights. It is only when a trial ends that the outer algorithm evaluates the rewards of all episodes in the trial and updates the weights of the inner algorithm so that it increases its chances of having a better reward across all episodes. For trials of three episodes such as the one presented in this figure, the outer algorithm forces the inner algorithm to learn the problem in three episodes.

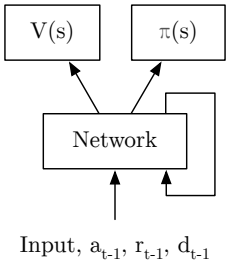


Figure 4.3: The meta-learning A2C agent. The key difference with a standard A2C agent is that it receives the values $[a_{t-1}, r_{t-1}, d_{t-1}]$ in addition to the state observation. It is of course also trained in a different way to a standard A2C agent.

how to solve a problem, but in a very low number of episodes rather than the many episodes needed to train a standard reinforcement learning agent. This will incentivize the agent to understand the structure of a problem and develop a strategy that allows it to estimate the parameters of the problem as fast as possible. Generally, we will call the learned policy the inner algorithm, and the algorithm that learns the policy the outer algorithm.

In our manually designed strategies, we use previous actions and rewards to update the policy and make it better. Similarly, in order to make meta-RL work, the agent has to receive as input the previous reward and the action that led to that reward, but it also needs to carry on some sort of memory of past actions and rewards. We do this manually in ϵ-greedy by keeping track of the average reward yielded by each arm. In the case of meta-RL, this is done by using a recurrent network which carries a hidden state from the first episode in the trial to the last.

The general setup presented by Wang et al. uses a recurrent neural network of which the weights, once trained "slowly" over several trials of multiple episodes, will encode the "fast" learning policy. Figure 4.3 shows the meta-learning A2C agent presented in Wang et al. [27]. There are two differences to note compared to Figure 3.3: the first one

is the recurrent connection, and the second one is the set of values we input to the agent at each time step. In addition to the state of the environment, we add the previous action, previous reward and a termination flag which is set to 1 when an episode has reached a terminal state; 0 otherwise.

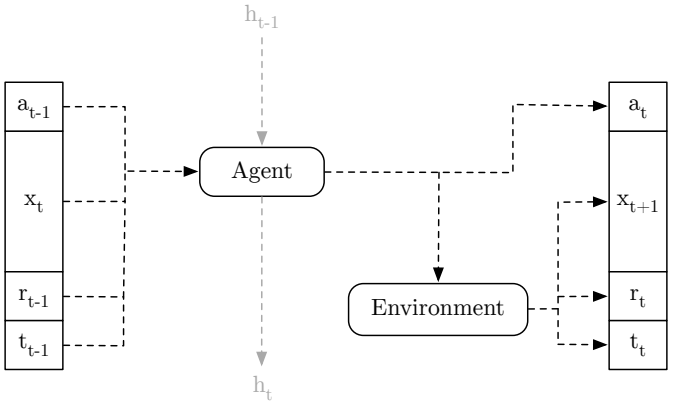


Figure 4.4: An unrolled timestep in the meta-RL training setting. At each timestep, the agent receives an observation of the state of the environment as well as the previous action taken, the associated reward, and the termination flag. It also receives its previous hidden state if and only if the previous timestep was part of the same trial (this could still mean that the last timestep was from a different episode which just terminated)

It is important to emphasize on the fact that the agent passes on its hidden state between different episodes of the same trial (and if episodes last for more than one timestep, between timesteps as well), but **not** between trials (see Figure 4.2). The reason why the inter-episode connection is needed is because the agent might want to deploy a different policy given the results of previous episodes as it has to optimise its expected reward across multiple episodes.

4.2 Agent architecture

We use the same A2C agent that Wang et al. [27] have used for their bandit experiments. It is described in Figure 4.5. The input vector is a concatenation of the state observation, the previous action taken, the reward obtained at the previous timestep and the termination flag :

$$i = [s_t, a_{t-1}, r_{t-1}, d_{t-1}]$$

The action taken is converted to a one-hot encoding (converting the action index into a vector of length $|\mathcal{A}|$ containing zeros except a single 1 at the index of the action). This vector is then fed to a LSTM sized at 48 hidden units, followed by two branches :

1. a fully connected softmax layer with $|\mathcal{A}|$ units (the policy output). Actions are selected by sampling according to the distribution defined by the policy, rather than by taking the action with the highest probability.

2. a fully connected linear layer with 1 unit (the value output).

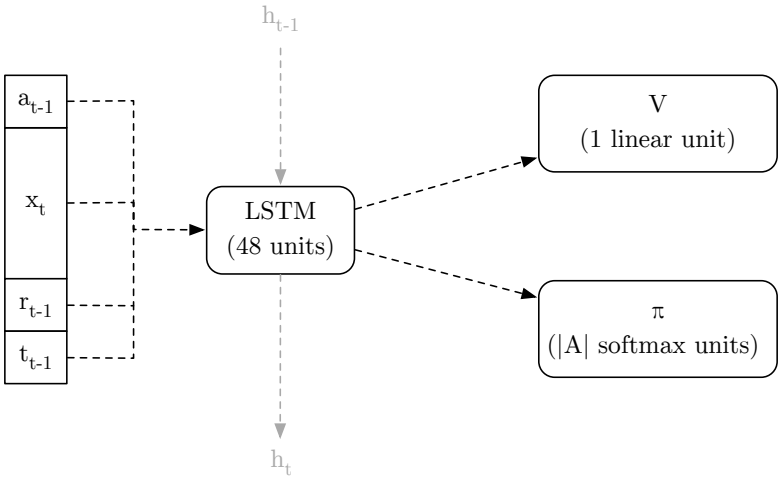


Figure 4.5: Architecture of the meta-learning A2C agent

The loss of the A2C agent is the following :

$$\mathcal{L} = \beta_v \mathcal{L}_v + \beta_p \mathcal{L}_p - \beta_e \mathcal{L}_e$$

where $\mathcal{L}_v$ is the value loss :

$$\mathcal{L}_v = (R_{t-1} - V(s_t))^2$$

$\mathcal{L}_p$ is the policy loss :

$$\mathcal{L}_p = \pi(a_t \mid s_t)(R_t - V(s_t))$$

$\mathcal{L}_e$ is the entropy regularisation :

$$\mathcal{L}_e = \sum_{a \in A} \pi(a \mid s_t) \log(\pi(a \mid s_t))$$

and $\beta_v = 0.5$, $\beta_p = 1$, $\beta_e = 0.05$. An update is performed once after every trial using Adam [15] with a learning rate of 0.001.

Unless stated, the discount factor used in all experiments is $\gamma = 0.9$.

4.3 Meta-learning dependent bandits

Let us come back to the problem stated previously as an example. This will prove useful as it is the same setting used by Wang et al. [27]. We can then compare our results with theirs to introduce the current state of the art, test our implementation and verify that their results can be reproduced. This setting has the advantage of being relatively simple so it will help us understand meta-learning before jumping to a harder problem.

The setting is the following: a dependent bandit with 2 arms of which the reward distribution is a Bernoulli distribution with the following parameters :

$$\begin{cases} P(r \mid \text{arm}_1) = p_b \\ P(r \mid \text{arm}_2) = 1 - p_b \end{cases}$$

We define a training distribution of dependent problems with $p_b \in [0.1, 0.2, 0.8, 0.9]$, creating the following set of dependent bandit problems:

Arm #1	Arm #2
0.1	0.9
0.2	0.8
0.8	0.2
0.9	0.1

The agent will play trials of 100 episodes - meaning that we choose one bandit problem and let the agent pull either one of its arms 100 times, then reset the hidden state and start again. Since the problem is stateless, the agent only receives the last action taken, the previous reward and the termination flag as input. The discount factor γ is set to 0.8 to replicate Wang et al.'s experiment [27].

Wang et al. [27] achieved creating a meta-learning agent that could learn a dependent problem as the one defined above in only a few episodes (one episode being one pull of the bandit). For some experiments, their agent outperforms the state of the art; and a look over the behaviour of their agent shows only a minimal number of episodes spent exploring before exploiting only.

Figure 4.6 shows that after some time, the agent figures out a way to yield an excellent reward over all trials: for trials using bandits with $p_b \in [0.2, 0.8]$, even if the agent pulled the best arm for each episode (which rules out exploration completely), the total reward expectation is 80, and for trials using bandits with $p_b \in [0.1, 0.9]$, the total reward expectation is 90. Scoring an average trial reward between 80 and 85 looks very close to optimal, if not fully optimal.

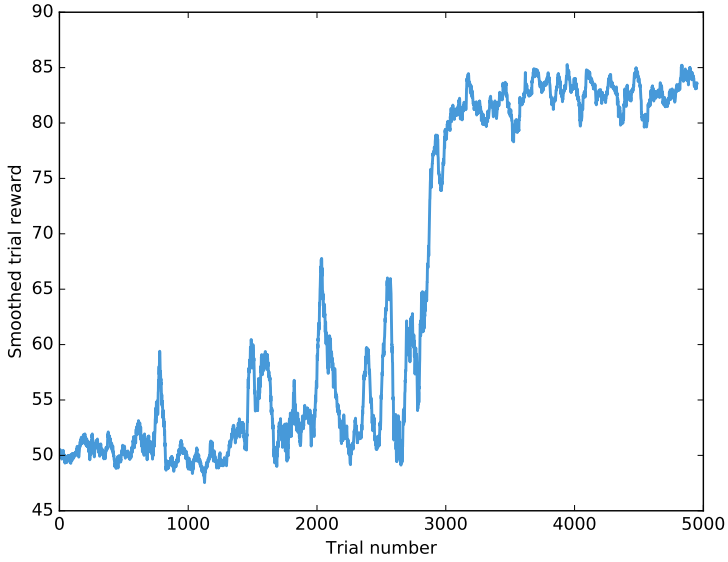
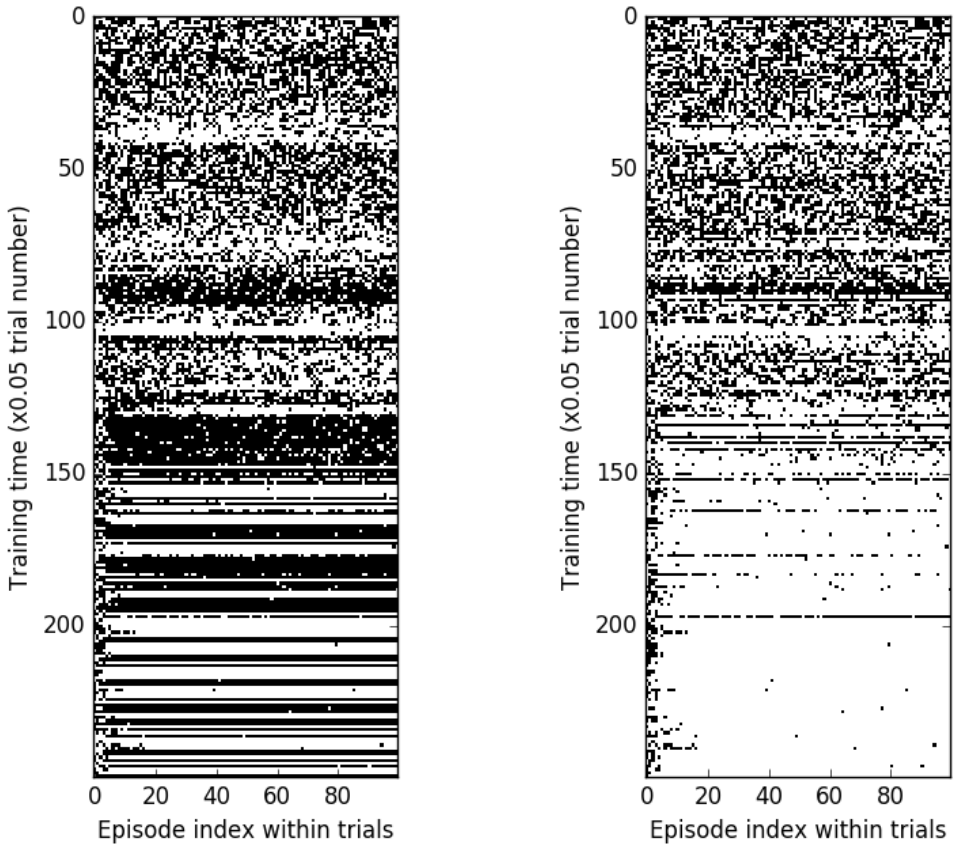


Figure 4.6: Evolution of the smoothed reward of trials during training on dependent bandit problems

If we look at the evolution of decisions taken by the agent during training on Figure 4.7, and especially on Figure 4.7b, we see that the agent evolves from playing randomly each arm to testing both arms for just a few episodes at the start of the trial to then always play what it judges is the best episode.



(a) Pulled arm

(b) Optimality of the pulled arm

Figure 4.7: Insight into the evolution of which arms get pulled and whether they are optimal during training. On the left, a black dot represents the fact that arm #1 has been pulled and a white dot represents the fact that arm #2 has been pulled. On the right, a white dot signifies that the best arm has been pulled. This figure shows that after a bit of training, the agent learns to quickly identify which arm is the optimal arm and pulls that arm only after only a few episodes.

We should emphasize on the importance of these results. If we had used the ϵ-greedy method as is, the difference in performance would be significant on several aspects:

- with a fixed ϵ, sticking to the optimal arm would have been impossible (as a random action is required from time to time), hindering performance while it can be very clear which arm is the best
- even if we chose a small ϵ to minimise performance loss in the exploitation phase, we would have increased dramatically the probability of not exploring enough at

the start of the episode

- similar reasoning can be used to argue about the number of trials over which ϵ could have been annealed from 1 to 0.

Using meta-learning allowed us to bypass all these design considerations. Furthermore, both Wang et al. [27] and Duan et al. [8] show that the strategy deployed by the meta-learning agent matches (or outperforms) the best hand-designed strategies such as UCB and Gittins.

It is worth noting however that we have only been considering known and seen bandit problems in our discussion so far. Nevertheless, we see on Figure 4.8 that the meta-learning agent can generalise to all other values of the parameter p_b with very good performance. For the performance analyis of Figure 4.8, we let the agent play 100 trials of 20 equally spaced values of p_b in the range $[0.5, 0.975]$. For each trial, the optimal arm was randomly shuffled to be either the first or second arm. We compare the average reward obtained to the optimal expected reward (the theoretical expected reward obtained by exclusively pulling the optimal arm, which is not practically possible since the agent doesn't know which arm is the optimal arm and has to explore for several episodes).

The agent performs very well on unseen values of p_b, even harder ones which are closer to 0.5, even though it has only ever seen $p_b = 0.8$ and $p_b = 0.9$.

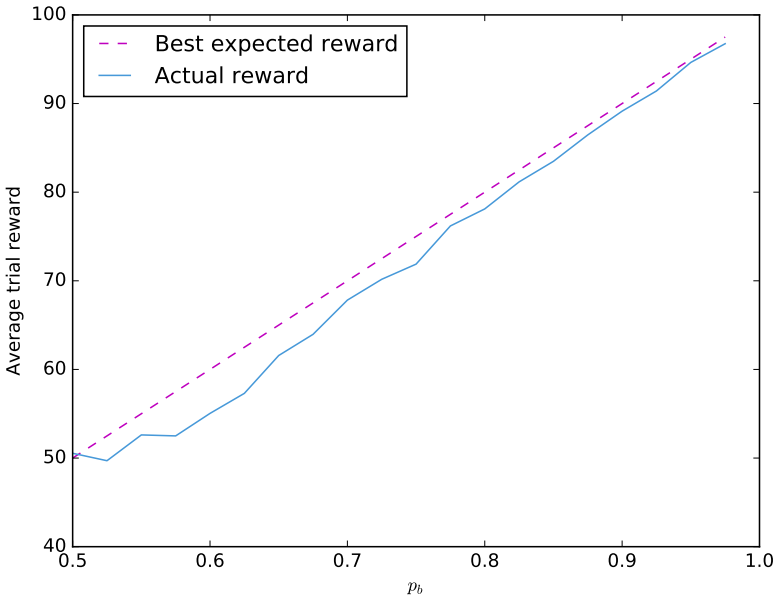


Figure 4.8: Average performance of the meta-learning agent on the whole range of the parameter p_b. The agent has only seen $p_b \in [0.8, 0.9]$. The performance is compared to the theoretical expected reward if the agent selected the optimal arm for each episode.

Summary

Now that we have laid out the stakes and goals of meta reinforcement learning, explained the training process and the architecture of the agent, and recreated state of the art

experiments to verify both our implementation and the results obtained in the literature; we will extend the application of meta learning to a new class of problem. We will derivate a distribution of tasks from the CartPole problem and analyse the performance and dynamics of meta-learning applied to this case.

Chapter 5

Knowledge gain across episodes

> "*Failure is simply the opportunity to begin again, this time more intelligently.*"
>
> **Henry Ford**

5.1 Meta-learning CartPole setting

We will now go beyond the bandit experiments presented in Wang et al. [27] and Duan et al. [8] and extend the study of the performance of meta-learning on multi-timestep episodes; and for that purpose, we will use various distributions of the CartPole problem. The CartPole environment describes a cart, rolling on a frictionless rail (see Figure 1.2). A pole is attached to the cart and has to be kept balanced above the cart for more than 195 timesteps by also keeping the cart within a defined section of the rail. The agent has two actions at its disposal: nudge the cart left or right. The observation of the state is a 4-valued vector containing the position of the cart, its velocity, the angle of the pole and the velocity of the pole at its tip. At each time step, the environment emits a reward of +1; so if the agent keeps the pole and the cart in their respective acceptable ranges for 20 timesteps, it will receive a total reward of 20.

5.1.1 Generating a distribution of CartPole problems

To generate a distribution of similar, yet different CartPole problems, we will play with two different parameters.

Permutations In standard reinforcement learning, the state observation vector is always ordered, meaning that each component of the vector always represents the same value (in CartPole for example, the first component represents the position of the cart). We can generate a set of $4! = 24$ CartPole problems where for each problem, the state observation vector is permutated with a unique permutation. This means that the agent will receive a vector of 4 values without knowing which is which.

Table 5.1: State permutations used for training and testing

(a) Training distribution

[0, 1, 2, 3]	[1, 0, 2, 3]	[2, 0, 1, 3]
[0, 1, 3, 2]	[1, 0, 3, 2]	[2, 0, 3, 1]
[0, 2, 1, 3]	[1, 2, 0, 3]	[2, 1, 0, 3]
[0, 2, 3, 1]	[1, 2, 3, 0]	[3, 0, 1, 2]
[0, 3, 1, 2]	[1, 3, 0, 2]	[3, 0, 2, 1]
[0, 3, 2, 1]	[1, 3, 2, 0]	[3, 1, 0, 2]

(b) Testing distribution

[2, 1, 3, 0]
[2, 3, 0, 1]
[2, 3, 1, 0]
[3, 1, 2, 0]
[3, 2, 0, 1]
[3, 2, 1, 0]

Actions inversion We will also introduce the inversion of the agent's actions. Just like the input vector, the policy output is ordered (the first component of the policy corresponds to the first action which is always the same). We will study how inverting actions affects training : instead of always associating the first component of the policy vector with going left, we will duplicate each of the 24 problems stated above by inverting the left and the right action. This adds up to a distribution of 48 CartPole problems.

Figure 5.1 shows how the single-timestep framework presented in Figure 4.2 adapts to episodes with multiple timesteps.

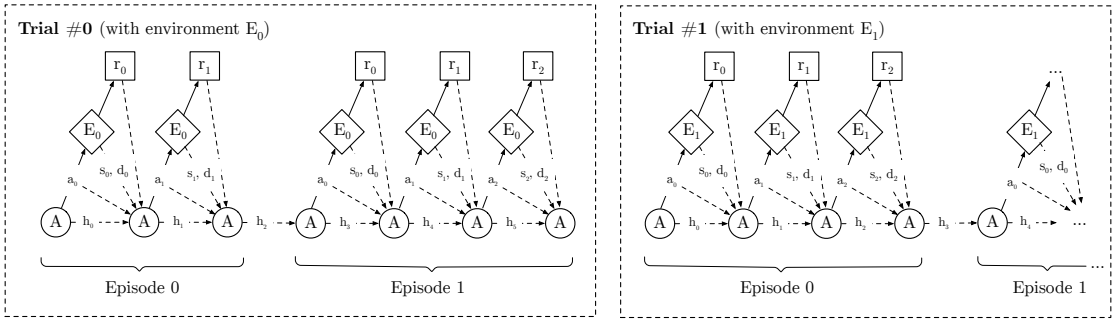


Figure 5.1: Meta-learning for multi-timestep episodes. A is the agent, E_0 is the problem selected from the distribution for the duration of a trial, s_t, is the state of the environment, r_t, the reward, a_t is the chosen action, d_t the terminal state and h_t is the hidden state at timestep t. Note that the same environment is used throughout all episodes of the same trial.

5.2 Performance gain when playing multiple episodes

We will first and foremost analyse how a reinforcement learning agent performs when it is confronted to playing a single episode of one of the CartPole problems. We will train the agent on a distribution of 18 permutations (shown in Table 5.1), at first without inverting the agent's action.

As a control experiment, we first need to check whether the agent can learn to discover which permutation of the state has been selected and to keep the pole balanced within one single episode. Surprisingly, as shown in Figure 5.2, the agent quickly man-

ages to reach a very good performance level, only failing a small percentage of the time.

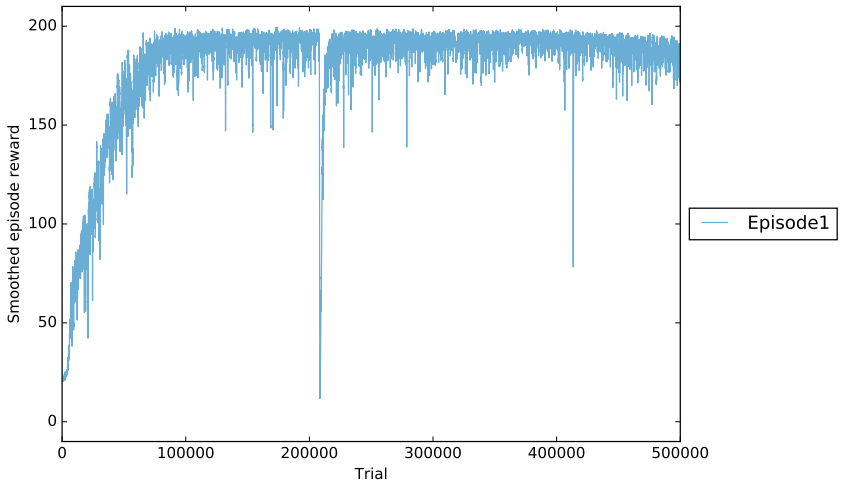


Figure 5.2: Training of an agent on trials of 1 episode. The curve shows a moving average over 200 trials.

When the agent is trained on trials of 2 episodes, we expect that its performance will improve (however good it already is). Looking at the graph of Figure 5.3, we can only deduce two things :

1. the performance of the first episode of every trial drops significantly compared to trials of one episode;

2. the performance of the second episode of every trial matches closely the performance of single-episode trials.

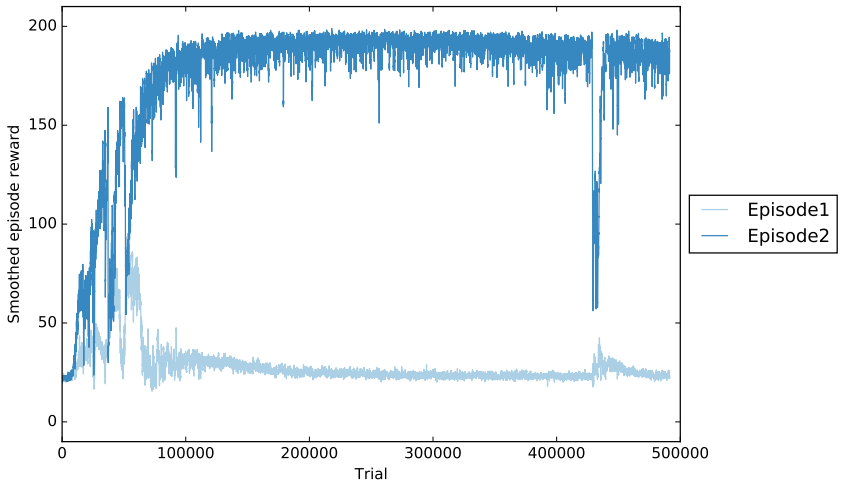
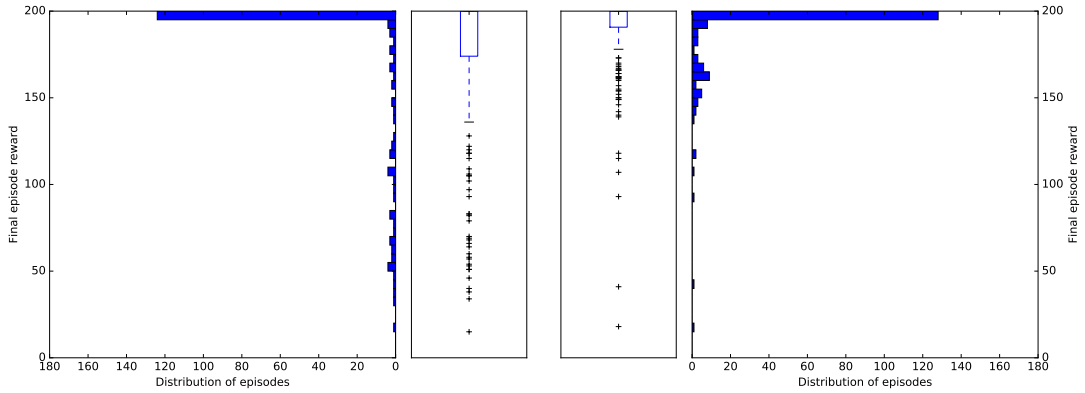


Figure 5.3: Training of an agent on trials of 2 episodes

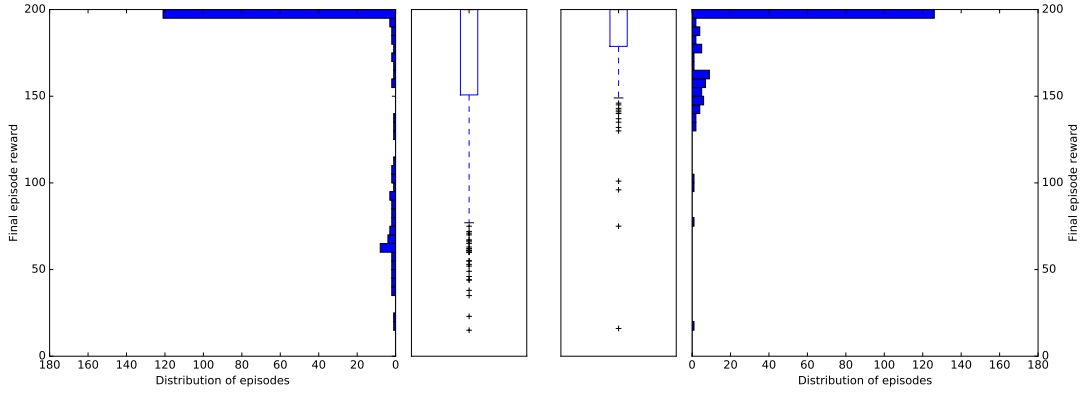
The first of these considerations will be addressed in chapter 6. Let us analyse the second one in more detail as the average reward graph doesn't give us enough insight in the final reward distributions.

Let us compare the final reward distribution for the only episode of single-episode trials and for the second episode of dual-episode trials. For now, we are not interested in the reward distribution of the first episode in dual-episode trials as we want to see whether or not the agent managed to gain knowledge during that first episode (potentially hurting its first episode reward) to improve its performance for the second episode.



(a) Trials of 1 episode (b) Trials of 2 episodes

Figure 5.4: Distributions of the total reward accumulated during the last episode of trials. 180 trials are played both in the case of single-episode trials and dual-episode trials. This shows the distribution of rewards obtained after playing 10 times each permutation of the **training** set. No action inversion is performed.



(a) Trials of 1 episode (b) Trials of 2 episodes

Figure 5.5: Distributions of the total reward accumulated during the last episode of trials. This shows the distribution of rewards obtained after playing 30 times each permutation of the **test** set. No action inversion is performed.

To compare the two settings, we fix the agent's learnable parameters and make it play 180 trials (resetting its hidden state before every trial). Every CartPole problem

in the distribution is played the same amount of times. Fixing the agent's learnable parameters means that we do not change any of the weights of the neural network that represents the meta-learning agent. We should perhaps emphasize that this means we do not train the agent anymore; and that we expect the agent to learn **after training** still, even though its weights are fixed.

Figure 5.4 shows both a histogram and a boxplot describing the final reward distributions in the single and dual episode situations. Although the difference is slight, and seeing that the number of successes (a final reward of 195 or more, corresponding to the uppermost bar) is almost exactly the same, there is definitely a difference in when the failures occur. For the single episode setting, all failures are distributed in a relatively uniform way spanning from 50 to 195; whereas they cluster slightly above 150 in the dual episode setting. The boxplots clearly show a more compact distribution (and so a generally higher reward) in the dual episode setting.

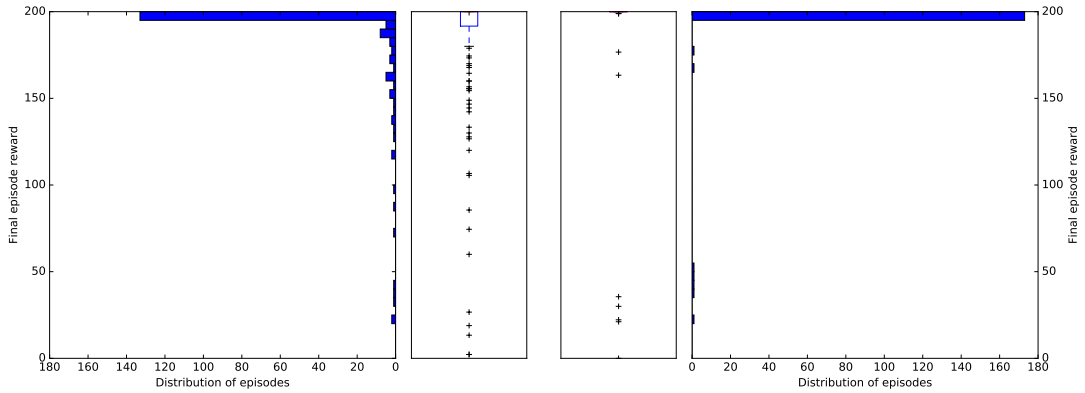
The agent seems to learn from the first episode to improve its performance in the second. This is even clearer when we run the same experiment on unseen permutations. As shown on Figure 5.5, where the agent played every permutation of the test set for 30 trials, playing 180 trials in total to allow for comparison to the previous experiment, the agent playing single-episode trials fails more often with a reward around 50. The dual-episode setting shows again a capability to learn from the first episode to improve performance in the second. We will later analyse performance with longer trials containing more episodes.

Even more impressingly, if we add the inversion of the left and right actions with a probability of 0.5 at the start of every trial, this effect is tremendously magnified. For seen permutations (Figure 5.6), there are about 40 more successes (out of 180) when playing two episodes than when playing only one; for unseen permutations, this number climbs to about 70 out of 180. The single-episode agent succeeds shy of 100 trials out of 180 where the dual-episode agent comes close to 170 out of 180.

The reader might have noticed that the absolute performance for the agent playing dual-episode trials is better when the problem is harder (with inverted actions) as it succeeds close to 170 trials out of 180 whereas for the simpler problem (without inverted actions), it only succeeds for around 120 trials out of 180. We can only speculate as to why this result occurs. We suggest this might be due to the fact that the harder problem is impossible to be learned in one epiosde, thus forcing the agent to really become a meta-learning agent, making it perform extremely well during the second episode; whereas the simple problem might only be learned like a standard reinforcement learning problem.

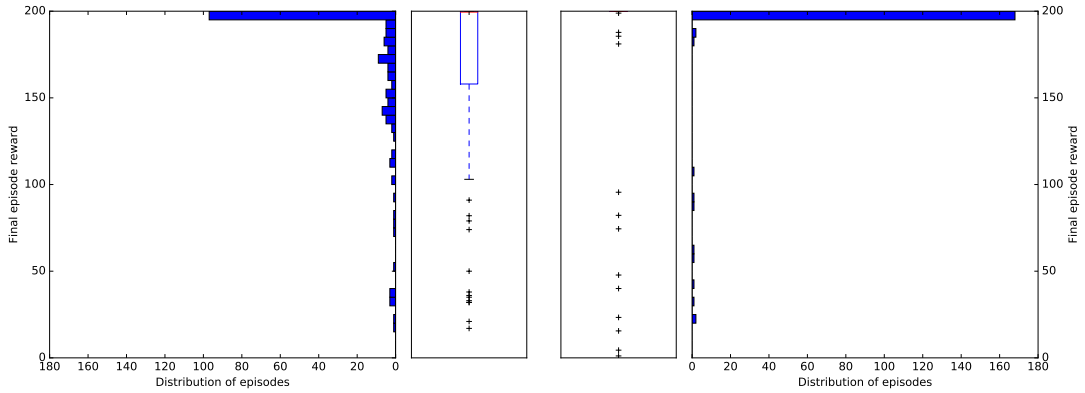
There are several surprises in these results. The first one is that a single episode appears to be enough for an agent to discover how the observation has been permutated in time to take action so that the pole stays balanced (at least for a majority of trials), but playing a second episode improves the performance of the agent (although at the cost of the reward of the first episode – this will be analysed in chapter 6).

The second one seems to be that in the setting of dual-episode trials, the harder the problem is, the higher the number of successes will be. Indeed, the performance of agents



(a) Trials of 1 episode (b) Trials of 2 episodes

Figure 5.6: Distributions of the total reward accumulated during the last episode of trials. This shows the distribution of rewards obtained after playing each permutation of the **training** set 5 times with action inversion and 5 times without action inversion.



(a) Trials of 1 episode (b) Trials of 2 episodes

Figure 5.7: Distributions of the total reward accumulated during the last episode of trials. This shows the distribution of rewards obtained after playing each permutation of the **test** set 5 times with action inversion and 5 times without action inversion.

playing problems with inverted actions is significantly higher than the performance of agents playing problems without.

Conclusion

We have seen that applying meta-learning to a problem such as CartPole yields interesting results on several aspects. The first one is that we can indeed see information being passed from episode to episode to improve performance (in essence, learning is happening), but at the same some behaviours deserve looking into:

- the reward per episode seems to change quite a lot as a function of the number of episodes per trial and shows curious dynamics
- harder problems seem to make meta-learning more effective than simple problems; not only is the difference between non-meta-learning and meta learning larger, but the absolute performance of the meta-learning agent on a harder problem is better than the absolute performance of the meta-learning agent on a simpler problem.

We will now look into more details as to why these behaviours occur.

Chapter 6

Episode-wise reward dynamics

> "*The Panic Monster is dormant most of the time, but he suddenly wakes up when a deadline gets too close or when there's danger of public embarrassment, a career disaster, or some other scary consequence.*"
>
> **Tim Urban**, on procrastination

Let us now look at why the reward of the first episode in a dual-episode trial drops to random policy level when we know the agent is able to receive an almost perfect reward in a single-episode trial. To understand why, we will keep the same setting as previously, but we will vary the length (in number of episodes) of the trials and the discount factor γ.

6.1 Inherent laziness

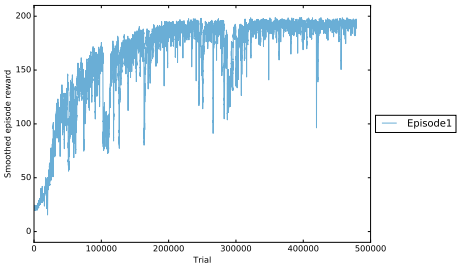
There is a recurrent pattern visible in Figure 6.1. Indeed, no matter the number of episodes in each trial, the last one will always reach optimal or near-optimal performance while all previous episodes will receive a reward which corresponds to following a random policy.

This is confirmed by the plots of Figure 6.2 which show the rewards obtained by playing several trials of 2 and 5 episodes (after having trained on trials of corresponding length) over all the permutations of the training set. Only the last episode of the trial will succeed or almost succeed.

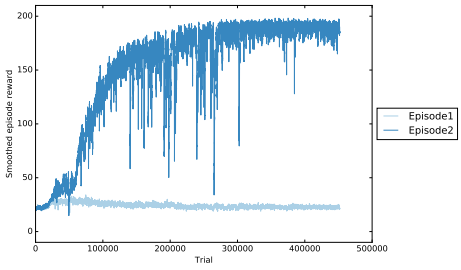
We will explain why the reward structure of the CartPole problem (+1 at every time step) inevitably produces such an undesired behaviour and look for ways to counter it.

6.1.1 Problems with a 1-per-timestep reward

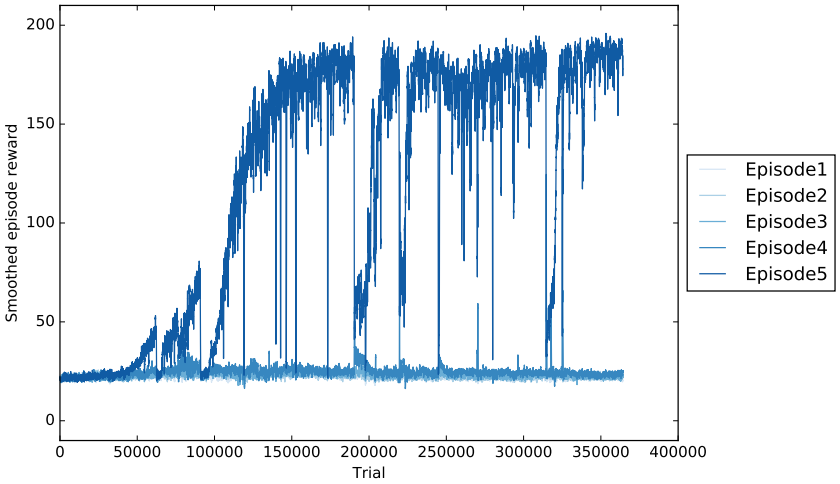
This behaviour is linked to the dynamics of the expected future reward, the chosen discount factor and the reward structure of the environment. In our case, the environment gives a constant +1 reward at every timestep of a CartPole episode. If we choose a discount factor $\gamma = 0.9$, the discounted future reward at timesteps which are more than 50 timesteps away from the end of the episode will be almost exactly equal (see Figure 6.3).



(a) Training over 1 episode

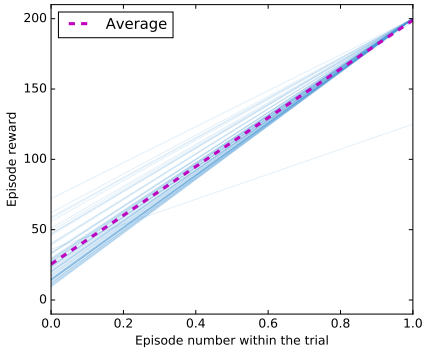


(b) Training over 2 episodes

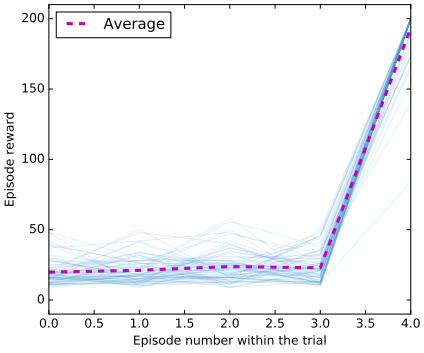


(c) Training over 5 episodes

Figure 6.1: Episode-wise reward evolution when training on trials of different lengths



(a) 2 episodes



(b) 5 episodes

Figure 6.2: Testing performance of agents trained on trials of different lengths. Several runs are displayed in blue on the same plot, and their average is shown as a red dashed line.

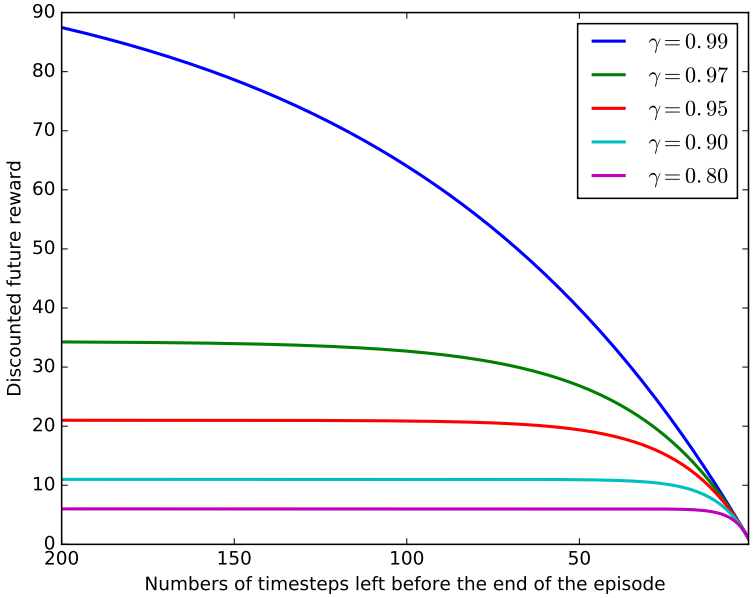


Figure 6.3: Value of the discounted expected reward R_t as a function of the number of timesteps left before the end of an episode in which a reward of $+1$ is given at every timestep; shown for different values of γ.

This means that if an action that defines whether the episode ends or continues at time t has to be taken before time $t - 50$, there is no way to link the end of the episode with the action that was wrongly taken. This can be turned around to say that actions do not really matter as long as the end of the constant reward stream is more than 50 timesteps away for $\gamma = 0.9$. In our case, this explains why all episodes prior to the last one achieve low reward. Because in every case, if the last episode is longer than 50 timesteps, the end of the reward stream is too far for the second-to-last episode to take it into account (if $\gamma = 0.9$).

This should make the reader wonder why the last episode lasts more for more than 50 ticks. As stated above, in episodes that are not the last one, a failure doesn't lead to a loss in expected reward; at the contrary, the reward stream keeps going after the end of the episode. For the last episode however, the agent might notice that the current state of the environment will lead to the end of the episode **and** of the trial, so the end of the reward stream if it doesn't take any action.

This reasoning assumes that the agent knows when it is playing the last episode; and the results show that, in some way or another, the agent has learned to count either the termination flags, or the number of timesteps needed to get to the last episode. This will be analysed further in section 6.2.2.

6.2 Encouraging the agent to succeed

There are two main parameters that directly affect this lazy behaviour : the number of episodes in a trial and the discount factor. In the following experiments, we will select

Table 6.1: State permutations used for training and testing

(a) Training distribution			(b) Testing distribution
[0, 1, 2, 3]	[0, 1, 3, 2]	[1, 0, 2, 3]	[1, 0, 3, 2]

a small subset of the training set previously used to accelerate training. This subset is shown in table 6.1.

6.2.1 Tuning the discount factor

If we want to enforce good performance for non-terminal episodes, selecting higher values for the discount factor is the first solution that comes to mind. Unfortunately, besides the fact that this only enforces k last episodes to have a high reward (instead of trying to maximise performance as soon as possible), this solution doesn't seem to work, as Figure 6.4 shows.

In this experiment, we selected 3 training permutations (see Table 6.1) and trained our agent for 5 episodes. Several trainings were run with different values for the discount factor γ.

Even though the rewards of previous episodes clearly climb higher as we increase the discount factor, they all eventually decline to random policy-like reward. Once the discount factor reaches values where the second-to-last episode should be involved in having a high reward (around 0.97 and above), training diverges and the agent never manages to reach high scores.

High values of γ directly lead to higher returns, and this has a double effect on the training:

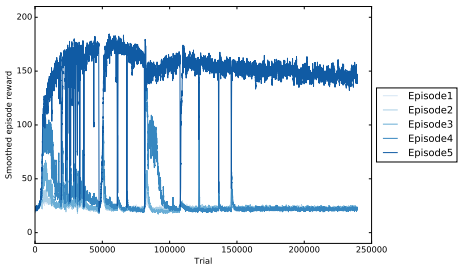
1. gradients will be scaled by a bigger factor, and as stated previously, bigger steps might make the training generally less stable and decrease the probability of reaching a good optimum
2. the variance of returns will also increase, making the training targets for the network less stable.

6.2.2 Training on more episodes

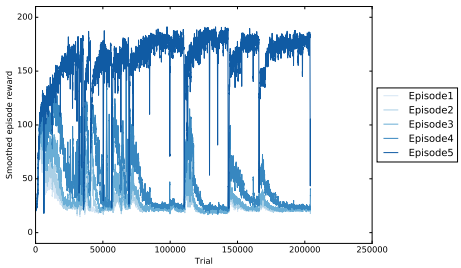
As said before, we made the hypothesis that the agent knew when it was playing the last episode. What happens if we increase the number of episodes in a trial? We trained an agent on the permutations shown in table 6.1 on trials of 2, 5, 10 and 15 episodes.

If we look at the training graphs for this experiment (Figure 6.5), we can identify two main trends:

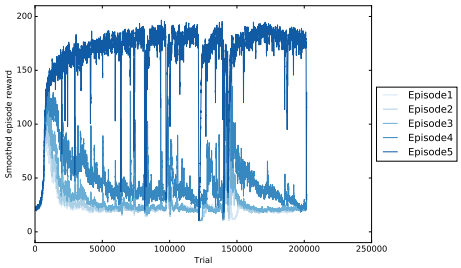
- the reward of the last episode doesn't reach optimal performance as the number of episodes increases;
- the almost last episodes do not fail completely as the number of episodes increases; in fact, in the case of 15 episodes, the penultimate episode seems to climb about as much as the last episode.



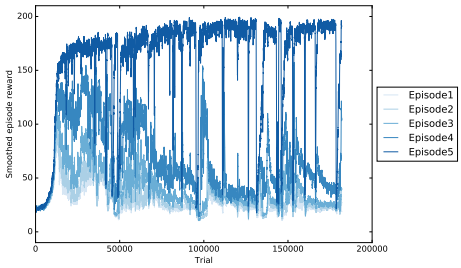
(a) $\gamma = 0.82$



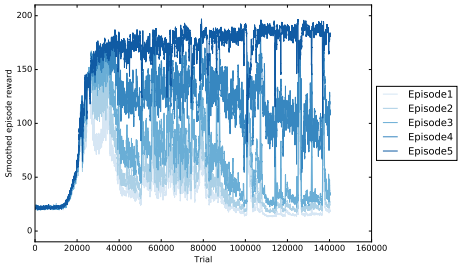
(b) $\gamma = 0.85$



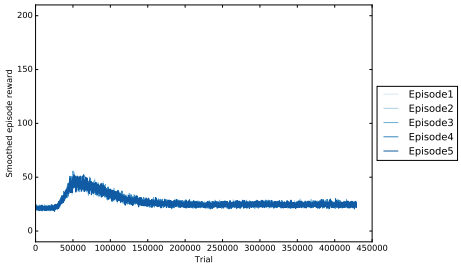
(c) $\gamma = 0.88$



(d) $\gamma = 0.91$

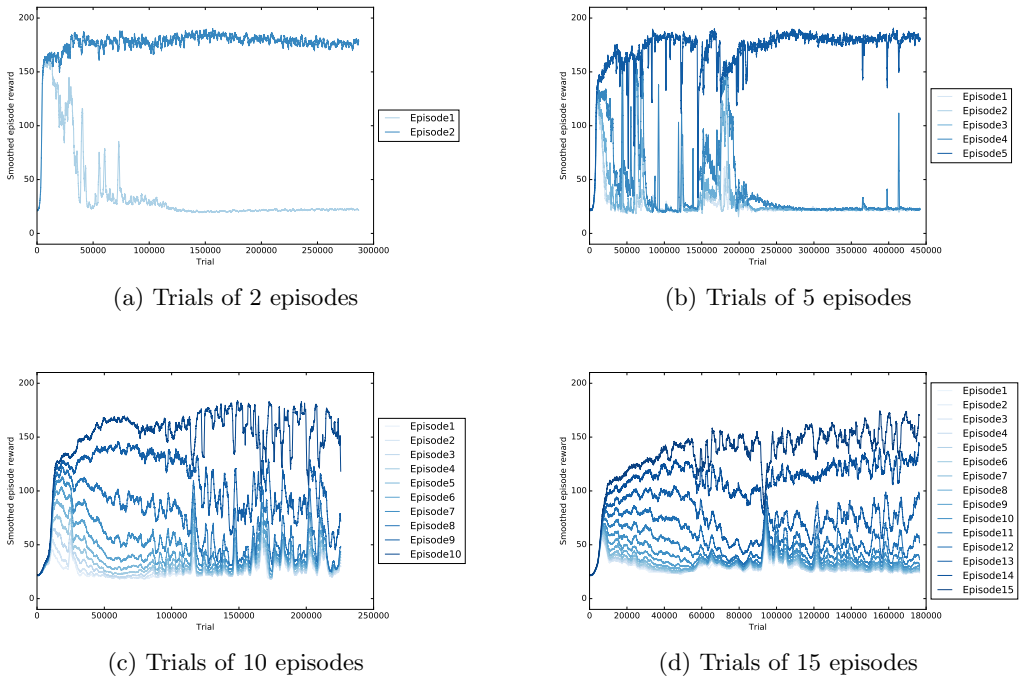


(e) $\gamma = 0.94$



(f) $\gamma = 0.97$

Figure 6.4: Variation of the episode-wise reward evolution in trials of 5 episodes when tuning the discount factor γ.



(a) Trials of 2 episodes

(b) Trials of 5 episodes

(c) Trials of 10 episodes

(d) Trials of 15 episodes

Figure 6.5: Episode-wise reward during training with trials of different lengths

From what we can see on Figure 6.5, it looks like a gain in performance for the last episode always happens at the expense of the reward of previous episodes. We make the hypothesis that the agent "counts" the **timesteps** it takes to reach the last episode. This yields good performance when the number of timesteps prior to the last episode is stable, meaning that all previous episodes have either failed quickly or succeeded. Since the former solution is the simplest to achieve, and it is supported by the drop of reward in the first episodes shown on Figure 6.5, we believe that is what the agent does.

6.3 Testing performance after the training horizon

So far, we have tested our agent only within the limits of its training. What happens if we leave it running for more episodes than what it has been trained for?

We have trained an agent on the 3 permutations listed in table 6.1 without inverted actions, over trials of 5 episodes. We can test the agent on trials of more episodes - meaning that the hidden state will be carried on until the end of the 20th episode. Figure 6.6 shows the result of this experiment.

There is no surprise in the rewards obtained for episodes 0 to 4. The pattern observed from episode 5 onwards, however, deserves commenting. The first, obvious comment to make is that the average reward of the 5th episode is dramatically low. There still seems to be mild success after the 5th episode until all runs level off to what looks like a random policy level of reward.

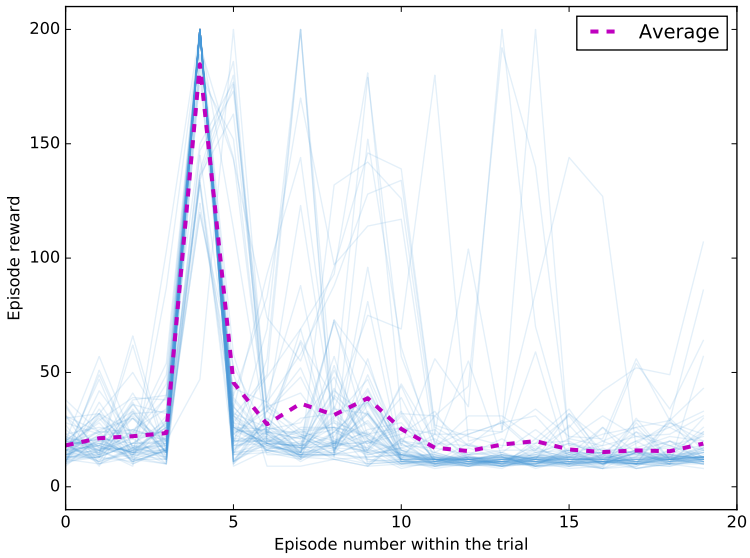


Figure 6.6: Testing performance over 20 episodes of an agent trained on trials of 5 episodes only. Several runs are shown in blue, their average is plotted in red dashes.

6.3.1 Restoring recurrent weights

One solution to maintain a high reward after the training horizon would be to restore the recurrent weights to their state prior to the last episode of the training horizon at the beginning of each episode. As Figure 6.7 shows, this solution works.

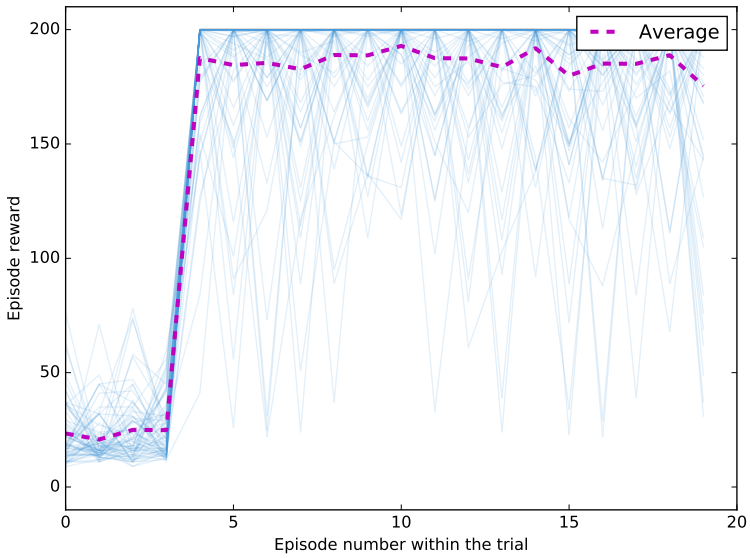


Figure 6.7: Testing performance over 20 episodes of an agent trained on trials of 5 episodes only. The recurrent weights are saved prior to the 5th episode and restored prior to each episode from the 6th episode onwards. Several runs are shown in blue, their average is plotted in red dashes.

This result is not a surprise, and this solution should rather be called a workaround. Indeed, restoring the weights at the beginning of each episode stops any learning to occur after the preset training horizon. Although the agent is supposed to have reached an optimal policy at the end of the training horizon, we cannot assume there is nothing left to learn as the agent navigates through later episodes.

Some remarkable feats occur when we we only increase the number of episodes the agent is trained on (see Figure 6.8). Although it seems the agent doesn't manage to reach the peak performance of the agent trained on 5 episodes, a few interesting things to notice are:

- the reward in the first 5 to 7 episodes seems to be slightly superior to the reward in the first episodes of the agent trained on 5 episodes
- some of the 9th episodes are successful - and unlike anything we have seen so far, they are the penultimate episode – not the last one.
- most importantly, the average reward beyond the horizon is significantly higher than what we have seen for the agent trained on 5 episodes.

These findings seem to support the hypothesis we made earlier that the agent tries to figure out whether or not it is playing the last episode. Pushing the horizon further seems to confuse the agent's count, forcing it to distribute its high performance on several of the last episodes instead of only the last one.

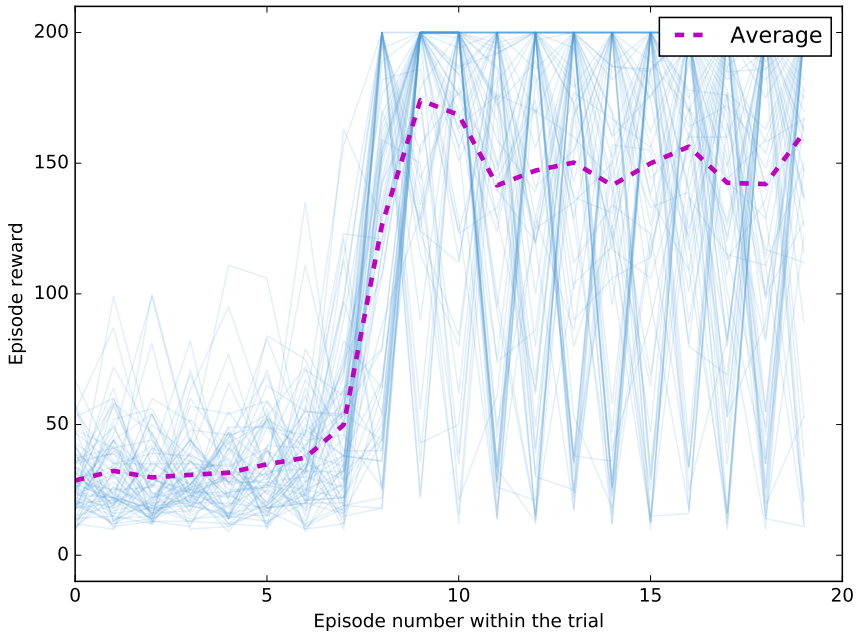


Figure 6.8: Testing performance over 20 episodes of an agent trained on trials of 10 episodes only. Several runs are shown in blue, their average is plotted in red dashes.

Conclusion

We have seen that for a small number of episodes per trial, only the last episode manages to reach an optimal performance at the expense of all previous episodes. This is due to the fact that CartPole generates a reward of +1 at every timestep and our setup doesn't allow the agent to know that an episode has finished too early (and that it is a bad thing). All it ever sees is a constant stream of positive reward until it reaches the last episode where the stream is at risk of ending if the agent doesn't balance the pole for as long as it can.

We have tried tuning the discount factor, hoping that higher values would encourage the agent to perform well in non-terminal episodes, but even though a trend showed that this idea could have worked (at least to help the agent win the last k episodes, which is not as good as winning as soon as it can), training would diverge before we reached values for γ where we would see several terminal episodes reach an optimal reward.

Increasing the number of episodes per trial gave more interesing results; after a critical threshold, it looks like several terminal episodes reach high rewards, and another benefit of training over longer trials is the better performance of agents beyond their training horizon (playing more episodes than what they have been trained for). An agent trained on trials of 5 episodes would quickly return to a random policy-like reward after playing 5 episodes whereas agents trained for 10 episodes managed to keep a reward that was significantly higher than random – although still worse than optimal performance.

Chapter 7

Injecting an informational reward

> "*The ideal of behaviourism is to eliminate coercion: to apply controls by changing the environment in such a way as to reinforce the kind of behaviour that benefits everyone.*"
>
> **Burrhus Frederic Skinner**

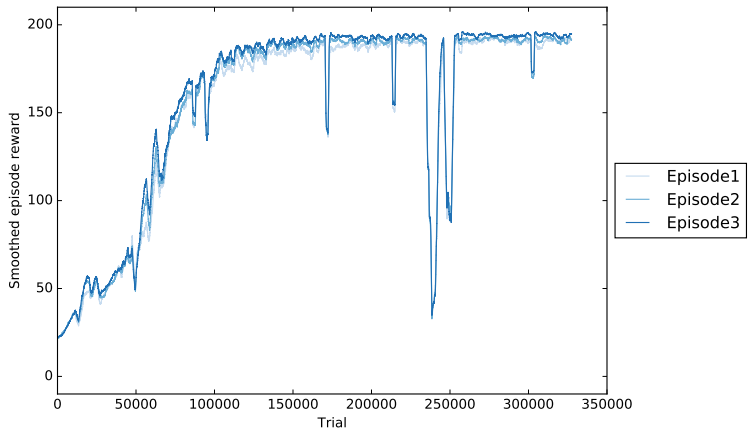
We have described in section 6.1.1 why a 1-per-timestep reward structure led to an undesired behaviour of laziness, in which the agent only maximised its reward on the last episode of its training trials.

We introduce a simple, yet effective way to allow problems with similar reward structures to be meta-learned. In hand-designed strategies aimed to solve such problems, the designer holds the implicit knowledge that a short episode is a bad thing, and treats the reward streams of each episode separately. In meta-learning, just like we feed back a termination flag, which is not an environment-level value but rather a "process-level" value (where the process is the training process we replicate in a trial); we can introduce a process-level reward to give information to the agent performing the trial.

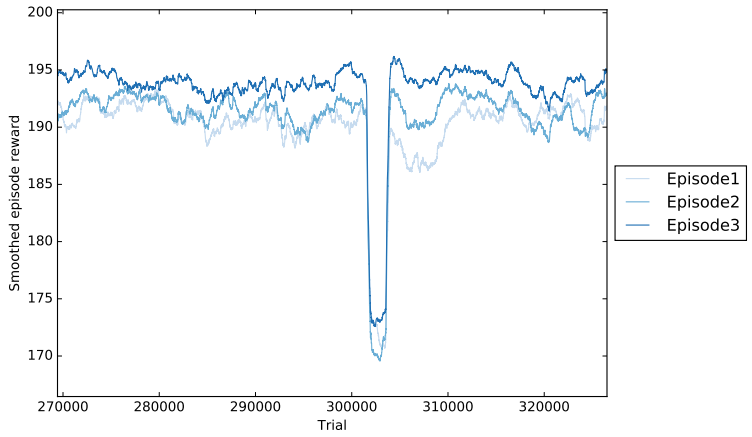
We propose to add a small negative reward at the end of each episode, interrupting the continuous reward stream to force the agent to perform well for each of the episodes in the trial.

Our agent is trained on the training set of 18 permutations listed on table 5.1, on trials of 3 episodes. The reward for the last timestep of each episode is set to -10. Figure 7.1 shows how the episode-wise reward curve evolves during training. We see that all three episodes reach a high reward, but more importantly, performance increases for the second and third episode.

This is clearer on Figure 7.2, where we see that episodes 2 and 3 get a higher average than episode 1. We also see however that letting the agent play beyond its training horizon leads to a generally lower average reward. A second surprise is the fact that the average reward for the third episode when testing on unseen permutations is below the average reward of second episodes. It is however still above the average reward for first episodes. We suspect this variance in the results is due to the closeness in performance of all three episodes.

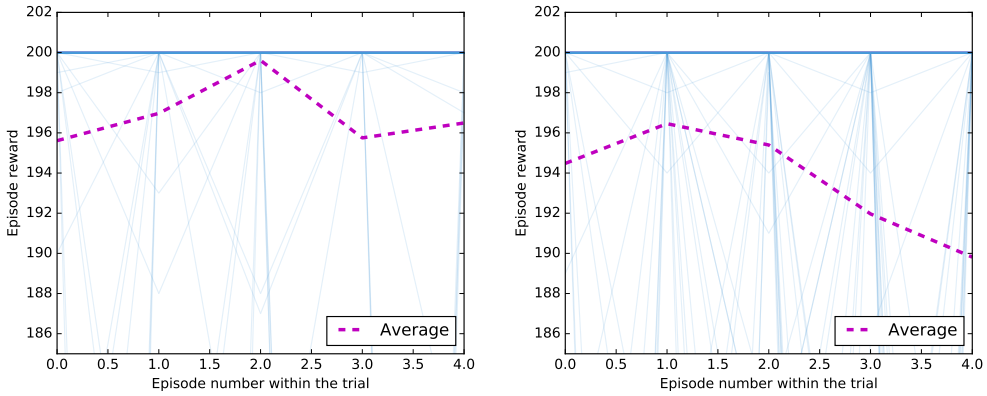


(a) Full training graph



(b) Close-up of the last trials

Figure 7.1: Episode-wise reward evolution during the training of an agent playing trials of 3 episodes with a small negative reward at the end of each episode.



(a) Training permutations

(b) Testing permutations

Figure 7.2: Testing performance of the agent trained with the reward injection. Several runs are displayed in blue on the same plot, and their average is shown as a red dashed line.

Even though this is a success, we think there might be a better solution for problems with a similar reward structure. For example, Wang et al. [27] present an experiment (bandits with dependent arms II) where the optimal long-term strategy is to start by playing a suboptimal action voluntarily to gain information that will then allow the agent to play optimal moves in future episodes. The setting of our experiment doesn't lend itself to the obligation of abandoning short-term reward for a higher long-term reward, but adapting it to make room for that type of experiment could be an interesting lead to follow in future work.

We are aware that the results shown so far could seem like they miss one of the key promises of meta-learning that were made in the introduction : the ability to perform tasks that are fundamentally different. Indeed, one could argue that meta-learning is not really needed in the setting presented in this chapter as the average reward for the first episode is already almost optimal. This is why we propose to test the exact same agent that was used in the experiments of Figures 7.1 and 7.2 for a different CartPole game it has actually never been trained to play by inverting one parameter : the reward. For this experiment, we start giving the agent a reward of +1 at every timestep only from the moment at which the environment considers it has failed the episode, until the end of the 200-timesteps episode. To our surprise, and perhaps we should emphasize this : **without having been retrained**, the agent learns to fail as quickly as it can after the first episode. Figure 7.3 shows this result.

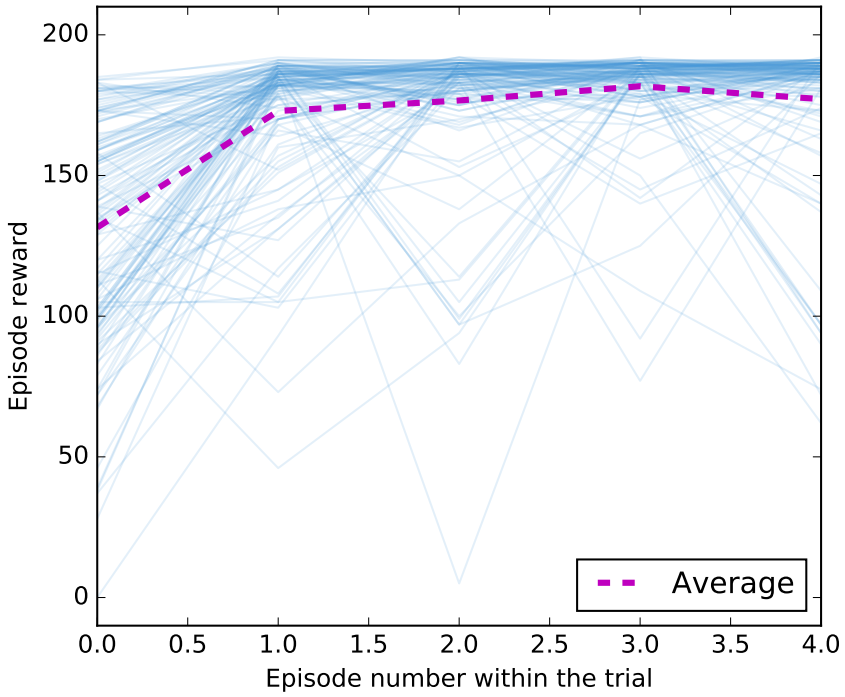


Figure 7.3: Reward per episode for an agent trained to balance the pole, tested on a task where a reward is generated as soon as it fails at doing what it was trained to do. Each different run is shown in light blue; the average of all runs is shown as a dashed red line.

Summary

In this section, we have proposed and successfully tested injecting an informative process-level negative reward at the end of each episode, indicating to the outer algorithm that it should try to keep each episode running for as long as possible. We trained an agent using this injected reward on trials of 3 episodes and found that it performed well on each episode, but that performance generally increased between episodes.

We also tested this agent on a completely new problem, where it was implicitly asked to fail at what it had been trained for as quick as possible. The results showed that the agent quickly learned to fail after the first episode, increasing dramatically its average reward over following episodes.

Chapter 8

Conclusions

> "*By far the greatest danger of artificial intelligence is that people conclude too early that they understand it.*"
>
> **Eliezer Yudkowsky**

After having reviewed the foundations of artificial neural networks and reinforcement learning, we went through the reasons why meta-learning is interesting and useful: not only it allows one to avoid choosing, designing or parametrising complex task-related strategies to accelerate training, but most importantly, it allows for agents to learn highly efficiently problems that have the same structure and to generalise over the parameters of the structure – bluntly put, one can reuse an agent without retraining it, as long as the problem is similar and the meta-learning agent has seen enough variation in the training distribution.

We have reproduced one experiment originally setup by Wang et al. [27] and Duan et al. [8] which consisted in meta-learning dependent 2-armed bandit problems. While we achieved the same results as the two seminal papers cited above, we extended the results discussion to the dynamics of meta-learning such a problem, showing how the meta-learning agent develops its strategy to learn a bandit problem faster and faster. We also studied its generalisation capability for the whole range of parameters of this problem.

Meta-learning was extended to a new set of experiments based on the CartPole environment. We designed a distribution of CartPole problems by shuffling the state observation received by the agent (without informing the agent of the mapping between values and their meaning), and by permutating the agent's actions randomly at the start of training trials.

We have shown that even though, surprisingly, the agent was able to learn to discover which problem it was playing in one episode, meaning that:

1. it was able to make sense of an unordered set of values in input;
2. it was able to learn the consequences of its actions;
3. it could perform 1) and 2) in time to be able to balance the pole for a long enough time to succeed the episode.

Although the performance of the agent was near optimal, we found that letting the agent play for at least two episodes increased its performance as it was able to learn about the environment and take consequential action in later episodes to improve its success rate. Quite surprisingly also, the meta-learning agent performed better in environments which were more difficult, and the difference between single-episode and multi-episode trials increased as the problems grew harder and both were tested on previously unseen problems.

There are still parameters to choose manually when designing a meta-learning agent, and their tuning provides for some interesting dynamics in the performance of a trained agent. We have studied how the discount factor and the number of episodes per trial played an important role in the evolution of episode-wise reward during the training of a meta-learning agent. We have also tried to understand how a meta-learning agent handled playing more episodes than what it had been trained for.

This deep look into the workings of our agent and its reaction to different hyperparameters helped us understand an inherent flaw in the reward structure of the CartPole problem forbidding the agent to perform at its best for as many episodes as possible, as soon as it could. Indeed, a continuous stream of rewards of +1 at each timestep drowned the information of "lost episodes" to the meta-learning agent.

This discovery led us to propose a simple way to inject an informative reward at the end of each episode to allow the agent to play at its best performance as soon as it could, learning and passing information on to following episodes so that they could in turn perform even better. After having successfully trained an agent to balance the pole in CartPole with this informative reward, we tested it on the novel, unseen task of failing at what it had been trained to do by starting to give it positive reward only after its failure. To our surprise, the agent managed to learn to fail within the first episode of its trials, later taking action to fail its episodes as quickly as it could. This new problem was sucessfully learned only within **one** episode without having to be retrained.

8.1 Future work

There is still much to explore in meta-learning. For instance, could a meta-learning agent learn to play problems generated from different environment? What would be the similarity constraints for it to work? If the agent does manage to learn to perform well in different environments, could it then learn to play in a totally unseen environment? Once again, in that case, what would be the requirements in terms of similarity to ensure success?

Furthermore, could environments with fundamental differences (different dimensionalities of the state and action sets) be learned by the same agent?

We are very confident in the multi-tasking capabilities of the meta-learning agent, and its ability to add an interpretation level between itself and the state, both input-wise (when it receives observations) and output-wise (when it performs actions); and hope that there will be a strong interest in researching the limits of its performance.

We suspect that adapting the architecture presented by Wang et al. to allow the outer algorithm to access more information that is implicitly or explicitly available to strategy designers (e.g. have knowledge of total episode rewards instead of timestep-only rewards) could lead to an overall better learning algorithm.

8.2 Source code

The source code for our agent, and also the code used to run all the experiments of this work has written in Python 3 [1] using Tensorflow [2]. It is available at the following url: `https://github.com/fhennecker/meta-reinforcement-learning`.

Bibliography

[1] Python software foundation. python language reference, version 3.5.2. available at `http://www.python.org`.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

[4] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Artificial neural networks. chapter Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems, pages 81–93. IEEE Press, Piscataway, NJ, USA, 1990.

[5] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Trans. Neur. Netw.*, 5(2):157–166, March 1994.

[6] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[7] Thomas Degris, Patrick M. Pilarski, and Richard S. Sutton. Model-free reinforcement learning with continuous action in practice. In *American Control Conference, ACC 2012, Montreal, QC, Canada, June 27-29, 2012*, pages 2177–2182, 2012.

[8] Yan Duan, John Schulman, Xi Chen, Peter L. Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl^2: Fast reinforcement learning via slow reinforcement learning. *CoRR*, abs/1611.02779, 2016.

[9] Author(s) J. C. Gittins and J. C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, pages 148–177, 1979.

[10] S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Diploma thesis, Institut für Informatik, Lehrstuhl Prof. Brauer, Technische Universität München, 1991.

[11] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[12] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[13] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991.

[14] Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] Yann Lecun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 5 2015.

[17] V. Mnih, A. Puigdomenech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv preprint arXiv:1602.01783*, 2016.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[19] 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, February 2015.

[20] Michael C. Mozer. Backpropagation. chapter A Focused Backpropagation Algorithm for Temporal Pattern Recognition, pages 137–169. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[21] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. Omnipress, 2010.

[22] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems. Technical report, 1994.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[24] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[25] W.R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3–4):285–294, 1933.

[26] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[27] Jane X. Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z. Leibo, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *CoRR*, abs/1611.05763, 2016.

[28] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.