Using multistability to solve fading memory problems in reinforcement learning

Auteur : De Geeter, Florent
Promoteur(s) : Drion, Guillaume
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"
Année académique : 2020-2021
URI/URL : http://hdl.handle.net/2268.2/11556

University of Liège

Faculty of Applied Sciences

Using multistability to solve fading memory problems in reinforcement learning

Author: Florent De Geeter

Supervisor: Prof. Guillaume Drion

Jury members:

Prof. Damien Ernst, Antoine Wehenkel

Master thesis submitted for the degree of
MSc in Computer Science and Engineering

Academic year 2020-2021

Abstract

Using multistability to solve fading memory problems in reinforcement learning

This thesis focuses on two important topics in machine learning. First, recurrent neural networks (RNN), which are neural networks with memory, i.e. which can propagate information through the time. Second, reinforcement learning (RL), a machine learning technique in which agents have to learn to perform the best actions by interacting with environments. The goal of the thesis is to use RNNs in a reinforcement learning setting to learn to retain information which will be useful in the future for selecting the best actions.

More specifically, this thesis studies the benefits of using RNN agents whose memories are based on multistability when dealing with environments that require a long-lasting memory. Indeed, the memory of RNNs usually fades with time because it relies on their dynamics, but it is no longer true when multistability is used.

The goal of this paper is twofold: first we show the interest of the multistability-based memory by testing a new type of RNN called the nBRC built for being multistable on a RL environment specifically made to require a long-lasting memory. This new RNN shows a great generalization capability thanks to its memory: by only being trained on a small version of the environment, it is able to generalize its knowledge and to play correctly on longer versions.

The second objective of this thesis it to introduce and test a pretraining algorithm, called the *multistability warmup*, which is supposed to force a RNN to become multistable. This algorithm is applied to several types of RNNs, including the well-known GRU and LSTM. Then these RNNs are tested on the same environment as previously, and we observe that their results are improved, especially for GRU which is able to compete with the nBRC.

Finally, we discuss about what could be done next. On one hand, multistability gave us great observations, and it could be very interesting to test it on more complex environments to see what it has to offer. On the other hand, the *multistability warmup* could still be improved, as it does not work very well with each type of RNN.

Acknowledgments

I would like to thank my supervisor, Prof. Guillaume Drion, and Nicolas Vecoven, for all the time they invested in this project. They guided me all year long and their many tips helped me a lot.

I would also like to thank all my friends and my family for their support.

Contents

Chapter 1

Introduction

Machine Learning (**ML**) can be defined as the *automated detection of meaningful patterns in data* [1]. It is widely used in problems in which it is impossible to provide a precise set of instructions to the computer, either because humans cannot explain how they do it (for instance recognizing faces on images) or because the problem is too complicated to be solved *by hand* (by a human). ML brings a concept present in living beings into our computers: the ability to learn, i.e. the ability to acquire knowledge through experience. ML algorithms can thus be trained on tasks, starting with no knowledge at all about these tasks and ending by performing very well on them. People have made major breakthroughs in Machine Learning during the past decades, especially in Deep Learning (**DL**) and neural networks. Indeed, as our computers are becoming more and more powerful, deep neural networks which were only a concept at the end of the 20th century can now be used in multiple domains and applications, including for instance cars' autopilot and machine translation. A big community has been built up over the years around these fields and people are developing more and more techniques which keep improving the performance of DL and ML algorithms.

This paper focuses on a special ML technique, Reinforcement Learning (**RL**), and a special type of neural networks, Recurrent Neural Networks (**RNN**). In RL, agents evolve inside environments by performing actions. Their goal is to learn to maximize the reward obtained when playing in these environments. On the other hand, RNNs can be seen as neural networks with memory, i.e. the ability to propagate information through time. However, the current types of RNNs are (normally) not able to keep an information for a long time if they are not stimulated periodically. 2 solutions to this problem are tested in this paper:

- The first one involves using a RNN cell (the nBRC) based on the bistability of human neurons and designed to have a long-lasting memory;
- The second one consists of a pretraining algorithm which, when applied on a RNN cell, trains it to increase the period during which it can keep an information, i.e. forces it to become multistable.

Experiments using these solutions are made on a RL environment (the *T-maze)* designed for testing the duration of the memory of the cells. The behavior of the RNNs on this environment is then analyzed.

This paper is structured as follow: In chapter 2, a global overview of Machine Learning and Deep Learning is given. Then in chapter 3 are described the RNNs. Some architectures of RNNs are introduced in this chapter, like the well-known GRU or the new nBRC. Chapter 4 presents more formally the Reinforcement Learning and our RL environment on which will be made the experiments, the *T-maze*. The first experiment is described in chapter 5. It consists of testing the GRU and nBRC cells on the *T-maze* environments, and then to analyze their behavior. Chapter 6 presents the second solution, i.e. the pretraining algorithm, and tests it on the *T-maze* environment. Finally, chapter 7 concludes this paper.

The codes used to make the experiments described in this report are available in this repository: https://github.com/FloDg/using-multistability-to-solve-fading-memory-problems-in-reinforcement-learning

Chapter 2

Machine learning and Deep learning

This chapter aims at giving a global overview of Machine learning and Deep learning, while the following two chapters focus on two specific fields, i.e. Recurrent neural networks and Reinforcement learning. The most well-known techniques of ML are introduced in the first section, while the concept of *Artificial Neural Network* is described in the second section.

2.1 Machine learning

2.1.1 Overview

Machine Learning (**ML**) brings a new capability to our computer programs, the ability to learn, in order to solve a given task. More formally, it is *the study of computer algorithms that improve automatically through experience* [2]. Usually, when a program is written to perform a certain task, it contains precise instructions describing how to perform this task. But sometimes it may be too difficult to describe precisely how to perform a certain task, because it is too complex for instance. This is where ML comes in with a different approach: instead of telling the computer how to perform the task, it is given an algorithm that allow it to learn to perform well at this task. However, computers, like living beings, need past experiences to be able to learn. This is a concept known in ML as *training data*, and it is very important. Indeed, even the best ML algorithm will perform badly on a task if it has *bad* training data. Determining if the training data used is *bad* or *good* will be discussed in the next section, because it depends on which ML technique is used.

The complete process of training a ML model[1] for a given task is divided in four parts (figure 2.1):

1. the **Data acquisition** part during which the training data is acquired,
2. the **Data preprocessing** part which modifies the training data in order to facilitate or to improve the training of the model,
3. the **Training** part which actually trains the model using the preprocessed data,

[1] a *model* is an output of a ML algorithm, i.e. it is the model that performs the task.

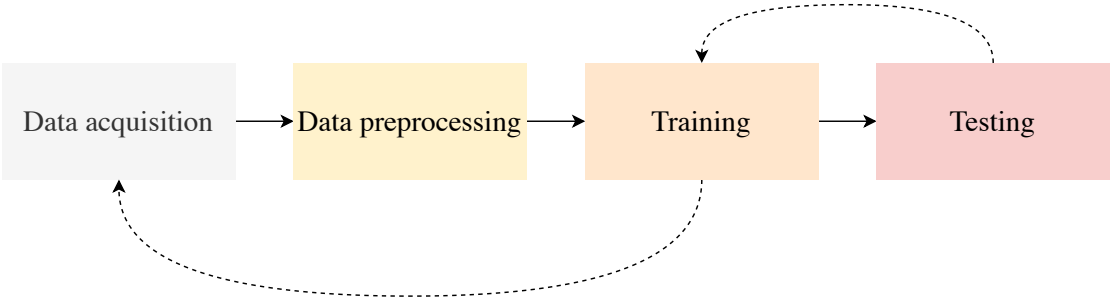


Figure 2.1: General steps when training a ML model. Dotted arrows are possible cycles.

4. the **Testing** part which tests the trained model and evaluates its performance.

Note that this decomposition is intended to be general. In practice, new parts can be added or some can be removed. For instance, it is not always necessary to preprocess the data, but it is often a good idea. Lots of researches have been made in order to improve data preprocessing [3][4]. Moreover, it is sometimes necessary to repeat certain parts: for instance, when the training data is generated by the model that is currently trained, it is better to loop over the three first steps several times. Once more, this process depends a lot on the ML technique used. Finally, when testing a model, it is very important to test it on data which have not been used to train it in order to correctly estimate the performance.

ML can be split into several techniques, depending mainly on the task and the training data available. These techniques are explained in section 2.1.2.

2.1.2 Techniques

The three best known ML techniques are the Supervised Learning (**SL**), Unsupervised Learning (**UL**) and Reinforcement Learning (**RL**). These are all described in this section, but the last one is explained in more details in chapter 4.

Supervised learning SL is about learning an unknown function f, whose domain is $\mathcal{X}$ and image domain is $\mathcal{Y}$. In this case, the training data is labeled, which means that each sample of the dataset contains a set of features (the input) $x \in \mathcal{X}$ and an output $y \in \mathcal{Y}$ such that $y \sim f(x)$. Indeed, as f may not be deterministic, it is possible to have several y generated from the same x. The goal of the ML algorithm is to output a model $\hat{f}$ which best approximates the function f given the training data. Once this model obtained, it can be used to predict outputs from unseen features, $\hat{y} = \hat{f}(x)$. However, if there are not enough samples in the training data, or if it is not representative enough of the function f, then the outputted model will not provide a good approximation of f. To estimate the performance of the model, an evaluation function is used. This function takes as inputs the predictions $\hat{y}$ of some x's as well as the real outputs y of these x's. It then outputs a score telling how good (or bad) are the predictions compared the real outputs.

There are two basic types of tasks in SL: classification tasks and regression tasks. In classification, the outputs y are symbols, also known as *classes*. The goal of the model is

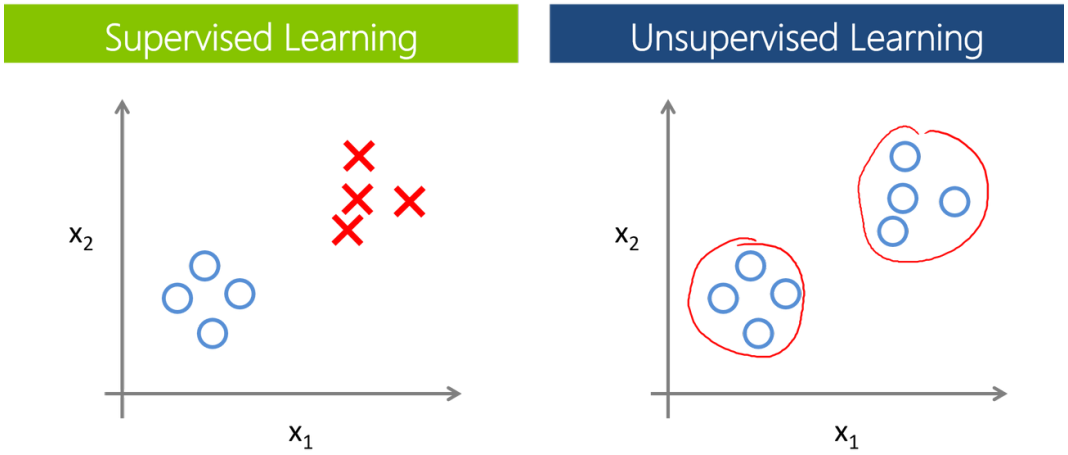


Figure 2.2: Comparison between supervised learning and unsupervised learning. The SL example showed here is an example of binary classification, while the UL one shows a clustering task. Source: [5]

thus, given a set of features x, to predict the class y of this set. One example of evaluation function in this case is the accuracy, i.e. the ratio of correctly predicted classes by the model over the total number of predictions. If the accuracy reaches 1, then the model is 100% accurate. But in certain tasks this evaluation method may not be correct. For instance, let's take the famous example of the rare disease detection task: the goal of the model here is, given some information about a patient, to predict if he suffers from this disease (class *true*) or not (class *false*). Assume a naive model that always predicts *false*, its accuracy will be close to 1 because there are very few patients who suffer from that disease. But this model is completely useless because it never detects a sick patient. The accuracy is not adapted to this task, it may be better to compute the ratio of the sick patients who have been predicted as *true* over the total number of sick patients. This is called the *recall*. In the other basic type of SL, the regression, the outputs y are numbers. An example of evaluation function in this case is the mean squared error MSE: given a set of real outputs/predictions pairs $\{y_i, \hat{y}_i\}, i \in [1, N]$,

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

The closer to 0, the better the model is.

Unsupervised learning Unlike SL, UL deals with unlabeled data. There are thus no output in the samples, only features, and the goal of UL is to extract hidden patterns from this data (see figure 2.2). UL is especially useful in tasks where labelling the data is costly (there are not a lot of labelled data) but unlabelled data is largely available. This is the case for instance in Machine Translation (**MT**), where the models are trained to translate a sentence in a source language to a target language. Indeed, labelling the data, i.e. translating sentences, takes time, while, for a given language, there are a lot

of available sentences that are not translated (which constitutes a *monolingual dataset*). Some researches have already been made about using UL techniques in MT [6]. An other common problem in UL is clustering, i.e. grouping the samples or the features in different subsets (also known as *clusters*).

An other ML technique called Semi-Supervised Learning is located between SL and UL. Its goal is to use both labelled and unlabelled samples in order to build a model. One general idea is to iteratively train a model on labelled dataset, use this model to label some unlabelled data (*pseudo-labelling*) which are added to the labelled dataset and then retrain a new model on the new dataset. This technique can be used for instance in image classification [7] or in machine translation [8].

Reinforcement learning *The idea that we learn by interacting with our environment is probably the first to occur to us when we think about the nature of learning* [9]. This sentence summarizes fairly well the idea of RL. While SL and UL view *learning* as building models from existing datasets, RL takes a different approach. In this type of ML, a model is viewed as an agent that interacts with an environment, described by some hidden dynamics: at a given timestep t, the agent receives an observation x_t of the current state of the environment. Based on this observation, the agent performs an action a_t in the environment, which changes the state of the environment: this is called a *transition*. The latter finally gives the new observation x_{t+1} to the agent as well as a reward signal r_t telling how good this transition was. This reward signal is used by the agent to learn the good actions to perform. Indeed its goal it to maximize the sum of the rewards it will get while interacting with the environment. In RL, a policy π is a function that selects at each timestep an action to perform. RL is thus concerned by, given an environment, finding the optimal policy π^*, i.e. the policy that maximizes the cumulative reward obtained. All of these notions will be explained in more details in chapter 4. One difference between RL and the two other types of ML is the origin of the training data: while in SL and UL the data is provided, in RL the agent usually generates its own training data. In section 2.1.1, we have seen the 4 big steps when building a ML model (figure 2.1). We have also seen that it was sometimes possible to have cycles in this process. This typically happens in RL: indeed, several RL algorithms alternate between generating new data by playing on the environment and training the agent with the stored data. Due to its nature, RL is often associated with games [10], where the agent controls what is usually controlled by the player and tries to maximize the score, and robots [11], where the agent observes its environment through sensors and performs actions by controlling actuators. However, researches are made in order to use RL techniques in more "supervised" tasks, like for instance machine translation [12].

2.2 Deep learning

2.2.1 Overview

Deep Learning (**DL**) [13] is a branch of ML used in a lot of types of ML tasks. While traditional ML algorithms, like the *decision tree* [14] or the *K-nearest neighbors* [15] algorithms, are not well suited for dealing with some types of data, like images or se-

quences, DL provides methods for dealing with these special types of data. Furthermore, DL achieves impressive results in lots of different tasks. DL models are called Artificial Neural Networks (**ANN**), which are inspired from human brains. These are universal function approximators, which means that no matter the function, it will always be possible to find a neural network that approximates it. DL is thus concerned with the creation and the training of such networks with the aim that they will approximate some unknown function $y = f(x)$, which can sometimes be very complex. ANNs are described in more details in the next section.

This branch of ML only started to be really used in the last few decades because it requires a big amount of computational power to train its models and also a lot of data. Indeed, neural networks can be very big and can contain up to billions of parameters. For instance, the *GPT-3* model [16], which is used to model languages, contains about 175 billions of parameters. The texts generated by this model are so well written that it is hard to distinguish them from real texts, written by humans. This is one of the many examples where DL methods can compete, or even outperform, humans.

The next section describes the components of the neural networks, as well as how these models are trained in practice.

2.2.2 Artificial neural networks

ANNs can be seen as black boxes, that take as inputs some values and output some other values depending on their inputs. The goal of this section is to introduce the different components that compose ANNs.

An ANN consists of several layers, and each layer is made up of several artificial neurons, also known as *units*. The unit is the smallest component of ANNs. It is connected to other units, usually units from the previous layer, like neurons are connected to other neurons. To each connection between two units is associated a value, also known as a *weight*. The weights are the parameters of the neural network and changing them modifies the values outputted by the model, thus allowing the network to *learn*. Each unit can compute a value based on the values of the units it is connected to it and the weights of these connections. The only exceptions are the units of the first layer, namely the *input* layer, which are not connected to other units. Their values are directly taken from the inputs. Finally, the outputs of the neural network are the values of the unit of the last layer, i.e. the *output* layer. Thus, how does an ANN compute its outputs given some inputs ? First the inputs are given to the units of the first layer. Then the units of the second layer computes their value from the ones of the first layer and their weights. This is repeated with the third, fourth, etc layers until the last layer is finally reached. The outputs are the values of this last layer. We observe that the process of computing the outputs of an ANN consists of a propagation of values from the input layer to the output layer. This is called a *forward pass*. This concept of *propagation through the network* is very important, because it is also used during the training.

An example of a neural network is represented in figure 2.3. Note that the layers comprised between the input and the output layers are called the *hidden* layers. There can be 0, 1 or more hidden layers in a neural network. The network showed in figure 2.3 is called a *multi-layer perceptron* (**MLP**). The *perceptron* [17] is one of the first types of

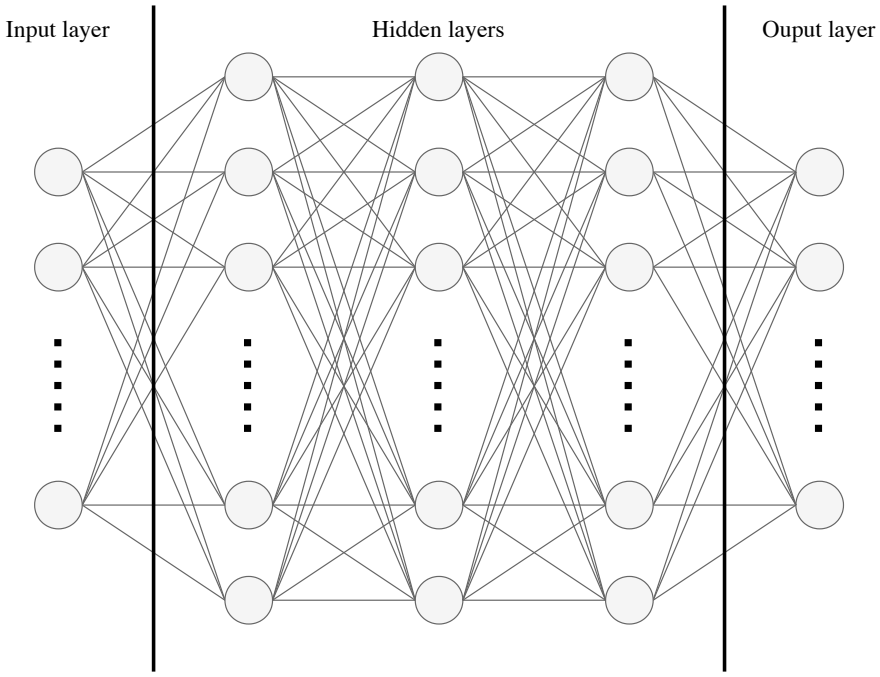


Figure 2.3: An artificial neural network, more precisely a MLP.

artificial neuron. It is defined as follow:

$$y = \begin{cases} 1 \text{ if } \sum_{i=1}^{n} w_i \cdot x_i + b \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

where y is the output, the x_i's are the inputs, the w_i's the weights and b the bias, i.e. a *weight* which is not connected to any unit. Originally, the MLP designated a neural network with multiple layers composed of perceptrons, but now as the units have evolved, it can designate any *fully connected feed-forward network*. The terms *fully connected* refer to the type of layer used in the network. Indeed, there exist different types of layers. As its name suggests, in such a layer, a unit is connected to all the units of the previous layer.

2.2.3 List of units

Since the perceptron has been created, new types of units have appeared, but they can almost all be summarized in one general formula:

$$\begin{aligned} y &= f\left(\sum_{i=1}^{n} w_i \cdot x_i + b\right) \\ &= f(\mathbf{w} \cdot \mathbf{x} + b) \end{aligned}$$

which introduces a new function f, called the activation function. There exist units that are not based on this formula, we will see such an example in chapter 3. y (also denoted

sometimes as a) is called the activation of the unit. The different units have thus different activation functions. For instance, if we use the binary step activation function:

$$f(x) = \begin{cases} 1 \text{ if } x \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

then we retrieve the perceptron unit. Here is a list of the main activation functions:

- The *sigmoid* function, which is a soft and differentiable version of the perceptron:

$$S(x) = \frac{1}{1 + e^{-x}}$$

- The *tanh* function, which squeezes inputs in range $]-1, 1[$:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The *Rectified Linear Unit ReLU*, probably the most used activation function for the hidden layers:

$$ReLU(x) = max(0, x)$$

- The *Softmax* function, which computes probabilities from real values (also called *logits*):

$$Softmax(x_1, ..., x_n, j) = \frac{e^{x_j}}{\sum_{i=1}^{n} e^{x_i}}$$

  Note that, unlike the other functions, the *Softmax* value of one unit depends on the other units' values, because this function makes a normalization in order to output a probability distribution, i.e.:

$$\sum_{i=1}^{n} Softmax(x_1, ..., x_n, i) = 1$$

Of course, this is not an exhaustive list, but these are the most known activation functions.

2.2.4 Backpropagation

Thus we now know what are the basic components of a neural network, but there is still a problem: how can this thing learn to perform a task ? The most used algorithm, which allows to train up to very big networks of billions of parameters, is known as *backpropagation* (**BP**), introduced in [18]. We will briefly describe this method.

We know that a neural network can learn by changing the weights of its units. The goal is thus to find the weights that perform the best on the task. And how do we measure the performance of an agent of a given task ? With a function that has to be minimized (*loss* function) or maximized (*objective* function) by the model, such that if the model reaches a global minimum/maximum of this function, then it cannot be improved anymore. Thus the next question is: Given a neural network and such a function, how

can I find the optimal weights ? It is almost impossible to find these optimal weights in one step, thus the solution is to make several small steps that improve the model. Practically, this can be done by making a small step in the direction of the gradient of the function (in case of an objective function) or in the opposite direction (in case of a loss function). Mathematically, if θ_t designates the model's weights at time t and $\mathcal{F}$ the function, then:

$$\theta_{t+1} = \theta_t + \alpha \cdot \nabla_{\theta_t} \mathcal{F} \text{ if } \mathcal{F} \text{ is an objective function}$$
$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_{\theta_t} \mathcal{F} \text{ if } \mathcal{F} \text{ is a loss function}$$

where α is called the *learning rate* and controls the importance of the update. These methods are respectively called *gradient ascent* and *gradient descent*. Usually, the value of the function is computed from a small *batch* of data. For instance, in SL a loss function is usually used (like the *MSE*, introduced in 2.1.2). For each update of the model, a batch, i.e. some number of features/label pairs (usually 32 or 64), is taken from the training dataset. Then the loss value is computed from the real labels and the predicted labels, and finally the update is made with gradient descent. On the other hand, in RL, sometimes an objective function is used, rather than a loss. In our experiments however, we note that the DQN algorithm (described in section 4.3.2) also uses a loss function. In practice, *optimizers* are used to update the weights of a neural network. They use variants of gradient descent/ascent by adding different notions, like for instance *momentum*, to increase the efficiency or the stability of the training. For the experiments made in this project, the *Adam* optimizer [19] has been used.

The last problem concerns the way of computing the derivative of the function with respect to each weight of the network. This is where BP comes up. We saw previously that neural networks make a *forward propagation* in order to compute the predictions. Therefore, the outputs of each layer depend on the outputs of the previous layer. The idea of BP is thus to propagate the gradients using the chain rule of derivatives from the output layer to each weight of the network, which thus results in a *backward propagation* or, in shorter form, *backpropagation*. Once the gradient computed for each weight, these can be updated with gradient ascent/descent. The BP algorithm can be summarized in 4 equations:

$$\delta^L = \nabla_{a^L} \mathcal{F} \odot f^{L'}(z^L) \quad \text{BP1}$$
$$\delta^l = \left(\left(W^{l+1}\right)^T \delta^{l+1} \right) \odot f^{l'}(z^l) \quad \text{BP2}$$
$$\frac{\partial \mathcal{F}}{\partial b^l_j} = \delta^l_j \quad \text{BP3}$$
$$\frac{\partial \mathcal{F}}{\partial w^l_{j,k}} = a^{l-1}_k \delta^l_j \quad \text{BP4}$$

where L is the output layer, a^l is the vector of units' activations of layer l, a^l_k is the activation of the kth unit of layer l, f^l if the activation function of layer l, z^l is the argument of the activation function at layer l (i.e. $a^l = f^l(z^l)$), W^l is the weights matrix of layer l, b^l_j is the bias of the jth unit of layer l and $w^l_{j,k}$ is the weight between the jth

unit of layer l and the kth unit of layer $l-1$. These equations use a lot of symbols indeed, but their main logic can be quickly explained: Like the units' activations are propagated through the network during a forward pass, the δ's are propagated from the last layer to the first one. BP1 computes the first δ, i.e. the one of the output layer, from the gradient of $\mathcal{F}$ with respect to the outputs of the network. BP2 allows to propagate the δ's: indeed, δ^l is computed from δ^{l+1}. Finally BP3 and BP4 compute the derivatives of $\mathcal{F}$ with respect to the bias and the weights from the δ's. Note that BP2 multiplies δ^{l+1} by the weights matrix, meaning that each time we pass though a layer, we multiply by the weights matrix. This will be discuss in section 3.1, because problems arise from this.

2.2.5 Other types of layers

In this section have been introduced several units, but only one type of layer, the *fully connected* (**FC**) one. These layers are usually used with compressed forms of data (also known as *features*) because the number of parameters of such layers increases very quickly. Indeed, if the number of units of the FC layer is N and the previous layer has M units, then the number of parameters is $\mathcal{O}(N \cdot M)$. For instance, assume the input of the network is an RGB image with a resolution of 100x100, then if a FC layer of 100 units is used, then this layer will have about $3 \cdot 100 \cdot 100 \cdot 100 = 3 \cdot 10^6$ parameters, which is pretty big. That is why an other type of layer is used when dealing with images, the *convolutional* layers [20]. They will not be described here but they allow to decrease drastically the number of weights, compared to a FC layer. These layers basically compress the input image by extracting the main features. Usually, FC layers are added after convolutional layers in order to make computation on the features extracted. The last type of layer that will be introduced is the *recurrent* layer, also known as *recurrent cell*, which allows an ANN to maintain some form on memory by propagating information through the time. These are explained in the next section.

Chapter 3

Recurrent neural networks

The predictions of the neural networks that have been described until now only depend on the current input. But in some cases, it may be useful to allow the neural network to *memorize* some information. For instance, when dealing with sequences of variable lengths, like sentences, having such network allows to feed it with one element of the sequence at a time. Once the last element of the sentence has been passed to the network (or at every step, it depends on what the network is supposed to do), we can ask it to perform a task, like classification for instance. Those networks are called *Recurrent Neural Networks* (**RNN**), and they use a special type of layer, called the *recurrent cell*. The first mention of such an idea has been written by Minsky and Papert in 1969 in [21] (section 13.2), while the first appearance of the term "Recurrent Net" is in [22]. But since these papers, lots of researches have been made on this field, and new types of recurrent cell have been created. This section first describes globally how recurrent cells work and how they are trained and then presents several types of recurrent cells.

3.1 Overview

The RNNs create *memory* by propagating information between forward passes. In practice, this information consists of a vector of values, namely the *hidden state*. More formally, if ϕ is the RNN, x_t the input at time t and h_t the hidden state at time t, then:

$$h_t = \phi(x_t, h_{t-1})$$

Thus the input of a RNN consists of the concatenation of the previous hidden state h_{t-1} and the *real* input x_t. Then the RNN outputs the new hidden state, which will be used at the next timestep (recurrent connection). Usually, one or more FC layers are added after the recurrent cell in order to compute the predictions from the hidden state. Also, as the next hidden state depends on the previous one and on the current input, it is possible to train the network to retain in its hidden state some important information, thus propagating it until the moment it will need that info. In section 2.2.2 has been introduced the backpropagation BP algorithm, which allows to efficiently compute the gradient of a function with respect to all the weights of the network, with the aim of improving the performance of the model by updating its weights "in the good direction".

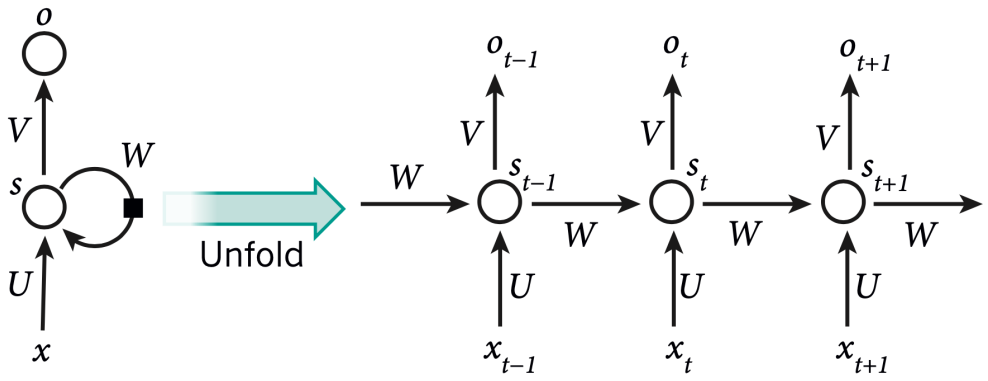


Figure 3.1: Unrolling (or unfolding) a recurrent neural network. Source: [13].

However, when dealing with non-recurrent neural networks, there is no time dependency, i.e. the current output only depends on the current input, and thus to update the weights, only the current value of the objective/loss function has to be considered. But in the case of RNNs, the computations made at a certain timestep t can have an impact on the value of the function at a later timestep, because of the propagation of the hidden state. These are time dependencies, and the training algorithm has to ensure that the RNN will be able to learn them in order to improve itself. Nevertheless, there is a simple solution to this problem: BP propagates gradients through the network, and it is possible to *unroll* a RNN, which makes it look like a deep feedforward neural network whose depth depends on the length of the sequence and whose layers share the same weights. Figure 3.1 shows this transformation. In this figure, U, V and W are weights matrices, and the hidden states h and the outputs o of this RNN are computed as follow:

$$h_t = f_h\big(U^T x_t + W^T h_{t-1} + b_h\big)$$
$$o_t = f_o\big(V^T h_t + b_o\big)$$

where f_h and f_o are two activation functions and b_h and b_o two biases. This RNN is known as the *Elman network* [23], and the recurrent cell used here is simply a FC layer which has two weights matrices: the usual one for the input (U), and a new one for the previous hidden state (W). From the unrolled representation of a RNN, it is simple to understand how the BP algorithm will be applied: At each timestep, the RNN outputs something, thus at each timestep we can compute the value of the objective/loss function $\mathcal{F}$. Using the BP equations, we can propagate the gradients from a given timestep t to the previous timesteps. This process is known as *backpropagation through time* (**BPTT**).

3.1.1 Problems of BPTT and solutions

However, BPTT suffers from two problems: *vanishing* and *exploding* gradients [24]. Both result in the impossibility of training correctly the network. We have seen in equation BP2 that each time we pass through a layer, we multiply by the weights matrix of that layer. But in this case, all the layers share the same weights matrix (W), which can result in instabilities (either exploding or vanishing gradients) when dealing with

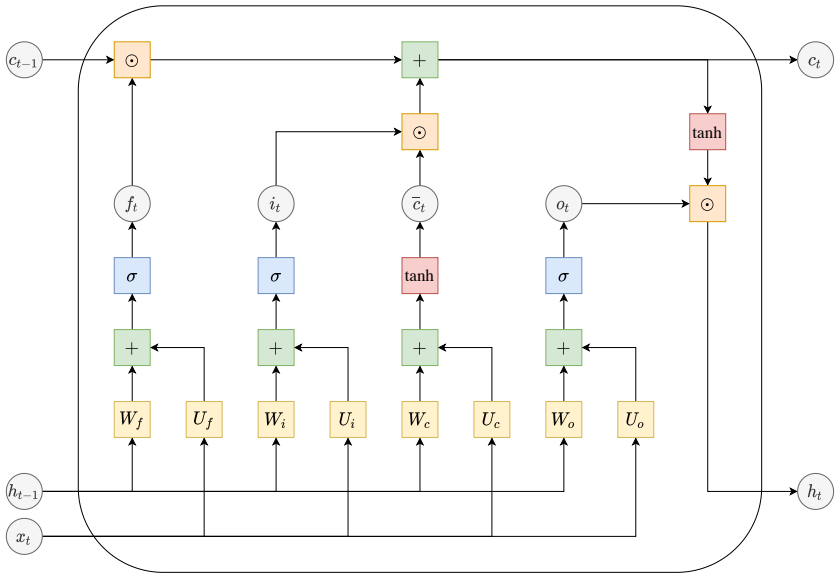


Figure 3.2: Computation graph of a LSTM cell.

very long sequences. One solution to these problems is to use a truncated version of BPTT: instead of backpropagating through the entire sequence, the backpropagation only occurs in small chunks. But this has the drawback that long term dependencies will not be learned. Initializing the weights matrix orthogonally can also help, because orthogonality ensures that the powers of such a matrix will be stable, they will neither converge towards 0 nor explode.

Finally, one solution used by current recurrent cells are *gates*, i.e. *pass-through*'s which avoid the activation function. Indeed, non-linear activation functions (like the *sigmoid*) can saturate, in which case their derivative is almost zero. As BP multiplies by these derivatives (BP2), this results in vanishing gradients. The *gates* solution is described in the next section.

3.2 Types of cells

Up to now, only the Elman's RNN has been introduced. But it is not often used in practice, due to the gradients problems. This section thus aims at introducing some other recurrent cells. Note that all of them use *gates* in their architecture.

3.2.1 LSTM

Created in 1997 by Hochreiter and Schmidhuber [25], the *Long-Short Term Memory* (**LSTM**) recurrent cell is still used nowadays. It was designed to be able to learn long time dependencies and is based on the gating system. The specificity of the LSTM is its hidden state, which has two components: the *real* hidden state, i.e. the output of the cell, and the cell state, which is internal to the cell and is used for long-term memory. The gates are used to control what is added/removed to/from the memory of the cell.

In practice they consist of vectors whose elements are between 0 and 1 and which are multiplied element-wise (*Hadamard* product, $\odot$) with a "data" vector to control what to keep from this vector. There are three gates in a LSTM cell:

- the **forget gate** f_t which controls what to keep and what to forget about the last cell state c_{t-1};
- the **input gate** i_t which selects what to add to the memory from the current input x_t;
- the **output gate** o_t that decides what to output from the memory (the cell state c_t).

The computation graph of a LSTM cell is shown in figure 3.2. Previously we saw the BPTT algorithm and its problems related to the gradients. We also said that using gates could help to solve these problems. BPTT uses the hidden states to propagate the gradients through the time. With LSTM, this is done via the cell states. In figure 3.2, we can see how the next cell state c_t is computed from the previous one c_{t-1} (the top line). They are no weights multiplications or activation functions that are used there, which thus increase the stability of the BPTT. This top line can be used as a kind of highway for the gradients to go back in time. And then the gradients of a particular timestep t can be computed by propagating through the rest of the cell. The LSTM equations are:

$$
\begin{aligned}
f_t &= \sigma\big(W_f h_{t-1} + U_f x_t + b_f\big) \\
i_t &= \sigma\big(W_i h_{t-1} + U_i x_t + b_i\big) \\
o_t &= \sigma\big(W_o h_{t-1} + U_o x_t + b_o\big) \\
\overline{c}_t &= tanh\big(W_c h_{t-1} + U_c x_t + b_c\big) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \overline{c}_t \\
h_t &= o_t \odot c_t
\end{aligned}
$$

The three gates are computed in a similar fashion and $\overline{c}_t$ represents the new data. Each gate is applied to a "data" vector: f_t to c_{t-1}, i_t to $\overline{c}_t$ and o_t to c_t.

LSTM is a quite old and complex recurrent cell, but it is still used nowadays. However, lots of researches are made in order to create new recurrent cells that are better and less complex than LSTM. The next introduced cell is its most famous concurrent, GRU.

3.2.2 GRU

Introduced in 2014 by Cho et al. in [26], the *Gated Recurrent Unit* (**GRU**) is a more simple recurrent cell than LSTM that achieves similar performance. It does not have a cell state, and it uses only two gates:

- the **update gate** which controls the update ratio, i.e. the lower it is the more the previous hidden state is forgotten;
- the **reset gate** that can decide to reset or drop some parts of the memory.

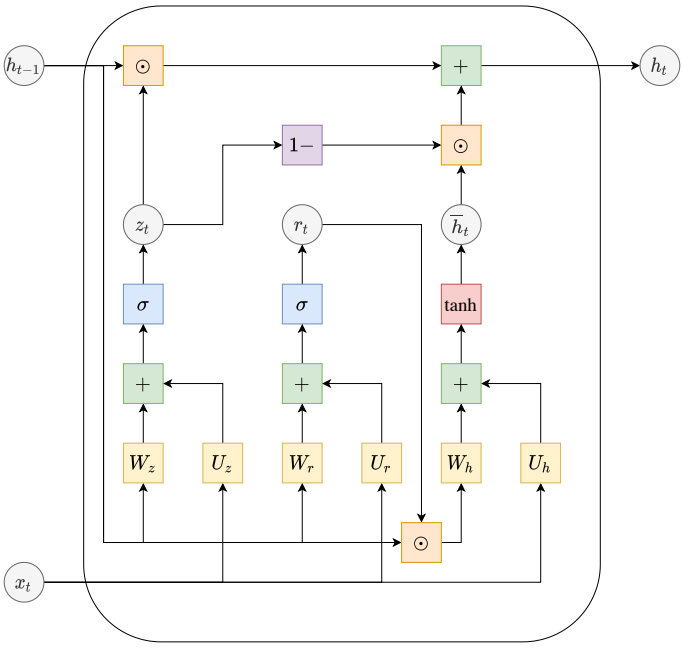


Figure 3.3: Computation graph of a GRU cell.

The computation graph of a GRU is given in figure 3.3, and here are the corresponding equations:

$$
\begin{aligned}
z_t &= \sigma\big(W_z h_{t-1} + U_z x_t + b_z\big) \\
r_t &= \sigma\big(W_r h_{t-1} + U_r x_t + b_r\big) \\
\overline{h}_t &= tanh\big(W_h(r_t \odot h_{t-1}) + U_h x_t + b_h\big) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t
\end{aligned}
$$

The two gates are computed in the same way as the LSTM's gates. The top line used to propagate the gradients though the time is still present in the computation graph. From the equations, we observe that the LSTM's forget and input gates have been merged to form the GRU's update gate, which performs a linear combination of the previous hidden state h_{t-1} and the new data $\overline{h}_t$.

LSTM and GRU have improved a lot the results obtained by RNNs, but they both suffer from the same problem: if their memory (i.e. their hidden state) is not stimulated periodically, then it will fade away. In practice, it means that if LSTM and GRU receives an information at a certain timestep, then it will always be possible to find a another timestep such that they both have forgotten this information. This is true most of the time, but it is sometimes possible for GRU and LSTM to keep information for an infinite period. This happens quite rarely, and this phenomenon will be discussed in the experiments. The two recurrent cells presented in the next section are based on a property which allows to get rid of this problem.

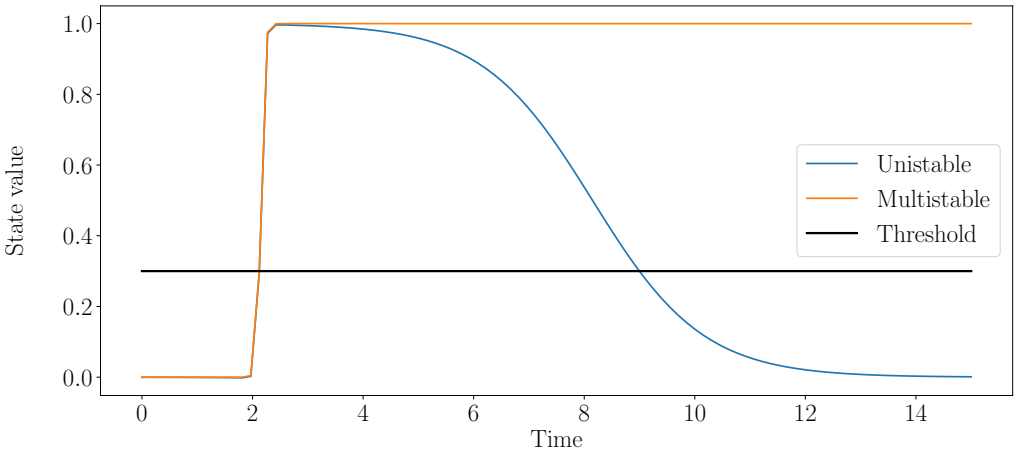


Figure 3.4: Example to demonstrate the utility of having a multistable RNN. This example has not been obtained with real RNNs, it has been created by hand. Assume two RNNs, one unistable and one multistable, and a indication received at $t = 2$. The goal of the RNNs is to retain as long as possible this information. At $t = 2$, both RNNs' states are perturbed and both memorize the indication. But after a certain time, the unistable RNN starts converging towards its stable state (in 0). After having passed a certain threshold, it will not be able anymore to remember the indication. On the other hand, the multistable one converges toward its second stable state (1), and is thus able to memorize the information for an infinite time.

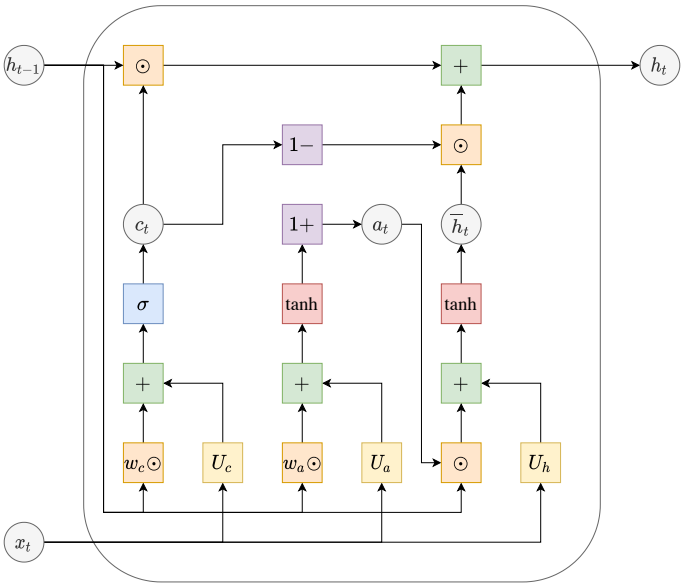


Figure 3.5: Computation graph of a BRC cell.

3.2.3 BRC & nBRC

Created by Vecoven et al. [27], the *Bistable Recurrent Cell* (**BRC**) is inspired by the neurons' bistability property of our brains in order to provide a never-fading memory. Indeed, our neurons have the possibility to switch between two stable states, providing thus some sort of cellular memory [28]. To understand how this could be used in the RNNs, we have to stop seeing them as neural networks, but more as closed-loop systems, where the input sequence is a perturbation and the hidden state a feedback. Such systems can have multiple stable states, switching from one to another. Having different stable states implies the capacity of keeping information for an infinite time. As said previously, LSTM and GRU have (most of the times) a fading memory: they only have one stable state. Therefore, when LSTM and GRU receive new information (a perturbation), their state moves away from the stable one for a certain duration, which thus creates memory. But if they do not receive any more perturbations, then their state will eventually converge back to the stable one. Their memory thus fades away. With more than one stable state, it is possible to jump from one to another when the system is perturbed (receives new information). As the system's state will converge to a different stable state, it will possible to remember forever the received information. An example is shown in figure 3.4. BRC and nBRC are based on this idea, and their multistability originates from the bistability of each of their units/neurons (cellular memory). It is also possible for GRU and LSTM to become multistable, but this is harder for them as, unlike BRC and nBRC, they have not been built to have this property. This will be discussed in the experiments.

The computation graph of BRC is given in figure 3.5, and its equations are:

$$
\begin{aligned}
c_t &= \sigma\big(w_c \odot h_{t-1} + U_c x_t + b_c\big) \\
a_t &= 1 + tanh\big(w_a \odot h_{t-1} + U_a x_t + b_a\big) \\
\overline{h}_t &= tanh\big(a_t \odot h_{t-1} + U_h x_t + b_h\big) \\
h_t &= c_t \odot h_{t-1} + (1 - c_t) \odot \overline{h}_t
\end{aligned}
$$

Like GRU, BRC uses two gates: c_t and a_t. The first one plays the same role as GRU's update gate z_t, while the second one controls the bistability of each unit (see [27]): when a_t is lower (greater) than 1, the unit is monostable (bistable). There are two big differences between GRU and BRC:

- the first one is the matrix multiplications of h_{t-1} that are replaced by Hadamard products which implies that a unit cannot affect the other ones (cellular memory constraint);
- the second one is the range of the second gate (r_t for GRU and a_t for BRC): with BRC, this gate can take value between 0 and 2, while GRU allows values between 0 and 1.

The *recurrently neuromodulated Bistable Recurrent Cell* (**nBRC**) is an improved version of the BRC, where the cellular memory constraint has been relaxed: the Hadamard products with h_{t-1} have been replaced by matrix multiplications, which makes nBRC more similar to GRU. nBRC's computation graph is given in figure 3.6 and its equations

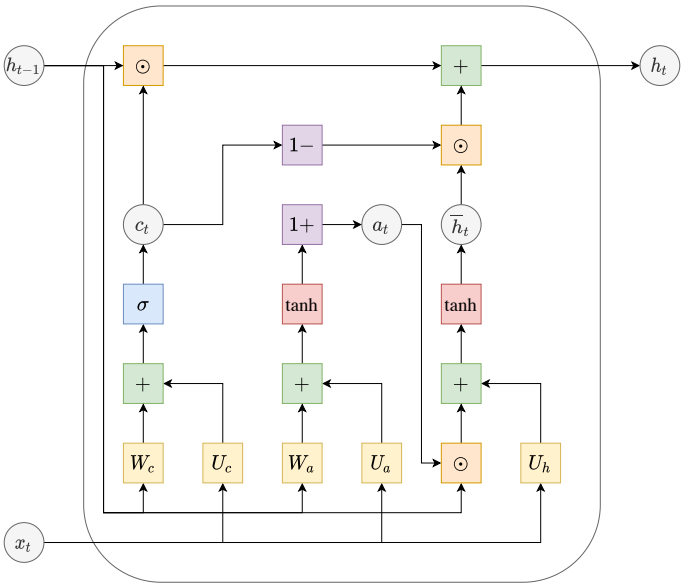


Figure 3.6: Computation graph of a nBRC cell.

are listed below:

$$
\begin{aligned}
c_t &= \sigma\big(W_c h_{t-1} + U_c x_t + b_c\big) \\
a_t &= 1 + tanh\big(W_a h_{t-1} + U_a x_t + b_a\big) \\
\overline{h}_t &= tanh\big(a_t \odot h_{t-1} + U_h x_t + b_h\big) \\
h_t &= c_t \odot h_{t-1} + (1 - c_t) \odot \overline{h}_t
\end{aligned}
$$

The nBRC is more powerful than the BRC, that is why we have used this cell in the experiments.

3.2.4 Other cells

As said previously, the RNNs and the recurrent cells are an active field of research. These researches mostly focus on the simplification or the improvement of the equations of known recurrent cells and on the initialization of the hidden state. An example of such cell, which is a simplification of the LSTM, is the JANET [29]. This cell only keeps the forget gate of the LSTM and also uses a special type of initialization, called the *chrono-initialization* [30]. JANET will be used in the experiments made in chapter 6. Other examples of new cells are for instance the LMU [31], the Laguerre and Ladder networks [32] or the GORU [33].

Chapter 4

Reinforcement learning

As stated in section 2.1.2, Reinforcement Learning (**RL**) is one of the main techniques in ML. As it is the technique used in the experiments, it is important to detail it. This chapter thus describes RL as well as the RL algorithm used in the experiments, namely the Deep Q-learning. The advantage of using RNNs in RL tasks will be addressed and at the same time we will take the opportunity to introduce the environment used in the experiments called the *T-maze*. Meta-Reinforcement learning, a subtopic of RL, is also introduced, because it will be discussed in the experiments.

4.1 Overview

As already said, RL is concerned with training an agent (i.e. a model) to survive or evolve in a given environment. RL is about learning from interactions: each time an agent interacts with an environment, it receives a feedback, called the reward signal. Its goal is to maximize the rewards it will get while interacting with the environment. But before diving into the details of RL and its algorithms, it is important to define more formally the notions of environment, agent, reward, etc. An example of basic interaction between the agent and the environment is given in figure 4.1.

4.1.1 RL Environment

An environment is characterized by several concepts:

- the **state space** X, which contains all the possible states of the environment. This space can be continuous or discrete. It is important to know that the agent does not always receive the full state of the environment, it may only perceive a part of it, i.e. an observation. This defines the notion of *observability* of an environment: an environment is said to be *fully-observable* when the observation received by the agent completely describes the current state, while the observations of a *partially-observable* environment only partially describe its current state. In this second case, it is necessary to define an **observation space** O, i.e. the set of all observations o_t the agent can receive, and the **observation probability distribution** $o(o_t, x_t)$, representing the probability of observing o_t while being in x_t.

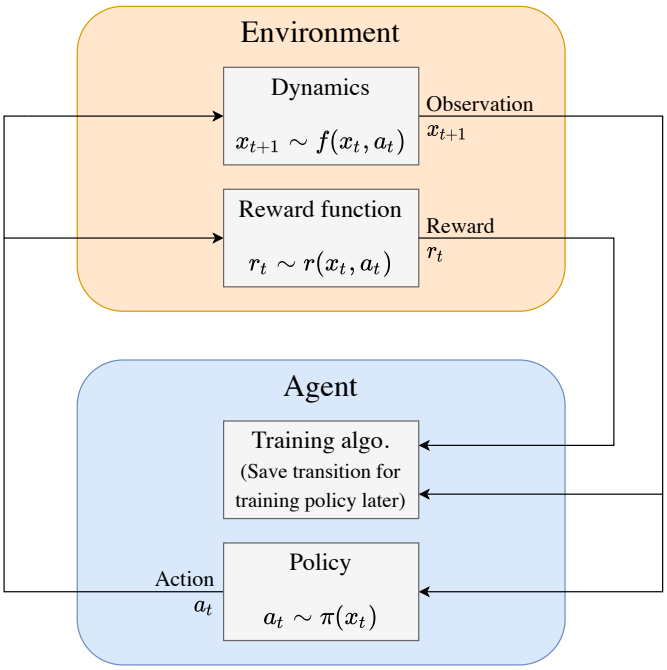


Figure 4.1: Basic interaction between the (fully-observable) environment and the agent in RL. Note that the agent is using a stationary policy here.

- the **action space** A, which contains the actions the agent can perform. Like the state space, the action space can be either continuous or discrete. When choosing a RL algorithm, it is important to know whether or not the action space is continuous, because some RL algorithms only work on discrete action spaces, while others only work on continuous action spaces.

- the **dynamics** f, that supervises the transitions of the environment: given the current state x_t and the chosen action a_t, the new state is computed as follows:

  $$x_{t+1} \sim f(x_t, a_t)$$

  In some environments, for the same state and the same action, it is impossible to reach two different states. Such environments are said *deterministic*. On the other hand, an environment is said to be *stochastic* if different states may be reach from a same state and a same action. In this case, it is common practice to define the notion of **transition probability distribution** $p(x_{t+1}, a_t, x_t)$, which represents the probability of switching from state x_t to state x_{t+1} when doing action a_t. Therefore, in a deterministic environment, for a given pair of state x_t and action a_t, there is only one state x_{t+1} with a non-zero transition probability, i.e. $p(x_{t+1}, a_t, x_t) = 1$.

- the **reward function** r, that computes the reward based on the state and the chosen action, i.e.:

  $$r_t \sim r(x_t, a_t)$$

The same remark as the one made for the dynamics applies here: for a given state x_t and a given action a_t, the reward might always be the same or not.

- the **initial state space** which is a subset of the state space. It contains all the states the environment could be in before the first interaction with the agent.
- the **terminal state space** which is also a subset of the state space. The states it contained are called *terminal* states. When such a state is reached, the agent cannot interact with the environment anymore. Therefore, an initial state marks the beginning of what is called an *episode*, i.e. a series of transitions, while a terminal state marks its end.
- the **discount factor** γ ($0 < \gamma < 1$), that defines the importance of the future rewards: indeed, at each transition, the agent receives a reward, and its goal is to maximize the discounted cumulative reward, i.e.:

$$R_{0 \to T} = \sum_{t=0}^{T} \gamma^t r_t$$

The closer γ is to 1, the more important the futures rewards are.

Due to their nature, fully-observable environments can be represented as *Markov Decision Processes* (**MDP**) $\mathcal{M}\big(S, A, p, r, \gamma\big)$, while the partially-observable ones can be modeled as *Partially Observable Markov Decision Processes* (**POMDP**) $\mathcal{P}_{\mathcal{O}}\big(S, A, p, r, O, o, \gamma\big)$.

4.1.2 Agent and policy

In turn, the agent is defined by its policy π, which selects the actions to perform. A policy is said *stationary* if the action a_t depends only on the current state x_t (or observation o_t), and not on the past (denoted as h_t, the history of the past transitions). A policy may be *deterministic*, if for the same input, the policy always outputs the same action, or, on the contrary, *stochastic*. The agent can also contain a training algorithm, which updates the policy based on the transitions made while exploring the environment.

The ultimate goal of an agent is to find the optimal policy π^*, i.e. the policy that maximizes the expected return, which is the expectation over the discounted cumulative rewards:

$$\pi^* = argmax_\pi \; E\Big\{ \sum_{t=0}^{T} \gamma^t r\big(x_t, a_t \sim \pi(h_t)\big) \Big\}$$

where T is called the *time-horizon* and represents the period during which the obtained rewards are taken into account to search for the optimal policy. For instance, if $T = 10$, the optimal policy will maximize the discounted cumulative reward over 10 timesteps. Note that T can tend to the infinity. The policy used in this equation is a non-stationary one, but in practice it is sometimes sufficient to use a stationary one. It is the case when the environment is fully-observable.

Now that the notions of environment and agent in RL have been defined, as well as the goal of RL which is to find the optimal policy, the training algorithms that can be used in such problems can be presented. This is the topic of the next section.

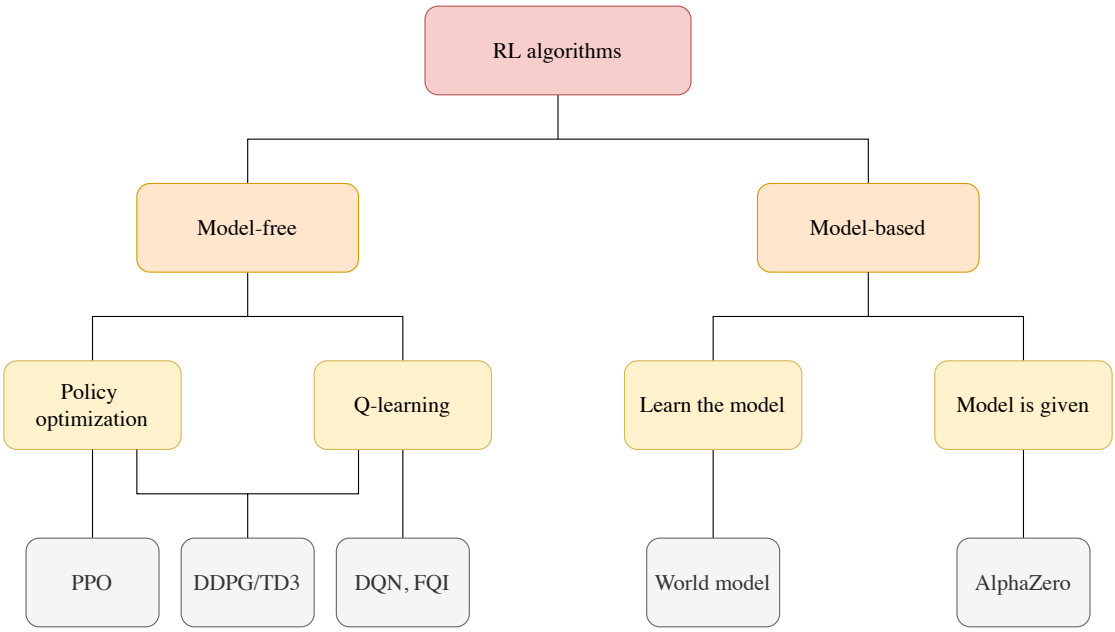


Figure 4.2: Taxonomy of the RL algorithms with some examples. Inspired from [34]

4.2 Types of RL algorithms

There exist several ways to improve the policy of an agent. Lots of different algorithms have been created in the past years, but these can classified into different subcategories. Figure 4.2 shows these different categories as well as some examples of algorithms. The two main categories of RL algorithms are the *model-based* and the *model-free* ones. In the first one, the algorithm uses a model of the environment to train the policy. A *model* means that the algorithm is able to simulate the environment. To do that, an access to the transition probability distribution $p(x_{t+1}, a_t, x_t)$ and to the reward function $r(x_t, a_t)$ is required. There are thus two possibilities: either a model is given to the algorithm, or it is learned during the training. For instance, *AlphaZero* [35] is an algorithm developed at DeepMind which has achieved superhuman performance in the games of Go, Chess and Shogi. It is model-based, and it is given the rules of the games, enabling it to simulate games. On the other hand, the *world model* algorithm created by Ha and Schmidhuber [36] is able to learn to simulate an environment only by observing it.

The second main category of RL algorithms does not use such models to train the agents. They do not rely on this ability of simulating episodes of the environment. The model-free algorithms can be split in two classes: the ones that directly optimize the policy, these are called the *Policy optimization* algorithms, or the ones that first learn to approximate an intermediary called the Q-function and then derive the policy from this function. These ones are based on the *Q-learning* algorithm and are described in the next section. Among the policy optimization algorithms is the *REINFORCE* algorithm [37], which is a policy gradient method. There exist also algorithms that belong to the two classes: for instance, the *Deep Deterministic Policy Gradient* (**DDPG**) [38] and its improved version the *Twin Delayed DDPG* (**TD3**) [39] are actor-critic methods. They

train an actor network that represents the policy via policy gradient and a critic network that approximates the Q-function with Q-learning.

4.3 Q-learning

4.3.1 Overview

The Q-learning algorithm [40] is based on dynamic programming and is concerned with training agents to act optimally in an environment by learning to evaluate the quality of the actions given the state. Before going into more details, note that only stationary policies will be considered here, as well as fully observable environments, to ease the notations. The Q-function, also called the state-action (value) function, is directly linked to this notion of quality: given a state x, an action a and the current policy π of the agent, the state-action function $Q^\pi(x, a)$ gives the expected return (Q-value) if the agent performs action a in the state x and then follows its policy π. Intuitively, the Q-function is an answer to the question "What can I expect to receive in the long term if I do action a in state x ?" and the best action to choose in a given state is the one that leads to the highest Q-value. Mathematically,

$$Q^\pi(x,a) = E\Big\{\sum_{t=0}^{T}\gamma^i r(x_t,a_t)\Big\}, \text{ with } \begin{cases} x_0 = x, a_0 = a, \\ x_{t+1} \sim f(x_t,a_t), \\ a_t \ \sim \pi(x_t) \text{ for } t > 0 \end{cases}$$

where T can tend to the infinity. The expectation is taken over the reward function (as it may be stochastic) but also over the transition probability distribution and the policy (as the environment and the policy may be stochastic). It is also possible to recursively define the Q-function:

$$Q^\pi(x,a) = E\Big\{r(x,a) + \gamma \cdot Q^\pi\Big(f(x,a), \pi\big(f(x,a)\big)\Big)\Big\}$$

where $Q^\pi(x, a) = 0$ if x is a terminal state (base case of the recursion).

As said previously, the goal of a RL algorithm is to find the optimal policy π^*, which has an associated optimal Q-function Q^* such that:

$$\begin{aligned} &Q^*(x,a) = max_\pi \, Q^\pi(x,a), \\ &\pi^* = argmax_\pi \, Q^\pi(x,a), \\ &\forall x \in X, \forall a \in A \end{aligned}$$

Furthermore, the optimal Q-function must respect a condition, known as the Bellman optimality equation, which comes from dynamic programming:

$$Q^*(x,a) = E\Big\{r(x,a) + \gamma \cdot max_{a'} Q^*\big(f(x,a),a'\big)\Big\}. \tag{4.1}$$

If this condition is not satisfied by the policy, than this policy cannot be the optimal one. The goal of Q-learning is thus to train an agent to approximate the optimal Q-function,

because once this function is known, the optimal policy can be easily computed from it. The idea is to start with a dummy approximation $\hat{Q}_0^*(x,a) = 0 \ \forall x, a$ and then to iteratively update it from a set of saved transitions (x_k, a_k, r_k, x'_k) with a variation of equation (4.1) [41]:

$$\hat{Q}^*(x_k, a_k) \leftarrow (1-\alpha) \cdot \hat{Q}^*(x_k, a_k) + \alpha \cdot \left(r_k + max'_a \, \hat{Q}^*(x'_k, a')\right)$$

where α is the learning rate and controls at which speed is updated the approximation. This equation is based on *Temporal Difference* (**TD**) learning, because it uses the approximated value of the next step $\hat{Q}^*(x'_k, a')$ to update the current one. This is a general algorithm for Q-learning, and many different ones have been created from it, each one with its own concepts, like for instance the *Fitted Q Iteration* (**FQI**) algorithm [42]. This one performs a certain number of iteration over a training set of transitions, the approximation $\hat{Q}^*$ getting closer to Q^* at each iteration.

Originally, the approximation of the Q-function was made by using a table: for each state x and action a, the value of $\hat{Q}^*(x,a)$ was stored in this table. But this only worked for discrete action and state spaces, and it was not very memory-efficient, especially when there were a lot of different states and actions. When dealing with a big discrete state space or a continuous state space, function approximators can be used, like trees for instance. In our case, neural networks have been used in the experiments, thus a small discussion about Q-learning with neural networks (also known as Deep Q-learning) can be made.

4.3.2 Deep Q-learning

Most Q-learning-based algorithms nowadays use neural networks as function approximators. One of the most famous is known as the *Deep Q-Network* (**DQN**) agent, created by Mnih et al. [43], which works with discrete action spaces. In the experiments made, a variant of DQN has been used, it is thus important to understand how it works. Unlike FQI that learns the Q-function from a training set of transitions, the DQN creates its own dataset by interacting with the environment. The algorithm 1 gives the pseudocode of this algorithm. Basically, DQN alternates between generating transitions to save them in a buffer called the *replay buffer* and training the current network on minibatches sampled from the replay buffer in a supervised way.

As it can be seen in the pseudocode, the agent plays several episodes, and each episode has a maximum number of timesteps. When interacting with the environment, transitions are collected and added to the replay buffer in order to be used later for training. These collected transitions are thus very important: for instance, if the agent always chooses the same action for a given state, then lots of transitions will be the same, and the agent will not sufficiently explore the environment. This problem is known as the *Exploitation-Exploration tradeoff*: exploitation happens when the agent follows what it already knows, while exploration happens when the agent is trying new actions, which it may think is a bad action but which could turn out to be a very good action. For instance, if the agent always chooses the action that maximizes its approximation of the Q-function, then it will always choose the same action for a given state; in this case, there is not enough exploration. On the other hand, if the agent always picks a

Algorithm 1: DQN, inspired from [43]

Initialize state-action function Q with random weights θ
Initialize target state-action function Q_{targ} with weights $\theta_{targ} = \theta$
Initialize empty replay buffer R
for *episode* $e = 1$ *to* N **do**
  Reset environment and place initial state in x
  for *timestep* $t = 1$ *to* T **do**
    Choose random action a with probability ϵ
    Otherwise choose action $a = argmax_{a'}\ Q(x, a'; \theta)$
    Perform action a in the environment and store new state x' and reward r
    Add transition (x, a, r, x') in R
    if x' *is a terminal state* **then**
      break
    else
      Set $x = x'$
    end
    if *Time to train* **then**
      Sample minibatch of transitions (x_i, a_i, r_i, x'_i) from R
      for *each transition* (x_i, a_i, r_i, x'_i) *in minibatch* **do**
        if x'_i *is a terminal state* **then**
          Set $y_i = r_i$
        else
          Set $y_i = r_i + \gamma \cdot max_a\ Q_{targ}(x'_i, a; \theta_{targ})$
        end
      end
      Perform a gradient descent on $\big(y_i - Q(x_i, a_i; \theta)\big)^2$ with respect to θ
    end
    if *Time to update target* **then**
      Set $\theta_{targ} = \theta$
    end
  end
end

random action, it may never reach the very good rewards if they are far from the initial state. Think about a long corridor with a treasure at the end, if the agent always picks a random action, it will never reach the end of this corridor: in this case there is not enough exploitation. Nevertheless, there is a simple solution to this problem: the ϵ-greedy exploration policy. Given a ϵ between 0 and 1, this policy selects a random action with probability ϵ, otherwise it selects the action that maximizes Q. By playing on the value of ϵ, it is possible to adjust the tradeoff between exploitation and exploration. In practice, the value of ϵ varies during the training: it starts at a high value and decreases to a minimal value. This technique is used in the DQN.

Concerning the training part of the algorithm, it was said that the agent was trained in a supervised way. But as said in section 2.1.2, this technique needs training samples that consist of *features* and *labels*. In this case, the model takes as input a state x and an action a, these are thus the features, and outputs a estimation of Q, which is thus the label. The states and the actions are saved in the replay buffer thus they are already available, but how can the target labels, i.e. the targets Q, be computed ? Like in traditional Q-learning, a variant of equation (4.1) is used: given a transition (x_i, a_i, r_i, x'_i) sampled from the replay buffer, the target Q can be computed as follows:

$$y_i = r_i + \gamma \cdot max_{a'}\ Q(x'_i, a')$$

where $max_{a'}\ Q(x'_i, a')$ is manually set to 0 when x'_i is a terminal state. By preparing a minibatch of transitions using this formula and then performing a gradient descent step using the MSE loss, the agent will improve its predictions. There is still one problem: the model that has to be trained is also used to generated the targets. This means that each time the model is updated, the target changes. This can cause instabilities in the training, especially when the model is updated very often. A simple solution to this is to use a target network, which is initialized with the same weights as the model at the beginning of the training. This target network is then periodically updated during training by copying the weights of the current model into it. This target network allows to make the training more stable, and is used by DQN.

The version of DQN used in the experiments is different from the real one, notably because RNNs have been used. This version is described in the experiments (chapter 5), while the next section describes the utilization of RNNs in RL.

4.4 Mixing reinforcement learning and recurrent neural networks

4.4.1 Motivations

At the beginning of this chapter, The distinction between stationary and non-stationary policies has been made. The action selected by a stationary policy only depends on the current state of the environment, while with a non-stationary policy, the action may depend on the entire history of the episode, i.e. the previous states and possibly the actions. The stationary policies are thus easier to use, and they are sufficient when working with fully-observable environments, because in such environments, the optimal action that should be taken is always the same for a given state. But sometimes all

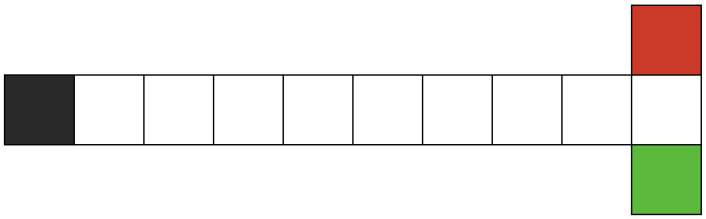


Figure 4.3: *T-maze* environment with a corridor length of 10. The agent is at the beginning of the corridor (black square) and the treasure if at the bottom of the junction (green square).

required information may not be present in the state (partially-observable environment), and it may be useful to retrieve some past information in order to choose the action (non-stationary policy). A naive solution is to keep the whole history, and to use a table to choose the action as in the beginning of Q-learning with the difference that this table is indexed not by current state but by history. Of course this is not tractable, the size of this table can explode with the length of the episode. To improve this, a maximum length of the history can be set, thus keeping only track of the last n steps. But long-term dependencies would be missed with this technique. A better solution would be to compress the past history into a fixed size buffer, and that is exactly what RNNs do. Thanks to BPTT, RNNs can learn what past information is important in the selection of the future actions, and they can thus encode such information in their hidden states.

The goal of this work is to test different types of RNNs on a RL environment with long-term dependencies, and to see how they handle these dependencies. This environment can now be introduced.

4.4.2 *T-maze* environment

The *T-maze*, created by Bakker [44], is a very simple RL environment, where all the difficulty lies in the ability of the agent to keep simple information for a very long time, without any stimulation. Its formalization is given in appendix A.1.

The *T-maze* consists of a horizontal corridor of squares ending with a T-junction, as it can be seen in figure 4.3. A treasure is placed either at the top or the bottom of the junction. The agent moves from square to square, it can thus move up, down, right or left. The goal of the agent is to reach the treasure.

Initially, the agent is at the beginning of the corridor and receives an indication telling it if the treasure is placed at the top of the junction or at its bottom. Once the agent moves right, the state changes to tell it that it is in the corridor. This state is the same no matter where is placed the treasure. As long as it is in the corridor, the state remains the same and at each step, it receives a penalty reward (-0.1). Once it reaches the junction, the state changes, indicating it that it has reached the end of the corridor. Once again, this state is the same no matter the position of the treasure. At this point the agent has to choose between going up or down. To make the good choice, it has to remember the

information it received at the beginning, thus making *T-maze* a long-term dependency environment. If the agent makes the good choice, it receives a good reward (equals to the length of the corridor). But if it makes a mistake, it receives a bad reward (-0.1). In both cases the episode is ended. The maximum non-discounted cumulative reward the agent can get is therefore equal to $l - (l-1) \cdot 0.1$, where l is the corridor length.

This environment is thus a partially-observable environment as it requires to retain an indication given at the very beginning of an episode. The goal of the RNNs is to maintain this information while the agent is going through the corridor. By playing on the corridor length, the range of the long-term dependency can be adjusted.

Some tests were also made on a custom variant of the *T-maze*, called the *Xor-T-maze*, whose formalization is given in appendix A.2. The difference is that the treasure is no longer uniquely determined by the information given at the beginning: indeed, the position of the treasure is determined by the combination of the initial indication and an other one given at some place in the corridor. For instance, if the two indications are the same, then the treasure is at the top of the junction, otherwise it is at the bottom. This environment allows to check the ability of RNNs to generalize with respect to the time at which is given an indication. Ideally, the RNNs behavior should not change when the indication is given later or earlier.

It can be important for an agent to be able to generalize, i.e. to adapt when its environment changes a little bit. The Meta-Reinforcement learning is a subtopic of RL and is related to these generalization abilities. As generalization will be discussed in the experiments, it is useful to get a brief insight of it.

4.5 Meta-Reinforcement learning

Meta-Reinforcement learning is part of a broader concept called Meta-learning, which is also known as *learning to learn*. Meta-learning [45] [46] is concerned with the development of ML algorithms that allows a model that was originally trained for a specific task to quickly adapt to a new task that has some similarities with the first one. This thus avoids retraining a model from scratch, but also decreases the quantity of data required to train on the new task. A good example [47] to understand the idea of Meta-learning is when a person wants to learn how to ride a motorcycle: if this person knows how to ride a bike, than he or she will learn it more quickly and more easily than if he or she does not.

Meta-RL [48] is thus the application of Meta-learning to RL environments. Meta-RL will not be formalize in details here, but its goal is to give the agent the ability to quickly adapt when their environment changes. In our case, these changes in the environment are the variations of the corridor length in the *T-maze* and of the moment the second indication is given in the *Xor-T-maze*. The goal is to see if a RNN pretrained on a *T-maze* with a small corridor length is able to play on a longer *T-maze*, or at least if it allows it to learn faster on longer corridors. In the experiments we will discussed about the *generalization capabilities* of the different agents that have been tested.

Chapter 5

GRU VS nBRC on *T-maze*

This chapter describes the first series of tests made with the GRU and nBRC recurrent cells (sections 3.2.2 and 3.2.3) on the *T-maze* and its variant the *Xor-T-maze* (section 4.4.2). The goal here is to analyze if the ability of nBRC to be multistable, i.e. having several stable states and switching from one to another, really helps the agent to perform well on these environments. This chapter first describes the experiment, then some observations on the experiment results will be discussed. To understand these results, an analysis of the hidden states of the RNNs is made. This chapter finally ends with some tests on the *Xor-T-maze*.

5.1 Experiment

5.1.1 Description

Initially, it was planned to first train some agents with both cells on the *T-maze* with a corridor length of 10, to see how they were doing on it, and then to retrain these agents on a longer *T-maze* to see if the nBRC agents have it easier to generalize on longer corridors. But as we will see in the next sections, it was not necessary to train on longer corridors to get great results, which was a nice surprise.

To measure the performance of the agents on the *T-maze*, the score used is the non-discounted cumulative reward. For instance, on a *T-maze* with a length of 10, the best score is 9.1, if the agent finds the treasure without making unnecessary moves. If the agent chooses the wrong side at the junction, it will get a score of -1 or less. To avoid never-ending episodes, a maximum number of timesteps has been set. However this number must be high enough to let the agent properly explore the environment. When training with a length of 10, this value has been set to 100, therefore if an agent timeouts, it will receive a score of -10, which is the worst achievable score. The most important is not that the agents achieve the best possible score, but that they remember the treasure position. This is easy to detect as it is the only way to get a positive score.

Concerning the neural network implementing the agents, it simply consists of a recurrent cell (either GRU or nBRC) followed by a fully-connected layer of 8 units and a ReLU activation. Its output layer is an other fully-connected layer of 4 units (one for each action) and a linear activation. This layer thus outputs the predictions of the

Q-values, one for each action. The algorithm used to train the agent is presented in the next section.

5.1.2 RL algorithm used

The training algorithm used in this experiment is a variant of DQN (section 4.3.2). Indeed, the original DQN has been created to train feed-forward networks, i.e. networks without recurrent connections. This type of network implements stationary policies, explaining why DQN stores transitions in its replay buffer. However, when dealing with RNNs, the entire episode must be retained, as important information can be given at a certain timestep but only be useful at a future timestep (which is obviously the case with the *T-maze*). If the network is only trained with transitions, it will be impossible to learn these time dependencies. Therefore, the algorithm used here retains entire episodes in its replay buffer. Also, when performing a gradient descent step to update the weights, the *features* are not anymore a list of states and their *labels* a list of Q-values. Instead, the *features* are a list of sequences of states, each sequence containing all the states observed during an episode, and their *labels* are a list of sequences of Q-values, each Q-value being associated with the state in the same position in the sequence. This allows the RNNs to learn the time dependencies. The Q-values are computed in the same way as with DQN.

As entire episodes have to be used to train the agent, it was decided to not train the agent very frequently like in DQN but to make bigger updates. Indeed, a training session consists of a certain number of iterations. In each iteration, some number of episodes is made to fill the replay buffer. After that, all the retained episodes are taken, and the samples are prepared like in DQN. Finally the agent is trained on these samples for some epochs. Note that no target network is used in this version as the updates are not very frequent.

Concerning the exploration, the ϵ-greedy exploration policy is also used here, with a varying ϵ. Indeed, ϵ decreases during the training by following an decreasing exponential function:

$$\epsilon(x) = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min})\ exp(-\alpha \cdot x)$$

where ϵ_{max} is the initial value of ϵ, ϵ_{min} the value toward which it converges and α the decay factor. In this algorithm, the ϵ value is updated at each new episode, and the parameters were chosen in order to start with an ϵ of 0.9 and to finish with about 0.05.

Now that the experiment has been correctly described, the results can be discussed. Note that all the technical details of the agents trainings are given in appendix B.1.

5.1.3 Observations during training

The figures 5.1 and 5.2 show the evolution of the score during the training of, respectively, 4 GRU agents and 4 nBRC agents on a *T-maze* with a corridor length of 10. There is no visible difference between the two types of cells, both converge easily towards the best score (9.1). It is possible to split these curves into 3 main phases. In the first one, the agent does not know how to reach a terminal state. It thus often timeouts and when it does not, it is thanks to the randomness of the ϵ-greedy policy. Once it has learned that it can reach a terminal state, it enters the second phase: it does not timeout anymore,

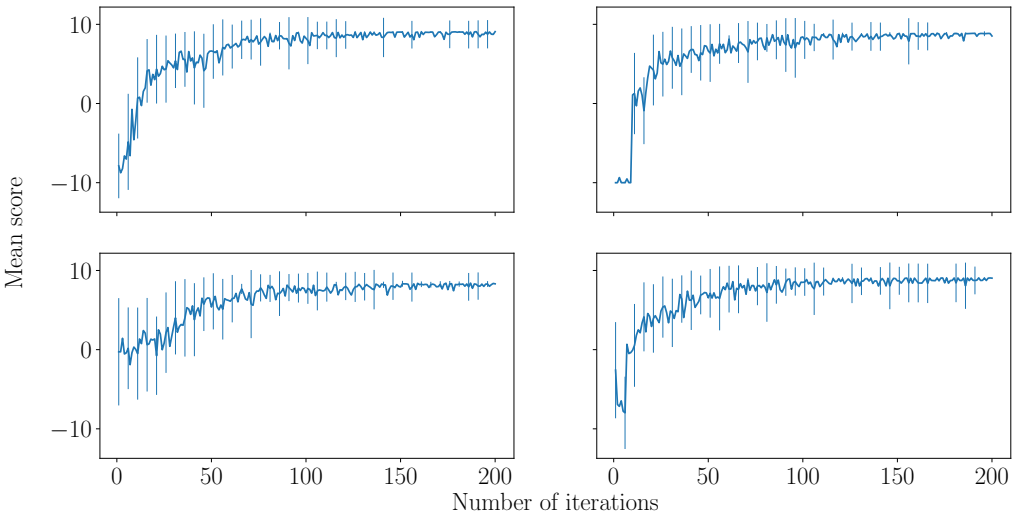


Figure 5.1: Evolution of the mean scores obtained by 4 GRU agents during the training episodes.

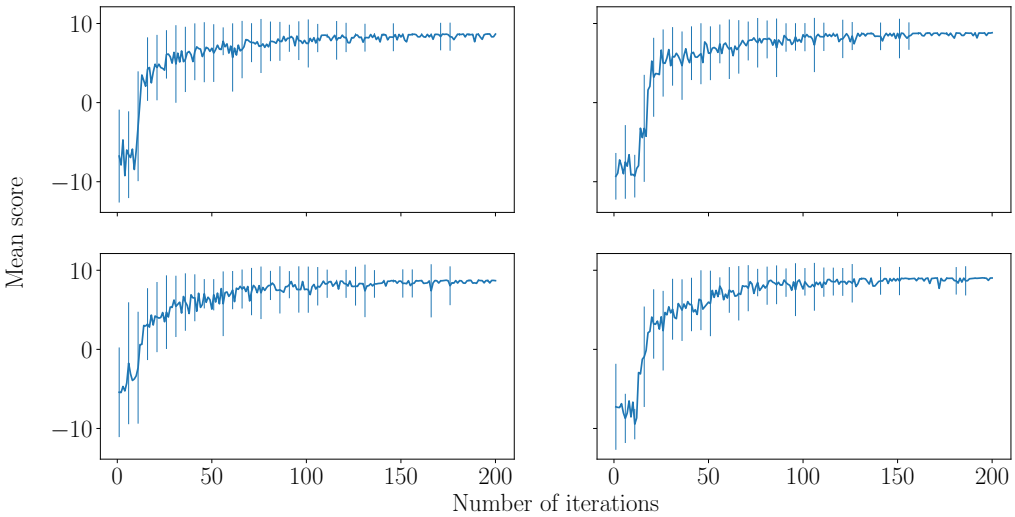


Figure 5.2: Evolution of the mean scores obtained by 4 nBRC agents during the training episodes.

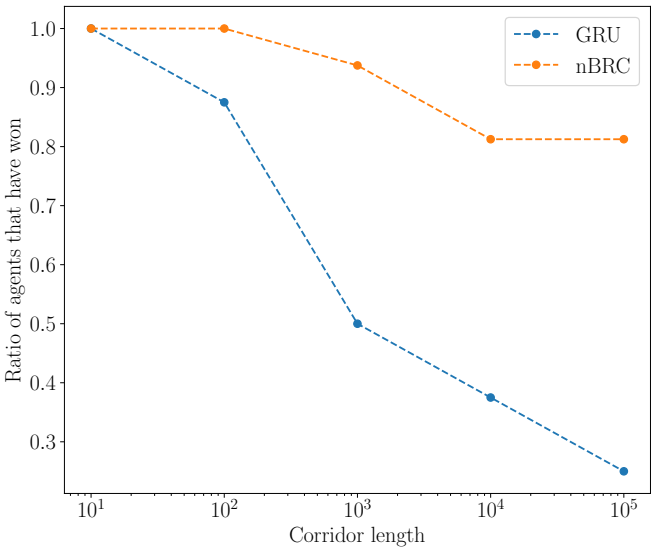


Figure 5.3: Evolution of the ratio of GRU and nBRC agents that always win in the *T-maze* environment to the total number of agents trained (16) with respect to the corridor length.

which increases its score. But it hasn't learned yet the relation between the initial state and the position of the treasure. Sometimes it wins, sometimes it loses. It finally reaches the third phase once it has learned this relation. In this phase, which is the longest, it always wins. Its goal in this last phase is to learn to win as quickly as possible, without making unnecessary actions.

As both cells achieved great scores on this environment without one being better than the other, we decided to test them on longer corridors without retraining them to see if they are able to generalize.

5.1.4 Test on longer *T-mazes*

The GRU and nBRC agents were thus tested on longer corridors with lengths of 100, 1000 and more and we observed that both are able to generalize. However, the GRU agents often start failing after a certain length, while the nBRC ones seem to be able to generalize to infinite lengths. To make more precise observations, a new test was run: 16 agents for both cells have been trained on a *T-maze* of length 10. Then these agents have been tested on several *T-mazes* whose lengths are powers of 10, starting from 10 to 100000. Figure 5.3 shows the results, i.e. the ratio of agents that managed to win no matter the treasure position with respect to the corridor length.

From this figure, several observations can be made:

- Firstly, both agents are able to generalize, as expected, but nBRC is clearly better.
- nBRC agents are almost always able to generalize to lengths of 100000 and probably

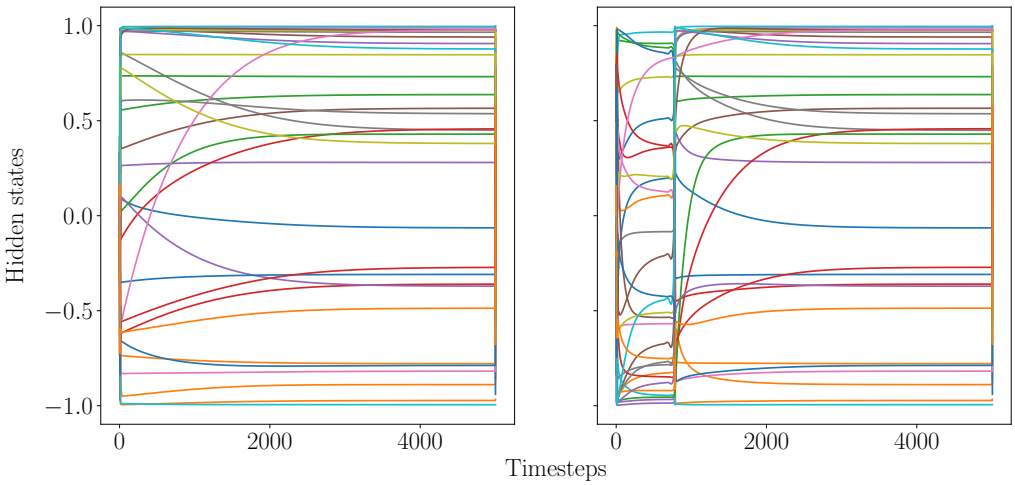


Figure 5.4: Evolution of the hidden states of a 32-units GRU agent on two *T-maze* environments with a corridor length of 5000, one with the treasure at the top of the junction, and the other with the treasure at the bottom. The agent loses when the treasure is at the bottom (right figure). Information is clearly lost around the 1000th timestep.

beyond. Is it due to the multistability property of nBRC ? This will be investigated in the next section.

- Finally, the GRU agents were not expected to be able to generalize to very long *T-mazes*. Indeed, as GRU is not supposed to be multistable, it should not be able to maintain its memory for a long period unless it has been trained for. This will be also investigated in the next section.

In this section have been obtained impressive and unexpected results: by only having been trained on a length of 10, some agents are able to generalize to lengths of 100000 and beyond. The next section aims at analyzing the hidden states of such agents to discover how they manage to do that.

5.2 Hidden states analysis

5.2.1 With 32 units in the recurrent cell

Firstly, a nBRC agent that generalizes correctly and a GRU agent that fails when the length becomes greater than a certain value (≈ 1000) have been selected. They have both played 2 episodes on a *T-maze* of length 5000, one with the treasure at the bottom, the other at the top. At each timestep, their hidden state has been added to a buffer. Their hidden states have finally been plotted in a graph with respect to the timesteps. The results are presented in figure 5.4 for the GRU agent and figure 5.5 for the nBRC agent.

Concerning the GRU agent, it wins when the treasure is at the top (left graph), but loses, i.e. chooses the wrong side, when the treasure is at the bottom (right graph),

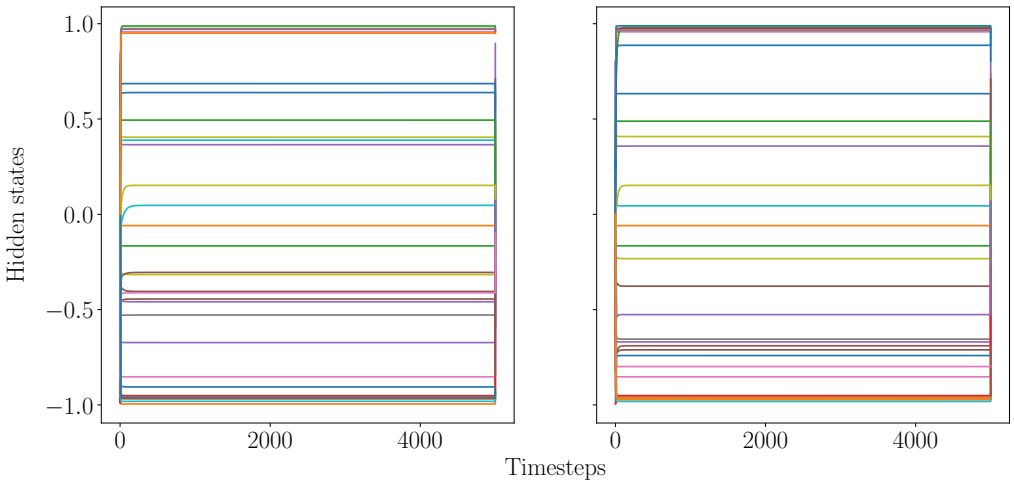


Figure 5.5: Evolution of the hidden states of a 32-units nBRC agent on two *T-maze* environments with a corridor length of 5000, one with the treasure at the top of the junction, and the other with the treasure at the bottom. The agent wins in both environments.

meaning that it has forgotten the initial indication. Previously it was said that such things can happen when the GRU cell is not stimulated for a long time: its hidden state then converges back to its unique stable state. This can clearly be observed in the right graph: around the 1000th timestep, the hidden state starts converging towards the same state as in the left graph. Its memory is thus fading. Once the end of the corridor is reached, the hidden state is the same in both graphs, it is thus impossible to differentiate the two cases, and the agent always chooses the same side (in this case the top).

On the other hand, the nBRC clearly converges towards different stable states in the two cases. A stable state being attractive, the hidden state will never leave it, and thus the agent will remember forever the initial indication.

The multistability of nBRC thus really helps in this type of environments. The GRU's memory relies on its dynamics [49][50][51]: To encode information, GRU moves its hidden state following a certain direction depending on the information. This is more flexible as it allows to encode easily several indications by moving in different directions. But this memory cannot survive without stimulation, it fades as it has been seen. To retain information for a longer period, GRU has to be trained. Unlike GRU, nBRC uses stable states to encode information: for each combination of indications is associated a stable state. For instance, in this case there is only one indication, the position of the treasure, it thus needs only two stable states. Its memory is less flexible, but there is no problem of fading memory. GRU will thus be preferred in tasks where there are a lot of information to encode but not for a long time, like in machine translation for instance where a RNN has to encode each word of a sentence. nBRC will be preferred in tasks where a punctual information has to be retained for a long time.

5.2.2 With 3 units in the recurrent cell

To further investigate these behaviors and to ease the observations, agents with less units in the recurrent cell have been trained: 2 units is apparently too little for the nBRC to be able to encode correctly the treasure position, but with 3 units it works. Agents with 3 units in the recurrent cell have thus been trained and the evolutions of their hidden states are shown in several figures: the figure 5.6 shows 2 GRU agents while the figure 5.7 shows 2 nBRC agents.

In the figure related to GRU, the bottom graphs show the same behavior as in figure 5.4: It is apparently the *orange* value of the hidden state that encodes the initial indication. When the indication is received, this value increases or decreases depending on which indication it is. But after a certain number of timesteps (about 15), it is observed in the right graph that the orange value starts increasing and converging towards the same value of the left graph, i.e. the stable state. Once again, the memory is fading, and when the agent reaches the junction, its hidden state is the same in both cases, it is thus impossible for it to always pick the correct side. However, what is happening in the top graphs is more surprising: it looks like GRU is able to become multistable. In these graphs, the corridor length is rather small, thus it may give an impression of multistability while it just converges very slowly. But this agent has been tested on much longer corridors and the observation stays the same: here GRU has become multistable. Actually, GRU can have this property but it is pretty rare. In section 3.2.3, when BRC and nBRC have been introduced, it has been shown how nBRC can be multistable: each of its units can be bistable, thus if it has n units, it can in theory have up to 2^n different stable states, where each unit retain a bit of information (cellular memory). Therefore the multistability of nBRC comes from the bistability of each of its units. However, the units of GRU do not have this bistability property, thus the GRU's multistability that is observed here cannot originate from cellular memory. In this same section, we proposed to see recurrent cells as closed-loop systems, with the hidden state acting like a feedback. It is known that a closed-loop system can become multistable by using positive feedbacks [52], and that is what GRU is doing here. But playing with positive feedbacks can be dangerous as it can lead to instabilities. That is why it is quite hard for GRU to become multistable. This will be more investigated in chapter 6.

Concerning the nBRC agents (figure 5.7), the one related to the top graphs behaves the same way as the one in figure 5.5: It encodes the initial indication by using two different stable states. By observing the evolution of the three values (*blue*, *orange* and *green*) of the hidden state, some things can be deduced:

- It seems that the *green* one is responsible for encoding the treasure position. In the next section we will see that it is possible to alter agent's memory by manually changing this value.
- The *orange* one is apparently used to detect the end of the corridor, as it drops very quickly when the junction is reached.
- The *blue* one has no particular usage. It is maybe used as a kind of *interneuron* that is used by the others.

The agent related to the bottom graphs timeouts when the treasure is at the top. It is

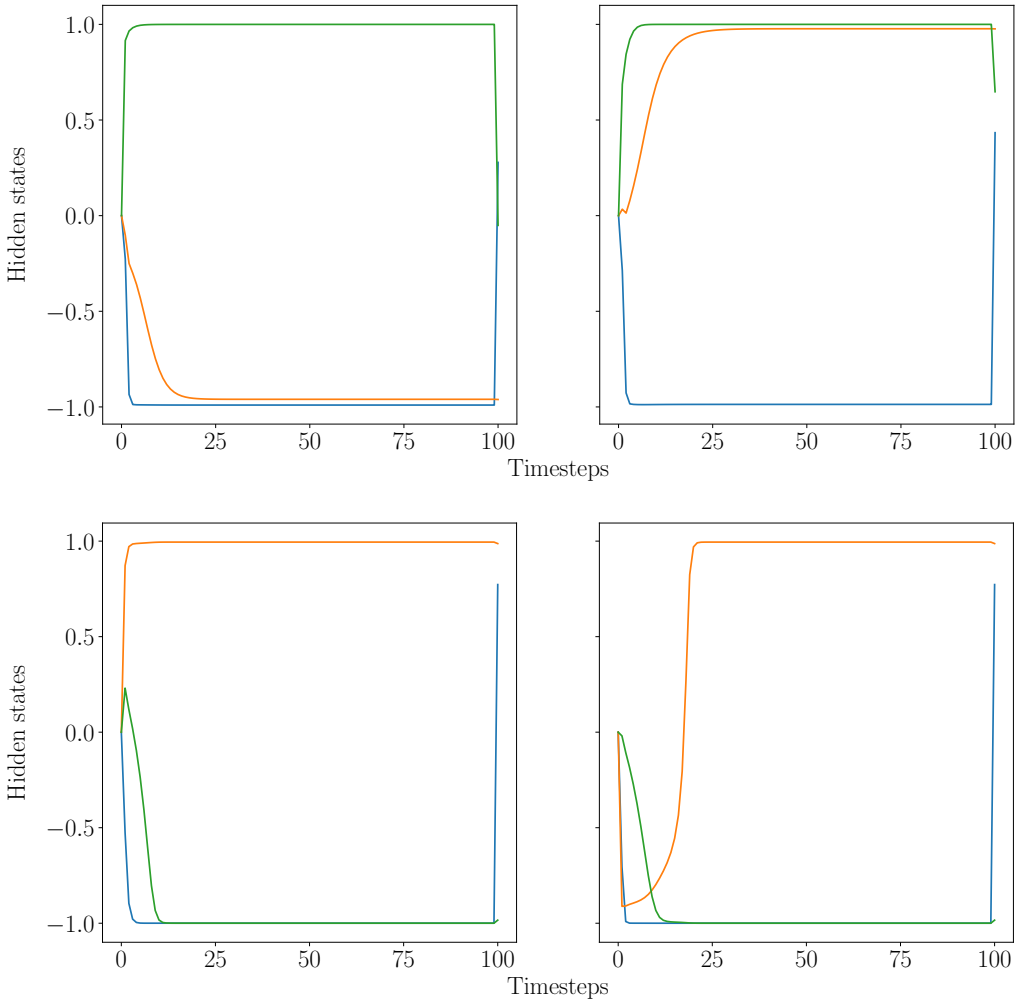


Figure 5.6: Evolution of the hidden states of two 3-units GRU agents on 2 *T-mazes* with a corridor length of 100 and the treasures at the bottom (left figures) or at the top (right figures). The agent related to the top figures wins in both environments, while the other one loses when the treasure is at the top (bottom-right figure).

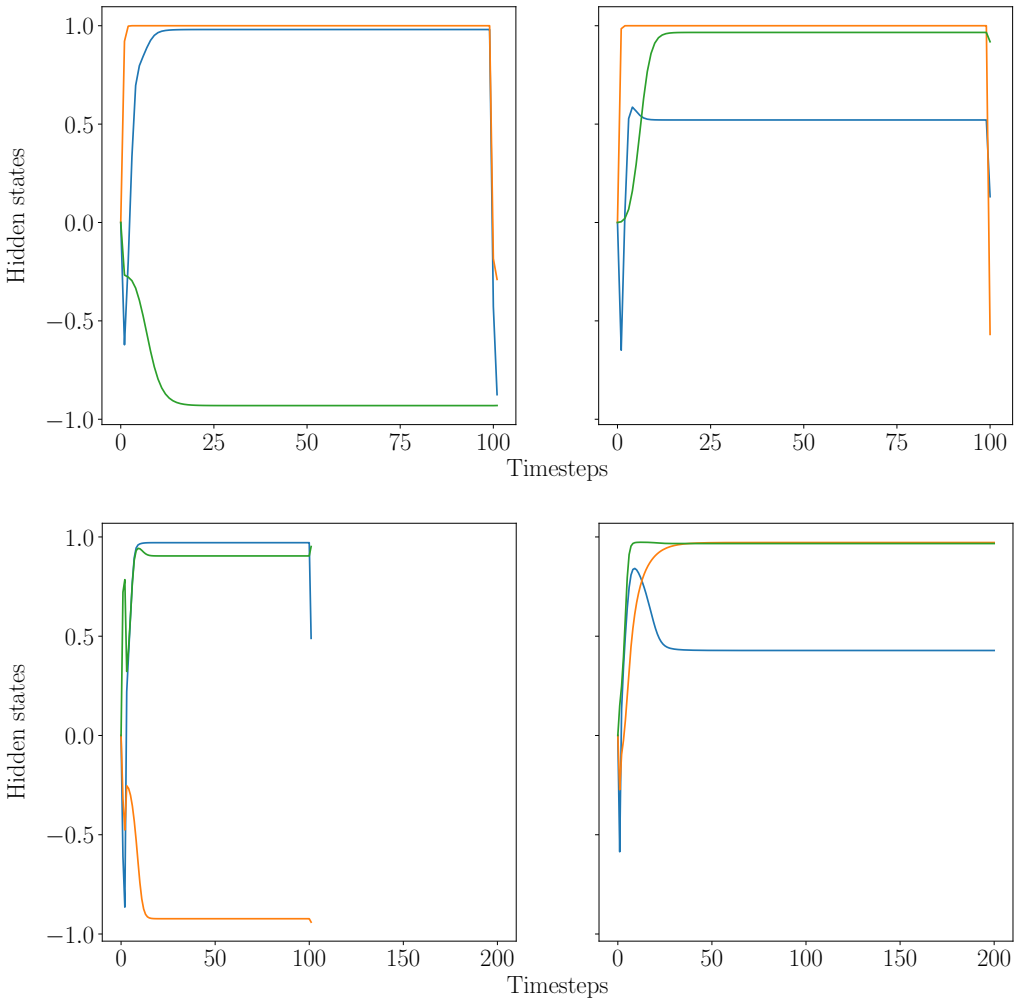


Figure 5.7: Evolution of the hidden states of two 3-units nBRC agents on 2 *T-mazes* with a corridor length of 100 and the treasures at the bottom (left figures) or at the top (right figures). The agent related to the top figures wins in both environments, while the other one timeouts (i.e. never reaches a terminal state) when the treasure is at the top (bottom-right figure).

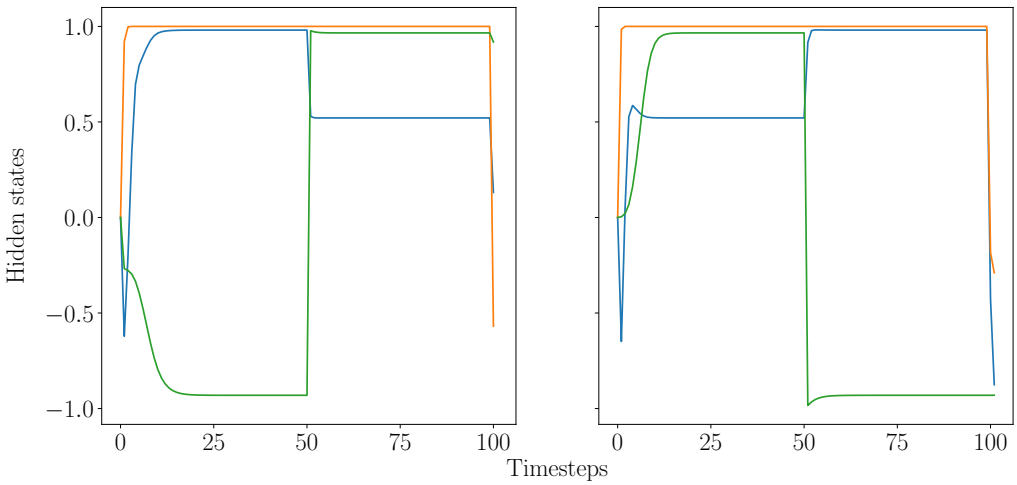


Figure 5.8: Evolution of the hidden states of a 3-units nBRC agent on two *T-mazes* with a corridor length of 100 and the treasures at the bottom (left figure) and at the top (right figure). In normal circumstances, the used agent wins no matter where the reward is, but here the hidden states have been manually changed at the 50th step. More precisely, the *green* hidden state has been set to 1 in the left figure and -1 in the right one. This makes the nBRC to change its current state from one stable state to the other. In both cases, the agent chooses the bad side at the junction, because its memory has been altered.

not the same problem as with GRU (i.e. fading memory) because here there are clearly two stable states. It is more probably due to the fact that when this agent was training on the T-maze of length 10, its hidden state did not finish converging towards the second stable state when the junction was reached. It has therefore not learned to use it to retain the information. When inspecting the action it wants to perform, we see that once it has reached this stable state, it starts always outputting the action "up", even if it has not reached the end. However this behavior is quite rare, because nBRC usually generalizes easily as it has been seen.

5.2.3 Altering memory by manually changing the hidden state

To confirm what has been said about the *green* value of the hidden state in figure 5.7, i.e. that it is responsible for encoding the treasure position, we decided to make a new test where this value is manually changed when the agent reaches the middle of the corridor. The goal is to see if changing this value causes the agent to pick the wrong side at the junction. When the treasure is at the bottom, the *green* value is close to -1, while when the treasure is at the top, this value is close to 1. We thus swap these 2 values when the agent reaches the middle of the corridor. The evolution of the hidden states during this test is given in figure 5.8.

It is easy to observe that, by changing this *green* value, the nBRC changes to the other stable state in both cases: the *orange* do not change because it is the same in both states, while the *green* and the *blue* both converge towards their value of the other stable state. Once it has finished converging, it is as if he had received the opposite indication at

the beginning. Indeed, when it eventually reaches the junction, it chooses the wrong side. It is quite rare to be able to understand how a neural network encodes information, they are often viewed as "black boxes". This result is thus quite nice, because we understand how the information is encoded and we know how to alter it.

When analyzing the hidden states of the 3-units nBRC, it was also observed that the *orange* value seemed to be used to trigger the agent when the junction was reached. Decreasing manually this value might therefore trigger the agent like if it was at the junction. This was tested but did not directly worked: in practice, the *blue* value has to be also decreased. When both are changed, the agent starts performing "up" or "down" actions (depending on which indication is encoded) until the two values have converged back to their stable value.

Note that the results here have been obtained with one specific agent, and may be different with others. For instance, we tested this with an other agent and this one only required one value to be changed for triggering the "end of corridor" action. Thus even if there could be differences between agents, the conclusion is that by changing the correct value(s) of the hidden state, it is possible to alter the memory of the agent or to make it believe that it has reached the junction.

5.3 Test on *Xor-T-maze*

5.3.1 Description

In the previous tests, nBRC has shown a quite impressive generalization capability thanks to its multistability property. By using different stable states to retain information, there is no notion of time (except for the small periods during which the hidden state converges): once the stable state reached, there is no more variation until the next perturbation. Therefore this perturbation may come at different moments, there will be no difference in the reaction of the agent. This is not the case with GRU when it uses its dynamics to retain the information, because as we have seen, its hidden state can vary even where there is no perturbation. Therefore the moment at which the next perturbation is received can have an impact on the reaction of the agent.

The *Xor-T-maze* allows to test whether or not the agent is impacted by the moment at which is received a perturbation by playing on the moment at which the second indication is given to the agent. Its behavior should obviously not depend on this moment, but only on the indication. This test consists of training nBRC agents on a *Xor-T-maze* of length 10 with the second indication given at the middle, and then to see if there are able to generalize both on the length of the corridor and on the position of the second indication.

5.3.2 Results

Several agents have been trained and then tested to see if they were able to generalize. Some of them were, but not all, like in the test on the *T-maze*. We have not performed an exhaustive test to get a precise value of the ratio of agents that generalize correctly, but it seems that they are less than in the *T-maze* test, which makes sense as it is harder here, but they are still frequent. One of these agents has been chosen to analyze how its hidden state evolves when it plays on this environment. Figure 5.9 shows the evolution of the

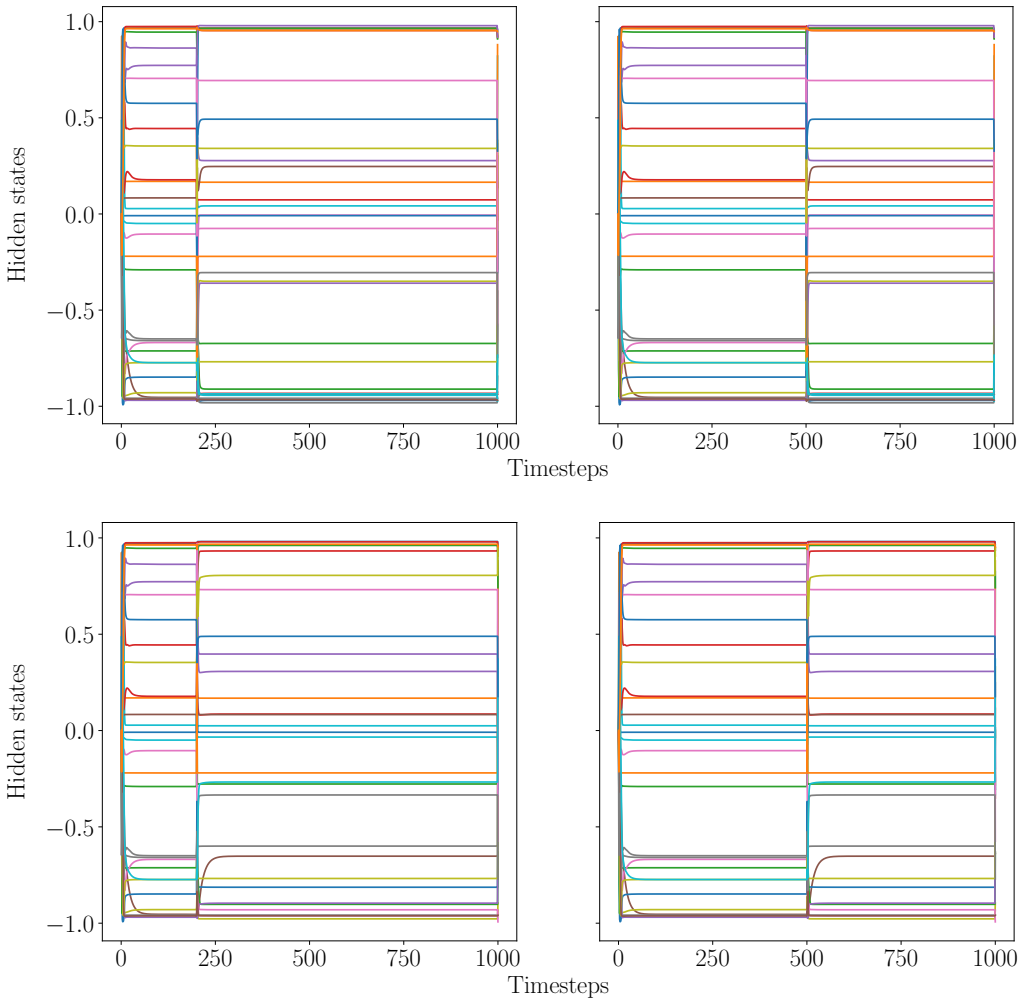


Figure 5.9: Evolution of the hidden state of a 32-units nBRC agent during 4 episodes on a *Xor-T-maze* of length 1000. In these 4 episodes, the initial indication is the same, only the second is different. In the top graphs, the treasure is at the bottom, therefore the second indication is different from the first one, while it is the opposite for the bottom graphs. Concerning the moment at which is given this second indication, it is given when the agent reaches the 200th square of the corridor for the left graphs and the 500th square for the right graphs.

hidden state of this agent during 4 episodes with small differences: the initial indication is always the same, but not the second one, which allows to change the treasure position (top graphs VS bottom graphs). The other difference is the moment at which is given the second indication (left graphs VS right graphs). Some observations can be made:

- Firstly, as the initial indication is the same for all episodes, there is no difference between the 4 graphs until the second indication.
- Then, we see that different second indications leads to different stable states, it is the same observation as the one made in the *T-maze* test. This encodes the treasure position.
- Finally, the position of the second indication has no impact on the agent's behavior, as it only results in a kind of "translation" in the graph.

As expected, nBRC's multistability allows to generalize on the time at which new information is received.

5.4 Conclusion of the experiment

In this chapter, we have made several experiments with the GRU and nBRC recurrent cells on the *T-maze* and *Xor-T-maze* environments. The first impressive result that we have obtained in the generalization capability of nBRC: by only having trained on a small *T-maze*, it is able to play on much longer *T-mazes*, thanks to its multistability property which prevents its memory from fading, i.e. time has no effect on it. We have also seen by testing nBRC on the *Xor-T-maze* that, thanks to this property, the moment at which information is given is not important. Finally, it has been observed that most of the GRU agents were unable to generalize to very long corridors. We saw however, by analyzing the hidden states of successful GRU agents, that they are able to develop multistability as well.

The next chapter focuses on this property of GRU, and introduces a pretraining algorithm specially made for forcing a recurrent cell to become multistable. Some tests are then made using this algorithm on GRU and other recurrent cells.

Chapter 6

Forcing the multistability

This chapter presents the *multistability warmup*, a pretraining algorithm that can be applied to any recurrent cell in order to train it to become multistable (if its equations allow it). This algorithm is then tested on the *T-maze* with different types of recurrent cells to see if it allows these cells to generalize like the nBRC.

6.1 Motivations

In the previous chapter, we have seen that nBRC was able to generalize easily on longer *T-mazes*. This is great because training on a long *T-maze* can take much more time than training on a short one because of the difficulty of finding the treasure when exploring such a big environment. Indeed, assume a *T-maze* of length 10000: once you know how it works, it is easy to solve it, but otherwise, if the actions are taken more or less at random, it is very unlikely that the terminal states will be reached. Furthermore, the agent has to understand that the indication that is given initially has an impact on the action it should take 10000 timesteps later or more. This also takes time because it is a very long time dependency. Thus, this generalization capability is very useful in this case.

We have also seen that GRU can become multistable, but this happens much less frequently than with nBRC, which is designed to be like that. Therefore, by designing an algorithm that is able to force GRU to become multistable, it should be possible to increase the ratio of GRU agents that can generalize. In practice, this algorithm should be applied first on the agent, to make it multistable, and then this agent should be trained on a small *T-maze*, hoping that it will learn to encode the treasure position like nBRC, i.e. by using different stable states. This is thus an alternative to a direct training on a long *T-maze*, which takes much more time. Note that the other recurrent cells, like LSTM or JANET for instance, can also become multistable by using the same technique as GRU (positive feedbacks). This *warmup* can thus be also applied to them and one goal of this experiment is to observe how each recurrent cell reacts to this algorithm.

6.2 Multistability warmup

6.2.1 Algorithm

The goal of this algorithm is to train a recurrent cell to become multistable, or at least to train it to increase its capacity to retain an information for a long period. A cell is multistable if, given different input trajectories, it converges towards different states when there is no perturbation for a long time. Therefore, we need to find a function such that the cell has to become multistable to maximize this function. The idea of this algorithm is to first feed the cell with the first states of different sequences. There is therefore one hidden state for each sequence. After that, the cell is fed with a fixed input for a certain number of timesteps. This fixed input is the same for all sequences. The goal here is to make each hidden state converge towards a stable state. Once this step is finished, the variance over the different hidden states is computed. As the hidden states are vectors, the variance will also be a vector, thus its mean is taken in order to have a scalar. If it is 0, it means that all the hidden states are the same, and thus there is only one stable state. If it is greater than 0, then the hidden states have converged towards at least two different states. By using gradient ascent, we can ask the cell to maximize this scalar. The cell will thus learn to maintain different hidden states when there is no perturbation for a long time. For this to be *real* multistability, it should be tested on infinite duration, but this is not feasible in practice. Therefore the cell may not really be multistable and yet have different states after a long but finite period. Indeed, by slowing down enough its dynamics, this can be done. This is one thing that we will observe when doing the experiment: does the cell become multistable or simply slow down its dynamics ? If it becomes multistable, then it can generalize like nBRC, which is the goal. Otherwise, this can still be useful: even if the cell is not able to generalize, it has still managed to increase a lot its memory duration. However, being multistable is not the only condition to be able to generalize: a cell may be multistable but may not encode the information using its multistability. This is one observation that we will made during the experiment: if the cell converges too slowly towards its stable state, then it may not have converged when reaching the junction of the *T-maze* during the training, and thus the agent will not learn to encode the indication with its stable states. This is the same problem that we have seen with nBRC in the bottom graph of figure 5.7, where the nBRC agent was multistable but was not able to generalize.

The algorithm 2 shows a general pseudocode for the multistability warmup. It takes as input the cell that has to be trained and a set that contains different sequences, which will be used as the first inputs to the cell. For instance, with the *T-maze*, these sequences begin with the initial indication followed by some "corridor" states. The algorithm consists of a certain number of iterations. At each iteration is done a gradient ascent step as explained above. The number of timesteps during which is applied the fixed input (also known as the warmup length) is take at random between 1 and an increasing upper limit. Indeed, it is too difficult for the cell to learn if it directly starts with a high warmup length. Therefore we start by asking it to maximize the variance on a small number of timesteps, and then increment this number little by little, until a maximum l_{max}.

When doing the experiment, the fixed input has been chosen to be the "corridor"

Algorithm 2: Multistability warmup

Given a recurrent cell ϕ of U units with weights θ
Given a set $\mathcal{S}$ containing sequences of the first K inputs to give to the cell
for *iteration* i *= 1 to* N **do**
  Sample a batch $\mathcal{B}$ of sequences from $\mathcal{S}$
  Initialize vector of hidden states h of length $|\mathcal{B}|$
  for *timestep* t *= 1 to* K **do**
    $h = \phi\big(\mathcal{B}[:, t], h; \theta\big)$ where $\mathcal{B}[:, t]$ is the vector containing the tth element of all the sequences contained in $\mathcal{B}$
  end
  Pick a random integer between 1 and $min(10 \cdot i, l_{max})$ and place it in l
  Create z, a vector where each element is the fixed input (for instance a zero input) and whose length is $|\mathcal{B}|$
  for *timestep* t *= 1 to* l **do**
    $h = \phi\big(z, h; \theta\big)$
  end
  Compute the mean variance of the hidden states contained in h, i.e.:
  $\mathcal{O} = \frac{1}{U} \sum_{u=1}^{U} V\{h[:, u]\}$, where $V\{h[:, u]\}$ is the variance over the uth value of the hidden states
  Perform a gradient ascent step on $\mathcal{O}$ with respect to θ
end

state, because it is the input that the recurrent cell will receive for a long period of time, thus it makes sense to maximize the variance on this input. Furthermore, we encountered a problem when applying this algorithm on JANET cells: unlike GRU, the hidden state's space of this cell is not bounded, therefore it is able to obtain an infinite variance between two states. And having infinite values is not good when training a neural network. We thus changed the objective function: instead of having to maximize the mean variance, the cell has to minimize the squared difference between the mean variance and 1. This does not change the goal of the algorithm but it prevents infinite variances.

6.2.2 Double layer model

Even if the multistability has a great advantage, i.e. creating never-fading memory, it has also an inconvenient as it has been discussed in 5.2.1: the memory it created is less flexible. A multistable cell is very good at retaining some information for a very long duration (which is perfect in our case), but it may perform badly in tasks that require to encode lots of different information for a small duration, while an usual recurrent cell will achieve very good results. To get the best of both worlds, we have designed an hybrid model, which is in a way inspired from the human brain and its long-term and short-term memories. This model is called the *double layer model*, and it consists of using two recurrent cells in parallel. One of these is pretrained with the warmup and must thus handle the long-term memory. The other one is not pretrained and takes care of the short-term memory. This model is represented in figure 6.1. We have tested it on the *T-maze*, to see if it was to perform as well as a model with a single recurrent cell

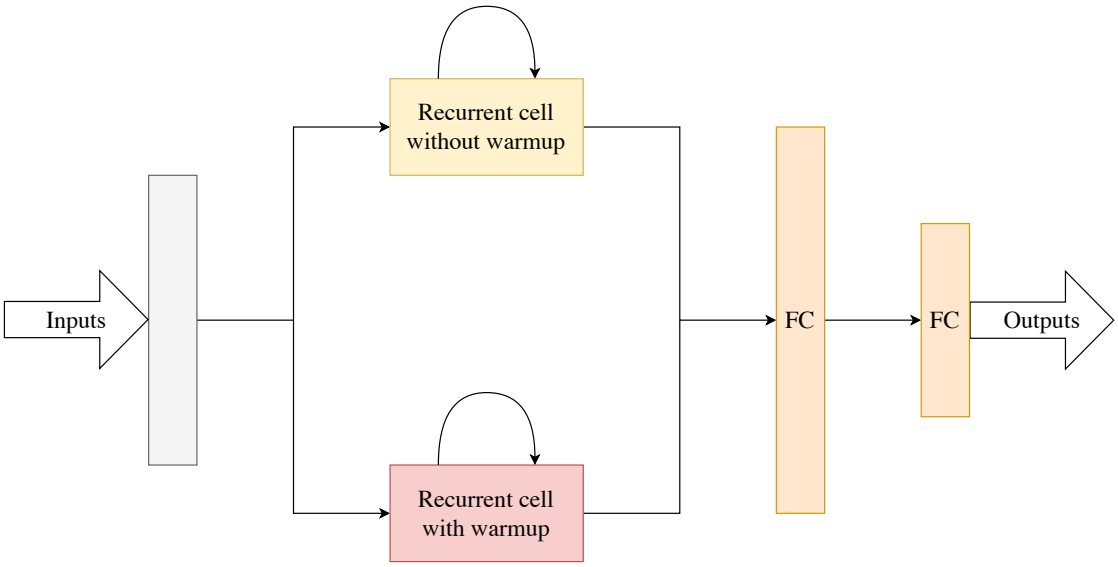


Figure 6.1: Double layer model used in the experiment. The inputs are passed to two recurrent cells including one that has been pretrained with the warmup. The outputs of these two cells are then concatenated and passed to two fully connected layers.

which has been pretrained.

6.3 Experiment

6.3.1 Description

The experiment made in this chapter is similar to the previous one: the agents are trained on a small *T-maze* (of length 10) and then they are tested on longer *T-mazes* to see if they are able to generalize and up to what length. The difference with the previous experiment is the addition of new types of recurrent cells (LSTM and JANET), the multistability warmup pretraining algorithm and the double layer model. To be more precise, for this experiment have been trained 16 agents for each of the following combinations: GRU with warmup, double GRU, LSTM without warmup, double LSTM, JANET without warmup and double JANET. Note that each cell is composed of 32 units and that *double GRU* refers to a double layer model with two GRU cells. Then each agent has been tested on longer *T-mazes* to compute the ratio of generalizable agents in order to complete figure 5.3. The values of the parameters used for training on the *T-maze* are the same as the previous experiment and are given in appendix B.1, while the values of the few warmup parameters are given in appendix B.2.

6.3.2 Observations during warmup and training

The first thing that we can observe is how each type of cells handles the warmup by looking at the evolution of the variance during the pretraining. Concerning the parameters of the warmup, it has been chosen to make 300 iterations and to have a maximum

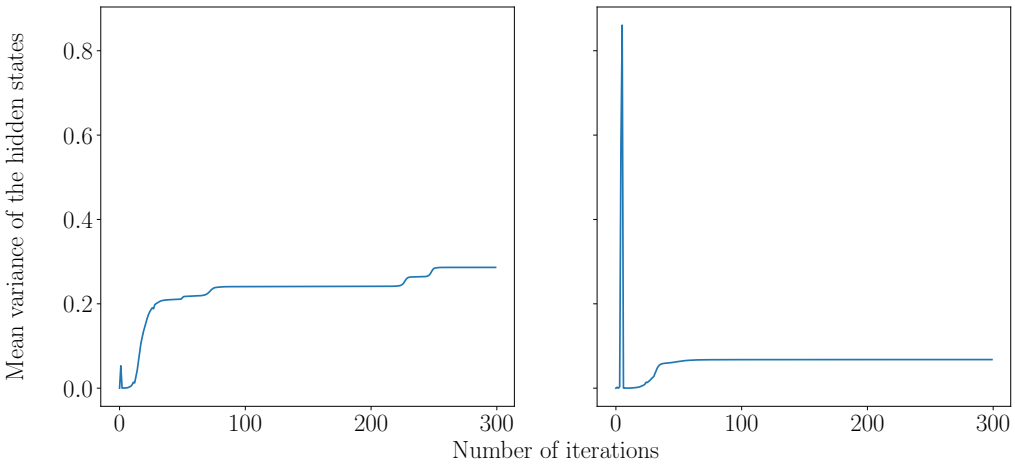


Figure 6.2: Evolution of the mean variance of the hidden states of two GRU cells during their warmup.

warmup length of 5000.

GRU is probably the cell which is the most stable during the warmup. As shown in the two graphs of figure 6.2, the mean variance of the hidden states is very stable: it usually makes a big jump at the beginning and then small jumps during the rest of the training. It also never decreases, or at least not significantly enough to be noticed. There can be pikes at the beginning of the warmup, this happens because the upper limit of the warmup length starts very low (at 10) and then increases up to the maximum warmup length. It is thus easy for the cells to maintain different hidden states on such small periods, which explains these pikes at the beginning.

Unlike GRU, LSTM does not handle well the warmup. From the three types of cells used here, it is the hardest to pretrain. The first problem is that LSTM sometimes cannot manage to increase its variance during the warmup, and this happens quite frequently. Because of that, many trainings of double LSTM agents had to be restarted, in order to only keep the ones where the warmup had an effect on. Furthermore, even when the cell has managed to increase its variance, it is often up to a small value. For instance, figure 6.3 shows the evolution of this variance during the warmup of the LSTM cells. The left graph shows what happens the most often: There are some pikes at the beginning, and then when it becomes stable, it stays at a very low value, which means that the states obtained at the end are very close to each other. This not ideal for keeping information for a long period of time. The right graph shows the warmup of a LSTM cell on which it worked pretty well: the cell has reached a high variance and then remains stable, which is the same behavior than GRU.

Finally, it seems that JANET is the easiest cell to warmup: it increases its variance very easily. But there are two problems with JANET: first, its hidden state space is not bounded, thus its variance can grow infinitely. If we keep the original objective function, i.e. that maximizes the variance, then the warmup can become unstable with JANET. But this is easily solved as it has been explained previously. The other problem is that the variance is very unstable during the training: thanks to the loss function, it converges

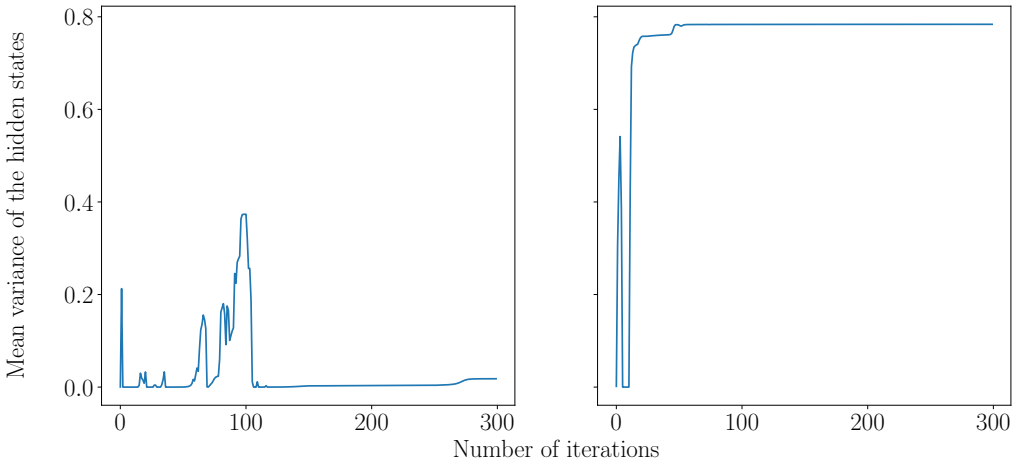


Figure 6.3: Evolution of the mean variance of the hidden states of two LSTM cells during their warmup.

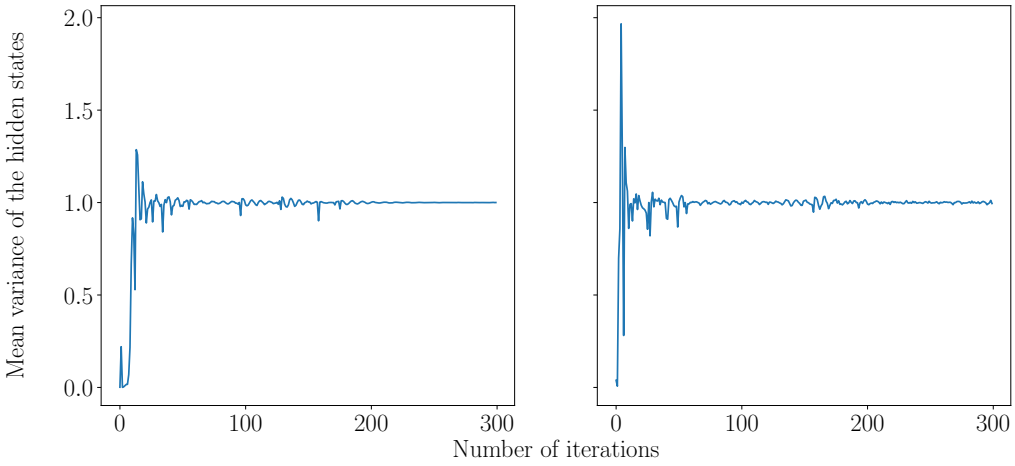


Figure 6.4: Evolution of the mean variance of the hidden states of two JANET cells during their warmup.

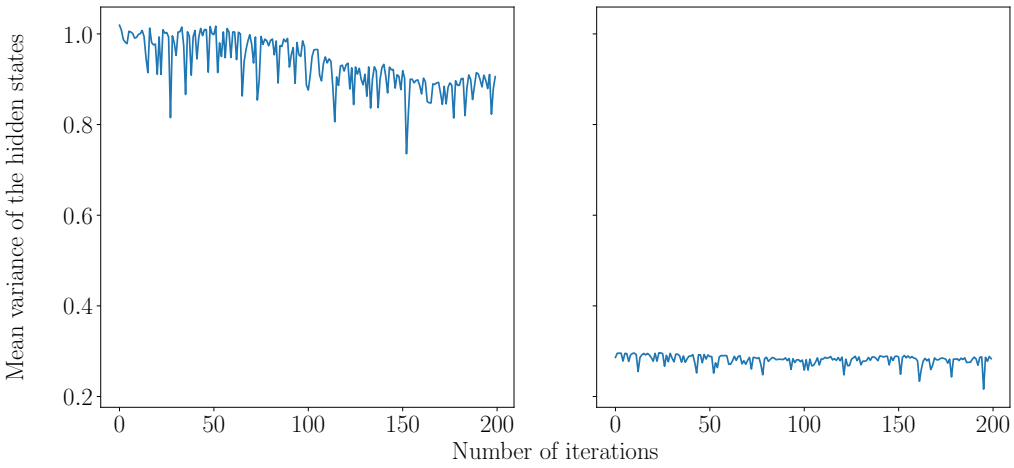


Figure 6.5: Evolution of the mean variance of the hidden states of a double JANET agent (left graph) and a double GRU agent (right graph) during the training on the small *T-maze*.

towards 1 and does not explode, but then it keeps oscillating around this value. This is very different from the behaviors of GRU and LSTM, which reach a value (not necessarily 1) and then remain quite stable.

The last thing that can be observed during the training is how the variance evolves while the agent is training on the *T-maze*. Indeed, once the agent has finished the warmup, it is supposed to have a non-zero variance. But it may loose it during its training. In practice, the agents rarely loose their variance during the training, no matter the type of their cells, which is great. Figure 6.5 shows the evolution of the variance during the trainings of a double JANET and a double GRU. The variances oscillate, they may increase or decrease but it is very rare to reach a variance of 0 during a training.

6.3.3 Test on longer *T-mazes*

Figure 6.6 shows the evolution of the ratios of agents that always win in the *T-maze* with respect to the length of the corridor for the different types of cells. Globally, it seems that the warmup algorithm is helping, as the green lines are above the blue ones. Also, the double layer model does not deteriorate the performance of the agents, as we can see in the figure by looking at the ratios of GRU with warmup (red line) and double GRU (green line with circles). By looking at each type of cell separately, we can make further observations. Firstly, GRU is the cell for which the warmup works the best. Indeed, as we can see in the graph, GRU with warmup and double GRU compete with nBRC. It seems that the warmup allows GRU to generalize more often, which is great. For the two other cells, the results are less perfect, but still good. As said previously, it is hard to train a LSTM cell with the warmup algorithm and its variance is usually not very high at the end of the warmup. When looking at the graph, we see that double LSTM performs better than LSTM, which means that warmup has helped it to increase its memory duration. However, it does not generalize as often as double GRU, as its ratio

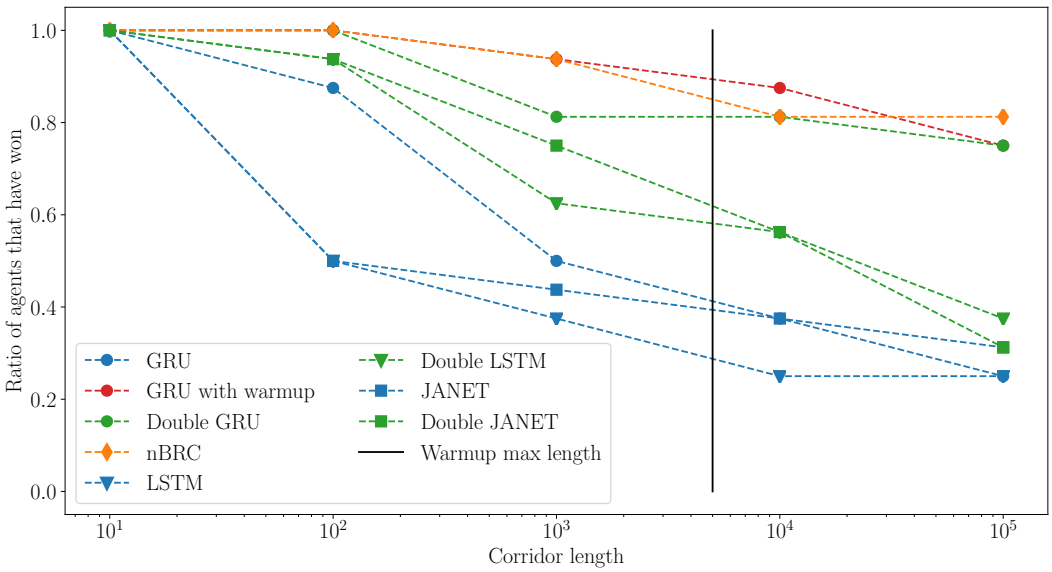


Figure 6.6: Evolution of the ratio of the agents that always win in the *T-maze* environment to the total number of agents trained (16) with respect to the corridor length. The circles are related to GRU, the diamonds to nBRC, the triangles to LSTM and the squares to JANET. The green lines correspond to the double layer models, while the blue ones correspond to the simple layer version with no warmup.

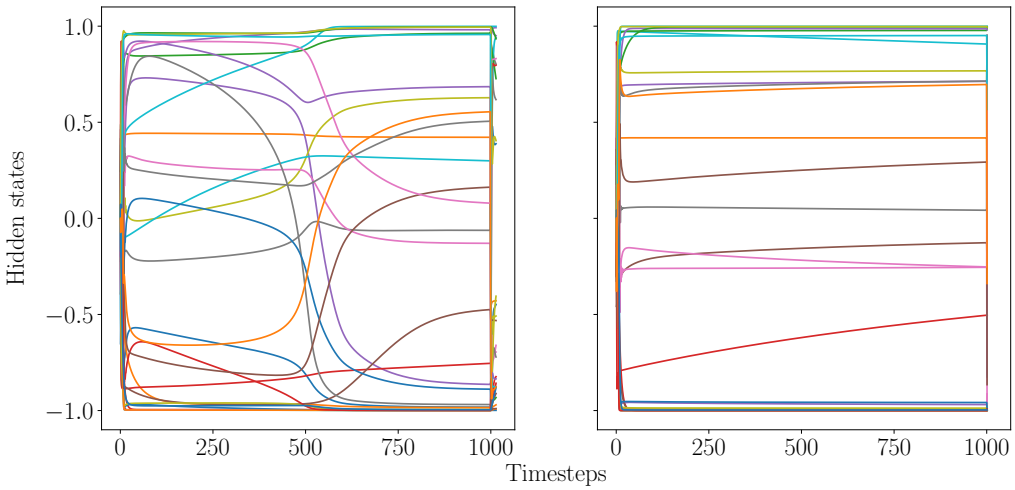
for the last length is quite low. It seems thus that the warmup has a positive effect on LSTM but not as significant as on GRU. Concerning the last cell, JANET, we have seen previously that it is very easy for this cell to increase its variance during the warmup, which suggests that it does not really become multistable but more likely that it can very easily adapt its dynamic to increase the memory duration. It is indeed what we observe in the graph: its results are significantly better for lengths smaller than the warmup max length, but once this length is passed, the results of double JANET drops, it is not able to generalize correctly.

6.4 Hidden states analysis

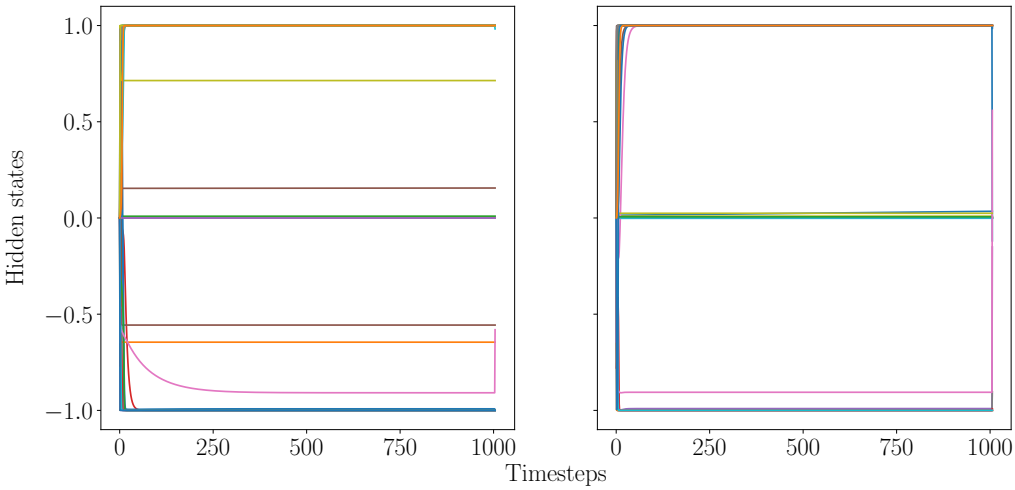
During the experiment, we have observed the behavior of the hidden states for each type of agents while they were playing on the *T-maze*. This section aims at analyzing what we have observed, in order to see how the warmup algorithm affects the behavior of the hidden states. As each type of cell seems to have a different reaction, we will look separately at each one.

6.4.1 GRU

Concerning GRU, we have already looked at some examples in section 5.2. We saw that in most of the cases, GRU only had one stable state, and thus after a certain period, its hidden state had to converge towards it, no matter the initial indication it had received.



(a) Simple GRU agent, loses when treasure is at the bottom (left graph).



(b) Double GRU agent, wins in both cases.

Figure 6.7: Evolution of the hidden states of a simple GRU agent and a double GRU agent.

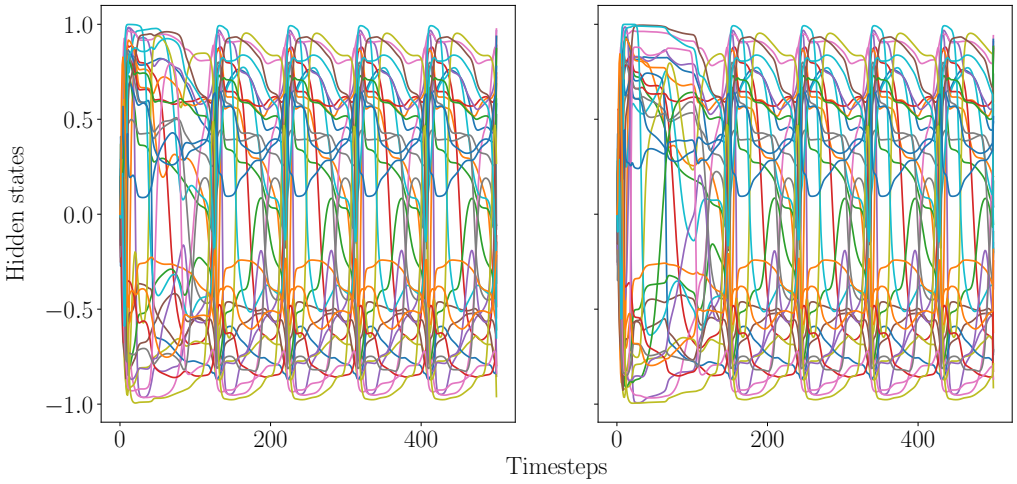
But we also saw that in some rare cases, GRU was able to become multistable, and thus to retain the treasure position for an infinite duration. The goal of the warmup was thus to force GRU to become multistable, in order to increase the ratio of multistable agents. In practice, this is exactly what happened, as we have seen in figure 6.6. Figure 6.7 shows the evolution of the hidden states for a simple GRU agent (with no warmup) and for a double GRU agent. Note that, for the double agent, the hidden states of the cell that has been warmed up are shown, not the hidden states of the second cell. For the simple agent, we see the same behavior as previously: after a certain point, the hidden state starts converging towards an unique stable state, no matter the initial state. Even if the states reached at the end in these two graphs are not exactly the same, the agent still chooses the same side (the top in this case) no matter the initial indication. On the other hand, the double agent shows *very clean* stable states: its hidden state converges very quickly towards a stable state which depends on the initial indication, and then stay constant. However, it may possible that some values of the hidden state still vary. For instance in the right graph, one of the values (the blue one) that are close to 0 tends to deviate from the 0 axis. It is not a problem, as long as enough other values stay constant.

6.4.2 LSTM

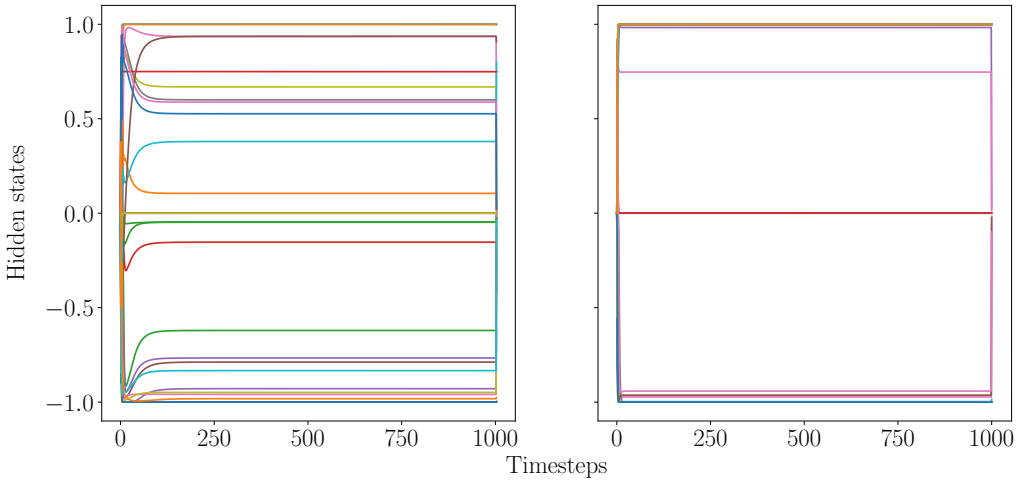
The figure 6.8 shows the hidden states of a simple LSTM agent and a double LSTM agent. It seems that when LSTM is not pretrained, its hidden state does not converge towards a stable state but instead starts to make a cycle (top graphs). Nevertheless, it seems that the same conclusion than the one made for GRU can be made here: after a certain point, the hidden state starts to make the same cycle no matter the initial indication, and thus the agent always chooses the same side (here bottom). We have seen previously that warmup works less often on LSTM than on GRU. We thus chose a double agent which was able to generalize correctly to see if its hidden state behaves the same way as the ones of a GRU that can generalize. The bottom graphs of figure 6.8 show the evolution of the hidden state of this agent. It behaves in the same as GRU's: it converges towards a stable state that depends on the initial indication, and then stay there. Therefore the warmup creates a big change here as it makes LSTM switching from a cycle to multiple stable states. Maybe this transition is hard to make, which may explain why the warmup less often works with LSTM.

6.4.3 JANET

Finally, JANET is the hardest type of cell to characterize, because it can have several behaviors and it is difficult to see the effect of the warmup. When there is no warmup, it seems that JANET can behave like GRU, with a single stable state, or like LSTM with a cycle, which is the case in the top graphs of figure 6.9. In the right graph, the hidden state varies a lot before reaching the cycle. Knowing that when a no-warmup agent is trained on the *T-maze* of length 10, the length of the longest sequence it may have to handle is 100, it is quite surprising to see such big variation in the hidden states around the 400th timestep. A similar observation can be made with the double agent and the bottom graphs of figure 6.9: during the training of a warmup agent, it will never have to handle sequences of more than 5000 timesteps, as 5000 is the maximum warmup length.

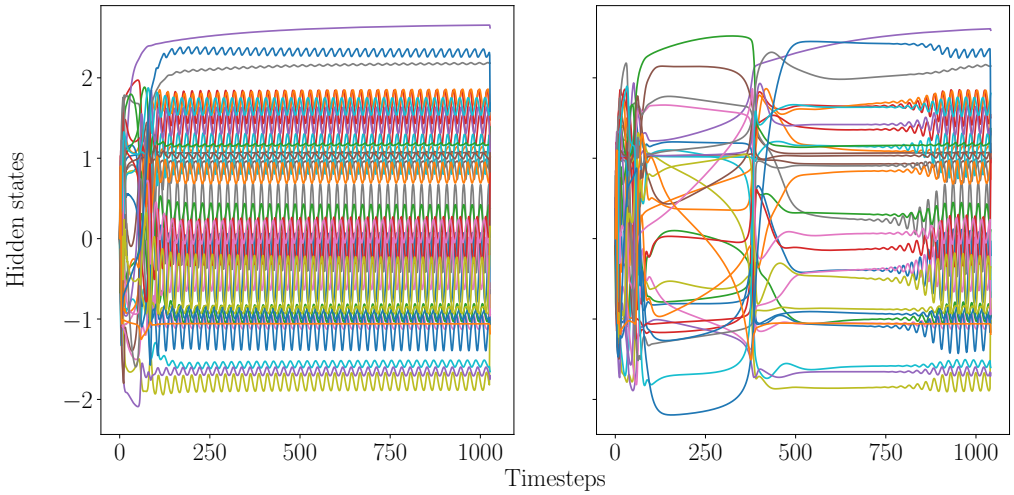


(a) Simple LSTM agent, loses when treasure is at the top (right graph).

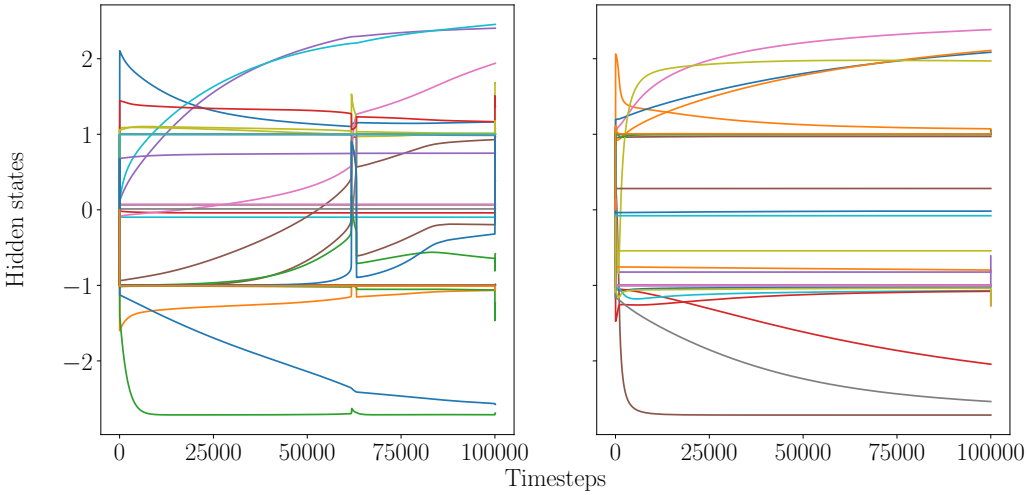


(b) Double LSTM agent, wins in both cases.

Figure 6.8: Evolution of the hidden states of a simple LSTM agent and a double LSTM agent.



(a) Simple JANET agent, loses when the treasure is at the bottom (left graph).



(b) Double JANET agent, loses when the treasure is at the bottom (left graph).

Figure 6.9: Evolution of the hidden states of a simple JANET agent and a double JANET agent.

But in the left graph we see a big variation in the hidden state around the 65000th step. Thus it seems that JANET can very easily adapt its dynamic to switch from short-term memory to long-term memory. Furthermore, it seems that some values of the hidden state converge towards different stable values depending on the treasure position, thus it could be possible to use them to retain the position of the treasure for an infinite period. However, this agent fails when the length of the *T-maze* is 100000: it chooses the wrong side when the treasure is at the bottom, thus it probably based its decision on values of the hidden states that are not stable. In other words, the network ends up being multistable, but converges too slowly compared to the length of the training T-Maze for the network to use those states as fixed points. Indeed, it is of course necessary to be able to distinguish the hidden state obtained at the junction when the treasure is at the top and the one when the treasure is at the bottom in order to select the correct action. But it is not the unique condition: when the agent is trained, it learns how to choose the correct action based on its hidden state. But the hidden state obtained at the junction may not be stable, and when testing on longer corridors, it may continue to vary, thus reaching new values never seen by the agent. It is probable that the action selected by the agent based on this new hidden state may not be the correct one. When the agent has learned to select its actions based on stable states, this problem does not arise, as the hidden state will be the same at the 100th and the 100000th timesteps. The hidden state being the same, the action will also be the same. We come to the same conclusion as in the tests on *Xor-T-maze* (section 5.3): the *multistability-based* memory's ability of being completely insensible to time is very useful when an agent is expected to generalize on longer sequences.

6.5 Conclusion of the experiment

In this chapter we have tested a new pretraining algorithm called the multistability warmup, which, when applied to a recurrent cell, is supposed to increase its memory duration and even forces it to become multistable. A new model, the double layer model, has also been introduced. Some tests with several types of recurrent cells have been made on the *T-maze* to analyze the effects of the warmup, i.e. to see if it allows the agents to better generalize on longer *T-mazes*. We have observed that warmup improves the performance of agents for all the types of cells, but not all cells become multistable. The best result has been obtained with GRU, which, when it is pretrained, becomes much more often multistable.

Furthermore, it is important to note that even if some cells did become multistable with warmup, they however had too slow dynamics for the length of the training *T-maze*: their stable states were not reached at the end of the corridor during training, preventing therefore the agent from learning on its stable states. This asks the question as to how this behavior could be avoided, and in particular, how can we make sure that fixed states are reached fast enough. An idea would be to compute variance of the state through-time per sample and to ask the cell to minimize it during the warmup. On one hand this would avoid LSTM cycles, and on the other hand this could force the network to speed up convergence, depending on the range of timesteps on which the variance is computed. This new warmup algorithm would thus ask two things to the recurrent cells:

- maximize the variance between the hidden states obtained after a certain number of timesteps in order to have different stable states, and
- minimize the variance between the different values taken by each hidden state through the time in order to force the cell to converge as quickly as possible to a stable state.

Chapter 7

Conclusion and perspectives

This thesis, located at the junction of two domains of Machine learning, i.e. Recurrent neural networks and Reinforcement learning, studies the benefits of using a *multistability-based* memory in RL environments that require a long-lasting memory. Indeed, a RNN's memory usually relies on its dynamics: information is like a perturbation which moves away the RNN's state from its stable state, which creates memory. The problem is that this memory can fade: if it is not stimulated periodically, the state converges back to the stable one and the RNN forgets. When using multistability to encode information, a RNN can learn to switch to a different stable state when receiving a particular indication. The RNN will never converge back to the previous stable state, thus the information will never been forgotten (never-fading memory). We have introduced two recently created recurrent cells, namely the BRC and the nBRC [27], that have been built to retain information using multistability. Two experiments have then been made in the context of this thesis on two RL environments, the *T-maze* [44] and a variant of it, the *Xor-T-maze*, which both need agents that are able to keep simple information for a long time.

In the first experiment, we highlighted the advantage of using a *multistability-based* memory by comparing the performance of the well-known recurrent cell GRU [26] and the new nBRC on the *T-maze*. We trained agents containing either a GRU or a nBRC on a *T-maze* of length 10, and then these agents have been tested on longer *T-mazes* without any further training. The nBRC agents showed an impressive generalization capability by being able to play on very long *T-mazes*, while most of GRU agents failed to. By analyzing the behavior of the nBRC agents, we confirmed that their multistability was the origin of this generalization capability. This property allows their memory to not be affected by time, and to remain intact as long as needed. Furthermore, by testing nBRC on the *Xor-T-maze*, we observed that this cell was able to generalize on the moment at which is given the information, which once again showed that time has no impact on a *multistability-based* memory. Finally, we discovered that GRU sometimes become multistable, but this does not happen frequently. This is why we tried to find a way to force GRU to become much more often multistable.

For the second experiment, we introduced a new pretraining algorithm, called the *multistability warmup*. Its goal is to train a recurrent cell to become multistable, with the aim that, when the agent will be trained on the *T-maze*, it will use its multistability

to encode information. A new model, the *double layer* model, was also presented. It uses two recurrent cells in parallel, one of which being warmed up. This cell is supposed to handle long-term memory, while the other one the short-term memory. We made the same test as in the first experiment, i.e. training on a small *T-maze* and then testing on longer ones, using different cells to observe the effects of the warmup and the double layer model. The first observation we made is that the double layer model performs as well as a model which contains a unique warmed up cell but not better, which suggests that the short-term memory layer is not needed in this task. Then we saw that the warmup pretraining improves the results for each type of cell. The best result was obtained with GRU which, when it is warmed up, can compete with nBRC in term of generalization capability. For the two other cells (LSTM [25] and JANET [29]), we saw some improvements when using the warmup, but not as significant as with GRU. This hints that the pretraining algorithm could be improved and gives a path for future works.

Concerning the perspectives, on one hand, we saw that *multistability-based* memory allows some kind of generalization because it does not fade away with time. This type of memory learns to treat information, no matter when it is received. We have observed this by playing on a simple environment, but it may be more interesting to see what multistability can offer in more complex environments. On the other hand, the multistability warmup has shown great results by allowing cells to generalize much more often. However it does not work very well for each type of cell, thus it may be improved. One problem we observed during the second experiment is that in some cases, while the cell succeeds in becoming multistable during the warmup, it converges slowly towards the stable states. Therefore, when trained on the small *T-maze*, the network is always in transition regime. Thus, the network never learns to use its stable states, but rather only the very slow transient dynamics to encode its policy. This leads the network to failing on longer T-mazes, despite having its multistability property obtained through warmup. One solution to this problem would be to train the cell to converge as quickly as possible towards one of the stable states, as discussed at the end of the second experiment.

Bibliography

[1] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: from theory to algorithms*. Cambridge University Press, 2014.

[2] Tom Michael Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[3] Sotiris Kotsiantis, Dimitris Kanellopoulos, and P. Pintelas. Data preprocessing for supervised learning. *International Journal of Computer Science*, 1:111–117, 01 2006.

[4] A. Famili, Wei-Min Shen, Richard Weber, and Evangelos Simoudis. Data preprocessing and intelligent data analysis. *Intelligent Data Analysis*, 1(1):3–23, 1997.

[5] Lakshaysuri. Machine learning: Supervised vs unsupervised learning. lakshaysuri.wordpress.com, Mar 2017.

[6] Alexis Conneau, Guillaume Lample, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *CoRR*, abs/1710.04087, 2017.

[7] Qizhe Xie, Eduard H. Hovy, Minh-Thang Luong, and Quoc V. Le. Self-training with noisy student improves imagenet classification. *CoRR*, abs/1911.04252, 2019.

[8] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. *CoRR*, abs/1511.06709, 2015.

[9] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction, Second edition*. The MIT Press, 2015.

[10] 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.

[11] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. How to train your robot with deep reinforcement learning; lessons we've learned. *CoRR*, abs/2102.02915, 2021.

[12] Yingce Xia, Di He, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. *CoRR*, abs/1611.00179, 2016.

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[14] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. *Classification and Regression Trees*. Taylor & Francis, 1984.

[15] Evelyn Fix and J. L. Hodges. Discriminatory analysis. nonparametric discrimination: Consistency properties. *International Statistical Review / Revue Internationale de Statistique*, 57(3):238, December 1989.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[17] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[18] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[20] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In David A. Forsyth, Joseph L. Mundy, Vito Di Gesù, and Roberto Cipolla, editors, *Shape, Contour and Grouping in Computer Vision*, volume 1681 of *Lecture Notes in Computer Science*, page 319. Springer, 1999.

[21] Marvin Lee Minsky and Seymour Papert. *Perceptrons*. MIT Pr., 1969.

[22] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1987.

[23] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[24] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[25] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[26] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[27] Nicolas Vecoven, Damien Ernst, and Guillaume Drion. A bio-inspired bistable recurrent cell allows for long-lasting memory. *CoRR*, abs/2006.05252, 2020.

[28] E Marder, LF Abbott, GG Turrigiano, Z Liu, and J Golowasch. Memory from the dynamics of intrinsic membrane currents. *Proceedings of the National Academy of Sciences of the United States of America*, 93(24):13481—13486, November 1996.

[29] Jos van der Westhuizen and Joan Lasenby. The unreasonable effectiveness of the forget gate. *CoRR*, abs/1804.04849, 2018.

[30] Corentin Tallec and Yann Ollivier. Can recurrent neural networks warp time? *CoRR*, abs/1804.11188, 2018.

[31] Aaron Voelker, Ivana Kajić, and Chris Eliasmith. Legendre memory units: Continuous-time representation in recurrent neural networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[32] Sergio Dorado-Rojas, Bhanukiran Vinzamuri, and Luigi Vanfretti. Orthogonal laguerre recurrent neural networks, 12 2020.

[33] Li Jing, Çaglar Gülçehre, John Peurifoy, Yichen Shen, Max Tegmark, Marin Soljacic, and Yoshua Bengio. Gated orthogonal recurrent units: On learning to forget. *CoRR*, abs/1706.02761, 2017.

[34] OpenAI. spinningup.openai.com.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[36] David Ha and Jürgen Schmidhuber. World models. *CoRR*, abs/1803.10122, 2018.

[37] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, May 1992.

[38] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2015.

[39] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

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

[41] Damien Ernst. Optimal decision making for complex problems, 2021.

[42] Damien Ernst, Pierre Geurts, and Louis Wehenkel. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6(18):503–556, 2005.

[43] 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.

[44] Bram Bakker. Reinforcement learning with long short-term memory. *Neural Information Processing Systems*, jan 2002.

[45] J Schmidhuber, Jieyu Zhao, and Marco Wiering. Simple principles of metalearning, feb 1999.

[46] Timothy M. Hospedales, Antreas Antoniou, Paul Micaelli, and Amos J. Storkey. Meta-learning in neural networks: A survey. *CoRR*, abs/2004.05439, 2020.

[47] Lilian Weng. Meta-learning: Learning to learn fast. lilianweng.github.io, Nov 2018.

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

[49] David Sussillo and Omri Barak. Opening the black box: Low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural Computation*, 25(3):626–649, March 2013.

[50] Andrea Ceni, Peter Ashwin, and Lorenzo Livi. Interpreting recurrent neural networks behaviour via excitable network attractors. *Cognitive Computation*, 12(2):330–356, March 2019.

[51] Niru Maheswaranathan, Alex H Williams, Matthew D Golub, Surya Ganguli, and David Sussillo. Reverse engineering recurrent networks for sentiment classification reveals line attractor dynamics. *Advances in neural information processing systems*, 32:15696—15705, December 2019.

[52] R. Sepulchre, G. Drion, and A. Franci. Control across scales by positive and negative feedback. *Annual Review of Control, Robotics, and Autonomous Systems*, 2(1):89–113, May 2019.

Appendices

Appendix A

Environments formalization

A.1 *T-maze*

Here is formalized the *T-maze* RL environment, which is used in the experiments. Note that a *T-maze* environment has a fixed corridor length L. It is deterministic: given its current state x_t and the action a_t performed by the agent, the next state x_{t+1} and the received reward r_t will always be the same. Furthermore, it is a partially observable environment: the agent does not have accessed to the real state x_t of the environment, but rather an observation of it o_t, computed with the observation function.

- **State space** X: the current state of a *T-maze* is entirely determined by $x = (p, i, b)$ with $p \in \{true, false\}$, $i \in [1, L+2]$ and $b \in \{true, false\}$:
  - p is a boolean which is *true* if the treasure is at the top and $false$ otherwise.
  - i the index of the square on which is currently located the agent. if $i \in [1, L]$, then the agent is on the ith square of the corridor. If $i = L + 1$ then it is on the top square, while if $i = L + 2$ then it is on the bottom square.
  - b is a boolean which is set to *true* once the agent has left the first square of the corridor. It is used in the observation function.
- **Observation space** O:

  $$O = \{(0, 1, 1), (1, 1, 0), (1, 0, 1), (0, 1, 0), (0, 0, 0)\}$$

  where:
  - $(0, 1, 1)$ is the initial observation when the treasure is at the top.
  - $(1, 1, 0)$ is the initial observation when the treasure is at the bottom.
  - $(1, 0, 1)$ is the *corridor* observation, i.e. the observation received by the agent when it is in the corridor but not at the junction.
  - $(0, 1, 0)$ is the *junction* observation.
  - $(0, 0, 0)$ is the terminal observation, used to tell the agent that the episode is finished.
- **Action space** A: $A = \{Up, Down, Right, Left\}$.

- **Initialization** The initial state $x_0 = (p_0, i_0, b_0)$ is computed as follow: p_0 is taken at random, $i_0 = 1$ and $b_0 = false$.
- **Dynamics** f: $x_{t+1} = (p_t, i_{t+1}, b_{t+1}) = f(x_t, a_t)$ where

$$i_{t+1} = \begin{cases} i_t + 1 \text{ if } a_t = Right \text{ and } i_t \in [1, L-1] \\ i_t - 1 \text{ if } a_t = Left \text{ and } i_t \in [2, L] \\ L + 1 \text{ if } a_t = Up \text{ and } i_t = L \\ L + 2 \text{ if } a_t = Down \text{ and } i_t = L \\ i_t \text{ otherwise} \end{cases}$$

$$b_{t+1} = \begin{cases} true \text{ if } i_t = 1 \text{ and } i_{t+1} \neq 1 \\ b_t \text{ otherwise} \end{cases}$$

- **Terminal state condition**: A state $x = (p, i, b)$ is a terminal one if $i \in \{L + 1, L + 2\}$, i.e. the agent is situated on one of the two sides of the junction.
- **Reward function** r:

$$\begin{aligned} r_t &= r\big(x_t = (p_t, i_t, b_t), a_t\big) \\ &= \begin{cases} L \text{ if } i_t = L,\ p_t = true \text{ and } a_t = Up \\ L \text{ if } i_t = L,\ p_t = false \text{ and } a_t = Down \\ 0 \text{ if } i_t \in \{L + 1, L + 2\} \\ -0.1 \text{ otherwise} \end{cases} \end{aligned}$$

  The two first cases happen when the agent chooses the good side at the junction and thus find the treasure. The third case is for handling the terminal states. Finally the last case happens when the agent makes an action that does not lead to a terminal state or chooses the wrong side at the junction.

- **Observation function** o:

$$\begin{aligned} o_t &= o\big(x_t = (p_t, i_t, b_t)\big) \\ &= \begin{cases} (0, 1, 1) \text{ if } p_t = true,\ i_t = 0 \text{ and } b_t = false \\ (1, 1, 0) \text{ if } p_t = false,\ i_t = 0 \text{ and } b_t = false \\ (0, 1, 0) \text{ if } i_t = L \\ (0, 0, 0) \text{ if } i_t \in \{L + 1, L + 2\} \\ (1, 0, 1) \text{ otherwise} \end{cases} \end{aligned}$$

- **Discount factor**: It has been chosen to use $\lambda = 0.9$.

A.2 *Xor-T-maze*

Here is formalized the *Xor-T-maze* RL environment, used in section 5.3. It is deterministic and partially observable like the *T-maze*. However, the big difference with the previously described environment is that the treasure position is determined from two indications. The first one is give in the initial state, while the second one is given when the agent reaches a certain position of the corridor. There are two possible types of indications, and if the two received indications have the same type, then the treasure is at the top, otherwise it is at the bottom. These two types will be referred as $type1$ and $type2$ in this formalization. Note that a *Xor-T-maze* environment has a fixed corridor length L and the position M at which is given the second indication is also fixed.

- **State space** X: the current state of a *T-maze* is entirely determined by $x = (p1, p2, i, b1, b2)$ with $p1 \in \{type1, type2\}$, $p2 \in \{type1, type2\}$, $i \in [1, L+2]$ and $b \in \{true, false\}$:
  - $p1$ is the type of the first indication.
  - $p2$ is the type of the second indication. The treasure is thus placed at the top if and only if $p1 = p2$.
  - i the index of the square on which is currently located the agent. if $i \in [1, L]$, then the agent is on the ith square of the corridor. If $i = L + 1$ then it is on the top square, while if $i = L + 2$ then it is on the bottom square.
  - $b1$ is a boolean which is set to $true$ once the agent has left the first square of the corridor. It is used in the observation function.
  - $b2$ is a boolean which is set to $true$ once the agent has received the second indication. It prevents from giving a second time the second indication to the agent.
- **Observation space** O:

$$O = \{(1,0,1), (1,1,0), (0,1,1), (0,0,1), (0,1,0), (1,0,0), (0,0,0)\}$$

  where:
  - $(1,0,1)$ is the initial observation giving an indication of type *type1*.
  - $(1,1,0)$ is the initial observation giving an indication of type *type2*.
  - $(0,1,1)$ is the *corridor* observation, i.e. the observation received by the agent when it is in the corridor but not at the junction.
  - $(0,0,1)$ is one of the two observations given when the agent reaches the position of the second indication (for the first time). It gives an indication of type $type1$.
  - $(0,1,0)$ is one of the two observations given when the agent reaches the position of the second indication (for the first time). It gives an indication of type $type2$.
  - $(1,0,0)$ is the *junction* observation.

– $(0, 0, 0)$ is the terminal observation, used to tell the agent that the episode is finished.

- **Action space** A: $A = \{Up, Down, Right, Left\}$.
- **Initialization** The initial state $x_0 = (p1_0, p2_0, i_0, b1_0, b2_0)$ is computed as follow: $p1_0$ and $p2_0$ are taken at random, $i_0 = 1$ and $b1_0$ and $b2_0$ are set to $false$.
- **Dynamics** f: $x_{t+1} = (p1_t, p2_t, i_{t+1}, b1_{t+1}, b2_{t+1}) = f(x_t, a_t)$ where

$$i_{t+1} = \begin{cases} i_t + 1 \text{ if } a_t = Right \text{ and } i_t \in [1, L-1] \\ i_t - 1 \text{ if } a_t = Left \text{ and } i_t \in [2, L] \\ L + 1 \text{ if } a_t = Up \text{ and } i_t = L \\ L + 2 \text{ if } a_t = Down \text{ and } i_t = L \\ i_t \text{ otherwise} \end{cases}$$

$$b1_{t+1} = \begin{cases} true \text{ if } i_t = 1 \text{ and } i_{t+1} \neq 1 \\ b1_t \text{ otherwise} \end{cases}$$

$$b2_{t+1} = \begin{cases} true \text{ if } i_{t+1} = M \\ b2_t \text{ otherwise} \end{cases}$$

- **Terminal state condition**: A state $x = (p1, p2, i, b1, b2)$ is a terminal one if $i \in \{L+1, L+2\}$, i.e. the agent is situated on one of the two sides of the junction.
- **Reward function** r:

$$\begin{aligned} r_t &= r\big(x_t = (p1_t, p2_t, i_t, b1_t, b2_t), a_t\big) \\ &= \begin{cases} L \text{ if } i_t = L,\ p1_t = p2_t \text{ and } a_t = Up \\ L \text{ if } i_t = L,\ p1_t \neq p2_t \text{ and } a_t = Down \\ 0 \text{ if } i_t \in \{L+1, L+2\} \\ -0.1 \text{ otherwise} \end{cases} \end{aligned}$$

The two first cases happen when the agent chooses the good side at the junction and thus find the treasure. The third case is for handling the terminal states. Finally the last case happens when the agent makes an action that does not lead to a terminal state or chooses the wrong side at the junction.

- **Observation function** o:

$$
\begin{aligned}
o_t &= o\big(x_t = (p1_t, p2_t, i_t, b1_t, b2_t)\big) \\
&= \begin{cases}
(1,0,1) \text{ if } p1_t = type1,\ i_t = 0 \text{ and } b1_t = false \\
(1,1,0) \text{ if } p1_t = type2,\ i_t = 0 \text{ and } b1_t = false \\
(0,0,1) \text{ if } p2_t = type1,\ i_t = M \text{ and } b2_t = false \\
(0,1,0) \text{ if } p2_t = type2,\ i_t = M \text{ and } b2_t = false \\
(1,0,0) \text{ if } i_t = L \\
(0,0,0) \text{ if } i_t \in \big\{L+1, L+2\big\} \\
(0,1,1) \text{ otherwise}
\end{cases}
\end{aligned}
$$

- **Discount factor**: It has been chosen to use $\lambda = 0.9$.

Appendix B

Technical details of the trainings

Here are given the values of the parameters used for training the agents in the two experiments, starting with the parameters of the training on the *T-maze* (and the *Xor-T-maze*) used in both experiments, and then the parameters of the warmup algorithm, used in the second experiment.

B.1 Parameters for trainings on *T-maze* and *Xor-T-maze*

Name	Value	Description
iters	200	Number of iterations made of the algorithm
batch_size	32	Size of the batches
games	32	Number of games made at each iteration
memory_size	512	Size of the replay buffer
epochs	10	Number of epochs made at each iteration
gamma	0.9	Discount factor of the RL environment
optimizer	*Adam*	Optimizer used to train the models
learning_rate	0.001	Learning rate
timeouts	100	Maximum number of timesteps for each episode
max_eps	0.9	Maximum value of ϵ
min_eps	0.01	Minimum value of ϵ
decay_eps	0.0005	Decay of ϵ

B.2 Warmup parameters

Name	Value	Description
warmup_iters	300	Number of iterations made during warmup
warmup_max_length	5000	Maximum warmup length
batch_size	32	Size of the batches