Introducing biological neuronal dynamics and neuromodulation in artificial neural networks.

Nicolas Vecoven

Advisor : Guillaume Drion

Co-advisor : Damien Ernst

PhD dissertation

The University of Liège

2021

Jury members

Prof. Louis Wehenkel (president)	*Université de Liège, Belgium;*
Prof. Guillaume Drion (advisor)	*Université de Liège, Belgium;*
Prof. Damien Ernst (co-advisor)	*Université de Liège, Belgium;*
Prof. Timothy O'Leary	*Cambridge, United Kingdom;*
Prof. Michal Valko	*Deepmind, Paris;*
Prof. Alessio Franci	*UNAM, Mexico;*
Prof. Pierre Geurts	*Université de Liège, Belgium;*
Prof. Gilles Louppe	*Université de Liège, Belgium;*

Acknowledgements

I really enjoyed working on this thesis, and am very grateful to have had the opportunity to do so in such a great environment. This manuscript would never have seen daylight without the help, support and insights of people around me, who I would like to thank here.

First, I would like to thank my advisor, Guillaume Drion. To begin with, I would like to thank him for letting me work on such an interesting topic. Although it was not an easy task to conciliate our different domains in the beginning, his availability as well as his positivity were incredible assets that led to successfully do so. As such, I want to thank him for all the interesting discussions and for giving me so much of his time. I would also like to thank him for giving me so much freedom in my research. And last but not least, I would like to thank him for his enthusiasm, which has been a great driver for me throughout those years, it was absolutely great working with him.

Of course, the writing of this thesis would have been impossible without my co-advisor Damien Ernst. In particular, I would like to thank him for his many feedbacks, his great mathematical rigour, his availability and for all the interesting discussions I was given with him. It makes no doubt that my scientific writing and reasoning has greatly improved thanks to his many advices. I would also like to thank him for introducing me to the world of reinforcement learning, a topic which I particularly liked working on. Finally, I would also like to thank him for the great times we had around a beer.

I would also like to thank all the members of the jury for reading this manuscript, for their interest in it as well as for their feedback. A special thanks goes to Pierre Geurts for his comments and for having spiked my interest in research during my master's thesis. I probably would not have started this work if it wasn't for the great time I had working with him.

I would also like to thank Gilles Louppe for the many discussions we had and advices he gave to me throughout the years.

Of course, I can not forget to thank all my Montefiore colleagues and friends. First, I would like to give a special thanks to Antoine Wehenkel for the many discussions we had and his support/interest in my work. I would also like to thank him for proofreading this thesis, and express him my gratitude for being such a good friend. I would like to thank Pascal for his friendship and for making it so enjoyable at Montefiore as well as Anais for all of our "little" chats. I would also not only like to thank Antonio for our interesting discussions, Laurine for her enthusiasm and Vân Anh for the time spent working with me but also Marie, Anthony, Romain, Jean-Michel, Pierre Sacré, Laurent Mathy, Michael, Jonathan, Ulysse, Arnaud, Adrien, Renaud and all the others I forgot to mention here for making the day-to-day life at Montefiore so great.

I would also like to thank my friends for their support and, last but not least, my family for their unconditional love. In particular, I would like to thank my father for driving me towards the path of computer science and engineering and my mother for her support. Finally, last but not least, I would like to thank my wife Marine for always being there for me and supporting me throughout this journey.

Abstract

The present thesis takes a step towards enriching artificial neural networks with bio-inspired mechanisms. To this end, high-level abstractions of important biological rules, modelled using control theory, will be introduced and linked to artificial networks. In particular, it will be discussed that introducing neuronal bistability and neuromodulation into artificial neural networks provides different benefits. As a first step, Part I will first introduce necessary machine learning background.

Part II of this thesis will focus on the ability of recurrent neural networks to learn long-term dependencies, something which usually proves difficult. Bistable recurrent cells will be introduced as a way to help towards solving such issues. Furthermore, supported by the results obtained with those cells, a more generic method to promote multistability with usual recurrent cells is proposed. This part highlights the importance of dynamics in recurrent neural networks and in particular, right after initialisation, for easily learning long-term dependencies.

Part III of this thesis is dedicated to introducing neuromodulation in artificial neural network. This important biological mechanism is often associated to the robust control of continuous behaviours, allowing biological systems to adapt very quickly to changing context, something which remains very difficult for usual artificial agents. As such, a neuromodulated architecture, specifically designed for its adaptive capabilities (i.e. robustness towards changing environment or context), is proposed. It will be shown to exhibit much more stable performance and to converge towards better policies than classical recurrent networks. Furthermore, this part discusses that these architectures are also implicitly able to learn a continuous representation of the different contexts in which they evolve. Finally, some other very recently proposed architectures and their benefits will be briefly mentioned as well.

As a last note, it is important to mention that Part II and Part III of this thesis could potentially be linked. As such, the last part of this thesis will be dedicated to provide ideas for future works, specifically aimed at closing the gap between Part II and Part III.

Contents

Chapter 1

Introduction

One of the most popular definition of machine learning has been given by Mitchell & McGraw (1997):

"*A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.*"

Machine learning has been a rapidly evolving field for the past few decades, growing into an extremely vast and rich domain of research over the years. So rich, in fact, that machine learning now comprises a multitude of different algorithms, models, settings and fields. As an extremely brief introduction, the most common machine learning settings can be listed as follows.

- **Supervised learning:** a setting in which an annotated dataset of input-output pairs is given, corresponding to experience E. In such a case, the task T is to map new inputs to correct outputs, as assessed by some performance measure P. To this end, machine learning models can be trained on the dataset in order to map new unseen inputs to their output as correctly as possible. A simple example of this setting would be handwritten digit recognition, as for example with the MNIST dataset (LeCun & Cortes, 2010).
- **Reinforcement learning (RL):** a setting in which no dataset is given. Rather, an RL agent gets to interact with an environment gathering experience E and receiving rewards in the process. Its task T is to interact with the environment in order to maximize its received rewards. Thus, P can be defined as the expectation of the sum of reward signals received by the agent. An example of this setting would be an agent learning how to achieve high-score in an Atari game, as noteably done by Mnih et al. (2013).
- **Unsupervised learning:** as for the supervised learning setting, a dataset is given, but without annotation. The task T in this case is thus to find patterns in the provided dataset. Depending on the task, multiple performance measures exist. A currently popular example of this setting would be generative modelling. Broadly defined, sampling a generative model should allow to produce samples as

if they arised from the same distribution than that used to create the provided dataset. For example, generative adversarial networks (a particular machine learning model) can be trained to produce highly accurate images (Goodfellow et al., 2014). Clustering (Rokach & Maimon, 2005) could be taken as another example of finding pattern in a non-annotated dataset.

- **Semi-supervised learning:** a setting at the intersection of supervised and unsupervised learning in that only parts (usually very few) of a dataset are annotated. The goal is often the same as for supervised learning, while making use of some non-annotated samples to learn patterns which could improve model performance.

It is important to stress that this list is non exhaustive, and that due to the increasing richness of the domain, it has become more and more difficult to keep track of it. However, some authors have done an excellent didactic job at summarizing the field. In particular, Hastie et al. (2009) went through the explanation of most machine learning concepts in a very comprehensive book and Murphy (2022) gave a different, more probabilistic, perspective on machine learning basics in a book of his own, also including newer concepts.

It is out of the scope of this thesis to detail such a broad field. Rather, this thesis focuses on two of the previously introduced settings of machine learning, namely supervised and reinforcement learning. Both of these settings can be tackled using machine learning models. These models can either be parametric or non-parametric, are all trained through different learning algorithms and all come on a different footing when it comes to their interpretability and predictive performance. In this thesis we focus on a particular type of parametric model: artificial neural networks (ANNs).

First introduced with the perceptron (Rosenblatt, 1958), ANNs were initially designed to attempt mimicking the human brain, hence their interest for this thesis. Since their introduction, ANNs have however been driven away from this goal and modified towards having better mathematical properties. Over the years, these changes as well as the advent of computational power and data availability have allowed ANNs to gain fame, thanks to increasing predictive accuracy. The rise of ANNs popularity first came thanks to a performance breakthrough on the image classification benchmark “imagenet” achieved by Krizhevsky et al. (2012). This performance leap showed that ANNs could become competitive and even outperform other types of machine learning models, which were considered as state-of-the-art at the time. For about ten years now, the field of ANNs has been growing exponentially and has received a tremendous amount of interest, providing breakthroughs and state-of-the-art performance in a large number of fields such as computer vision, time-series analysis and natural language processing.

Despite all these progress, neural networks are still far from achieving perfect performances for a variety of tasks. This thesis proposes to take insights from important biological concepts to enhance performance of artificial neural networks on specific tasks. As such, the goal of this thesis is thus not to compete with state of the art, but rather to highlight the impact that some biological concepts could potentially bring to artificial neural networks. This goal leads to two important choices made throughout this thesis. First, the biological concepts will always be introduced such that they are as close as possible to usual artificial architectures. This allows to clearly target the specific concept to be analysed, and make sure it is linked with the differences observed. Second, some benchmarks are specifically built for standard architectures to fail. This allows not to focus on performances compared to state of the art, but rather, to highlight the impact of the introduced concept in specific cases. When combined, those two choices allow for a clear understanding of the proposed concept interests. In conclusion, the resulting architectures often do not vary much from more usual ones, and thus, the biological concepts are not so much seen in the resulting architectures than in the process carried to build them. This thesis uses the preceding approach to tackle two main subjects.

Tuning neuronal dynamics for long-term memory On the one hand, biological neurons have been modelled as exhibiting different dynamical properties, one of such being bistability. In Part II, this property will be discussed in the case of artificial recurrent neural networks, providing insights and methods for learning long temporal dependencies, something which can currently remain difficult with usual recurrent cells.

In many applications and important problems, be them theoretical or practical, sequentiality plays an important role in the data structure. As such, lots of different neural architectures have already been designed towards exploiting such characteristic as best as possible. In particular, recurrent neural networks have been put forward early as such an architecture. Providing state-of-the-art results on many tasks over the years, they have become a staple tool when working with artificial neural networks (Lipton et al., 2015). However, they are known to be difficult to train, especially when patterns in the sequential data grow longer (Pascanu et al., 2013). Furthermore, due to the high complexity of such architectures, they remain black-box models and it is usually arduous to understand their underlying dynamics. Nevertheless, recent body of work has focused on using control theory to get a better grasp of the workings such networks, highlighting the importance of fixed points in their prediction process (Sussillo & Barak, 2013; Ceni et al., 2020; Maheswaranathan et al., 2019).

Fixed points, and more particularly attractors, were also shown to be of importance when modelling high-level behaviours of human neurons. The first work behind Part II aims at taking inspiration from those high-level models to create a biologically inspired bistable recurrent cell (BRC). The resulting BRC was shown to provide great performance when learning long-term patterns. As such, following this work, Part II also proposes a more generic technique, allowing to endow usual cells with similar dynamic properties as those of BRCs, and consequently, enhancing their ability to learn longer patterns.

Neuromodulation in neural networks for adaptive capabilities On the other hand, neuromodulation is a well known biological mechanism, usually thought to be highly important for the adaptive capabilities of humans. A big challenge behind neuromodulation however, is its high complexity. Indeed, its modelling often comprises high-dimensional, highly non-linear dynamics. Nevertheless, recent advances have been made on the modelling of mechanisms allowing the robustness and modulation of neuron intrinsic properties. Part III takes advantage of this research and will focus on the introduction of high-level rules of such mechanisms in artificial neural networks. It will be discussed that this can provide benefits in terms of adaptive capabilities of artificial neural network, which also remains a challenge as of today.

Thanks to many breakthroughs in the past few years, it is now possible to train artificial agents to achieve super-human performance on a wide variety of tasks. However, this often requires tremendous amounts of data and computational power. Whereas humans are able to make use of past accumulated knowledge in order to learn how to perform new tasks efficiently, this remains difficult for machine learning models. When presented with a new task, these will often have to start a new learning process altogether. Due to the sheer amounts of data needed for such training, this quickly becomes limitant and is the reason why studying adaptive capabilities of artificial neural networks is an active and important field of research.

Neuromodulation is the ability of neurons to tune their input-output properties to reshape signal transmission at the cellular level, generally in response to an external signal. This processus was shown to be highly important in the functioning of the human brain, allowing to regulate many critical nervous system properties which can not be controlled through synaptic plasticity alone. Neuromodulation is also often associated to the robust control of continuous behaviours, and as such, to "adaptive capabilities".

It is thus natural to study the introduction of neuromodulatory principles in artificial neural networks, and more particularly to endow them with the ability to better adapt. The work behind Part III precisely aimed towards this goal. Concurrent works, as well as more recent ones, have also taken this line of research with success, further showing the importance of endowing artificial neural networks with neuromodulation.

Thesis organisation

Part I aims at giving the necessary background about relevant machine learning concepts for understanding the remaining of the manuscript. To this end, the objective of Chapter 2 is to give a more formal and detailed view on the supervised and RL settings, as well as on artificial neural networks. This chapter should provide enough information to the reader for precisely understanding machine learning settings discussed in this thesis as well as the standard ANNs architectures and training methods. Furthermore, we will also shortly discuss the interpretability of ANNs. Indeed, sometimes the comprehension of the underlying model can be more important than its predictive accuracy. Then, in Chapter 3, different methods to solve RL tasks are presented. Due to the complexity of the setting, this chapter is necessary to provide information to the reader on some of the state-of-the-art methods required to tackle the RL problems in Part III.

Part II will dive deeper into recurrent neural networks, and, more precisely, in the long-term memory capabilities of such networks. To this end, first, background and state-of-the-art will be presented, highlighting common troubles of such networks when temporal dependencies grow longer. Afterwards, a novel biologically-inspired bistable cell, as well as a more generic method will be shown to help alleviate these issues.

Part III will focus on adaptive capabilities of artificial neural networks, and in particular, will look at the benefits of neuromodulation for such ability. First, in the form of meta-learning and meta-reinforcement learning, usual settings for assessing adaptive capabilities learning will be presented. Afterwards, basic concepts of neuromodulation will be detailed, leading to the proposition of a neuromodulatory architecture, specifically designed for adaptation. Finally, Part III will discuss the benefits of such architecture, as well as briefly highlighting those obtained with other neuromodulatory approaches in concurrent works.

Finally, Part IV will summarise the main findings of this thesis. Importantly, this part will also step towards closing the gap between Part II and III through potential future works avenues. This thesis ends with a brief and more general discussion on the approach taken throughout its genesis.

Publications

The following publications have led to the core of this thesis:

- Vecoven, N., Ernst, D., Wehenkel, A., & Drion, G. (2019). Cellular neuromodulation in artificial networks. In Proceedings of the NeurIPS 2019 Workshop Neuro AI.

  (This publication originated Chapter 8)
- Vecoven, N., Ernst, D., Wehenkel, A., & Drion, G. (2020). Introducing neuromodulation in deep neural networks to learn adaptive behaviours. PloS one, 15(1), e0227922.

  (This publication also contributed to Chapter 8)
- Vecoven, N., Ernst, D., & Drion, G. (2021a). A bio-inspired bistable recurrent cell allows for long-lasting memory. Plos one, 16(6), e0252676.

  (This publication originated Chapter 5)
- Vecoven, N., Ernst, D., & Drion, G. (2021b). Warming-up recurrent neural networks to maximize reachable multistability greatly improves learning. arXiv preprint arXiv:2106.01001.

  (This publication originated Chapter 6)

The following publication was also worked on during this thesis.

- Vecoven, N., Begon, J.-M., Sutera, A., Geurts, P., & Huynh-Thu, V. A. (2020). Nets versus trees for feature ranking and gene network inference. In International conference on discovery science (pp. 231–245).

PART I

Background

Chapter 2

Machine learning

This chapter aims at formalizing the machine learning settings and models used throughout this thesis. We note that, as the topics covered hereunder are quite large, state-of-the-art will not be discussed in this chapter. Rather, we will introduce relevant state-of-the-art in the following chapters as we dive deeper into some parts of the general concepts presented here. First, Section 2.1 will go through supervised learning basics. Section 2.2 will then go through ANNs' basics by detailing a standard ANN architecture, training algorithm. Finally, Section 2.3 will provide a formalisation of the RL setting, introducing different categories of RL algorithms and easing the reader into Chapter 3, which presents relevant state-of-the-art RL methods used in this manuscript.

2.1 Supervised learning

As previously introduced the goal of supervised learning is to learn a mapping between inputs (often called features) and outputs, given a dataset of such pairs. Formally, let $\mathcal{X}$ denote the set of all possible features and $\mathcal{Y}$ the set of all possible outputs. Given an unknown distribution $\mathbb{P}$ over $\mathcal{X} \times \mathcal{Y}$ defining the task at hand and a training set $\mathcal{D}$ of n samples drawn from it,

$$\mathcal{D} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\} \in (\mathcal{X} \times \mathcal{Y})^n,$$

the goal of supervised learning is to learn a function $f(\cdot) : \mathcal{X} \to \mathcal{Y}$ such as to minimize the expected prediction error defined as:

$$\mathop{\mathbb{E}}_{(\mathbf{x},y)\sim\mathbb{P}} L(y, f(\mathbf{x}))$$

where L is a function of dissimilarity between its arguments, called loss function. Depending on the wanted result and task at hand, different losses can be used. In this dissertation, we tackle both usual regression and classification benchmarks. As such, we use two very common loss functions to assess the predictive performance of our models.

- **Regression:** when $\mathcal{Y}$ is a continuous domain, then the loss can be defined as a squared error such that

$$L(y, \hat{y}) = ||y - \hat{y}||^2 \quad .$$

- **Classification:** when $\mathcal{Y}$ is a discrete domain and y takes its value in $\{o_1, \ldots, o_n\}$, then the loss can be the 0-1 error loss which is equal to one if its arguments are not equal, and else to zero:

$$L(y, \hat{y}) = \begin{cases} 0 \text{ if } \hat{y} = y, \\ 1 \text{ otherwise.} \end{cases}$$

  The expectation on this loss is called the miss-classification error rate. We note that, in this manuscript, the accuracy will often be reported for classification tasks instead. This measure is equal to one minus the miss-classification error rate and simply represents the proportion of samples correctly classified by the model.

When training a model in a supervised setting, one has to take particular attention to the bias-variance trade-off (Geurts, 2002). Shortly put, models with high bias do not learn a fine-enough representation of the training data and thus have poor performance on training and testing data. This is often referred to as underfitting. On the other hand, models with high variance learn a too fine representation of the training data, capturing noise in the training data. As such, those models have a very good predictive performance on the training data, but generalize extremely poorly to unseen data. In other words, their representation will be highly dependant on the realisation of the training set sampled from the underlying distribution. This is referred to as overfitting.

Hence why, for all experiments in this dissertation, sufficiently large testing sets are generated for evaluating performance and experiments are run multiple times with different training sets. As such, we make sure that our results do not depend on "lucky runs" for which, by coincidence, a model would have been able to overfit the training data and keep its predictive accuracy on testing data.

2.2 Artificial neural networks

In this section three main concepts of artificial neural networks are presented. First, the most basic architecture of a neural network is introduced, then the training procedure is explained.

2.2.1 Standard architecture

In essence, an artificial neural network can simply be defined as a parametric function with the main particularity that its parameters define a neural architecture. In its most basic form, a neural architecture is built with multiple layers, each comprising multiple neurons and connected using weight matrices, as shown on Figure 2.1.

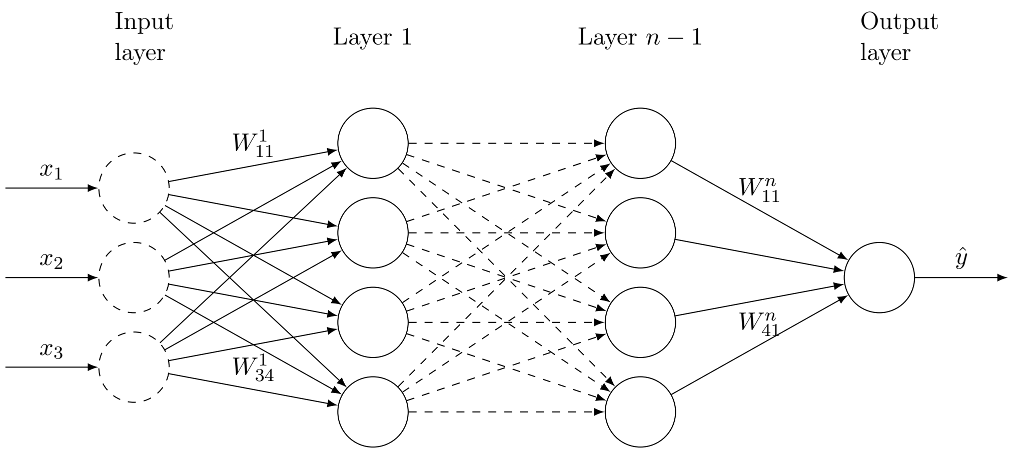


Figure 2.1: Neural network

With this architecture, each neuron takes a weighted sum of the previous layer's outputs as input to its activation function $g : \mathbb{R} \rightarrow \mathbb{R}$ and layers are said to be fully connected. Let o^l_i denote the output of neuron i of layer l comprising $dim(l)$ neurons, then we define the output of layer l as $\mathbf{o}^l = [o^l_1, \ldots, o^l_{dim(l)}]$, with

$$\mathbf{o}^l = g(W^l \mathbf{o}^{l-1} + \mathbf{b}^l) \quad ,$$

where $\mathbf{b}^l \in \mathbb{R}^{dim(l)}$ is the offset vector of layer l and where $g(\mathbf{x})$ represents the function g applied element-wise on $\mathbf{x}$ such that $g(\mathbf{x}) = [g(x_1), \ldots, g(x_n)]$ for ease or writing. These vectors are needed in order to be able to offset the output of each neuron with respect to their input. The activation function is one of the most important piece of any artificial neural network. Indeed, without this function, the network would simply output a linear combination of the input. It is thus highly important to select a non-linear activation function, such that the network itself can create a non-linear mapping. Some of the activation functions g used in this manuscript are

- the *sigmoid*

$$g(x) = \frac{1}{1 + e^{-x}} \quad ;$$

- the *hyperbolic tangent*

$$g(x) = \tanh(x) \quad ;$$

- the *rectified linear unit (ReLU)*

$$g(x) = \max(0, x) \quad .$$

A computational graph of a neuron is provided on Figure 2.2.

Note that, when working with categorical features, one must create an encoding procedure for them to be used by neural networks. Potdar et al. (2017) compared various strategies and one of the most usual methods is the "one-hot encoding", a procedure where each of the feature's categories is attributed to an input neuron. For a given sample, all of these neurons will be equal to zero except the one corresponding to that sample's feature category, which will be equal to one.

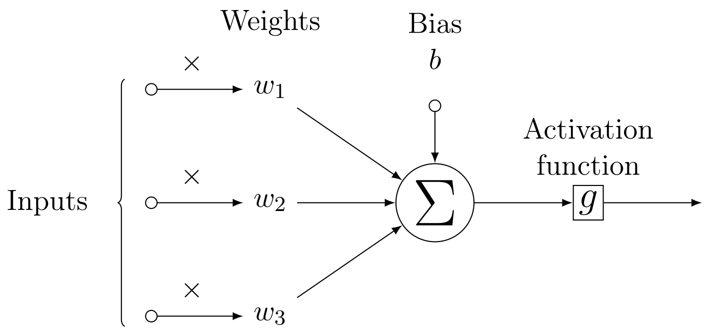


Figure 2.2: Computational graph of an artificial neuron.

We stress that only the most standard type of neural network has been described here. There exist many more architectures such as convolutional neural networks (LeCun et al., 1989), transformers (Vaswani et al., 2017), conditional processes (Garnelo et al., 2018) and hypernetworks (Ha et al., 2016a) among many others. All these architectures are designed to tackle different problems, often providing improved performance over more basic networks. In this thesis, a particular focus is given to a specific kind of architecture: recurrent neural networks (RNNs), which are specifically designed to handle sequential data. These networks are endowed with an internal state that serves as a memory. As the inputs are sequentially received by the network, this internal state gets updated (an input is passed to the network at each time-step). Unfortunately, such recurrent architectures are known to be hard to train when the temporal data spans too many time-steps. Part II is entirely dedicated to such networks. As such, more background on these architectures will be given in Chapter 4 while Chapter 5 and 6 will propose some possible solutions to cope with those training problems.

2.2.2 Training

Let $f(\mathbf{x};\theta)$ denote a neural network, where θ are the parameters of the network. Training such network in a supervised learning setting amounts to optimize θ through stochastic gradient descent on a differentiable loss. In this dissertation, the mean-squared error is used for regression problem, and the categorical cross entropy for classification problems. The latter is defined as

$$L(f(\mathbf{x};\theta),y) = -\sum_{i=1}^{k} y_i \log(f(\mathbf{x};\theta))$$

where k is the number of classes, and y_i is equal to 1 if the sample belongs to class i, else to 0. When using this loss, the network needs to output a probability distribution over the different classes (rather than a simple real-valued output). The mapping from $\mathbb{R}^k$ can be done by adding a softmax activation function $g : \mathbb{R}^k \to [0,1]^k$ to the final layer which, given a vector $\mathbf{x}$, maps each of its element x_i as

$$g(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

such that its real-valued output vector $[g(\mathbf{x})_1, \ldots, g(\mathbf{x})_k]$ sums to 1. With these loss defined, the stochastic gradient descent procedure is carried as follows. First, one uniformly samples a mini-batch $\mathcal{B}$ of k samples in $\mathcal{D}$ and updates the parameters as

$$\theta \leftarrow \theta - \alpha \frac{1}{k} \nabla_\theta \sum_{(\mathbf{x},y)\in\mathcal{B}} L(f(\mathbf{x},\theta),y) \ ,$$

where α is called the learning rate. This update is called a gradient step. Removing the samples of the mini-batch $\mathcal{B}$ from $\mathcal{D}$ and repeating the procedure until all elements of the dataset $\mathcal{D}$ have been seen once is called an epoch. Usually when training neural networks, a fixed number of epochs is predefined and gradient steps are repeated until that given number is reached.

Some variations of this procedure exist and, in particular, adding momentum to the gradient steps has been shown to improve training in most cases. As such, although the training procedure remains similar in this dissertation, networks are trained using ADAM (Kingma & Ba, 2014), one of the multiple ways to carry stochastic optimization with momentum. As a final note, it is worth mentioning that there exist many methods to reduce overfitting when training neural networks, such as weight regularization, dropout (Srivastava et al., 2014) and adaptive learning rates among others.

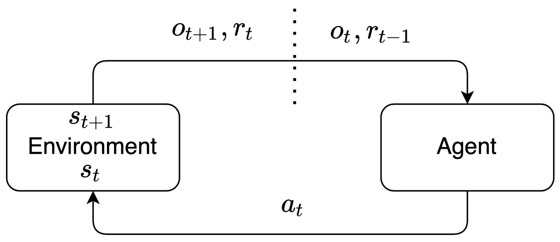


Figure 2.3: Small sketch of a RL setting.

2.3 Reinforcement learning

The goal of an RL agent is to learn how to maximize the reward it obtains while interacting with an environment. An RL problem can be defined as a Markov Decision Process (MDP). Let t denote the discrete time of such a process. In a RL setting (depicted on Figure 2.3), an agent receives an observation $\mathbf{x}_t \in \mathcal{X}$ of state $\mathbf{s}_t \in \mathcal{S}$ of the environment. Based on this observation, the agent will have to perform an action $\mathbf{a}_t \in \mathcal{A}$ on the environment, receiving reward $r_t \in \mathcal{R}$ and causing the environment's state to change. As such, a reinforcement learning setting can be defined through three auxiliary concepts. First is the system dynamics, second is the reward function and third is the policy. All these concepts are detailed in Subsection 2.3.1. Finally, Subsection 2.3.2 formalizes the goal of RL.

2.3.1 System dynamics, reward function and policy

Together, the system dynamics and reward functions describe an RL environment.

System dynamics On one hand, the system dynamics can be defined with two functions. First is the transition probability function written p, which describes the potential reactions of the environment to a given action of the agent, defined such that

$$\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{a}_t, \mathbf{s}_t) \quad .$$

Second is the observation function, written o, which describes the probability of the agent to observe variable $\mathbf{x}_t$ given the full-state of the environment $\mathbf{s}_t$, that is

$$\mathbf{x}_t \sim o(\mathbf{o}_t|\mathbf{s}_t) \quad .$$

If the observation $\mathbf{o}_t$ does not provide enough information to directly infer the full state of the environment $\mathbf{s}_t$, then the environment is said to be partially observable. In such a case, the RL setting becomes a partially-observable markov decision process (POMDP). It is important to note that the transition function can also be deterministic if $p(\mathbf{s}_{t+1}|\mathbf{a}_t, \mathbf{s}_t)$ is a Dirac for all possible pairs $(\mathbf{s}_t, \mathbf{a}_t) \in (\mathcal{S} \times \mathcal{A})$.

Reward function On the other hand, the reward function describes the reward received by the agent when performing a given action in a given state and going to a given state. It can thus be defined as

$$\rho(\mathbf{a}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) : \mathcal{A} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R} \quad .$$

In this dissertation, we will only tackle the case of bounded rewards, that is, $\rho(\cdot, \cdot, \cdot) < |R|$ with $R \in \mathbb{R}^+$ for all reward functions. Also in the context of this manuscript, we will only talk about time-invariant systems. Those are systems for which $o(\cdot|\cdot)$, $p(\cdot|\cdot, \cdot)$ and $\rho(\cdot, \cdot, \cdot)$ are independent of the discrete time t. Finally, we define the reward signal associated to the reward function at time-step t. This signal is equal to

$$\gamma^t \rho(\mathbf{a}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) \quad ,$$

where $\gamma \in [0, 1[$ is a decay factor, often called discount factor, that allows to bound the cumulative reward signal obtained by the agent.

Policy The policy, denoted π defines the RL agent's decision process. In other words, at time-step t, the agent samples action $\mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{h}_t)$ given the history

$$\mathbf{h}_t = (\mathbf{x}_0, \mathbf{a}_0, r_0, \ldots, \mathbf{a}_{t-1}, r_{t-1}, \mathbf{x}_t) \quad .$$

2.3.2 Objective of RL

The goal of RL is to learn a policy $\pi \in \Pi$, where Π denotes the domain of all possible policies, that maximizes its expected cumulative discounted reward signal J^π. Formally, we want to build a policy $\hat{\pi^*}$ such that $J^{\hat{\pi^*}}$ is as close as possible to J^{π^*} where

$$J^{\pi^*}(\mathbf{s}) = \max_{\pi \in \Pi} J^\pi(\mathbf{s}), \forall \mathbf{s} \in \mathcal{S} \quad ,$$

with

$$J^\pi(\mathbf{s}) = \lim_{T \rightarrow \infty} \mathop{\mathbb{E}}_{\mathbf{a} \sim \pi} [\sum_{t=1}^{T} \gamma^t * \rho(\mathbf{a}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) | \mathbf{s}_0 = \mathbf{s}] \quad .$$

Training There exist multiple methods to tackle this problem and they can be divided into two main categories.

First, **Model-based methods** for which the model of the environment has to be known. In other words, these methods can only be used if the transition and reward functions are fully-known. Some algorithms are very efficient at solving this type of tasks, and there is no need to interact with the environment to build a good policy. However, this setting is quite restrictive in practice as one rarely has access to the full knowledge of the system dynamics at hand. We refer the reader to Moerland et al. (2020)'s work for a survey on the subject.

Second, and more interestingly in the context of this manuscript, **Model-free methods** tackle the RL setting in a context where no prior information on the environment and reward function is given. In such a context, the agent has to learn solely based on its interactions with the environment and there exist two main ways to tackle the problem. On one hand, one can try to build a model of the environment, then use this approximation to use model-based methods (such procedures can sometimes be referred to model-based methods as a whole). Such model of the environment can for example trained by letting the agent play a random policy for many time-steps, then by using supervised methods to learn the transition and reward functions. Such methods have already proven to be very effective on multiple benchmarks (Ha & Schmidhuber, 2018). On the other hand, one can try to directly learn a policy, without ever trying to model the environment.

Next Chapter gives an overview of different such model-free methods, with a specific emphasis on the state-of-the-art methods used for solving RL problems with a continuous action space ($\mathcal{A}$ is a continuous set, as for the problems studied in Part III). As will be discussed, such setting is harder to tackle due to the infinite size of the action space. This infinite space makes it impossible to iterate over, something that is often done to solve RL problems with a discrete action space.

Importantly, we note that those methods are very general, but are unfortunately often very sample-inefficient. Usually, the agent will have to interact with the environment for millions of time-steps before achieving good performance. It can thus quickly become cumbersome to train such agents. Furthermore, an agent which has learned a good policy for a given environment will probably achieve very poor performance on a similar albeit slightly different environment. For these reasons, one has to wonder if, given multiple different environments, it is possible to acquire knowledge which would let an agent quickly learn how to tackle new similar tasks. Put otherwise, one could ask if it is possible to train an agent to quickly adapt to new environments (without requiring an extreme amount of time-steps), given some initial training on different environments.

This is precisely what the field of meta-reinforcement learning aims to solve and what Part III of this manuscript discusses. In particular Chapter 7, will formalize meta-reinforcement learning and present different ways of tackling such problematic, and especially how to do so by using RNNs.

Summary

This Chapter introduced the very basics of supervised and reinforcement learning, which are the two settings that are used for the experiments in this manuscript. In particular, Part II focuses on supervised problems whereas experiments in Part III are made in a reinforcement learning setting. This Chapter also introduced a standard NN architecture.Overall, having read this Chapter, the reader is expected to understand the characteristics as well as the goal (and performance measures) of the settings tackled in this manuscript, and to be familiar with the main concepts of NNs, such as their training process.

Chapter 3

Model-free reinforcement learning

In this Chapter, we first start by introducing two main categories of algorithms which can be used to solve model-free RL tasks. The first category encompasses the value iteration methods, for which one tries to learn a value function that is then used to derive a policy. The second category encompasses policy iteration methods, where one rather directly iterates over the policy space to find a good candidate. Basics of each category are respectively discussed in Section 3.1 and Section 3.2. While both categories have their advantages, they also have their drawbacks. More recently, a third class of algorithm was introduced: "actor-critic". The goal of such method is to directly iterate over the policy space, such as for policy iteration methods, while using guidance from a learned value function, as for value iteration methods. As such, those algorithms are supposed to combine advantages of both methods and usually provide good performance. Section 3.3 formalizes the actor-critic framework. Furthermore, Section 3.3 also deeply details an advantage actor critic framework (A2C) as proposed by Mnih et al. (2016), using proximal policy optimisation (Schulman et al., 2017) and generalised advantage estimation (Schulman, Moritz, et al., 2015). This framework is precisely described for the reader to fully understand the algorithm used for all experiments of Part III. Finally, it is important to note that for the sake of simplicity, all algorithms are explained in the setting of fully observable MDPs [1].

1. All algorithms remain similar when working with POMDPs. The main difference is that when working with POMDPs, function approximators will often take the history $\mathbf{h}_t$ as input, rather than the state $\mathbf{s}_t$. As such, the function approximators can capture as much information as possible about the true state $\mathbf{s}_t$.

3.1 Value iteration methods

In the early years of RL, the main focus was to study fully observable environments with small, discrete state and action spaces ($\mathcal{S}$ and $\mathcal{A}$). As such, one of the first and most immediate ways to solve such environments was through dynamic programming, as for example with the tabular Q-learning algorithm (Watkins & Dayan, 1992). This is probably the most well-known and simplest value iteration method in which one builds a $Q : (\mathcal{S} \times \mathcal{A}) \to \mathbb{R}$ function, also called state-action value function, which is the unique solution to the following Bellman equation:

$$Q(\mathbf{s}, \mathbf{a}) = \underset{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})}{\mathbb{E}} [\rho(\mathbf{a}, \mathbf{s}, \mathbf{s}') + \gamma \max_{\mathbf{a}' \in \mathcal{A}} Q(\mathbf{s}', \mathbf{a}')] \quad .$$

It is known that the policy π^* is optimal if and only if $\mathbf{a} \sim \pi^*(\mathbf{a}|\mathbf{s})$ is such that $\mathbf{a} \in \arg\max_{\mathbf{a} \in \mathcal{A}} Q(\mathbf{s}, \mathbf{a})$ and we define the state value function $V : (S) \to \mathbb{R}$ as

$$V(\mathbf{s}) = J^{\pi^*}(\mathbf{s}) = \max_{\mathbf{a} \in \mathcal{A}} Q(\mathbf{s}, \mathbf{a}) \quad .$$

Building Q In such a setting, let $Q_0(\mathbf{s}, \mathbf{a}) = 0$, one can get convergence of Q_N to Q thanks to the following recurrence equation:

$$Q_N(\mathbf{s}, \mathbf{a}) = \underset{\mathbf{s}' \sim p(\mathbf{s}'|\mathbf{s},\mathbf{a})}{\mathbb{E}} [\rho(\mathbf{a}, \mathbf{s}, \mathbf{s}') + \gamma * \max_{\mathbf{a}' \in \mathcal{A}} Q_{N-1}(\mathbf{s}', \mathbf{a}')], \forall N \geq 1 \quad .$$

In practice, the Q learning algorithm uses the previous recurrent relation to build an approximate $\hat{Q}$ of Q, without inferring the transition and reward functions. $\hat{Q}$ can be computed solely based on a historic of interactions with the environment $\mathbf{h}_t$ thanks to the following update:

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \leftarrow \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) + \alpha_t * TD(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \quad ,$$

where $\alpha_t \in [0, 1]$ is the learning rate and TD is called the temporal difference and is defined as

$$TD(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) = r_t + \gamma * \max_{\mathbf{a} \in \mathcal{A}} \hat{Q}(\mathbf{s}_{t+1}, \mathbf{a}) - \hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \quad .$$

Thanks to the law of large numbers and given some particular conditions on α_t and the history $\mathbf{h}_t$ (each state-action pair needs to be visited an infinite number of times when $t \to \infty$), it can be shown $\hat{Q}$ will converge towards Q.

Exploration-exploitation If the agent always plays the policy which maximizes its estimated return through $\hat{Q}$, then it will not explore correctly all sate-action pairs. This is the exploration/exploitation dilemma. Put otherwise, if the agent only does what it has learned to be best through the state-action value function (and thus only *exploits* his knowledge), it might miss and never learn about better opportunities. On the other hand, if the agent does not care about what it has previously learned and always plays differently, for example at random (hence *exploring*), it will never achieve good rewards. To handle this dilemma in Q learning, one often uses the epsilon-greedy approach. That is, a variable $\epsilon \in [0, 1]$ is used to give the probability of playing a random move versus that maximizing $\hat{Q}$. Usually, ϵ is initialized close to 1 as, at that time, the approximation of Q is supposed to be extremely poor since no prior information on the environment is given. Thus, when beginning to train, the agent has a high probability of playing randomly to explore well the environment. Then, ϵ is slowly diminished after each time-step to reach a threshold (usually small, such as 0.05) over time. As such, as $\hat{Q}$ is refined over time, the agent will be more and more inclined to chose the action which maximizes $\hat{Q}$, and thus to exploit its knowledge, hopefully achieving good rewards.

Limitations There are however two major limitations to the basic tabular Q learning algorithm explained above.

First, we note that the update of $\hat{Q}$ works well in the case of small state and action spaces as one can expect to visit each state-action pair sufficiently often for $\hat{Q}_N$ to approximate well Q as N grows large enough. However, when the state space grows or becomes continuous, this assumption is no longer respected and thus updating values of $\hat{Q}$ per state-action pair (for example by updating a cell of a $\mathcal{S} \times \mathcal{A}$ table) is no longer relevant. To counteract this problem, one can use a function approximator which would be expected to generalize better to unseen state-action pairs, allowing for better decisions of the policy in less frequently visited states. For example, Ernst et al. (2005) used random forests to model the Q function. More recently, it was also shown that using deep networks to model the Q function could prove very effective (Mnih et al., 2013; Van Hasselt et al., 2016). Overall, due to the increasing complexity of RL environments, function approximators have become an essential tool when tackling model-free RL with value iteration methods.

Second, when exploiting the value function to get the optimal action, the action is taken as

$$\mathbf{a} = \arg\max_{\mathbf{a}' \in \mathcal{A}} \hat{Q}(\mathbf{s}, \mathbf{a}') \quad .$$

When solving MDPs where the action space $\mathcal{A}$ is continuous, the arg max operation quickly becomes a limiting factor. Indeed, since the action space cannot be iterated over, it is difficult to compute the best action. Some recent work have focused on alleviating this problem. One of the most common ways to solve this problem is to use actor-critic algorithms, as described here under, however other methods exist as well. Among others, Gu et al. (2016) proposed to make the function approximator convex and Kalashnikov et al. (2018) proposed to use optimization to find local maxima.

It is important to note that only the most basic value iteration method has been described here. Many improvements have been proposed over the years in order to get better approximations of the true value functions. Among many others, Z. Wang et al. (2016) for example proposed using dueling networks while Van Hasselt et al. (2016) proposed to use two estimators to reduce the overestimation bias of Q which is known to happen with Q-learning. Also, not all value iteration algorithms focus on modelling the Q function. One can for example learn to approximate the state value function V rather than the state-action value function Q. Going one step further, Wiering (2005) showed that there can even be benefits to learning both the Q and V function. The concept behind all those methods remains similar however. The goal is always to learn a (or multiple) value function(s) which gives a value to a state (or state-action pair), and use these values to infer a policy.

3.2 Policy iteration methods

Similarly to tabular Q learning for value iteration methods, policy gradient algorithms have been used early as a way to solve model-free RL tasks. Let $\pi_\theta(\mathbf{a}|\mathbf{s})$ denote the policy used by the agent with parameters θ. This policy can for example be modelled by a neural network whose outputs are the means vector and variance matrix of a multivariate Gaussian distribution. Ideally, with such a policy, the goal of RL would be to update it as

$$\theta \leftarrow \theta + \alpha \nabla_\theta [\mathop{\mathbb{E}}_{\mathbf{s_0} \sim p_{\mathbf{s}_0}} J^{\pi_\theta}(\mathbf{s}_0)]$$

such as to maximize its expected return where p_0 is the distribution over initial states of the MDP defined over $\mathcal{S}$. This would be the ideal policy gradient, however, in a model-free setting one does not have access to this gradient. Hence, the goal of policy gradient methods is to approximate this update as close as possible. One of the simplest way to achieve this is with Monte Carlo estimation. For ease of writing, let $r(\mathbf{h})$ denote the sum of discounted rewards in an history such that, given $\mathbf{h} = [\mathbf{s}_0, \mathbf{a}_0, r_0, \ldots, \mathbf{s}_T, \mathbf{a}_T, r_T]$,

$$r(\mathbf{h}) = \sum_{t=1}^{T} \gamma^t r_t \quad .$$

The expected return of policy π_θ can thus be written as

$$\mathop{\mathbb{E}}_{\mathbf{s_0} \sim p_{\mathbf{s}_0}} J^{\pi_\theta}(\mathbf{s}_0) = \int \pi_\theta(\mathbf{h}) r(\mathbf{h}) d\mathbf{h} \tag{3.1}$$

where $\pi_\theta(\mathbf{h})$ is the probability of observing history $\mathbf{h}$ playing policy π_θ, that is

$$\pi_\theta(\mathbf{h} = [\mathbf{s}_0, \mathbf{a}_0, \ldots, \mathbf{s}_T, \mathbf{a}_T]) = p_{\mathbf{s_0}}(\mathbf{s_0}) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \quad .$$

The simplest way to derive a policy gradient is then approximate $\nabla_\theta \mathop{\mathbb{E}}_{\mathbf{s_0} \sim p_{\mathbf{s}_0}} J^{\pi_\theta}(\mathbf{s}_0)$ as follows,

$$\begin{aligned}
\nabla_\theta \mathop{\mathbb{E}}_{\mathbf{s_0} \sim p_{\mathbf{s}_0}} J^{\pi_\theta}(\mathbf{s}_0) &= \nabla_\theta \int \pi_\theta(\mathbf{h}) r(\mathbf{h}) d\mathbf{h} \\
&= \int \nabla_\theta \pi_\theta(\mathbf{h}) r(\mathbf{h}) d\mathbf{h} \\
&= \int \pi_\theta(\mathbf{h}) \nabla_\theta \ln \pi_\theta(\mathbf{h}) r(\mathbf{h}) d\mathbf{h} \\
&= \mathop{\mathbb{E}}_{\mathbf{h} \sim \pi_\theta, p_{\mathbf{s}_0}, p} [\nabla_\theta \ln \pi_\theta(\mathbf{h}) r(\mathbf{h})] \\
&= \mathop{\mathbb{E}}_{\mathbf{h} \sim \pi_\theta, p_{\mathbf{s}_0}, p} \left[r(\mathbf{h}) \sum_t \nabla_\theta \ln \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \right] \\
&= \mathop{\mathbb{E}}_{\mathbf{h} \sim \pi_\theta, p_{\mathbf{s}_0}, p} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \left(\sum_{t'=0}^{T} \gamma^{t'} r_{t'} \right) \right] \\
&= \lim_{N \to \infty} \sum_{i=1}^{N} \frac{1}{N} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \left(\sum_{t'=0}^{T} \gamma^{t'} r_{t'} \right) \right]_{\substack{\mathbf{s}_{t+1} \sim p(\mathbf{s}_t, \mathbf{a}_t) \\ \mathbf{s}_0 \sim p_{\mathbf{s}_0} \\ \mathbf{a}_t \sim \pi_\theta(\mathbf{s}_t)}}
\end{aligned}$$

As such, this value can be approximated by playing π_θ on the environment and collecting reward samples. In practice, it is of course impossible to play an infinite number of time-steps on the environment. Therefore, the agent will play the environment multiple times. Each time the agent plays, it will do so for T time-steps, the agent will then start a new trajectory on the environment from a new initial state $\mathbf{s}_0 \sim p_{\mathbf{s}_0}$. Each of these T interactions with the environment is called an episode. Denoting by $r_{i,t}$ the reward obtained by the agent (playing policy π_θ) at time-step t of episode i, the approximated gradient over N episodes finally writes

$$\nabla_\theta \underset{\mathbf{s}_0 \sim p_{\mathbf{s}_0}}{\mathbb{E}} J^{\pi_\theta}(\mathbf{s}_0) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \left(\sum_{t'=0}^{T} \gamma^{t'} r_{i,t'} \right) \right] \tag{3.2}$$

and the policy gets updated following

$$\theta \leftarrow \theta + \alpha \nabla_\theta [\underset{\mathbf{s}_0 \sim p_{\mathbf{s}_0}}{\mathbb{E}} J^{\pi_\theta}(\mathbf{s}_0)] \quad .$$

One of the most common policy gradient algorithms, REINFORCE (Williams, 1992), uses a very similar update, written below.

$$\theta \leftarrow \theta + \alpha \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'} \right) \right]$$

As such, the only difference with our derived policy gradient update, is that the term $\left(\sum_{t'=0}^{T} \gamma^{t'} r_{i,t'} \right)$ in Equation 3.2 gets replaced by the reward to go $\left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'} \right)$, based on the principle that an action taken at time-step t can only influence rewards at time-steps $t+1$ and beyond. Both updates lead to a similar expected value for the policy expected gradient, despite having different variance. The REINFORCE update is easy to understand in that, the higher the cumulative future return $\left(\sum_{t'=t}^{T} \gamma^{t'-t} r_{i,t'} \right)$ obtained after making an action $\mathbf{a}_{i,t}$ in state $\mathbf{s}_{i,t}$, the more likely the agent will be to play that action in that state again. There are multiple advantages to policy gradient:

1. Continuous action spaces are easily managed by the algorithm;
2. Exploration is handled automatically thanks to the variance of the policy. Whenever the agent does not manage to obtain good rewards, the variance of the policy will tend to increase, while it will tend to decrease whenever the agent finds a good suit of actions.

The main drawback of those methods however, is that they are highly subject to variance. Indeed, the rewards generated between runs of a given policy may vastly differ, sometimes leading to very bad updates. It is known that subtracting a baseline to the expected cumulative return when multiplying the gradient can help alleviate this problem. In a similar vein, the critic in actor-critic algorithms precisely helps at diminishing the variance of the policy gradient.

Finally, similarly to Q learning, it is important to note that many alternative ways to carry policy gradient have been proposed and that REINFORCE is only one of the most basic. For example, Schulman, Levine, et al. (2015) proposed to add constraints to the policy updates such that it cannot vary too abruptly. Also, Silver et al. (2014) proposed deterministic policy gradient algorithms in which a deterministic policy is used and where the gradient of the policy follows that of the state-action value function. There exist many more such alternatives (Casas, 2017; Kakade, 2001; Schulman et al., 2017) and next section focuses on three particular concepts: the advantage actor-critic (A2C) framework, using proximal policy optimisation (PPO) and generalised advantage estimations.

3.3 Advantage actor-critic

** Parts of this subsection have directly been adapted from the supplementary material of Vecoven et al. (2020)*

The goal of actor-critic algorithms is to simultaneously learn a policy and a value function. More formally, let $\theta \in \Theta$ and $\psi \in \Psi$ denote the parameters of the actor and critic respectively. We define π_θ and c_ψ as the policy and critic functions. Let us also define π_{θ_k} and c_{ψ_k} as the models for the policy and critic after k updates of the parameters θ and ψ, respectively. Finally, let R^{π_θ} denote the sum of discounted rewards obtained when playing policy π_θ on the MDP such that

$$R^{\pi_\theta} = \lim_{T\to\infty} \sum_{t=0}^{T} \gamma^t r_t \quad .$$

An AC algorithm should interact with its environment to find the value of θ that leads to high values of the expected return given a probability distribution over the initial states. This expected return is written as:

$$\mathop{\mathbb{E}}_{\substack{\mathbf{s}_0\sim p_{\mathbf{s}_0}\\ \mathbf{a}_t\sim\pi_\theta}} [R^{\pi_\theta}] \quad .$$

One the one hand and when working well, actor critic algorithms produce a sequence of policies $\pi_{\theta_1}, \pi_{\theta_2}, \pi_{\theta_3}, \dots$ whose expected returns increase as the iterative process evolves and eventually reaches values close to those obtained by π_{θ^*} with $\theta^* = \arg\max_{\theta\in\Theta} \mathop{\mathbb{E}}_{\substack{\mathbf{s}_0\sim p_{\mathbf{s}_0}\\ \mathbf{a}_t\sim\pi_\theta}} R^{\pi_\theta}$, which, if π^θ is flexible enough, are themselves close to those obtained by an optimal policy π^* defined as:

$$\pi^* \in \arg\max_{\pi \in \Pi} \mathop{\mathbb{E}}_{\substack{\mathbf{s}_0 \sim p_{\mathbf{s}_0} \\ \mathbf{a}_t \sim \pi_\theta}} [R^\pi] \quad .$$

On the other hand, and again for an efficient actor-critic algorithm, the value of the critic for $\mathbf{s}_t$, $c_\psi(\mathbf{s}_t)$, should be updated at each iteration k in a direction that provides a better approximation of $J^{\pi_{\theta_{k-1}}}(\mathbf{s}_t)$. Note that this "lag" between the critic and the policy is due to the fact that policy π_{θ_k} is used to generate the interactions which will allow to update ψ_k to ψ_{k+1}. Ideally, as k grows and $\mathop{\mathbb{E}}_{\mathbf{s}_0 \sim p_{\mathbf{s}_0}} J^{\pi_{\theta_k}}(\mathbf{s}_0)$ gets closer to $\mathop{\mathbb{E}}_{\mathbf{s}_0 \sim p_{\mathbf{s}_0}} J^{\pi_{\theta^*}}(\mathbf{s}_0)$, the critic should learn to approximate J^{π^*}.

Existing actor-critic algorithms mainly differ from each other by the way the actor and critic are updated. While in early actor-critic algorithms the critic was directly used to compute the direction of update for the actor's parameters (see for example the REINFORCE policy updates using the critic's output as estimated return), now it is more common to use an advantage function (Mnih et al., 2016). This function represents the advantage in terms of return of selecting specific actions given a trajectory history (or simply a state) over selecting them following the policy used to generate the trajectories. In the context of this manuscript, generalised advantage estimations (GAE) are used, as introduced by Schulman, Moritz, et al. (2015). More recently, it has been shown that avoiding too large policy changes between updates can greatly improve learning (Schulman, Levine, et al., 2015; Schulman et al., 2017). Therefore, while in classical AC algorithms the function used to update the actor aims at representing directly the gradient of the actor's return with respect to its parameters, such algorithms rather update the actor's parameters θ by minimising a loss function that represents a surrogate objective. In this manuscript, a surrogate function that is similar to the one introduced by Schulman et al. (2017) is used with an additional loss term that proved to increase (albeit slightly) the performance of the algorithm on our experiments in all cases.

In the context of this manuscript, the actors and critics are modelled by artificial neural networks. They are both updated through a gradient descent process which is described in the remaining of this Chapter.

3.3.1 Actor update

Let $\mathcal{H}_k$ be a set of the interactions used to update ψ_k and θ_k to ψ_{k+1} and θ_{k+1}. This set is usually built with multiple episodes of the agent. Let $\mathbf{s}_{i,t}$, $\mathbf{a}_{i,t}$ and $r_{i,t}$ respectively denote the state, action and reward of episode i at time-step t. In practice, it is obvioulsy impossible for the agent to interact with the environment for an infinite number of time-steps. Therefore, the stochastic gradient descent will be carried only on the $T \in \mathcal{N}_0$ first time-steps of each trajectory (using upwards to the $T' \in \mathcal{N}_0$ first time-steps of each trajectory to compute cumulative rewards). Let $B \in \mathbb{N}_0$ be the number of such trajectories used for the update, then we define $\mathcal{H}_k$ as

$$\mathcal{H}_k = \{\mathbf{h}_{Bk,T}, \ldots, \mathbf{h}_{B(k+1)-1,T}\} \quad ,$$

where

$$\mathbf{h}_{i,T} = \{\mathbf{s}_{i,0}, \mathbf{a}_{i,0}, r_{i,0}, \ldots, \mathbf{a}_{i,T-1}, r_{i,T-1}, \mathbf{s}_{i,T}\} \quad .$$

With these definitions, it is noted that $\mathcal{H}_k$ does not represent the full set of interactions used to update θ_k and ψ_k, but rather the set of interactions for which losses will be computed.

Generalised advantage estimations (GAE) In this context, one can use the critic to define an approximate temporal error difference for any two consecutive time-steps of any episode:

$$TD_t = r_t + \gamma * c_{\psi_k}(\mathbf{s}_{t+1}) - c_{\psi_k}(\mathbf{s}_t), \ \forall t \in \mathcal{N}$$

where ψ_k denotes the critic's parameters for playing the given episode. This temporal difference term represents, in some sense, the (immediate) advantage obtained, after having played action a_t over what was expected by the critic. If c_{ψ_k} was the true estimate of $J^{\pi_{\theta_k}}$ and if the policy played was π_{θ_k}, the expected value of these temporal differences would be equal to zero. We now define for each trajectory i, the GAE terms that will be used later in our loss functions:

$$GAE_{i,t} = \sum_{t'=t}^{T'} (\gamma * \lambda)^{t'-t} * TD_{i,t'}, \ \forall t \in \{0, \ldots, T\} \tag{3.3}$$

where $\lambda \in [0, 1]$ is a discount factor used for computing GAEs, $TD_{i,t}$ is the value of TD_t for trajectory i. It appears clearly here that T' has to be chosen in combination with T in order to have a value of $GAE_{i,t}$ that accurately approximates

$$\sum_{t'=t}^{\infty} (\gamma * \lambda)^{t'-t} * TD_{i,t'} \forall i, t \quad .$$

Note that the value chosen for T also needs to be sufficiently large to provide the loss function with a sufficient number of GAE terms. These GAE terms, introduced by Schulman, Moritz, et al. (2015), represent the exponential average of the discounted future advantages observed. Thanks to the fact that GAE terms can catch the accumulated advantages of a sequence of actions rather than of a single action, as it is the case with the temporal difference terms, they can better represent the advantage of the new policy played by the AC algorithm over the old one (in terms of future discounted rewards). It is important to note that it is also best to normalize GAE terms, such that they have a zero mean and standard deviation of 1. This allows for the policy learning rate to be independent of the magnitude of those terms.

Proximal policy optimization Once advantages have been computed, the values of θ_{k+1} are computed using updates that are strongly related to PPO updates with a Kullback Leibler (KL) divergence implementation (Schulman et al., 2017). The loss used in PPO updates is composed of two terms: a standard policy gradient term $L_{vanilla}$ and a penalisation term L_{ppo}. The first is defined as

$$L_{vanilla}(\mathbf{a}_{i,t}, \mathbf{s}_{i,t}; \theta) = -\frac{\pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})}{\pi_{\theta_k}(\mathbf{a}_{i,t}|\mathbf{s}_{i,t})} * GAE_{i,t} \quad . \tag{3.4}$$

One can easily become intuitive about Equation 3.4 as, given an history $\mathbf{h}_{i,t}$, minimising this loss function tends to increase the probability of the policy taking actions leading to positive advantages (i.e. $GAE_{i,t} > 0$) and decreases its probability to take actions leading to negative advantages (i.e. $GAE_{i,t} < 0$). As such, minimizing this loss has the exact effect to that of a REINFORCE update.

It has been found (Schulman, Levine, et al., 2015) that to obtain good performance with this above-written loss function, it is important to have a policy that does not change too rapidly from one iteration to the other. The penalisation term introduced in PPO is one way to do so, but other methods exist such as trust region policy optimisation proposed by Schulman, Levine, et al. (2015). Before explaining how this is achieved with the PPO update, it is useful to understand why it may be important to have slow updates of the policy. Taking a look back at the loss function given by Equation 3.4, one can see that minimising this loss function will give a value for θ_{k+1} that will lead to higher probabilities of selecting actions corresponding to high values of the advantages $GAE_{i,t}$. A potential problem is that these advantages are not really related to the advantages of the would-be new policy $\pi_{\theta_{k+1}}$ over π_{θ_k} but are instead related to the advantages of policy π_{θ_k} over $\pi_{\theta_{k-1}}$. Indeed, the advantages $GAE_{i,t}$ are computed using the value function c_{ψ_k}, whose parameters have been updated from ψ_{k-1} in order to better approximate the sum of discounted rewards obtained during the episodes played with policy $\pi_{\theta_{k-1}}$. It clearly appears that ψ_k has, in fact, been updated to approximate discounted rewards obtained through the policy $\pi_{\theta_{k-1}}$ (used to play episodes for update $k-1$). A solution to this problem is to constraint the minimisation to reach a policy $\pi_{\theta_{k+1}}$ that does not stand too far from π_{θ_k}. We may reasonably suppose that the advantage function used in (3.4) still correctly reflects the real advantage function of $\pi_{\theta_{k+1}}$ over π_{θ_k}. To achieve this, we add a penalisation term $L(\mathbf{s}_{i,t};\theta)$ to the loss function. In the PPO approach, the penalisation term is $L_{ppo}(\mathbf{s}_{i,t};\theta) = \beta_k * d(\mathbf{s}_{i,t};\theta)$, where:

- β_k is an adaptive weight, whose goal is to scale how strong the penalisation should be. A too low value of β_k could lead to too big updates, and thus too high variance in the training process. However, too high values of β_k would lead to too small updates, and thus to extremely slow learning.
- $d(\mathbf{s}_{i,t};\theta) = KL(\pi_{\theta_k(.|\mathbf{s}_{i,t})}, \pi_{\theta(.|\mathbf{s}_{i,t})})$, where KL is the Kullback-Leibler divergence. This term grows as the distribution π_{θ_k} and π_θ diverge.

In the context of the experiments of this manuscript, it was seen that adding another penalisation term (squared hinge loss) to L_{PPO} to further penalise the KL divergence, in cases where it surpasses $2 * d_{targ}$, improved algorithm performance. The final expression of the penalisation term is:

$$L_{ppo}(\mathbf{s}_{i,t};\theta) = \beta_k * d(\mathbf{s}_{i,t};\theta) + \delta * \max(0, d(\mathbf{s}_{i,t};\theta) - 2 * d_{targ})^2$$

where δ is a hyper-parameter that weights the third loss term. Finally, the loss function L that we minimise as a surrogate objective becomes:

$$L(\mathbf{a}_{i,t}, \mathbf{s}_{i,t};\theta) = L_{vanilla}(\mathbf{a}_{i,t}, \mathbf{s}_{i,t};\theta) + L_{ppo}(\mathbf{s}_{i_t};\theta) \tag{3.5}$$

One can then simply update the parameters θ_k of the neural network π_{θ_k} with standard stochastic gradient descent on the dataset $\mathcal{H}_k$ using the loss of Equation 3.5. Details on the computation of the KL divergence, as well as on the way the distribution π_{θ_k} is modeled with an ANN is given in Appendix A.

β_k is updated at each gradient step using a hyper-parameter $d_{targ} \in \mathbb{R}_0$ called the divergence target. Schulman et al. (2017) proposed the following procedure for this update:

$$\beta_{k+1} = \begin{cases} \frac{\beta_k}{1.5} & \text{if } \sum\limits_{\mathbf{s}_{i,t}\in\mathcal{B}} d(\mathbf{s}_{i,t};\theta) < \frac{d_{targ}}{2} \\ \beta_k * 1.5 & \text{if } \sum\limits_{\mathbf{s}_{i,t}\in\mathcal{B}} d(\mathbf{s}_{i,t};\theta) > d_{targ} * 2 \\ \beta_k & \text{otherwise} \end{cases} \tag{3.6}$$

where $\mathcal{B}$ is the mini-batch sampled for each gradient step and $\hat{\sum}$ denotes the average value over a set. With this update strategy, the penalisation term will tend to evolve in a way such that the KL divergence between two successive policies does not tend to go too far beyond d_{targ} without having to add an explicit constraint on d, as was the case in Trust Region Policy Optimization (TRPO) updates introduced by Schulman, Levine, et al. (2015).

3.3.2 Critic update

The critic is updated at iteration k in a way to better approximate the expected return obtained when following the policy π_{θ_k}, starting from a given trajectory history. To this end, a mean-square error loss is used as a surrogate objective for optimizing ψ. First, an approximate of the expected discounted cumulative return of policy π_{θ_k} is defined:

$$\hat{R}_{i,t}^{\pi_{\theta_k}} = \sum_{j=t}^{T'} \gamma^{j-t} * r_{i,t} \quad .$$

Note that if $T \to \infty$ this tends to the true discounted cumulative return of policy π_{θ_k}, $R_t^{\pi_{\theta_k}}$ where

$$R_t^{\pi_{\theta_k}} = \lim_{T\to\infty} \sum_{k=0}^{T} \gamma^k r_{t+k} \quad .$$

From there, the loss used to update the critic can simply be written:

$$L(\mathbf{h}_{i,t};\psi) = (c_\psi(\mathbf{s}_{i,t}) - \hat{R}_{i,t}^{\pi_{\theta_k}})^2 \quad . \tag{3.7}$$

In the case of the critic updates, the gradient descent is often not only carried on the set $\mathcal{H}_k$ but well on the sets $\mathcal{H}_{k-rb}, \ldots, \mathcal{H}_k$. As such, the critic not only uses the trajectories of update k but also all those played since update $k-b$. This is called a replay **b**uffer (hence, b), something that is often used in value iteration algorithms. Such buffers have for effect to smooth the critics' updates, often improving algorithm performance. Due to the replay buffer, the updates are not such that c_ψ directly approximates the average expected return of the policy π_{θ_k}. Rather, the updates are such that c_ψ directly approximates the average expected return obtained by the last $b+1$ policies played. In practice, this does not lead to problems, and efficiency of such algorithm is often very good.

As a final observation, note that the loss (3.7) is only computed on the $T \ll T'$ first time-steps of each episode, as was the case for the actor. The reason behind this choice is simple. The value function c_{ψ_k} should approximate $\sum_{t'=t}^{+\infty} \gamma^{t'-t} * r_{i,t'}$ for every $\mathbf{s}_{i,t}$. However, this approximation can become less accurate when t becomes close to T' since we can only guarantee $\hat{R}_{i,t}^{\pi_{\theta_k}}$ to stand in the interval: $[\sum_{t'=t}^{+\infty} \gamma^{t'-t} * r^{i,t} - \frac{\gamma^{T'-t}}{1-\gamma} R_{max}, \sum_{t'=t}^{+\infty} \gamma^{t'-t} * r^{i,t} - \frac{\gamma^{T'-t}}{1-\gamma} R_{min}]$, where R_{max} and R_{min} are respectively the minimum and maximum bounds of the reward function.

Summary

The two main classes of algorithm for solving model-free RL were presented in this Chapter. Both of them were formalized using the most basic corresponding algorithms, i.e. tabular Q learning for value iteration methods and REINFORCE for policy iteration methods. Through these introductions, the reader was exposed to the short-comings of both methods, leading to the presentation of a state-of-the-art algorithm for solving model-free RL. This algorithm, which will be used in Part III of this manuscript, is based on three recently introduced concepts. First, it uses an A2C framework; Second, policy updates are done using proximal policy optimisation; And third, advantages are computed using generalised advantage estimation. By reading this Chapter, the reader is expected to have understood the different strong points and short comings of classical model-free RL methods as well as the specific concepts used in this manuscript to address those issues.

PART II

Tuning neuron dynamics for long-term memory

Introduction

While there exists many datasets for which the input is unstructured, there also exists many more where the input is highly structured. Images, which hold a rich spatial structure, are a common example of the latter. While benefiting from such structure is not doable with all machine learning models, luckily, ANNs are very flexible. Thus, there exists a multitude of neural architectures which are all more or less well suited towards different types of data. As an example, convolutional neural networks (CNNs) are a great tool to introduce a bias towards the structure of images. In this type of architecture, filters are learned and applied onto patches of neighbouring pixels, extracting relevant spatial features from raw pixel values. The same filters are applied over all the image, patch by patch, effectively introducing spatial bias into the model over a fully connected network. This method was shown to greatly improve predictive accuracy in such cases.

A lot of interest has been given to a particular type of data, for which structure comes in the form of having ordered inputs or outputs: "sequential data". Natural language processing and time-series forecasting are two great examples of having sequential inputs. Indeed, when processing natural language, words (or letters) are often used as input features. In such a setting, it is easy to understand why introducing a bias towards the model taking word order into account is important. For example, if this is not the case, the two following sentences:

- "The disgusting looking apple I ate was good.";
- "The good looking apple I ate was disgusting.";

are exactly the same, despite having opposite meanings. Similarly, by definition, time-series are composed of data points ordered by time. As such, it is also important to use that order in the manner to find relevant temporal patterns that can arise from such data. It is noted that outputs can also be sequential, as would be the case for music generation and speech synthesis among others.

Due to the plethora of applications for which data is sequential, many neural architectures suited to tackle such structured data have been proposed. In particular, one-dimensional CNNs [2], position transformers (Liu et al., 2020) and RNNs are the first that come to mind. In this Part of the manuscript, a special interest is given to the latter.

2. One-dimensional CNNs are based on the same principle as standard CNNs. Instead of learning filters over patches of neighbouring samples in a spatial domain, one rather learns filters over patches of neighbouring samples in a one dimensional domain, for example, filters over consecutive data points in the context of time-series.

RNNs can be defined as neural networks for which neurons are connected to themselves (and others) through time. They have been discussed as soon as ANNs were introduced; in fact, McCulloch & Pitts (1943) already discussed the idea of neural nets with cycles when discussing their logical representation of neural networks. The first discussion on how to train such network was made as early as 1985 by Rumelhart et al. (1985). In that paper, it was shown that any recurrent neural network (over a finite period of time) can be represented by a non-recurrent neural network and that this property could be used to train such networks. Thanks to their temporal connections (called recurrent connections) RNNs can exhibit temporal dynamics, effectively endowing them with memory capabilities. This memory is what makes them specifically well-suited to handle sequential data. Indeed, as each input is fed to the network, it uses its current memory state (called internal state) and that input in order to produce an output and update its internal state. As such, when receiving an input, the network's state is always dependant on all the previous inputs received, as well as on their order.

Despite having exhibited state-of-the-art on multiple tasks over the years (Lipton et al., 2015), RNNs have been victim to some caveats.

- Mainly and most importantly, they are known to be difficult to train, especially for sequential data where patterns can span hundreds of inputs.
- Also, due to their inherent complexity, RNNs have often been seen as black boxes for which the underlying dynamics are unclear.

Many recent works have focused on improving the ease of training of RNNs. For example, over the years, one of the most common way to achieve this has become to use gated units (Cho et al., 2014; Hochreiter & Schmidhuber, 1997) as recurrent neurons. When it comes to addressing the second caveat, among others, Sussillo & Barak (2013) have recently focused on understanding the dynamics of trained RNNs, highlighting some important properties when it comes to their prediction process. Research in the field is still ongoing however, and the goal of this part is to introduce new solutions for easier training of RNNs on long-term dependencies. Furthermore, those solutions are well-motivated by non-linear control theory, allowing a deeper understanding on their workings.

In particular, Chapter 4 formalizes standard RNNs and introduces state-of-the-art methods for improving their training on long sequences. Furthermore, the Chapter will also present and discuss some important properties of trained RNNs following notions of dynamical systems theory. Chapter 5 will then focus on bistable recurrent cells (BRC), a new cell that is specifically designed for having bistability properties. The cell is shown to exhibit excellent performance on very long sequences, for which classical gated architectures fail to learn. This Chapter finally hints at some links that can in fact be made between gated cells and human neurons, despite their link being thought to be extremely shallow. Finally, Chapter 6 will introduce a more general technique called "warm-up" for handling long sequences with RNNs. As opposed to bistable recurrent cell, this technique can be used with any kind of recurrent neuron and it is shown to greatly enhance the performance of RNNs when sequences grow longer.

Chapter 4

Handling sequential data with RNNs

While Part I presented basic concepts on ANNs, this Chapter really focuses on recurrent networks. To this end, Section 4.1 will formalize RNNs and detail the difficulties that can arise when training such networks, providing more basic knowledge on RNNs. Section 4.2 will then introduce and discuss common methods to solve some of these issues. Finally, Section 4.3 will focus on the analysis of trained networks. Indeed, similarly to some of the decomposition based feature selection methods (which aim at understanding the role of each neuron towards predicting the output), there has been a rising interest towards understanding the underlying dynamics of trained RNNs. As such, Section 4.3 will discuss some recent findings on the subject, through the lens of control theory.

4.1 Recurrent neural networks

** A few parts of this Section have been adapted from Vecoven et al. (2021a).*

As mentioned earlier, RNNs are especially suited to problems where the input is ordered, and in particular, for problems with temporal structure such as time-series. In such a context, relevant information can only be captured by processing observations obtained during multiple time-steps. More formally, a time-series can be defined as $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ with $T \in \mathcal{N}_0$ and $\mathbf{x}_i \in \mathcal{R}^n$. To capture time-dependencies, RNNs maintain a recurrent internal state whose update depends on the previous internal state and current observation of a time-series, making them dynamical systems and allowing them to handle arbitrarily long sequences of inputs. Mathematically, RNNs maintain a hidden state $\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$, where $\mathbf{h}_0$ is a constant and θ are the parameters of the network. In its most standard form, an RNN is fully connected through time, and thus a standard update would be written as

$$\mathbf{h}_t = g(U\mathbf{x}_t + W\mathbf{h}_{t-1}) \quad , \tag{4.1}$$

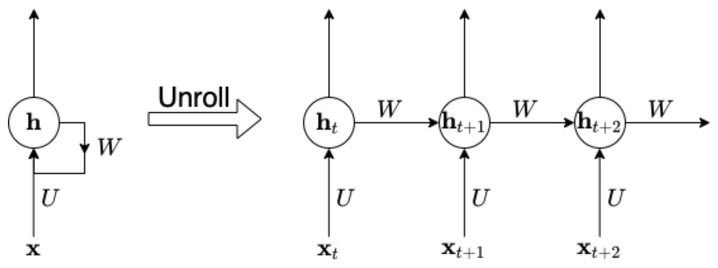


Figure 4.1: Unrolling of an RNN shown over three time-steps.

where g is a standard activation function such as a sigmoid or a hyperbolic tangent. Recurrent layers also output a value $\mathbf{o}_t = o(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta)$. Frequently, this output value is taken directly as the hidden state such that $\mathbf{o}_t = \mathbf{h}_t$. As a final note on recurrent architectures, it is worth to mention that one can of course stack multiple recurrent layers when building an RNN. As for feed-forward networks, these layers can be linked sequentially through $\mathbf{x}_t^i = \mathbf{o}_t^{i-1}$ with $\mathbf{x}_t^0 = \mathbf{x}_t$, where $\mathbf{o}_t^i$ denotes the output of layer i at time-step t and $\mathbf{x}_t^i$ its input.

Such networks can be trained as standard ANNs by using stochastic gradient descent. To achieve this, one can compute the gradient using backpropagation through time (Werbos, 1990) over an unrolled version of the RNN (as shown on Figure 4.1). When seen as such, the network thus becomes a standard fully connected network with layers using shared weights. In such a context, the depth of the network will be proportional to the number of time-steps unrolled, hence why backpropagating the gradient in such a network amounts to do it through time.

It is interesting to discuss the form that such gradient takes as the unrolling grows longer. Indeed, the mathematical expression of h_t can be written as

$$\mathbf{h}_t = \underbrace{f(\mathbf{x}_t, f(\mathbf{x}_{t-1}, f(\ldots); \theta); \theta)}_{t}$$

which can lead to some bad mathematical properties of the gradient when t grows too large. To highlight this, one can think of the gradient of $\mathbf{h}_t$ with respect to $\mathbf{x}_0$ in the case of a single layer RNN. First, to ease the writing for this discussion,

$$\mathbf{F}_n(\theta, \mathbf{h_{n-1}}(\theta)) = U\mathbf{x}_n + W\mathbf{h}_{n-1}(\theta)$$

is defined for all n in $\mathcal{N}_0$. A modified version, $\bar{\mathbf{F}}_n$, of $\mathbf{F}_n$ is also introduced. $\bar{\mathbf{F}}_n$ is equal to $\mathbf{F}_n$ for which $\mathbf{h}_{n-1}$ is considered constant and thus, does not depend on θ. This allows to write $\frac{\delta \bar{\mathbf{F}}_i}{\delta \theta}$ to denote the "immediate" partial derivative [1] of state $\mathbf{h}_t$ with respect to θ. In such a context,

$$\mathbf{h}_t = f(\mathbf{x_t}, \mathbf{h}_{t-1}; \theta) = g(\mathbf{F}_t) \quad .$$

Thanks to the chain rule, the jacobian of $\mathbf{h}_t$ with respect to θ can be expressed as:

$$\begin{aligned}
\frac{\delta \mathbf{h}_t}{\delta \theta} &= \frac{\delta \mathbf{h}_t}{\delta \mathbf{F}_t} \left(\frac{\delta \bar{\mathbf{F}}_t}{\delta \theta} + \frac{\delta \mathbf{F}_t}{\delta \mathbf{h}_{t-1}} \frac{\delta \mathbf{h}_{t-1}}{\delta \theta} \right) && (4.2) \\
&= diag(g'(\mathbf{F}_t)) \left(\frac{\delta \bar{\mathbf{F}}_t}{\delta \theta} + \frac{\delta \mathbf{F}_t}{\delta \mathbf{h}_{t-1}} \frac{\delta \mathbf{h}_{t-1}}{\delta \theta} \right) && (4.3) \\
&= diag(g'(\mathbf{F}_t)) \frac{\delta \bar{\mathbf{F}}_t}{\delta \theta} + diag(g'(\mathbf{F}_t)) \frac{\delta \mathbf{F}_t}{\delta \mathbf{h}_{t-1}} \frac{\delta \mathbf{h}_{t-1}}{\delta \theta} \quad . && (4.4)
\end{aligned}$$

where g' computes the derivative of g element-wise and $diag(\cdot)$ converts a vector into a diagonal matrix. Using $\mathbf{h}_0 = \mathbf{0}$, by recurrence, one can replace the expression of $\frac{\delta \mathbf{h}_{t-1}}{\delta \theta}$ in Equation 4.4, which leads to the following expression for all $t \geq 1$:

$$\begin{aligned}
\frac{\delta \mathbf{h}_t}{\delta \theta} &= diag(g'(\mathbf{F}_t)) \frac{\delta \bar{\mathbf{F}}_t}{\delta \theta} + \sum_{i=1}^{t-1} diag(g'(\mathbf{F}_i)) \left(\prod_{j=i+1}^{t} \frac{\delta \mathbf{F}_j}{\delta \mathbf{h}_{j-1}} diag(g'(\mathbf{F}_j)) \right) \frac{\delta \bar{\mathbf{F}}_i}{\delta \theta} && (4.5) \\
&= diag(g'(\mathbf{F}_t)) \frac{\delta \bar{\mathbf{F}}_t}{\delta \theta} + \sum_{i=1}^{t-1} diag(g'(\mathbf{F}_i)) \left(\prod_{j=i+1}^{t} W^T diag(g'(\mathbf{F}_j)) \right) \frac{\delta \bar{\mathbf{F}}_i}{\delta \theta} \quad . && (4.6)
\end{aligned}$$

To understand the problem with such an expression and for simplicity, one can look at the expression of the derivative of $\mathbf{h}_t$ with respect to the recurrent weights W in the context of a single neuron, that is, when W and $\mathbf{h}$ are scalar (hence written w and h in the following). In this context, Equation 4.6 leads to

1. This term was introduced in a similar context by Pascanu et al. (2013).

$$\frac{\delta h_t}{\delta w} = g'(F_t)h_t + \sum_{i=1}^{t-1} g'(F_i) \left(\prod_{j=i+1}^{t} wg'(F_j) \right) h_{i-1} \tag{4.7}$$

$$= g'(F_t)h_t + \sum_{i=1}^{t-1} g'(U\mathbf{x}_i + wh_{i-1}) \left(\prod_{j=i+1}^{t} wg'(U\mathbf{x}_j + wh_{j-1}) \right) h_{i-1} \quad . \tag{4.8}$$

From Equation 4.8, one can see that long-term components of the equation are likely to either vanish or explode. For example, in Equation 4.8 the value of the first input $\mathbf{x}_1$ only appears in the following term

$$g'(U\mathbf{x}_1 + wh_0) \left(\prod_{j=2}^{t} wg'(U\mathbf{x}_j + wh_{j-1}) \right) h_0 \quad , \tag{4.9}$$

which can, on the one hand,

- **vanish** if w is smaller than 1. Indeed, as already mentioned, activation functions used in RNNs are often either sigmoids or hyperbolic tangents (because they are bounded, which permits to avoid exploding states when doing a forward pass of the network). The sigmoid derivative has an image that belongs to $]0, 0.25]$ whereas the hyperbolic tangent derivative has an image that belongs to $]0, 1]$. In both cases, if $w < 1$, Equation 4.9 will tend towards 0 as t increases (due to the small values of the derivative, it might also be the case even if $w > 1$). This shows that in such cases, the first input does not influence the gradient at all, making it impossible for the network to learn long-term temporal patterns. On the other hand,
- **explode** if $w \gg 1$. In such a case, the gradient grows bigger as t increases, either leading to numerical instabilities or too big gradient steps, ultimately destabilizing training.

The same problems can arise when there are multiple neurons. Due to the non-linearity of the activation functions, it remains difficult however to find exact conditions on the weights matrices for the gradient to vanish or explode. An in-depth study on the subject has been made by Pascanu et al. (2013), who have thoroughly analyzed these issues using control theory. In the same paper, the authors summarize multiple solutions that have been introduced over the years in an attempt to solve some of those issues. The next section will now introduce some of the most commonly used, and in particular, the concept of gated recurrent cells.

4.2 Towards easily training RNNs

On the one hand, over the past years, a few methods have been proposed to solve exploding gradients issues. These methods however remain quite similar and straightforward, generally relying on clipping too big gradients (Pascanu et al., 2013; Mikolov et al., 2012) to a constant value. Despite the fact that this operation can change the direction of the gradient, it still proved to work well in practice. It is thus common practice to clip gradients when training RNNs, and as such, when needed, we use a similar approach for training our RNNs in next Chapters. Pascanu et al. (2013) go into more details on the subject and further discuss such methods.

On the other hand, there has also been an extensive amount of work to solve vanishing gradient problems. As such, different methods have been proposed, including introducing gating mechanisms (Cho et al., 2014; Hochreiter & Schmidhuber, 1997), maintaining orthogonality in recurrent weight matrices (Jing et al., 2019) and using new ways to initialise RNN parameters (Pascanu et al., 2013; Van Der Westhuizen & Lasenby, 2018; Marichal et al., 2009). The remaining of the section will focus on such methods. First, gating mechanisms will be introduced. Then, a special case of such mechanism will be used to highlight some ways of initialising RNNs such as to endow them with better training properties.

Long-short term memory Gating mechanisms were first introduced in the long-short term memory (LSTM) cell, proposed by Hochreiter & Schmidhuber (1997) as a way to solve vanishing gradients. The idea behind gating mechanisms is to build units for which the input, memory and output flows are controlled by different weight matrices, through sigmoid functions, denoted σ and called "gates". To understand why this denomination is used, one should study the update rules of LSTM cells, which are as follows:

$$\begin{cases} \mathbf{f}_t = \sigma(U_f \mathbf{x}_t + W_f \mathbf{h}_{t-1} + \mathbf{b}_f) \quad , \\ \mathbf{i}_t = \sigma(U_i \mathbf{x}_t + W_i \mathbf{h}_{t-1} + \mathbf{b}_i) \quad , \\ \mathbf{k}_t = \sigma(U_k \mathbf{x}_t + W_k \mathbf{h}_{t-1} + \mathbf{b}_k) \quad , \\ \mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(U_c \mathbf{x}_t + W_c \mathbf{h}_{t-1} + \mathbf{b}_c) \quad , \\ \mathbf{h}_t = \tanh(\mathbf{c}_t) \odot \mathbf{k}_t \quad , \end{cases} \tag{4.10}$$

where $\odot$ denotes the Hadamard product. From Equation 4.10, one can define three distinct gates.

1. The forget gate $\mathbf{f}$ which controls the internal state flow. Indeed, note that the image of sigmoid functions belongs to $]0,1[$. Therefore, the forget gate is there to drive what part of the previous cellular state $\mathbf{c}_{t-1}$ (an internal state that is specific to LSTMs) should be forgotten. The closer to 1, the more the previous cellular state will be kept into memory, whereas the closer to 0, the more it will be forgotten.
2. The input gate $\mathbf{i}$ that controls the input flow. This gate is basically used to choose what part of the new cellular state candidate (computed thanks to the previous state $\mathbf{h}_{t-1}$ and newly received observation $\mathbf{x}_t$) should be taken into account in the new cellular state.
3. Finally, the output gate $\mathbf{k}$ that drives the output flow of the cell state, similarly to both previous gates.

Since the introduction of LSTM cells, many other gated architectures have been introduced (Zhou et al., 2016; Jing et al., 2019; Dey & Salemt, 2017). Going one step further, an empirical search over hundreds of different gated architectures has even recently been carried by Jozefowicz et al. (2015), showing that different architectures could be better or worse suited to different tasks. This study also highlights the importance of well initialising parameters of LSTM cells, something that has also been discussed by Van Der Westhuizen & Lasenby (2018). Overall, LSTM still remain as one of the most well established recurrent cells and are heavily used in the field of deep learning. Another widely used recurrent cell that comes close to LSTMs in terms of widespread usage is the gated recurrent unit (GRUs), proposed by Cho et al. (2014).

Gated recurrent units Similarly to LSTMs, GRUs use gating mechanisms to control flow of information during the state updates. They however use less parameters and gates than LSTMs to do so. The update rules for GRUs are as follows:

$$\begin{cases} \mathbf{z}_t = \sigma(U_z\mathbf{x}_t + W_z\mathbf{h}_{t-1}) \quad , \\ \mathbf{r}_t = \sigma(U_r\mathbf{x}_t + W_r\mathbf{h}_{t-1}) \quad , \\ \mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1-\mathbf{z}_t) \odot \tanh(U_h\mathbf{x}_t + \mathbf{r}_t \odot W_h\mathbf{h}_{t-1}) \quad , \end{cases} \tag{4.11}$$

where $\mathbf{z}$ is called the update gate and $\mathbf{r}$ is called the regret gate.

- The role of the update gate is even clearer than that of LSTMs' gates. Indeed, from Equation 4.11, one can see that $\mathbf{z}$ drives the speed at which the current internal state is replaced by newly computed values (usually called candidate). The smaller $\mathbf{z}$ is, the quicker the internal state will be updated and past states will be forgotten.

- The role of $\mathbf{r}$ on the other hand is less immediate in that this gate controls the usage of the previous internal state when computing the new candidate.

Despite LSTMs and GRUs partly solving gradient issues, it can still remain difficult to train such cells on some tasks where very long temporal patterns exist and thus research in the field is still ongoing. As GRUs have exhibited good performance on multiple tasks, they also have inspired multiple works to build upon them.

Among others, Jing et al. (2019) proposed gated orthogonal recurrent units (GORUs), another type of gated recurrent cell heavily based on GRUs but making use of orthogonal matrices. This cell is detailed hereunder and serves to introduce orthogonal initialisation, another method that has recently been showed to enhance learning of RNNs.

GORUs The idea behind GORUs is to use a similar update to that of standard GRUs. In fact, the update of GORUs is given by Equation 4.11 where only the tanh function is replaced with a sort of modified ReLU (introduced by Jing et al. (2019)). Thus the main difference with GRUs is neither in the update, nor in the architecture. Rather, the core concept behind GORUs is to parameterize W_h such that it is (and remains throughout training) orthogonal.

This idea comes from an initialisation procedure which was shown to improve GRUs (among other types of gated cells) performance. Indeed, it is thought that initialising the recurrent weight matrix such that it is orthogonal provides better gradient (and feed-forward) properties. This is often discussed intuitively in the case of a standard RNN cell, with no input, no bias, $\mathbf{h}_0$ set to the identity vector and identity as activation function. In such a case, the following development can be made:

$$\begin{aligned} \mathbf{h_t} &= W\mathbf{h}_{t-1} && (4.12) \\ &= \underbrace{W(W(\dots(W(\mathbf{h}_0))))}_{t} && (4.13) \\ &= W^t I && (4.14) \\ &= W^t \quad . && (4.15) \end{aligned}$$

In this particular case, it appears that the value of h_t will explode if the absolute value of some eigenvalues of W are greater than 1 and vanish if these absolute values are smaller than 1. Since orthogonal matrices have eigenvalues strictly equal to one or minus one, it appears that they permit to avoid this problem. Of course many assumptions are made here, and this development is done for intuition only. Henaff et al. (2016) talk more in-depth on this particular subject. For completeness, note that GORUs were only taken as an example to introduce the benefits of orthogonal matrices in RNNs since they followed well GRUs' description. There has however been other works on RNNs focusing on using orthogonal matrices, such as that of Arjovsky et al. (2016) among others.

To conclude, other initialisation methods for RNNs have proven to work well (Tallec & Ollivier, 2018; Van Der Westhuizen & Lasenby, 2018; Marichal et al., 2009; Jozefowicz et al., 2015), and it is believed that having good properties at initialisation is highly important and allows for faster and easier training of such models. In this regard, Chapter 5 and 6 will introduce new ways of initialising RNNs near a good dynamical regime. Chapter 5 does so by introducing a new cell which inherently possess good dynamical properties for long-term memory, while Chapter 6 introduces a novel procedure before training to reach such properties with usual cells. Doing so, these Chapters will also highlight how dynamics of untrained RNNs, i.e. at initialisation, can strongly impact RNNs learning performance during training on specific benchmarks.

4.3 Dynamics of trained RNNs

** Parts of this Section have been adapted from Vecoven et al. (2021b).*

Despite the tremendous amount of work focusing on developing new types of cells and analysing their performance (Chung et al., 2014; Jozefowicz et al., 2015), they have predominantly remained seen as black-box models. Recently, there has however been a growing body of works focused on understanding the internal dynamics of trained RNNs (Sussillo & Barak, 2013; Ceni et al., 2020; Maheswaranathan et al., 2019), providing invaluable insights into their prediction process.

In particular, RNNs can be seen as dynamical systems and non-linear control tools can be used to understand the behaviour of such networks. This viewpoint has been used early to understand the difficulties of RNNs to capture long term dependencies (Doya, 1993; Bengio et al., 1993; Pascanu et al., 2013). More recently, this specific approach was taken by Sussillo & Barak (2013) as a way to analyze dynamics of trained RNNs, considering them as blackbox systems. This work led to very interesting findings and mainly, the authors highlight the importance of fixed points in the prediction process of RNNs.

Fixed points are well-known characteristics of dynamical systems that are defined as points in the phase space that map to themselves through the update function. Fixed points depend on the input of the system. In the case of RNNs, we say that a state $\mathbf{h}^*$ is a fixed point of a network in $\mathbf{x}$ if and only if:

$$\mathbf{h}^* = f(\mathbf{h}^*, \mathbf{x}; \theta)$$

Fixed points can either be fully attractive (attractors), fully repulsive (repellors), or combine attractive and repulsive manifolds (saddle points). Attractors and saddle points are the most useful when it comes to understanding RNNs dynamics. Attractors correspond to network steady-sates and are thought to be the allowing factor for RNNs to maintain long-term information (Pascanu et al., 2013; Sussillo & Barak, 2013; Maheswaranathan et al., 2019). They are defined as fixed points towards which the system converges for multiple starting conditions. The set of starting states for which the system converges to the attractor $\mathbf{h}^*$ is called basin of attraction of $\mathbf{h}^*$ and written as $\mathcal{B}_{\mathbf{h}^*}$. Basins of attraction are delimited by the stable manifolds of saddle-points. Mathematically, $\mathbf{h}^*$ is an attractor in $\mathbf{x}$ if its basin of attraction in $\mathbf{x}$, $\mathcal{B}^{\mathbf{x}}_{\mathbf{h}^*}$, is not a singleton and is such that:

$$\mathbf{h} \in \mathcal{B}^{\mathbf{x}}_{\mathbf{h}^*} \iff \lim_{n \to \infty} f^n(\mathbf{h}, \mathbf{x}; \theta) = \mathbf{h}^* \quad .$$

From this definition, one can introduce the notion of reachable attractors. In particular, we say that an attractor $\mathbf{h}^*$ is reachable in $\mathbf{x}$ if there exists an input to the system such that its final state belongs to $\mathcal{B}^{\mathbf{x}}_{\mathbf{h}^*}$. More formally, an attractor $\mathbf{h}^*$ is said to be reachable in $\mathbf{x}$ if there exists $\mathbf{X}^* = [\mathbf{x}^*{}_1, \ldots, \mathbf{x}^*{}_M]$ such that $\mathbf{h}_M \in \mathcal{B}^{\mathbf{x}}_{\mathbf{h}^*}$, where $\mathbf{h}_{i+1} = f(\mathbf{h}_i, \mathbf{x}^*{}_i)$ with $\mathbf{h}_0 = \mathbf{c}$, $M \in \mathcal{N}$ and $\mathbf{c}$ denoting a constant initial state. A system that has a single reachable attractor in $\mathbf{x}$ is said to be monostable in $\mathbf{x}$, whereas a system that has multiple reachable attractors in $\mathbf{x}$ is said to be multistable in $\mathbf{x}$.

In their work, Sussillo & Barak (2013) use linearization in order to find all types of fixed points (i.e. attractors, repellors and saddle points) and to study them. They show that attractors are heavily used by RNNs to store information about the prediction and that saddle points were often created by the network to move between their different basins of attractions. Intuitively, this result makes sense as, by definition, attractors are stable states, and thus, allow the network to store information for arbitrarily long period of times. As such, it is believed that these points are highly important for RNNs to learn when sequences become longer (Jozefowicz et al., 2015).

On the other hand, it is known and important to note that fixed points alone are not sufficient when it comes to learning complex temporal patterns. Indeed, one can easily think of the case of regression tasks. If a network only uses stable points as a way to store information, it would only be able to predict N different values with $N \in \mathcal{N}$ being the number of different reachable attractors of the network. Due to the continuous nature of regression, this would quickly limit the predictive accuracy of such a model. It is thus important to note that transient dynamics of RNNs are also very important, and of course necessary, when it comes to their predictions. Indeed, the transient regime allows for the continuous, albeit shorter-term, encoding of information.

Despite not being complete enough to fully understand RNNs dynamics, fixed point analysis is thus a great tool to understand RNNs. However, due to the complexity of these models, it is difficult and often computationally expensive to find and study these points. As such, multiple methods have recently been developed for finding fixed points in trained RNNs (Katz & Reggia, 2017; Pascanu et al., 2013; Ceni et al., 2020). Despite this increasing amount of work on trained RNNs and due to these constraints, there has however been only very little work focusing on the evolution of fixed points in RNNs during training. The goal of Chapter 6 is precisely to focus on this evolution, looking in particular at the number of different reachable attractors that the network possesses. Focusing on such particular fixed points, as opposed to finding all of them, allows for the definition of fast and easy to compute metrics, in turn allowing to easily check their evolution during training. In particular, Chapter 6 highlights a high correlation between the number of reachable attractors that a network possesses and its predictive accuracy.

Summary

The goal of this Chapter was to make the reader familiar with recurrent architectures and mainly, some of their common problems. The reader should also have a good understanding of some common state-of-the-art techniques designed towards solving these issues. In particular, the reader is expected to understand the two most commonly used gated architectures, i.e. GRUs and LSTMs, as well as some newly introduced initialisation (such as orthogonal initialisation) and gradients clipping methods (helping towards alleviating exploding gradients, which sometimes arise when training RNNs on long sequences).

Furthermore, this Chapter also aimed at discussing the dynamics of successfully trained networks and at providing tools to do so. In particular, the importance of fixed points in the prediction process of RNNs has been highlighted, and more precisely, that of stable points. These discussions aimed at easing the reader into the next Chapter in which a new cell, specifically designed towards being multistable and easily analysable through control theory, is presented.

Chapter 5

Bio-inspired bistable recurrent cells

This Chapter contains the core of the following publication (Vecoven et al., 2021a).

As mentioned in previous chapter, RNNs provide state-of-the-art performances in a wide variety of tasks that require memory, which are often achieved thanks to gated recurrent cells. Standard gated cells share a layer internal state to store information at the network level, and long term memory is shaped by network-wide recurrent connection weights. Biological neurons on the other hand are capable of holding information at the cellular level for an arbitrary long amount of time through a process called bistability. Through bistability, cells can stabilize to different stable states depending on their own past state and inputs, which permits the durable storing of past information in neuron state. In this Chapter, inspiration from biological neuron bistability is taken to embed RNNs with long-lasting memory at the cellular level. This leads to the introduction of a new bistable biologically-inspired recurrent cell that is shown to strongly improves RNN performance on time-series which require very long memory, despite using only cellular connections (all recurrent connections are from neurons to themselves, i.e. a neuron state is not influenced by the state of other neurons). The fact that only cellular connections are used also allows for an easy interpretation of the dynamics of such cells from a control perspective. Furthermore, equipping this cell with recurrent neuromodulation permits to link them to standard GRU cells, taking a step towards the biological plausibility of GRU. Section 5.1 introduces the motivation behind building bistable recurrent cells from a biological perspective. Section 5.2 then goes into deeper details on a high-level model of human neurons, from which we derive the bistable recurrent cell in Section 5.3. Finally, results and experiments are discussed in Section 5.4.

5.1 Introduction

Recently, there has been an increased interest in assessing the biological plausibility of neural networks. There has not only been a lot of interest in spiking neural networks (Tavanaei et al., 2019; Pfeiffer & Pfeil, 2018; Bellec et al., 2018), but also in reconciling more traditional deep learning models with biological plausibility (Bengio et al., 2015; Miconi, 2017; Bellec et al., 2019). RNNs are a promising avenue for the latter (Barak, 2017) as they are known to provide great performances from a deep learning point of view while theoretically allowing a discrete dynamical simulation of biological neurons.

RNNs combine simple cellular dynamics and a rich, highly recurrent network architecture. The recurrent network architecture enables the encoding of complex memory patterns in the connection weights. These memory patterns rely on global feedback interconnections of large neuronal populations. It was discussed in previous chapter that such global feedback interconnections are difficult to tune, and can be a source of vanishing or exploding gradient during training, which is a major drawback of RNNs. In biological networks, a significant part of advanced computing is handled at the cellular level, mitigating the burden at the network level. Each neuron type can switch between several complex firing patterns, which include e.g. spiking, bursting, and bistability. In particular, bistability is the ability for a neuron to switch between two stable outputs depending on input history. It is a form of cellular memory as discussed by Marder et al. (1996).

In the next Sections, a new biologically motivated bistable recurrent cell (BRC) is proposed and embeds classical RNNs with local cellular memory rather than global network memory. More precisely, BRCs are built such that their hidden recurrent state does not directly influence other neurons (i.e. they are not recurrently connected to other cells). To make cellular bistability compatible with the RNNs feedback architecture, a BRC is constructed by taking a feedback control perspective on biological neuron excitability (Drion, O'Leary, et al., 2015). This approach enables the design of biologically-inspired cellular dynamics by exploiting the RNNs structure rather than through the addition of complex mathematical functions.

5.2 Neuronal bistability: a feedback viewpoint

Biological neurons are intrinsically dynamical systems that can exhibit a wide variety of firing patterns. In this Chapter, particular focus is made on the control of bistability, which corresponds to the coexistence of two stable states at the neuronal level. Bistable neurons can switch between their two stable states in response to transient inputs (Marder et al., 1996; Drion, O'Leary, & Marder, 2015), endowing them with a kind of never-fading cellular memory (Marder et al., 1996).

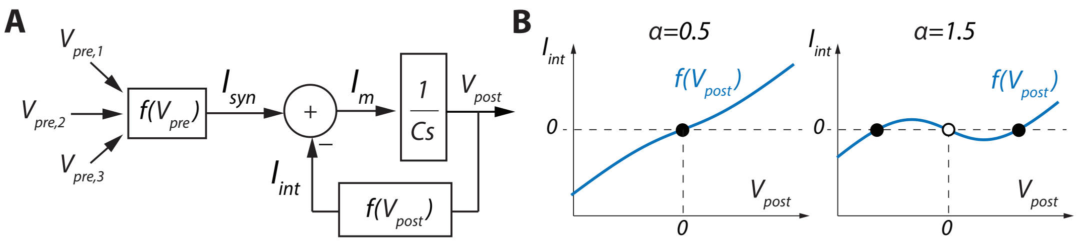


Figure 5.1: A. One timescale control diagram of a neuron. **B.** Plot of the function $I_{int} = V_{post} - \alpha \tanh(V_{post})$ for two different values of α. Full dots correspond to stable states, empty dots to unstable states.

Complex neuron firing patterns are often modeled by systems of ordinary differential equations (ODEs). Translating ODEs into an artificial neural network algorithm often leads to mixed results due to increased complexity and the difference in modeling language. Another approach to model neuronal dynamics is to use a control systems viewpoint (Drion, O'Leary, et al., 2015). In this viewpoint, a neuron is modeled as a set of simple building blocks connected using a multi-scale feedback, or recurrent, interconnection pattern.

A neuronal feedback diagram focusing on one time-scale, which is sufficient for bistability, is illustrated in Fig 5.1A. The block $1/(Cs)$ accounts for membrane integration, C being the membrane capacitance and s the complex frequency. The outputs from presynaptic neurons V_{pre} are combined at the input level to create a synaptic current I_{syn}. Neuron-intrinsic dynamics are modeled by the negative feedback interconnection of a nonlinear function $I_{int} = f(V_{post})$, called the IV curve in neurophysiology, which outputs an intrinsic current I_{int} that adds to I_{syn} to create the membrane current I_m. The slope of $f(V_{post})$ determines the feedback gain, a positive slope leading to negative feedback and a negative slope to positive feedback. I_m is then integrated by the postsynaptic neuron membrane to modify its output voltage V_{post}.

The switch between monostability and bistability is achieved by shaping the nonlinear function $I_{int} = f(V_{post})$ (Fig 5.1B). The neuron is monostable when $f(V_{post})$ is monotonic of positive slope (Fig 5.1B, left). Its only stable state corresponds to the voltage at which $I_{int} = 0$ in the absence of synaptic inputs (full dot). The neuron switch to bistability through the creation of a local region of negative slope in $f(V_{post})$ (Fig 5.1B, left). Its two stable states correspond to the voltages at which $I_{int} = 0$ with positive slope (full dots), separated by an unstable state where $I_{int} = 0$ with negative slope (empty dot). The local region of negative slope corresponds to a local positive feedback where the membrane voltage is unstable.

In biological neurons, a local positive feedback is provided by regenerative gating, such as sodium and calcium channel activation or potassium channel inactivation (Drion, O'Leary, & Marder, 2015; Franci et al., 2013a). The switch from monostability to bistability can therefore be controlled by tuning ion channel density. This property can be emulated in electrical circuits by combining transconductance amplifiers to create the function

$$I_{int} = V_{post} - \alpha \tanh(V_{post}), \tag{5.1}$$

where the switch from monostability to bistability is controlled by a single parameter α (Ribar & Sepulchre, 2019). α models the effect of sodium or calcium channel activation, which tunes the local slope of the function, hence the local gain of the feedback loop (Fig 5.1B). For $\alpha \in]0, 1]$ (where $]0, 1]$ denotes a continuous interval), which models a low sodium or calcium channel density, the function is monotonic, leading to monostability (Fig 5.1B, left). For $\alpha \in]1, +\infty[$, which models a high sodium or calcium channel density, a region of negative slope is created around $V_{post} = 0$, and the neuron becomes bistable (Fig 5.1B, right). This bistability leads to never-fading memory, as in the absence of significant input perturbation the system will remain indefinitely in one of the two stable states depending on the input history.

Neuronal bistability can therefore be modeled by a simple feedback system whose dynamics is tuned by a single feedback parameter α. This parameter can switch between monostability and bistability by tuning the shape of the feedback function $f(V_{post})$, whereas neuron convergence dynamics is controlled by a single feedforward parameter C. In biological neurons, both these parameters can be modified dynamically by other neurons via a mechanism called neuromodulation, providing a dynamic, controllable memory at the cellular level. The key challenge is to find an appropriate mathematical representation of this mechanism to be efficiently used in artificial neural networks, and, more particularly, in RNNs.

5.3 Cellular memory, bistability and neuromodulation in RNNs

The bistable recurrent cell (BRC) To model controllable bistability in RNNs, two main comparisons are made between the feedback structure Fig 5.1A and the GRU equations (Equation 4.11). First, it is noted that the reset gate r has a role that is similar to the one played by the feedback gain α in Equation 5.1. In GRU equations, r is the output of a sigmoid function, which implies $r \in]0, 1[$. These possible values for r correspond to negative feedback only, which does not allow for bistability for initial values of W_h (be it with an orthogonal or a more usual initialisation). The update gate z, on the other hand, has a role similar to that of the membrane capacitance C. Second, one can see through the matrix multiplications $W_z\mathbf{h}_{t-1}$, $W_r\mathbf{h}_{t-1}$ and $W_h\mathbf{h}_{t-1}$ that each cell uses the internal state of other neurons to compute its own state without going through synaptic connections. In biological neurons, the intrinsic dynamics defined by I_{int} is constrained to only depend on its own state V_{post}, and the influence of other neurons comes only through the synaptic compartment (I_{syn}), or through neuromodulation.

To enforce this cellular feedback constraint in GRU equations and to endow them with bistability, it is proposed to update h_t as follows:

$$\mathbf{h}_t = \mathbf{c}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{c}_t) \odot \tanh(U\mathbf{x}_t + \mathbf{a}_t \odot \mathbf{h}_{t-1}) \tag{5.2}$$

where $\mathbf{a}_t = 1 + \tanh(U_a\mathbf{x}_t + \mathbf{w}_a \odot \mathbf{h}_{t-1})$ and $\mathbf{c}_t = \sigma(U_c\mathbf{x}_t + \mathbf{w}_c \odot \mathbf{h}_{t-1})$. $\mathbf{a}_t$ corresponds to the feedback parameter α, with $\mathbf{a}_t \in]0, 2[$ (as $\tanh(\cdot) \in] - 1, 1[$). $\mathbf{c}_t$ corresponds to the update gate in GRU and plays the role of the membrane capacitance C, determining the convergence dynamics of the neuron. This updated cell is called the bistable recurrent cell (BRC).

The main differences between a BRC and a GRU are twofold. First, each neuron has its own internal state $\mathbf{h}_t$ that is not directly affected by the internal state of the other neurons. Indeed, due to the four instances of $\mathbf{h}_{t-1}$ coming from Hadamard products, the only temporal connections existing in layers of BRC are from neurons to themselves. In other words, layers are no longer fully connected through time, which enforces the memory to be only cellular. Second, the feedback parameter $\mathbf{a}_t$ is allowed to take a value in the range $]0, 2[$ rather than $]0, 1[$. This allows the cell to switch between monostability ($a \leq 1$) and bistability ($a > 1$) (Fig 5.2A,B). The proof of this switch is provided in Appendix B.1.

It is important to note that the parameters $\mathbf{a}_t$ and $\mathbf{c}_t$ are dynamic. Tests were carried with $\mathbf{a}$ and $\mathbf{c}$ as parameters learned by stochastic gradient descent, which resulted in lack of representational power. This dynamic scheme was the most evident as it maintains the cellular memory constraint and leads to the most similar update rule with respect to standard recurrent cells (Equation 4.11). However, as will be discussed later, other updates can be thought of.

Likewise, from a neuroscience perspective, $\mathbf{a}_t$ could well be greater than 2. Limiting the range of $\mathbf{a}_t$ to $]0, 2[$ was made for numerical stability and for symmetry between the range of bistable and monostable neurons. It is argued that this is not an issue as, for a value of $\mathbf{a}_t$ greater than 1.5, the dynamics of the neurons become very similar (as suggested in Fig 5.2A).

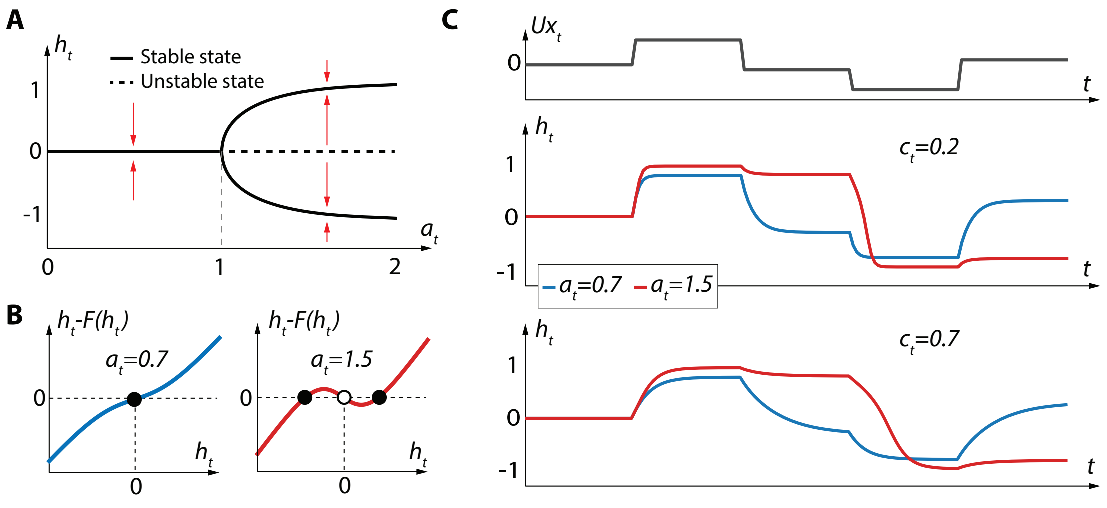


Figure 5.2: A. Bifurcation diagram of Equation 5.2 for $U\mathbf{x}_t = 0$. **B.** Plots of the function $h_t - F(h_t)$ for two values of a_t, where $F(h_t) = c_t h_t + (1 - c_t)\tanh(a_t h_t)$ is the right hand side of Equation 5.2 with $x_t = 0$. Full dots correspond to stable states, empty dots to unstable states. **C.** Response of BRC to an input time-series for different values of a_t and c_t.

Fig 5.2C shows the dynamics of a BRC with respect to a_t and c_t. For $a_t < 1$, the cell exhibits a classical monostable behavior, relaxing to the 0 stable state in the absence of inputs (blue curves in Fig 5.2C). On the other hand, a bistable behavior can be observed for $a_t > 1$: the cells can either stabilize on an upper stable state or a lower stable state depending on past inputs (red curves in Fig 5.2C). Since these upper and lower stable states do not correspond to an h_t which is equal to 0, they can be associated with cellular memory that never fades over time. Furthermore, Fig 5.2 also illustrates that neuron convergence dynamics depend on the value of c.

The recurrently neuromodulated bistable recurrent cell (nBRC) To further improve the performance of BRC, one can relax the cellular memory constraint. By creating a dependency of a_t and c_t on the output of other neurons of the layer, one can build a kind of recurrent layer-wise neuromodulation. This modified version of a BRC is referred to as an nBRC, standing for recurrently neuromodulated BRC. The update rule for the nBRC is the same as for BRC, and follows Equation 5.2. The difference comes in the computation of a_t and c_t, which are neuromodulated as follows:

$$\begin{cases} \mathbf{a}_t = 1 + \tanh(U_a\mathbf{x}_t + W_a\mathbf{h}_{t-1}) \\ \mathbf{c}_t = \sigma(U_c\mathbf{x}_t + W_c\mathbf{h}_{t-1}) \end{cases} \tag{5.3}$$

The update rule of nBRCs being that of BRCs (Equation 5.2), bistable properties are maintained and hence the possibility of a cellular memory that does not fade over time. However, the new recurrent neuromodulation scheme adds a type of network memory on top of the cellular memory.

This recurrent neuromodulation scheme brings the update rule even closer to standard GRU. This is highlighted when comparing Equation 4.11 and Equation 5.2 with parameters neuromodulated following Equation 5.3. A relaxed cellular memory constraint is still ensured, as each neuron past state h_{t-1} only directly influences its own current state and not the state of other neurons of the layer (Hadamard product on the $\mathbf{h}_t$ update in Equation 5.2). This is important for numerical stability as the introduction of a cellular positive feedback for bistability leads to global instability if the update is computed using other neurons states directly (as it is done in the classical GRU update, see the matrix multiplication $W_h\mathbf{h}_{t-1}$ in Equation 4.11).

Finally, it is noted that to be consistent with the biological model presented in Section 5.2, Equation 5.3 should be interpreted as a way to represent a neuromodulation mechanism of a neuron by those from its own layer and the layer that precedes. Hence, the possible analogy between gates z and r in GRUs and neuromodulation, an important aspect of biological brains (discussed in more details in Part III of this thesis). In this respect, studying the introduction of new types of gates based on more biological plausible neuromodulation architectures would certainly be interesting.

5.4 Analysis of BRC and nBRC performance

To demonstrate the impact of bistability in RNNs, four problems are tackled. The first is a one-dimensional toy problem, the second is a two-dimensional denoising problem, the third is the permuted sequential MNIST problem and the fourth is a variation of the third benchmark. All benchmarks are related to a supervised setting. The network is presented with a time-series and is asked to output a prediction (regression for the first two benchmarks and classification for the others) after having received the last element(s) of the time-series $\mathbf{x}_T$. Note that for the second benchmark the regression is carried over multiple time-steps (sequence-to-sequence) whereas, this prediction is given in a single time-step after receiving $\mathbf{x}_T$ for the other benchmarks. It is first shown that the introduction of bistability in recurrent cells is especially useful for datasets in which only time-series with long time-dependencies are available. This is achieved by comparing results of BRC and nBRC to other recurrent cells. LSTMs and GRUs are used as a baseline since they have already been established. As a state-of-the-art comparison, two other cells (GORUs and Legendre memory units (LMUs, proposed by Voelker et al. (2019)) are used. Finally, the dynamics inside the nBRC neurons are also discussed in the context of the denoising benchmark and they show that bistability is, as expected, heavily used by the neural network.

5.4.1 Results

For the first two problems, training sets comprise 40000 samples and performances are evaluated on test sets generated with 50000 samples. For the permuted MNIST benchmarks, the standard train and test sets are used. All averages and standard deviations reported were computed over three different seeds. It was found that there were only minor variations in between runs, and thus that three runs are believed to be sufficient to capture the performance of the different architectures. For all benchmarks, networks are composed of two layers of 128 neurons. Different recurrent cells are always tested on similar networks (i.e. same number of layers/neurons). The tensorflow (Abadi et al., 2015) implementation of GRUs and LSTMs were used. Finally, the ADAM optimizer with a learning rate of $1e^{-3}$ is used for training all networks, with a mini-batch size of 100. The source code for carrying out the experiments is available at `https://github.com/nvecoven/BRC`. All networks are trained for 50 epochs (which has proven to be enough to reach convergence on these particular benchmarks).

Copy first input benchmark In this benchmark, the network is presented with a one-dimensional time-series of T time-steps where $x_t \sim N(0,1),\ \forall t \in T$. After receiving x_T, the network output value should approximate x_0, a task that is well suited for capturing their capacity to learn long temporal dependencies if T is large. Note that this benchmark also requires the ability to filter irrelevant signals as, after time-step 0, the networks are continuously presented with noisy inputs that they must learn to ignore. The mean square error on the test set is shown for different values of T in Table 6.1. Despite their ability to learn longer patterns than standard RNN cells, one can see the limitation of LSTMs and GRUs when T becomes large (this is shown in particular for GRUs on Fig 5.3), as they are unable to beat random guessing performances (which would be equal to 1 in this setting [1]).

Furthermore, one can see that the gated orthogonal version of GRUs (GORUs) achieves better performances than GRU cells, as expected. It also appears that thanks to bistability, nBRCs and BRCs are able to learn effectively and achieve similar performances to those of LMUs.

1. Indeed, as x_0 is sampled from a normal distribution $N(0,1)$, guessing 0 would lead to the lowest error which would on average be equal to the standard deviation

T	BRC	NBRC	GORU	LSTMCell	GRUCell	LMU
5	0.005 ± 0.00	0.000 ± 0.00	0.000 ± 0.00	0.000 ± 0.00	0.000 ± 0.00	0.000 ± 0.00
50	0.082 ± 0.03	0.002 ± 0.00	0.019 ± 0.01	0.000 ± 0.00	0.997 ± 0.01	0.000 ± 0.00
300	0.086 ± 0.01	0.010 ± 0.00	0.308 ± 0.05	1.002 ± 0.01	0.876 ± 0.19	0.000 ± 0.00
600	0.099 ± 0.03	0.009 ± 0.00	0.323 ± 0.07	0.989 ± 0.01	0.999 ± 0.02	0.002 ± 0.00

Table 5.1: Mean square error ($\pm$ standard deviation) of different architectures on the test set for the copy input benchmark. Results are shown after 50 epochs and for different values of T.

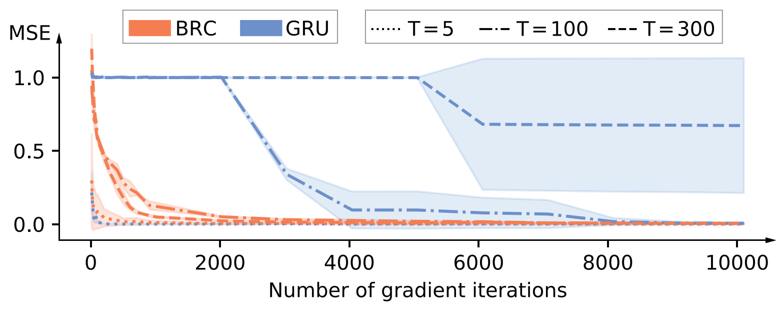


Figure 5.3: Evolution of the average mean-square error ($\pm$ standard deviation) over three runs on the copy input benchmark for GRU and BRC and for different values of T.

It is very important to note that theoretically, GRUs and LSTMs have the representational power to solve this benchmark and to possess attractors as well. They are however unable to learn how to solve this benchmark and remain mono-stable throughout training. This hints at the importance of the dynamics of RNNs at initialisation and while training and will be the subject of Chapter 6. To highlight this, those cells were tested on a modified version of the copy input benchmark, in which T is chosen uniformly between 1 and 600 for each sample. As such, for this modified version of the benchmark, some samples only require very little memory, while others require long-term memory. Results are shown in Table 5.2 and highlight that GRUs are indeed able to solve the benchmark (and thus, learn long-term dependencies) when guided by easier samples. In next Chapter, a method for allowing these cells to learn such benchmark (without requiring "short-term memory" samples) is introduced.

BRC	nBRC	GRU	LSTM
0.0010 ± 0.0001	0.0001 ± 0.0001	0.0373 ± 0.00371	0.3323 ± 0.4635

Table 5.2: Mean square error on test set after 50 epochs for different architectures on the modified copy input benchmark.

N	BRC	NBRC	GORU	LSTM	GRU	LMU
5	0.579 ± 0.03	0.016 ± 0.00	0.000 ± 0.00	0.655 ± 0.46	0.001 ± 0.00	1.004 ± 0.01
200	0.614 ± 0.12	0.071 ± 0.08	1.004 ± 0.00	0.996 ± 0.01	0.995 ± 0.00	1.000 ± 0.00

Table 5.3: Mean square error ($\pm$ standard deviation) of different architectures on the denoising benchmark's test set. Results are shown with and without constraint on the location of relevant inputs and after 50 epochs. Relevant inputs cannot appear in the N last time-steps, that is $\mathbf{x}_t[1] = -1, \forall t > (T - N)$. In this experiment, results were obtained with $T = 400$.

Denoising benchmark The copy input benchmark is interesting as a means to highlight the memorisation capacity of the recurrent neural network, but it does not tackle its ability to successfully exploit complex relationships between different elements of the input signal to predict the output. In the denoising benchmark, the network is presented with a two-dimensional time-series of T time-steps. Five different time-steps $t_1, \dots, t_5$, for which data should be remembered, are sampled uniformly in $\{0, \dots, T - N\}$ with $N \in \{5, \dots, T - 4\}$ and are communicated to the network through the first dimension of the time-series by setting $\mathbf{x}_t[1] = 1$ if $t \in \{t_1, \dots, t_5\}$, $\mathbf{x}_t[1] = 0$ if $t = T - 4$ and $\mathbf{x}_t[1] = -1$ otherwise. Note that the parameter N controls the length of the forgetting period as it forces the relevant inputs to be in the first $T - N$ time-steps. This ensures that $t_x < T - N,\ \forall x \in \{1, \dots, 5\}$. Also note that this dimension can be used by the network to know when its predictions influence the loss (whenever $\mathbf{x}_t[1] = 0$ as been seen).

The second dimension is a data-stream, generated as for the copy first input benchmark, that is $\mathbf{x}_t[2] \sim \mathcal{N}(0, 1), \forall t \in \{0, \dots, T - 5\}$ and $\mathbf{x}_t[2] = 0, \forall t \in \{T - 4, \dots, T\}$. At time-step $T - 4$, the network is asked to output $\mathbf{x}_{t_1}[2]$, at time-step $T - 3$ the network is asked to output $\mathbf{x}_{t_2}[2]$ and so on until time-step T at which it should output $\mathbf{x}_{t_5}[2]$. The mean squared error is averaged over the five values. That is, the error on the prediction is equal to $\sum_{i=1}^{5} \frac{(\mathbf{x}_{t_i}[2] - \mathbf{o_{T-5+i}})^2}{5}$ with $\mathbf{o_x}$ the output of the neural network at time-step x.

As one can see in Table 6.2 (generated with $T = 400$ and two different values of N), for $N = 200$ only bistable cells are able to learn how to achieve good performances. It is noted that for this benchmark, LMUs were not able to learn due to heavy overfitting.

BRC	NBRC	GORU	LSTMCell	GRUCell	LMU
0.662 ± 0.007	0.908 ± 0.006	0.902 ± 0.004	0.910 ± 0.002	0.908 ± 0.004	0.969 ± 0.001

Table 5.4: Overall accuracy (± standard deviation) on the permuted sequential MNIST benchmark's test set after 50 epochs and for different cell types.

N	BRC	NBRC	GORU	LSTMCell	GRUCell	LMU
72	0.968 ± 0.00	0.973 ± 0.00	0.977 ± 0.00	0.977 ± 0.00	0.977 ± 0.00	0.969 ± 0.00
472	0.960 ± 0.00	0.972 ± 0.00	0.198 ± 0.02	0.562 ± 0.32	0.591 ± 0.39	0.961 ± 0.00

Table 5.5: Overall accuracy (± standard deviation) on permuted sequential-line MNIST test set after 50 epochs for different architectures. Images are fed to the recurrent network line by line and N black lines are added at the bottom of the image after permutation. When N equals 72(472) the resulting image has 100(500) lines.

Permuted sequential MNIST In this benchmark, the network is presented with the MNIST images (LeCun & Cortes, 2010), where pixels are shown, one by one, as a time-series. It differs from the regular sequential MNIST in that pixels are shuffled, with the result that they are not shown in top-left to bottom-right order. This benchmark is known to be a more complex challenge than the regular one. Indeed, shuffling makes time-dependencies more complex by introducing lag in between pixels that are close together in the image, thus making structure in the time-serie harder to find. MNIST images are comprised of 784 pixels (28 by 28), requiring dynamics over hundreds of time-steps to be learned. Table 6.3 shows that cellular constraints do not hinder performances when compared to GRU cells, even for more complex standard benchmarks, in which specific long-term memory is not required. In this case, only LMUs provide significantly better performance than nBRCs, which are otherwise competitive with all other cell types.

Permuted line-sequential MNIST In this benchmark, the same permutation of pixels in the MNIST images as for the previous benchmark are used. The pixels are then fed to the RNNs line by line, thus allowing one to test the networks with a higher input dimension (28 in this case). Furthermore, to highlight once again the interest of bistability, N black lines are added at the end of the image. This has the effect of a forgetting period, as any relevant information for predicting the output will be farther from the prediction time-step in the time-serie. As for the copy input benchmark, one can see on Table 6.4 that only bistable cells and LMUs are able to tackle this problem correctly.

X	BRC	NBRC	GORU	LSTMCell	GRUCell	LMU
472	0.958 ± 0.00	0.970 ± 0.00	0.148 ± 0.02	0.630 ± 0.32	0.540 ± 0.43	0.180 ± 0.00

Table 5.6: Overall accuracy ($\pm$ standard deviation) on the permuted variable-sequential-line MNIST test set after 50 epochs for different architectures. Images are fed to the recurrent network line by line and $N \sim U\{0, \dots, X\}$ black lines are added at the bottom of the image after permutation.

Permuted variable-line-sequential MNIST Finally, to test the capacity of the network on variable-length sequences, a variation of this benchmark is also tested and is called permuted variable-sequential-line MNIST. In this variation, a random number N of black lines are added to the image (after permutation of the pixels) for each sample, where N is sampled uniformly in $\mathcal{U}\{0, \dots, X\}$. Additionally, all pixels of the last line are assigned a high positive value (greater than the value corresponding to that of a white pixel, so that this line can never appear in a standard image). This line can be used by the network for it to know it should output the class of the image for that particular time-step. We note that in this benchmark, samples are of variable lengths. In this case (Table 5.6), results are more similar to those obtained in the denoising benchmark.

5.4.2 Analysis of nBRC dynamic behavior

Until now, only the learning performances of bistable recurrent cells was discussed. It is, however, interesting to take a deeper look at the dynamics of such cells to understand whether or not bistability is used by the network. To this end, a random time-series from the denoising benchmark is chosen to analyse some properties of a_t and c_t. For this analysis, a network with 4 layers of 100 neurons each was trained, allowing for the analysis of a deeper network as compared to those used in the benchmarks. Note that the performances of this network are similar to those reported in Table 6.2. Fig 5.4 shows the proportion of bistable cells per layer and the average value of e_t per layer. The dynamics of the parameters show that they are well used by the network, and three main observations should be made. First, as relevant inputs are presented to the network, the proportion of bistable neurons tends to increase in layers 2 and 3, effectively storing information and thus confirming the interest of introducing bistability for long-term memory. As more information needs to be stored, the network leverages the power of bistability by increasing the number of bistable neurons. Second, as relevant inputs are presented to the network, the average value c_t tends to increase in layer 3, effectively making the network less and less sensitive to new inputs. Third, one can observe a transition regime when a relevant input is shown. Indeed, there is a high decrease in the average value of c_t, effectively making the network extremely sensitive to the current input, which allows for its efficient memorization.

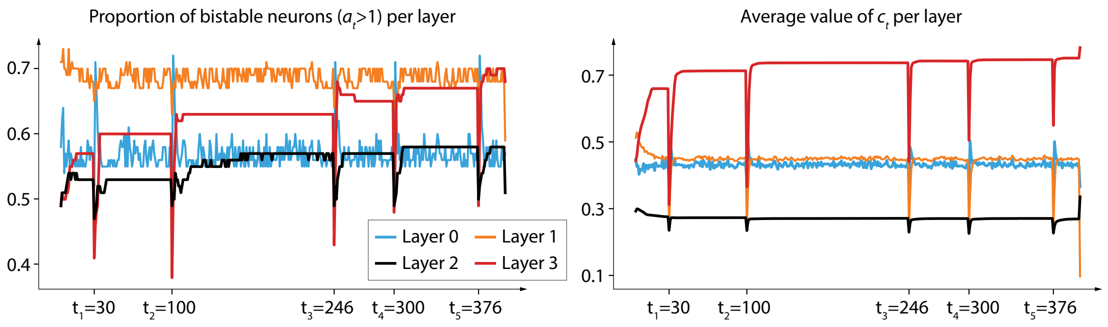


Figure 5.4: Representation of the nBRC parameters, per layer, of a recurrent neural network (with 4 layers of 100 neurons each), when shown a time-series of the denoising benchmark ($T = 400$, $N = 0$). Layer numbering increases as layers get deeper (i.e. layer i corresponds to the ith layer of the network). The 5 time-steps at which a relevant input is shown to the model are clearly distinguishable by the behaviour of those measures alone.

Summary

In this Chapter, two new important concepts from the biological brain were introduced into recurrent neural networks: cellular memory and bistability. This led to the development of two new cells, called the Bistable Recurrent Cell (BRC) and recurrently neuromodulated Bistable Recurrent Cell (nBRC) that proved to be very efficient on several datasets requiring long-term memory and on which the performances of classical recurrent cells such as GRUs and LSTMS were poor. Furthermore, through the similarities between nBRCs and standard GRUs, it is highlighted that gating mechanisms can be linked to biological neuromodulation.

Furthermore, it is worth it to note that even though, in the context of this Chapter, a particular focus was made on supervised benchmarks, bistable cells might be of great use for RL, and more precisely for RL problems with sparse environments (environments for which rewards or meaningful observations are rare). These problems have been known to be extremely hard to solve, on one hand due to the difficulty of exploration and on the other hand due to the difficulty of remembering relevant information across large periods of time-steps. Bistable cells are a promising avenue for solving the latter, and might be a worthwhile path to explore. In fact, some preliminary work has been carried on the subject by Lambrechts & Ernst (2021) and De Geeter & Drion (2021) showing promising results. Specifically, it was shown that such bistable cells could learn good policies in more complex specific environments than GRUs and LSTMs. The policies learned by multistable networks were also shown to generalize better in these specific environments. Although, this work is still preliminary. As such, a more thorough analysis should be carried on more standard environments to assess the performances of bistable cells in such a setting.

Chapter 6

Warming-up recurrent neural networks

This Chapter contains the core of the following publication (Vecoven et al., 2021b).

As seen in the previous Chapter, training standard gated cells such as GRUs and LSTMs on benchmarks where long-term memory is required remains an arduous task. In this Chapter, it is proposed to introduce a general way to initialize any recurrent network connectivity through a process called "warm-up" which improves their capability to learn arbitrarily long time dependencies. This initialization process is designed to maximize network reachable multistability, i.e. the number of attractors within the network that can be reached through relevant input trajectories. Warming-up is performed before training, using stochastic gradient descent on a specifically designed loss. We show that warming-up greatly improves recurrent neural network performance on long-term memory benchmarks for multiple recurrent cell types, but can sometimes impede precision. Therefore, a parallel recurrent network structure with partial warm-up is introduced, and is shown to greatly improve learning on long time-series while maintaining high levels of precision. This approach provides a general framework for improving learning abilities of any recurrent cell type when long-term memory is required.

First, Section 6.1 introduces a fast-to-compute and differentiable measure called variability amongst attractors (VAA) that counts the number of different reachable attractors within a network. It is shown that loss decrease during learning in long-term memory benchmarks is highly correlated with an increase in VAA, highlighting both the relevance of the measure and the importance of multistability for efficient learning. Second, stochastic gradient ascent is used on VAA before training as a way to maximize the number of reachable attractors within the network. It is shown that this technique strongly improves performance on long-term memory benchmarks, though at the cost of precision, the latter relying on the richness of network transient dynamics. Finally, a parallel recurrent network structure with partial warm-up is proposed and enables the combining of long-term memory through multistability on the one hand, and precision through rich transient dynamics on the other hand. Section 6.2 highlights the results of this architecture for multiple RNNs such as gated recurrent units (GRUs (Cho et al., 2014)) and long-short term memory (LSTMs (Hochreiter & Schmidhuber, 1997)). It is seen that this method indeed retains the benefits of warm-up, while improving predictive performances.

6.1 Variability amongst attractors and warm-up

VAA is proposed as a proxy measure for counting the number of attractors in dynamical systems. It is first detailed how VAA can be computed for any arbitrary dynamical system and then the usage of VAA in the case of RNNs is further discussed. It is then shown how an increase in VAA is highly correlated to loss decrease during RNN training on the denoising benchmark (see Section 5.4). Finally, the warm-up procedure is detailed, explaining how VAA can be used to maximize the number of reachable attractors in RNNs.

Variability amongst attractors. VAA is quite straightforward in essence. Given a batch of different initial-state conditions and a constant perturbation, VAA is computed as the proportion of different states towards which the system converges. States are considered different if their Euclidian distance in phase space is greater than a given threshold $\epsilon \in \mathcal{R}_0^+$. Concretely, a perturbation $\mathbf{x}$ is sampled and $n \in \mathcal{N}_0$ initial states $\{\mathbf{h}_{0,1}; \ldots; \mathbf{h}_{0,n}\}$ are sampled in phase space where n is a hyper-parameter. It is later detailed how to define these distributions for RNNs. For each initial state, the model converges over M time-steps with $\mathbf{x}$ as input. If we define δ as the minimal Euclidian distance between two attractors, M must be chosen large enough such that the Euclidian distance between all $\mathbf{h}_{M,i}$ and their corresponding attractor is smaller than $\delta/4$. This ensures that there exists a threshold ϵ that captures all states belonging to a same attractor while ensuring zero overlap between states belonging to different attractors. Finally, the number of unique vectors (given ϵ) in the final states $\{\mathbf{h}_{M,1}; \ldots; \mathbf{h}_{M,n}\}$ can be counted. This is done by first building a correspondence matrix C where $C_{i,j}$ is equal to 1 if $\mathbf{h}_{M,i}$ is close enough to $\mathbf{h}_{M,j}$ in Euclidian distance and else to 0. From this matrix, a vector $\mathbf{v}$ such that $\mathbf{v}[i]$ is equal to the number of corresponding states to $\mathbf{h}_{M,i}$ is built. Let m be the number of reached final states associated to an attractor. By definition each of these m states will be similar to $m-1$ other states. Thus, the number of unique states can simply be computed by inverting each element in $\mathbf{v}$ and summing them. Once the number of different states has been computed, it is divided by n to obtain the proportion of different states, and thus the VAA. Alternatively put:

$$\begin{cases} C_{i,j} = 1 \text{ if } ||\mathbf{h}_{M,i} - \mathbf{h}_{M,j}|| < \epsilon \text{ else } 0,\ \forall i, j \in \{1, \ldots, n\}^2 \\ \mathbf{v}[i] = \sum_{j=1}^{n} C_{i,j}\ \forall i \in \{1, \ldots, n\} \\ VAA = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\mathbf{v}[i]} \end{cases} \tag{6.1}$$

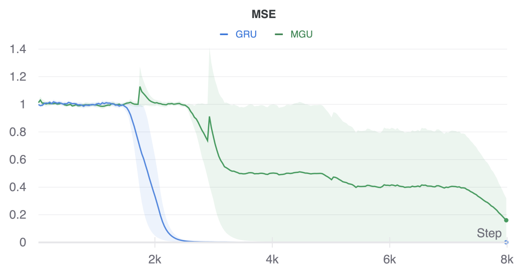
It is noted that VAA will vary between $\frac{1}{n}$ and 1. Indeed, if the system is mono-stable (has only one attractor) then the VAA will be equal to $\frac{1}{n}$, whereas if the network converges to different states for all input trajectories, VAA will be equal to 1.

When measuring the VAA on RNNs, the interest lies in the number of attractors that can be reached when receiving time-series from a given dataset as inputs. For this reason, when computing the VAA, a batch of n samples is used from the same dataset as that for training the model, for which each sample corresponds to a time-series $\mathbf{X}_i = \{\mathbf{x}_{0,i}, \ldots, \mathbf{x}_{T,i}\}$. Each time-series is then truncated at random before being fed to the RNN. The resulting final states of the system (one per time-series) are used as initial states for the VAA procedure detailed above. Furthermore, the perturbation $\mathbf{x}$ is sampled from a multivariate normal distribution. By repeating the VAA procedure multiple times and if VAA is greater than $\frac{1}{n}$ for all different sampled perturbations, we can then assume that the RNN is multistable for a wide range of perturbations. The full procedure for computing the VAA of an RNN is presented in Algorithm 1.

In terms of computations, measuring VAA is similar to a batched forward pass of the network and is thus very efficient, allowing its computation during training. To illustrate the relevance of this measure, a GRU network is trained on a denoising benchmark (see Section 5.4 for full details on the benchmark). Figure 6.1 shows a strong correlation between the ability of the network to learn (thus, a decrease in the mean-squared error) and its measured VAA. Interestingly, it is noted that each VAA measure reported in the Figure is computed with different perturbations, which suggests that when RNN become multistable, it is for a wide range of perturbations. Furthermore, it can also be observed that GRU are mono-stable at initialization and that multiple gradient steps are required to reach multistability. These observations motivate to propose warm-up, a procedure in which VAA is used to promote multistability in RNNs before training.

Algorithm 1: Computing VAA for an RNN

```
Data: $\mathcal{X}$ a set of $n$ time-series $\{\mathbf{X}_1, \ldots, \mathbf{X}_n\}$ sampled in the training set.
Parameters: $M \in \mathcal{N}_0$ the number of time-steps used for state convergence.
            $\epsilon \in \mathcal{R}_0^+$ tolerance when considering state similarity.
            $\theta$ the architecture and parameters of the network.
Result: VAA for a random perturbation, computed on the given data.
/* Compute the initial states.                                  */
$\mathcal{H} \leftarrow \{\}$
foreach $\mathbf{X}_i \in \mathcal{X}$ do
    $c \sim U\{1, \ldots, T\}$ where $T$ is the length of $\mathbf{X}_i$
    $\mathbf{h}_i \leftarrow 0$
    for $t \leftarrow 1$ to $c$ do
        $\mathbf{h}_i \leftarrow f(\mathbf{h}_i, \mathbf{x}_{t,i}; \theta)$ where $f(\cdot, \cdot; \theta)$ is the RNN's update rule.
    end
    $\mathcal{H} \leftarrow \mathcal{H} \bigcup \{\mathbf{h}_i\}$
end
/* Use initial states to compute VAA of each layer              */
$\mathbf{x} \sim N(\mathbf{0}, \mathbf{1})$
for $t \leftarrow 1$ to $M$ do
    $\mathbf{h}_i \leftarrow f(\mathbf{h}_i, \mathbf{x}; \theta), \forall \mathbf{h}_i \in \mathcal{H}$
end
$C_{i,j} \leftarrow 1$ if $||\mathbf{h}_i - \mathbf{h}_j|| < \epsilon$ else $0, \forall i, j \in \{1, \ldots, n\}$
$\mathbf{v}[i] \leftarrow \sum_{j=1}^{n} C_{i,j} \; \forall i \in \{1, \ldots, n\}$
$VAA \leftarrow \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\mathbf{v}[i]}$
```



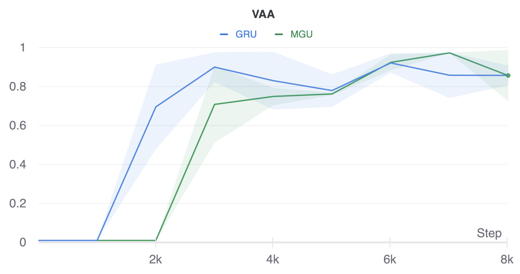


Figure 6.1: Evolution of the loss (left) and of VAA (right, computed with $n = 100$ and $M = 3000$) of muliple GRU and MGU networks trained on the denoising benchmark. The average over three runs is plotted ($\pm$ standard deviation). Learning, represented by a decreasing loss, only starts when the network becomes multistable (VAA greater than $1/n$).

Warming-up. The goal of warm-up is to maximize the number of reachable attractors of a RNN for a given dataset, that is, to maximize VAA. As is usually done for training neural networks, we propose using stochastic gradient descent so as to maximize VAA. However, SGD cannot be used directly on the VAA measure detailed in Algorithm 1 for two different reasons.

- First, it is important to note that the deeper recurrent neural networks are, the bigger M must become for reaching convergence. Indeed, one must wait for the shallower layers to converge before reaching convergence of deeper layers. This is still practical for computing VAA but can become too expensive for propagating the gradient back through time on such long sequences.
- Second, the correspondence matrix C is not differentiable due to the operations used to build it.

To solve the first problem, when warming up, each layer is treated as a separate dynamical system and the number of reachable attractors is maximized for each layer independently.

Solving the second problem can easily be done by introducing a differentiable proxy measure for VAA. The differentiable proxy measure for computing VAA is denoted as VAA^*. The only difference with the VAA measure is in the definition of C as that is the only non-differentiable computation. For all i, j in $\{1, \dots, n\}$, we approximate C as:

$$\begin{cases} \hat{C}_{i,j} = 1 - \frac{\max(0, \|\tanh(\mathbf{h}_{M,i}) - \tanh(\mathbf{h}_{M,j})\| - \epsilon)}{\|\tanh(\mathbf{h}_{M,i}) - \tanh(\mathbf{h}_{M,j})\|} \text{ if } i \neq j \\ \hat{C}_{i,j} = 1 \text{ else.} \end{cases} \tag{6.2}$$

It is noted that the value of $\hat{C}_{i,j}$ is strictly equal to 1 if $\mathbf{h}_{M,i}$ is close enough in Euclidian distance to $\mathbf{h}_{M,j}$. On the other hand, $\hat{C}_{i,j}$ will be close to 0 when $\mathbf{h}_{M,i}$ and $\mathbf{h}_{M,j}$ are different. We note that the $\hat{C}_{i,j}$ will never strictly be equal to 0, but will get closer as the distance between $\mathbf{h}_{M,i}$ and $\mathbf{h}_{M,j}$ increases since $\frac{\|\tanh(\mathbf{h}_{M,i}) - \tanh(\mathbf{h}_{M,j})\| - \epsilon}{\|\tanh(\mathbf{h}_{M,i}) - \tanh(\mathbf{h}_{M,j})\|}$ will get closer to 1. Although states being far apart from each others is not of a particular interest in itself, it appeared that this small bias of VAA^* (increasing slightly as distance between states grows) provides a good direction to the gradient for reaching multistability. Furthermore, one must note that this bias encourages using a saturating function (a hyperbolic tangent in this case) on the states. It permits to saturate attractor values even in non-saturated recurrent cells, avoiding extreme states when warming-up. The procedure for computing VAA^* for all layers of an RNN is given in Algorithm 2.

Algorithm 2: Computing the set of VAA^* for an RNN

```
Data: $\mathcal{X}$ a set of $n$ time-series $\{\mathbf{X}_1, \dots, \mathbf{X}_n\}$ sampled in the training set.
Parameters: $M \in \mathcal{N}_0$ the number of time-steps used for state convergence.
            $\epsilon \in \mathcal{R}_0^+$ tolerance when considering states similarity.
            $\theta$ the architecture and parameters of the network.
Result: $\mathcal{V}$ the set of $VAA^*$ of each layer for a random perturbation, computed on the
        given data.
/* Compute the initial states.                                                    */
$\mathcal{X} \leftarrow \{\}$
foreach $\mathbf{X}_i \in \mathcal{X}$ do
    $c \sim U\{1, \dots, T\}$ where $T$ is the length of $\mathbf{X}_i$
    $\mathbf{h}_i \leftarrow 0$
    for $t \leftarrow 1$ to $c$ do
        $\mathbf{h}_i \leftarrow f(\mathbf{h}_i, \mathbf{x}_{t,i}; \theta)$ where $f(\cdot, \cdot; \theta)$ is the RNN's update rule.
    end
    $\mathcal{H} \leftarrow \mathcal{H} \bigcup \{\mathbf{h}_i\}$
end
/* Use initial states to compute $VAA^*$ of each layer                              */
$\mathcal{V} \leftarrow \{\}$
foreach layer $l$ in $\theta$ do
    $\mathbf{x}^l \sim N(\mathbf{0}, \mathbf{1})$
    for $t \leftarrow 1$ to $M$ do
        $\mathbf{h}_i^l \leftarrow f^l(\mathbf{h}_i^l, \mathbf{x}^l; \theta^l), \forall \mathbf{h}_i^l \in \mathcal{H}$ where $f_l(\cdot, \cdot; \theta^l)$, $\mathbf{h}^l$ and $\theta^l$ are respectively the update
          function, hidden state and parameters of layer $l$.
    end
    $\hat{C}_{i,j} \leftarrow 1 - \frac{\max(0, || \tanh(\mathbf{h}_{M,i}^l) - \tanh(\mathbf{h}_{M,j}^l)|| - \epsilon)}{|| \tanh(\mathbf{h}_{M,i}^l) - \tanh(\mathbf{h}_{M,j}^l)||}$, $\forall i, j \in \{1, \dots, n\}, i \neq j$
    $\hat{C}_{i,i} \leftarrow 1, \ \forall i \in \{1, \dots, n\}$
    $\mathbf{v}[i] \leftarrow \sum_{j=1}^{n} \hat{C}_{i,j} \ \forall i \in \{1, \dots, n\}$
    $VAA^* \leftarrow \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\mathbf{v}[i]}$
    $\mathcal{V} \leftarrow \mathcal{V} \bigcup VAA^*$
end
```

Stochastic gradient descent can then be used to get the VAA^* of each layer as close as possible to $k \in [0,1]$. In practice, we use $k = 0.95$ as this proved, empirically, to maximize the number of attractors while avoiding too extreme states that could arise from the approximation of C. This is done with an usual MSE loss, defined as

$$\mathcal{L}(\mathcal{U}, M, \theta) = \sum\!\!\!\!\!\!\!\!\;\;\; \bigg|_{v_l \in \text{VAA}^*(\mathcal{U}, M, \epsilon, \theta)} (v_l - 0.95)^2$$

where $VAA^*(\mathcal{U}, M, \epsilon, \theta)$ represents the procedure for computing VAA^* with the corresponding parameters and data depicted in Algorithm 2 and where $\sum\!\!\!\!\!\!|\;$ represents an average over a set. Batches are sampled in the training set and to avoid over-fitting problems, M is sampled uniformly in $U(1, \dots, M^*)$ at each gradient step. M^* is a variable initialized at 1 and increased by a constant $c \in \mathcal{N}_0$ after each gradient step. This progressive increase, driven by the curriculum learning speed c, is required to smoothly reach multistability, avoiding gradient problems. Furthermore, this progressive increase is motivated by some results obtained in previous Chapter, showing that guidance by "short-term memory" samples could help standard gated cells learn how to handle longer relations. Algorithm 3 details the whole warm-up procedure.

Algorithm 3: Warming-up an RNN

```
Data: $\mathcal{D}$ a training set of time-series.
Parameters: $S \in \mathcal{N}_0$ the number of gradient steps.
            $lr \in \mathcal{R}_0^+$ the learning rate.
            $c \in \mathcal{N}_0$ constant driving the curriculum learning speed.
            $M \in \mathcal{N}_0$ maximum convergence steps for VAA computation.
            $\theta$ parameters of the network.
Result: Updates $\theta$ for multistability in different layers.
$M^* \leftarrow 1$
for $t \leftarrow 1$ to $S$ do
    $\mathcal{X} \leftarrow U^n(\mathcal{D})$ where $U^n(\mathcal{D})$ denotes a set of $n$ elements sampled uniformly in $\mathcal{D}$.
    $L \leftarrow \mathcal{L}(\mathcal{X}, M^*, \theta)$
    $\theta \leftarrow \theta - lr * \frac{\delta L}{\delta \theta}$
    $M^* \leftarrow \min(M, M^* + c)$
end
```

It is shown on Figure 6.2 that the warm-up procedure effectively increases the VAA^* of each layer in an RNN. Furthermore, and most importantly, one can also see on Figure 6.2 that as the warm-up procedure is carried out, the true VAA measure of the RNN also increases, even reaching 1 as the warm-up procedure ends.

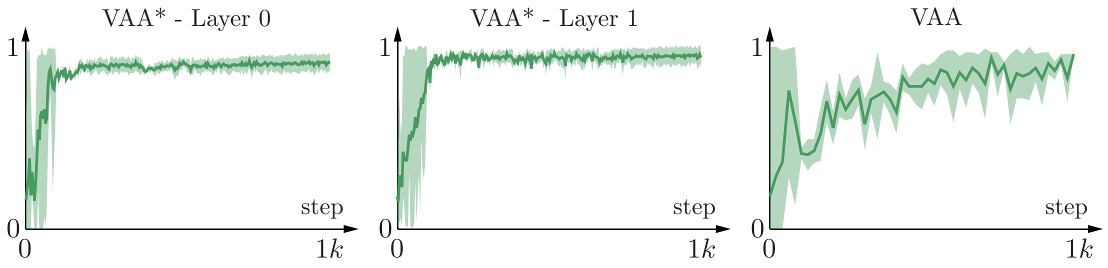


Figure 6.2: Evolution of the VAA^* for a two-layer RNN (left and middle) and of the VAA of the network (right) during warm-up. This network was warmed up on the denoising dataset and results were averaged over three runs.

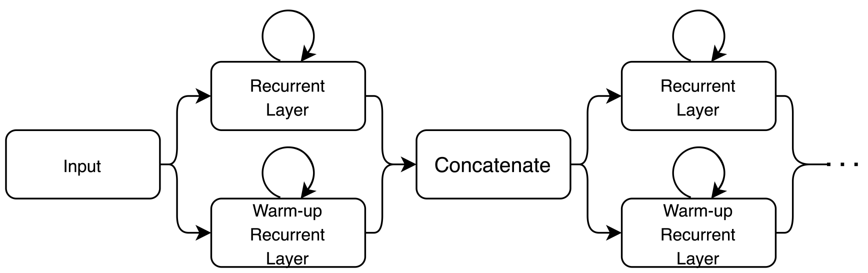


Figure 6.3: Scheme of a double-layer architecture. Each recurrent layer is split in two equal parts, only one of which has its parameters warmed-up. The output of the recurrent layer is then computed as the concatenation of the outputs of each of its parts. This effectively divides a recurrent layer into two separate dynamical systems.

Double-layers Until now, only attractors have been discussed. However, as mentioned in Chapter 4, Sussillo & Barak (2013) also pointed the importance of the transient dynamics of RNNs for prediction. It was observed that when warming up neural networks, they tend to lose predictive accuracy, to the benefit of easier training on longer time-series. To alleviate this problem and obtain precise predictions while maintaining the benefits of warm-up, a double-layer architecture is proposed. Each recurrent layer is simply split into two equal parts and only one of them is warmed up, that is, VAA^* is only computed on those parts of the network and solely their variables are updated when warming up. This enables the endowment of some part of each layer with multistability, while the rest remains mono-stable with richer transient dynamics. As a mean of visualisation, a double-layer structure is depicted on Figure 6.3.

6.2 Results

To demonstrate the impact of warming up RNNs, the same benchmarks previously as introduced in Chapter 5 are used. These benchmarks were proposed in (Vecoven et al., 2021a) to test the ability of RNN cells to learn long-term dependencies and the reader is referred to Section 5.4 of Chapter 5 for a precise description of each of the benchmark tackled in this Section. In this Chapter, warm-up is tested on three different types of cells. To this end, LSTMs, GRUs and minimal gated units (MGUs, proposed by Zhou et al. (2016)) are trained without warm-up, with warm-up and with double-layer warmup (DLWU) on these benchmarks and it is shown that their performance is greatly improved with warm-up, in a single or double-layer setting. Parameters for warm-up can vary depending on architectures and needs, however $lr = 1e^{-2}$, $c = 10$, $S = 100$ and $M = 200$ were found to be a good choice for an effective and fast warm-up on our benchmarks. For the first benchmark, networks were made of one 128 neuron recurrent layer. For the other two benchmarks, networks were made of two recurrent layers, each of 256 neurons. All averages and standard deviations reported were computed over three different seeds and training was done with the ADAM optimizer and a learning rate of $1e^{-3}$. Concerning warm-up, it is noted that in some rare cases, if the VAA^* is too close to 1 after warming up, RNNs become stuck and unable to learn. This is due to a bad fitting of the VAA^* measure and thus internal states becoming too extreme. Although very rare, this issue was solved by restarting any run for which VAA^* is too high for any layer after warm-up (0.98 was empirically chosen as a threshold in the paper).

Copy first input benchmark This benchmark allows for a simple proof of concept that warming up RNNs provides certain benefits for training. Indeed, one can see in Table 6.1 that warm-up greatly improves the performances of all RNNs as T increases. This is highly interesting as it highlights that, when long-term memory is required, warming up seems to initialise RNNs near a good dynamical regime.

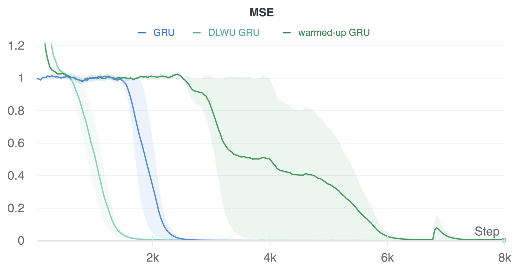
Denoising benchmark In this benchmark, the amount of information the network must store is much greater as it needs to store five real values. This allows for a demonstration of the effectiveness of the double-layer architecture combined with warm-up. Indeed, Table 6.2 shows that warm-up is required for the networks to learn when N increases. It is important to note that the standard deviation obtained for MGUs and LSTMs without double-layers is due to failed runs. That is, despite a successful warm-up, the network is not able to learn and loses its multistability properties. This is likely due to the nature of the problem which requires precise transient dynamics for outputting the prediction. One can see that adding a double-layer architecture solves this problem.

T	Warm-up	MGU	LSTM	GRU
50	None	0.831 ± 0.324	0.634 ± 0.427	0.997 ± 0.005
	warmed-up	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$
	DLWU	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$
300	None	0.98 ± 0.002	0.95 ± 0.019	1.003 ± 0.002
	warmed-up	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$
	DLWU	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$
600	None	1.017 ± 0.004	0.977 ± 0.008	1.017 ± 0.003
	warmed-up	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$
	DLWU	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$	$\mathbf{0.000 \pm 0.000}$

Table 6.1: Mean square error ($\pm$ standard deviation) of different architectures and different warm-up strategies on the test set for the copy input benchmark. Results are shown after 50 epochs and for different values of T.

N	Warm-up	MGU	LSTM	GRU
5	None	0.005 ± 0.008	1.001 ± 0.003	$\mathbf{0.000 \pm 0.000}$
	warmed-up	0.002 ± 0.006	0.032 ± 0.011	$\mathbf{0.000 \pm 0.000}$
	DLWU	$\mathbf{0.002 \pm 0.001}$	$\mathbf{0.025 \pm 0.007}$	$\mathbf{0.000 \pm 0.000}$
100	None	1.004 ± 0.003	0.996 ± 0.005	0.995 ± 0.003
	warmed-up	0.32 ± 0.641	0.338 ± 0.561	0.001 ± 0.002
	DLWU	$\mathbf{0.024 \pm 0.023}$	$\mathbf{0.013 \pm 0.125}$	$\mathbf{0.000 \pm 0.000}$

Table 6.2: Mean square error ($\pm$ standard deviation) of different architectures on the denoising benchmark's test set. Results are shown with and without constraint on the location of relevant inputs and after 50 epochs. Relevant inputs cannot appear in the N last time-steps. In this experiment, results were obtained with $T = 200$.



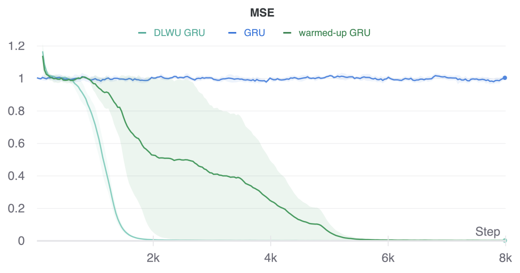


Figure 6.4: Evolution of the loss for GRU networks trained on the denoising benchmark with $N = 5$ (left) and $N = 100$ (right). The average over three runs is plotted ($\pm$ standard deviation). Learning, represented by a decreasing loss is the fastest with the double-layer architecture. Meanwhile, we see that warming-up makes it harder for the network to generate necessary transient dynamics. Once again, it also appears that warming-up is necessary for learning longer time-dependancies.

Warmup	MGU	LSTM	GRU
None	0.896 ± 0.004	0.907 ± 0.002	$\mathbf{0.925 \pm 0.004}$
warmed-up	0.897 ± 0.001	0.402 ± 0.012	0.102 ± 0.061
DLWU	$\mathbf{0.901 \pm 0.003}$	$\mathbf{0.909 \pm 0.005}$	0.914 ± 0.014

Table 6.3: Overall accuracy ($\pm$ standard deviation) on the permuted sequential MNIST benchmark test set after 70 epochs for different warm-up methods and for different cell types.

To further highlight the impact of the double-layer architecture, as well as that of warming up, Figure 6.4 shows the training curve of GRU networks for both values of N.

Permuted sequential MNIST Table 6.3 shows that warm-up with a double-layer architecture provides equivalent performances than without warming up. The slight decrease in GRU performance can be explained by the lower number of parameters in the double-layer architecture as compared layers connected fully recurrently. This benchmark shows the importance of a double-layer structure when warming up RNNs.

Permuted line-sequential MNIST Finally, Table 6.4 further highlights the importance of the double-layer architecture when training on more complex time-series. Again, the impact of warming-up the network also appears.

N	Warm-up	MGU	LSTM	GRU
72	None	$\mathbf{0.968 \pm 0.001}$	$\mathbf{0.977 \pm 0.002}$	$\mathbf{0.977 \pm 0.002}$
	warmed-up	0.893 ± 0.132	0.890 ± 0.017	0.873 ± 0.016
	DLWU	0.967 ± 0.021	0.976 ± 0.032	0.974 ± 0.012
472	None	0.198 ± 0.021	0.562 ± 0.328	0.591 ± 0.388
	warmed-up	0.534 ± 0.592	0.942 ± 0.009	0.493 ± 0.556
	DLWU	$\mathbf{0.828 \pm 0.152}$	$\mathbf{0.961 \pm 0.007}$	$\mathbf{0.973 \pm 0.001}$

Table 6.4: Overall accuracy (±standard deviation) on permuted sequential-line MNIST test set after 70 epochs for different architectures and warm-up methods. Images are fed to the recurrent network line by line and N black lines are added at the bottom of the image after permutation. We note that when N equals 72(472) the resulting image has 100(500) lines.

Summary

In this Chapter, a procedure for warming-up recurrent neural networks was proposed to improve their ability to learn long time-dependencies. The procedure is motivated by recent work that showed the importance of fixed points and attractors for the prediction process of trained RNNs. To this end, a lightweight measure called VAA and that can be optimized to endow RNNs with multistable dynamics was introduced. Warm-up can be used with any type of recurrent cell and it was shown to vastly improves their performance on long-term memory benchmarks when combined with a double-layer architecture. As this procedure is general and easy to implement, it could easily be further tested on multiple benchmarks.

As future work, similarly to bistable recurrent cells, an area of application that is promising for such an approach is sparse reinforcement learning. In fact, warming up RNNs would allow them to remember information for much longer, and they could thus be more robust to complex exploration in such a setting.

Furthermore, it is worthwhile to note that the double-layer architecture might be worth exploring with different types of cells. It was discussed here that there are benefits of using different types of initialization for the same type of cell. This might hint at the possibility of having similar benefits when using different types of cells, each with different dynamical properties, in RNNs. One could for example study the benefits of introducing bistable cells in RNNs mixed with standard gated cells such as GRUs and LSTMs.

Finally, in this Chapter, the aim was to maximize the number of attractors through warming up before training. However, in some rare cases, it was observed that networks can lose their multistability properties when training. To avoid this, using VAA as a regularization loss during training could be interesting. Likewise, when warming-up, one could choose to aim for a number of attractors that is optimized for a given benchmark. This provides further room for improvement of algorithm performance.

PART III

Neuromodulating neural networks for adaptation

Introduction

Despite the tremendous progress made in the field of machine learning over the years, there is however one specific type of task natural to humans and where machines still struggle to show reasonable performance: the adaptive control of continuous behaviors in unknown, time-varying contexts. Compared to humans, there are two main aspects to which this difference in performance can be attributed.

- First, RL agents often require a massive amount of data to learn decent policies. On the opposite side, humans can often achieve decent performance with very little experience in a specific environment.
- Second, and which can be linked to the first point, RL agents often specialize on a single domain, whereas humans can flexibly react to changing task conditions. Indeed, an RL agent trained on a specific task will often exhibit extremely poor performance if tested on a similar, albeit very slightly different one. A human, on the other hand, would easily be able to adapt to small changes without much of a performance loss.

In other words, standard machine learning methods usually lack the capacity to learn general representations which would allow them to quickly learn how to tackle similar tasks. Recently, there has been work towards solving these issues. The machine learning settings which are closely related to this issues have been called "meta-learning" and "meta-reinforcement learning" (related to their non-meta version). In such settings, one does not have to learn a single task, but rather to learn over a distribution of such tasks.

In the case of meta-learning, the goal is to train a model to learn a task as quickly as possible. For example, few-shot learning, a setting in which a model has to learn how to distinguish samples with only very few of them labelled, can be associated to meta-learning. In this setting, the train set consists of samples coming from multiple tasks. The test set is composed of different (but similar) tasks, for which very few labelled samples are available. As such, the model has to learn how to quickly adapt (or quickly learn) such new tasks, based on the knowledge acquired when training on other tasks. A high-level view sketch of this setting is depicted on Figure 6.5

Similarly, in the case of meta-reinforcement learning, the agent does not interact with a single environment, but rather with a distribution of such environments. This setting is thus very well suited for testing adaptation capabilities of RL agents. Indeed, to obtain good rewards in such benchmarks, the agent must be able to perform well in environments it has never seen during training (i.e. to be robust and adapt to these environments with very few interactions).

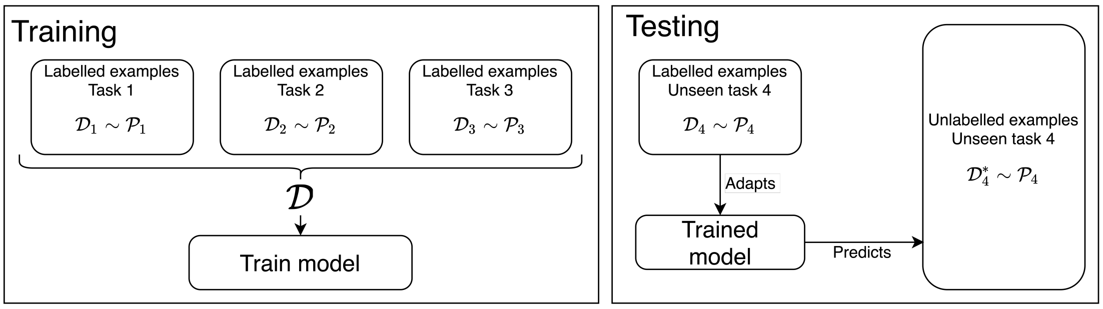

Training
Labelled examples Task 1
$\mathcal{D}_1 \sim \mathcal{P}_1$
Labelled examples Task 2
$\mathcal{D}_2 \sim \mathcal{P}_2$
Labelled examples Task 3
$\mathcal{D}_3 \sim \mathcal{P}_3$
$\mathcal{D}$
Train model
Testing
Labelled examples Unseen task 4
$\mathcal{D}_4 \sim \mathcal{P}_4$
Adapts
Trained model
Predicts
Unlabelled examples Unseen task 4
$\mathcal{D}_4^* \sim \mathcal{P}_4$

Figure 6.5: Sketch of a meta-learning problem

In the nervous system, the robust control of continuous behaviors is often associated with a second class of physiological mechanism as important as synaptic plasticity but less studied and much less understood, called neuromodulation (Bargmann & Marder, 2013). Neuromodulation is the ability of neurons to tune their input-output properties to reshape signal transmission at the cellular level, generally in response to an external signal carried by biogenic chemicals called neuromodulators (Bargmann & Marder, 2013; Marder et al., 2014). Neuromodulation regulates many critical nervous system properties that cannot be achieved by synaptic plasticity alone. One recent and striking example is in regards to the critical role of neuromodulation in the voluntary control of locomotion : using a brain-spine interface, researchers have recently been able to aleviate walking deficits in rodents (Van den Brand et al., 2012) and primates (Capogrosso et al., 2016) suffering from paralysing spinal cord injury by specifically restoring neuromodulation below the lesion site. Although experimental studies have highlighted the ubiquity of neuromodulation in all nervous systems, the basic principles governing this mechanism have not been formulated to date, and the potential impact of neuromodulation-inspired mechanisms on research strategies in engineering has yet to be fully explored.

One big obstacle in the study of neuromodulation is the high complexity of the underlying mechanisms, both from physiological and dynamical viewpoints. Although basic rules of synaptic plasticity have been quickly formulated and shown to adequately describe biological mechanisms (Abbott & Nelson, 2000), neuromodulation deals with high-dimensional, highly nonlinear dynamical systems and targets a vast number of different cellular and network properties, making it difficult to tackle. For instance, even the most recent, qualitative modeling attempts of brain signaling lack neuromodulation (Markram et al., 2015). However, recent advances have been made on the mechanisms underlying the robustness and modulation of neuron intrinsic properties (Franci et al., 2013a,b; Drion, O'Leary, & Marder, 2015; Drion, O'Leary, et al., 2015; O'Leary et al., 2013; O'Leary et al., 2014). These advances bring us closer to the extraction of basic rules of neuromodulation, a necessary step for the design of neuromodulation-inspired strategy and methods in artificial intelligence.

In fact, there has recently been a growing interest in introducing neuromodulation mechanisms into artificial neural networks, for different purposes. The works of Tsuda et al. (2021), Geadah et al. (2020), Beaulieu et al. (2020) and Wilson et al. (2018) are very recent examples that suggest these mechanisms to provide benefits for deep neural networks. As such, these works will be further discussed in Chapter 8.

Indeed, Part III focuses on the introduction of neuromodulation in deep neural networks as a mean to improve their capabilities to adapt. To this end, Chapter 7 formalizes meta-reinforcement learning. Chapter 7 will also discuss some recent architectures and methods to tackle meta-learning as well as meta-reinforcement learning. The goal of Chapter 8 is then to propose an architecture, based on neuromodulation, to tackle such tasks. This study highlights the benefits of designing an architecture specifically designed towards easily adapting (and thus, the interest of neuromodulation) over using more standard ones.

Chapter 7

Adaptation capabilities of artificial neural networks

Meta-learning and, more particularly, meta-reinforcement learning are known to be very difficult settings. As such, there has been a growing body of work regarding the capabilities of neural networks to solve such tasks, leading to the proposal of multiple new training algorithms. It is important to note that it is possible to formalize such a setting in multiple ways. Therefore, Section 7.1 starts by introducing the formalization of meta-reinforcement learning used in this manuscript. It will also be discussed that such a setting can in fact be associated to a regular POMDP. Then, Section 7.2 shortly describes the main classes of methods for tackling such a setting. Some examples of each class will also be shortly introduced.

7.1 Meta-reinforcement learning

In this section, the meta-reinforcement learning (meta-RL) problem is formalized along the lines of the setting used by J. X. Wang et al. (2016). One should note that other variants can also be thought of as relevant instances of the generic meta-RL problem. To end the section, the link with a standard RL setting and a partially observable environment will also be discussed.

Meta-RL formalization In a meta-RL setting, an agent has to interact through a sequence of episodes with MDPs drawn from a distribution η. All of these MDPs have different transition and reward functions and are assumed to have the same state space $\mathcal{S} \subseteq \mathcal{R}^n$ and the same action space $\mathcal{A} \subseteq \mathcal{R}^m$. At the beginning of a new episode i, an element from η is drawn to define an MDP with which the meta-RL agent interacts afterwards. Let us refer to such an element as $\mathbb{M}$. We refer to the transition and reward function of this MDP as $p_{\mathbb{M}}(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ and $\rho_{\mathbb{M}}(\mathbf{a}_t, \mathbf{s}_t, \mathbf{s}_{t+1})$ respectively. Finally, note that, as for the classical RL settings discussed in Part I, all the reward functions are also assumed to be bounded, i.e. $\rho_{\mathbb{M}}(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in [R_{min}, R_{max}]$ $\forall \mathbf{s}_t, \mathbf{s}_{t+1} \in \mathcal{S}, \mathbb{M} \in \eta$, $\mathbf{a}_t \in \mathcal{A}$ and with $R_{min}, R_{max} \in \mathcal{R}$.

The interaction process between the meta-RL agent and $\mathbb{M}$ is defined as follows:

1. The initial state $\mathbf{s}_0$ is drawn according to the distribution over the initial states $p_{\mathbb{M}_{s_0}}(\cdot)$.
2. At each time step, the agent selects an action $\mathbf{a}_t \in \mathcal{A}$, for which it transitions from state $\mathbf{s}_t$ to a new state $\mathbf{s}_{t+1}$ and receives a reward r_t.
3. The agent can observe the different states encountered and the rewards obtained. That is, all MDPs that belong to the support of η are considered to be fully observable, and thus the observation $\mathbf{x}$ received by the agent are equal to $\mathbf{s}$.
4. As for the formalization of RL proposed in Chapter 2, the interaction lasts an infinite number of time steps.

The only information that the agent collects on the transition function of $\mathbb{M}$ and its reward function is through observing the states crossed and the rewards obtained at each time-step.

Let $\mathbf{s}_{i,t}$, $\mathbf{a}_{i,t}$ and $r_{i,t}$ refer to the values of $\mathbf{s}_t$, $\mathbf{a}_t$ and r_t of episode i. Also let

$$\mathbf{h}_{i,t} = \{\mathbf{s}_{i,0}, \mathbf{a}_{i,0}, r_{i,0}, \mathbf{s}_{i,1}, \ldots, \mathbf{a}_{i,t-1}, r_{i,t-1}, \mathbf{s}_{i,t}\}$$

be the history of the interaction of the meta-RL agent the MDP of episode i up to time step t and $\mathcal{H}$ denote the set of all possible such histories. Finally, let $\pi : [\mathcal{H} \times \mathcal{U}] \to [0,1]$ be the policy played by the RL agent during episode and Π denote the set of all such policies. Without loss of generality, we assume that it is a (possibly degenerated) probabilistic policy that selects the action $\mathbf{a}_{i,t}$ to be played at time t based on the knowledge of $\mathbf{h}_{i,t}$ such that $\mathbf{a}_{i,t} \sim \pi(\mathbf{a}|\mathbf{h}_{i,t})$. The return of policy π during episode i called $R^{\pi}_{\mathbb{M}^i}$ writes as:

$$R^{\pi}_{\mathbb{M}^i} = \lim_{T\to\infty} \sum_{t=0}^{T} \gamma^t r_{i,t}$$

where $\gamma \in [0,1[$ the discount factor, common to all MDPs belonging to the support of η and $\mathbb{M}^i$ denotes the MDP drawn at episode i. The goal of the meta-learning agent is to maximise the expected value of the sum of returns it can obtain over a budget $E \in \mathcal{N}$ of episodes. As we will see later in this document, the policy π^i computed by the algorithm should implicitly, at time t, use the history $\mathbf{h}_{t-1}$, to *adapt* its behaviour to $\mathbb{M}_i$.

Meta-RL seen as a POMDP It is worthwhile to note that this formalization could well correspond to a standard RL problem where the environment is not fully observable. Thus this RL setting could in fact be formulated as a POMDP which could be defined as follows.

In a way to represent the different MDPs with which the agent interacts, a non-observable variable α is introduced. Similarly to MDPs in the previous formalization, this variable is sampled at time $t = 0$ of each episode in a distribution η^* and remains constant throughout the episode. It is noted that the information contained in α should be rich enough to encode the different transition and reward functions of the MDPs belonging to the support η. As such, the full state of the environment $\mathbf{s}_t$ could be described as the concatenation of $\mathbf{x}_t$ and α, that is, $\mathbf{s}_t = [\alpha, \mathbf{x_t}]$. Put otherwise, α is hidden and codes for the fully observable task currently at hand while $\mathbf{x}_t$ fully describes the current state of that task.

The main particularity of this POMDP setting is thus that α remains constant throughout the episode and that every other part of the state is fully observable, as opposed to a more standard POMDP in which non-observable parts of the full state can freely change during an episode. To highlight this, the transition, reward and observation functions of the environment in this setting can be written as:

$$\begin{cases} o(\mathbf{x}_t|\mathbf{s}_t) = o(\mathbf{x}_t|\mathbf{x}_t, \alpha) = \delta(\mathbf{x}_t) \\ p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{a}_t, \alpha) \quad , \\ \rho(\mathbf{a}_t, \mathbf{s}_t, \mathbf{s}_{t+1}) = \rho(\mathbf{a}_t, \mathbf{x}_t, \mathbf{x}_{t+1}, \alpha) \quad . \end{cases} \tag{7.1}$$

Looking at Equations 7.1, it thus clearly appears that, for a given episode, the goal of a meta-RL agent is to infer the value α (which will be further referred to as "context") from past observations and rewards as quickly as possible. Doing so, the full state of the environment becomes known and well-performing actions can be played.

7.2 ANNs for solving meta-RL

Due to their increasing popularity for a wide range of applications, it is a logical development that many works focused on using neural networks to tackle meta-learning. Hospedales et al. (2020) provide a great in-depth survey on the subject, going through many of the different recently proposed methods. The authors introduce a new complete taxonomy, allowing to link and/or distinguish all recent meta-learning algorithms along many different axes. For simplicity and conciseness, a more standard taxonomy is used here. This taxonomy separates the different meta-learning algorithms intro three categories:

1. optimisation based methods,
2. model-based / Blackbox methods and
3. metric learning methods.

The following Chapter focuses on a particular type of model-based method. However, to provide more context, each category is shortly depicted in the following subsections through some of their most common respective algorithms.

7.2.1 Optimisation based methods

These methods are probably the most complex of the three, as they rely on two stages optimisation processes. Indeed, in such methods, the adaptation process is done through optimisation, as would be done for usual training. For such methods, there thus exists an “inner” optimisation task, corresponding to adaptation, as well as an “outer” optimisation task, associated to meta-training. Obviously, the core concept of these methods is therefore to extract as much knowledge as possible during outer optimisation to learn how to make the inner optimisation task as efficient as possible. At test time, when presented a new task, only the inner optimisation part will be carried as a way of adaptation, and thus for a well-working algorithm, this optimisation process should show much greater performance over using a standard method. performance can be improved along multiple axes, depending on the method used. For example, some methods focus on improving sample efficiency while others aim at better convergence or robustness, in between others. One of the most common methods for this class of algorithm, which focuses on improving sample efficiency, has been proposed by Finn et al. (2017) and is called “Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks” (MAML).

Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks In their work, Finn et al. (2017) proposed a meta-optimisation process to learn a set of initial model (focusing primarily on deep neural networks) parameters. It was shown that it is possible to find such a set that few gradient steps (and especially, very few data) are necessary in order to generalise well (i.e. adapt) to new tasks. Following this idea, further works have also aimed at learning the step size and training recurrent networks to compute inner gradients, in between others. On a side note, it is interesting to note that this further highlights the impact that good initial parameters can have on training a neural network for a given task. This echoes, in some sense, discussions on RNNs of Part II.

7.2.2 Metric-learning methods

These methods are mostly relevant for the few-shot learning applications of meta-learning, and are therefore less relevant when it comes to meta-RL. However, as they are still relevant for some meta-learning problems, they are nevertheless shortly described here for informative purposes. The core concept of such methods is to perform a non-parametric "learning" (at adaptation time) by using relevant metrics to compare validation points and training points. These methods thus rely on learning the best metrics possible during meta-training (hence, before adaptation). A common example of such metric can be computed thanks to siamese networks.

Siamese Neural Networks for one-shot image recognition Koch et al. (2015) proposed to use siamese networks, first introduced by Bromley et al. (1993), as a way to build strong discriminative features. In his thesis, Koch et al. (2015) used siamese networks to do a forward pass on two images, resulting in two high level encodings of those images. In essence, siamese networks have two heads with similar parameters and activation functions, and as such, both images are encoded through the same function. Koch et al. (2015) trained its siamese networks to distinguish whether the two presented images are similar (of the same class) or not. This training procedure leads the siamese network to learn high level discriminative features, which can be used to build relevant metrics on unseen classes. Indeed, at test-time, both images (in one-shot learning, one would be labelled but not the other) can be of classes never seen during meta-training, however, the network should still be able to predict whether or not these two images are of the same class. Thus, the model should correctly predict whether or not the non-labelled example is of the same class as that of the labelled example, effectively achieving one-shot "learning".

7.2.3 Model-based / Blackbox methods

This final class of method is that of blackbox models, in which, the adaptation (or inner learning) is embedded in the forward pass of the model. That is, the model is trained to adapt by itself, effectively learning how to learn. At test-time, the model is presented with a labelled dataset (or equivalently, task) and a non-labelled validation sample belonging to that particular dataset (or task). With these methods, the model is trained to embed the current presented dataset (task) into its activations, such that the prediction of the validation sample can be made accordingly. Typical neural architectures used with these methods include RNNs, CNNs and hypernetworks (Ha et al., 2016a). The next Chapter precisely aims at introducing such an architecture, based on neuromodulation and specifically designed for adaptation purposes. This method will be compared to using a standard RNN in the learning to reinforcement learn (RL2) framework, as proposed by J. X. Wang et al. (2016).

Learning to reinforcement learn In the RL2 framework, the authors propose to train an RNN to solve the meta-RL setting presented in Section 7.1. The authors showed that the RNNs trained were able to come up with their own "inner" RL procedure. That is, after training, the RNN agents embedded an RL procedure in their dynamics such that, when presented a new task, they were able to adapt to it with very few samples (this adaptation can effectively be associated to "inner" reinforcement learning). This embedded procedure benefits from the fact that it is configured to highly exploit the training domain and as such, is highly biased towards solving tasks belonging to that particular domain. Over a more standard RL procedure, this allows the inner reinforcement learning procedure (i.e. adaptation) to be much more efficient on similar tasks, but also makes it much harder to extrapolate that inner procedure if tasks are too different than those of the training domain. This is the case for other meta-learning methods, but holds especially true for this class of methods. In fact, blackbox methods are known to be simpler to optimise than optimisation methods (as they do not require second order gradients) but are also usually less able to generalise to out of distribution tasks over optimisation methods.

PEARL In the RL2 framework, the context is implicitly represented by a recurrent network, through its hidden state. More recently, it was shown that representing this context through a latent variable allows to drastically increase sample efficiency in meta-RL. Rakelly et al. (2019) introduced an algorithm called probabilistic embeddings for actor-critic RL (PEARL) in which the past observations obtained from a given environment are compressed by a kind conditional neural process (CNP, proposed by Garnelo et al. (2018) [1]) into a latent variable, encoding the said environment as a context vector. The authors show that by adding this context to the agent's inputs, one can leverage off-policy algorithms (and thus their sample efficiency) even in a meta-RL setting. However, this approach suffers a limitation due to the aggregation method of the observed samples. Indeed, even though CNPs allow for a very computationally efficient and easily trainable framework, they suffer from huge inertia, which makes

1. As a brief introduction, similarly to RNNs, conditional neural process can handle arbitrary long sequences of inputs. To achieve this, each input is encoded through the same learned function. Afterwards, all such values are averaged over the input sequence, leading to a final representation of the input sequence. One will note that, as opposed to RNNs, order of inputs is not taken into account due to the commutativity property of the average operator.

them ineffective to adapt quickly in case of a rapid context switch. This could make the method proposed in PEARL inefficient in the context of meta-RL if, at test time, there is no way to know when a context switch occurs. Although this is not a problem with the meta-RL formalisation proposed here (as tasks remain constant throughout an episode), one could easily remedy this by using an RNN, as in RL2, to encode this latent variable. This problem of changing contexts would be harder to tackle with optimisation methods for example.

There exists other architectures and algorithms which fall in this category of *model-based / black box* methods. For example, hypernetworks (Ha et al., 2016a) have also been used in this context. Again, the reader is referred to (Hospedales et al., 2020) for a more complete review on the subject. In the next Chapter, the focus is made on the RL2 framework. Indeed, the goal is to highlight the interest of using a neuromodulated architecture over a standard RNN in such a context, thus the standard RL2 framework seemed well-suited for this study. As will be discussed, it is worth noting that the proposed architecture could well be extended to the PEARL algorithm, which would certainly be a worthwhile and very interesting study in itself.

Summary

This Chapter formalised the meta-RL setting that will be tackled in the remaining of this manuscript. It was also shown that this setting could be linked to a more standard partially observable RL setting. Afterwards, the most common meta-learning methods, as well as their advantages and drawbacks, were introduced and examples were given for each class of such methods. At this point, the reader is thus expected to well understand the meta-learning setting and more precisely, the meta-reinforcement learning as well as the challenges to solve such problems. Despite having only provided a brief overview of different existing methods, the reader should also be aware of the main concepts that are usually used in meta-learning algorithms. The RL2 framework will be more deeply detailed in next Chapter, but for further details on other methods, the reader is once again referred to (Hospedales et al., 2020).

Chapter 8

Neuromodulation of artificial neural networks

The core of this Chapter has been copied and slightly adapted from Vecoven et al. (2020).

In biological nervous systems, adaptation capabilities have long been linked to neuromodulation, a biological mechanism that acts in concert with synaptic plasticity to tune neural network functional properties. It has been shown as being critical to the adaptive control of continuous behaviours, such as in motor control, among others (Marder et al., 1996). In this Chapter, a new neural architecture is introduced, it is specifically designed for DNNs and inspired from cellular neuromodulation and is called NMN, standing for "Neuro-Modulated Network".

At its core, the NMN architecture comprises two neural networks: a main network and a neuromodulatory network. The main network is a feed-forward DNN composed of neurons equipped with a parametric activation function whose parameters are the targets of neuromodulation. It allows the main network to be adapted to new unforeseen problems. The neuromodulatory network, on the other hand, dynamically controls the neuronal properties of the main network via the parameters of its activation functions. Both networks have different inputs: the neuromodulatory network processes feedback and contextual data whereas the main network is in charge of processing other inputs.

Originally introduced in 2019, at the time of publication, the proposed architecture could be related to previous works on different aspects. In (Miconi et al., 2018), the authors took inspiration from Hebbian plasticity to build networks with plastic weights, allowing them to tune their weights dynamically. Miconi et al. (2020) extended their work by learning a neuromodulatory signal that dictates which and when connections should be plastic. The neuromodulatory architecture is also closely related to hypernetworks (Ha et al., 2016b), in which a network's weights are computed through another network. It is also worth noting that other works focused on learning fixed activation functions, such as those of Agostinelli et al. (2014); Lin et al. (2013).

Finally, it must be highlighted that concurrently and after the proposed NMN architecture, there has also been an increasing interest towards the introduction of neuromodulatory principles in ANNs. As such, the next section aims at pointing to some more recent works on neuromodulation in ANNs.

8.1 Neuromodulation in ANNs

Despite this growing interest for neuromodulation, the topic is still quite new. Therefore, research is still ongoing and only a handful of works have already looked at using neuromodulation mechanisms for adaptation in artificial neural networks. Some of the most relevant ones are introduced hereunder. It should be stressed that the list below is certainly non exhaustive and that the field is still quickly growing. As such, its goal is solely to point to the most relevant recent advances with respect to adaptation using neuromodulation, providing a bit more context on the current state of the field.

First, the work of Geadah et al. (2020) takes interest in the modulation of activation functions. More precisely, the authors introduced a new family of activation functions for which gain and saturation are parametric and can be learned. This modulation of activation functions is reminiscent of that used in NMN, although more expressive. Furthermore, in this work, they highlight that modulation of those parameters alone can help the network to adapt to new tasks, hinting at the potential interest of such mechanisms for transfer learning.

Second, Tsuda et al. (2021) have used neuromodulators in such a way to scale the weights of RNNs. That is, they input neuromodulatory signals which scale all the weights belonging to different neurons subpopulation of the network. The authors show that this mechanism alone is sufficient to tackle different tasks, whereas one could have expected that simply scaling the weights would lead to more linear behavioural changes.

Third, tackling different meta-learning tasks, Beaulieu et al. (2020) proposed “A Neuromodulated Meta-Learning Algorithm” (ANML). Despite not using the same training procedure, this method and the NMN are similar in that they both introduce a neuromodulatory network to gate the forward pass of another network. ANML was introduced a bit after NMN and the method proved to give state-of-the-art performances in continual learning tasks, exhibiting great resistance to catastrophic forgetting.

Finally, Wilson et al. (2018) used an artificial gene regulatory network as a neuromodulatory agent. This agent is trained and tasked to modify the optimiser parameters (such as the momentum parameters of ADAM) of each layer, at each step of training. The authors showed that this technique could help reduce overfitting, and demonstrated that the location-dependent and time-specific qualities of neuromodulation are important for deep learning, as they are in human brains.

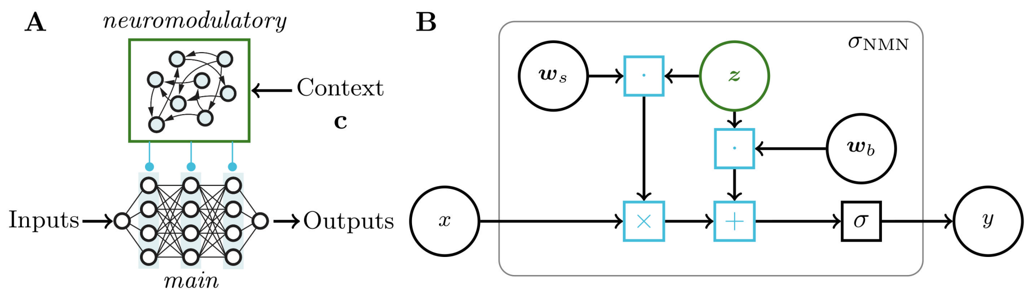


Figure 8.1: Sketch of the NMN architecture. **A.** The NMN is composed of the interaction of a *neuromodulatory* neural network that processes some context signal (top) and a *main* neural network that shapes some input-output function (bottom). **B.** Computation graph of the NMN activation functions σ_{NMN}, where $\mathbf{w}_s$ and $\mathbf{w}_b$ are parameters controlling the scale factor and the offset of the activation function σ, respectively. $\mathbf{z}$ is a context-dependent variable computed by the neuromodulatory network.

8.2 NMN architecture

The NMN architecture revolves around the neuromodulatory interaction between the neuromodulatory and main networks. Biological cellular neuromodulation (Drion, O'Leary, et al., 2015) is mimicked in a DNN by assigning the neuromodulatory network the task to tune the slope and bias of the main network activation functions.

Let $\sigma(x) : \mathbb{R} \to \mathbb{R}$ denote any activation function and its neuromodulatory capable version $\sigma_{\text{NMN}}(x, \mathbf{z}; \mathbf{w}_s, \mathbf{w}_b) = \sigma\left(\mathbf{z}^T(x\mathbf{w}_s + \mathbf{w}_b)\right)$ where $\mathbf{z} \in \mathbb{R}^k$ is a neuromodulatory signal and $\mathbf{w}_s, \mathbf{w}_b \in \mathbb{R}^k$ are two parameter vectors of the activation function, respectively governing a scale factor and an offset. In this work, it is proposed to replace all the main network neuron activation functions with their neuromodulatory capable counterparts. The neuromodulatory signal $\mathbf{z}$, where size k is a free parameter, is shared for all these neurons and computed by the neuromodulatory network as $\mathbf{z} = f(\mathbf{c})$, where $\mathbf{c}$ is a vector representing contextual and feedback inputs. The function f can be any DNN taking as input such vector $\mathbf{c}$. For instance, $\mathbf{c}$ may have a dynamic size (e.g. more information about the current task becomes available as time passes), in which case f could be parameterised as a recurrent neural network (RNN) or a conditional neural process (Garnelo et al., 2018), enabling refinement of the neuromodulatory signal as more data becomes available. The complete NMN architecture and the change made to the activation functions are depicted in Figure 8.1.

Notably, the number of newly introduced parameters scales linearly with the number of neurons in the main network whereas it would scale linearly with the number of connections between neurons if the neuromodulatory network was affecting connection weights, as seen for instance in the context of hypernetworks (Ha et al., 2016b). Therefore this approach can be extended more easily to very large networks.

8.3 Experiments

Experiments are carried using the meta-RL setting described in Chapter 7. This Section first details the training procedure used for the meta-RL agent, then the precise benchmarks used to test our agent capabilities.

8.3.1 Training

In J. X. Wang et al. (2016), the authors tackle this meta-RL framework by using an advantage actor-critic (A2C) algorithm. This algorithm revolves around two distinct parametric functions: the actor and the critic. The actor represents the policy used to interact with the MDPs, while the critic is a function that rates the performance of the agent policy. All actor-critic algorithms follow an iterative procedure that consists of the three following steps.

1. Use the policy to interact with the environment and gather data.
2. Update the actor parameters using the critic ratings.
3. Update the critic parameters to better approximate a value function.

In J. X. Wang et al. (2016), the authors chose to model the actor and the critic with RNNs, taking $\mathbf{h}_t$ as the input. In this work, we propose comparing the NMN architecture to standard RNN by modelling both the actor and the critic with NMN. To this end, the feedback and contextual inputs $\mathbf{c}$ (i.e. the neuromodulatory network inputs) are defined as $\mathbf{h}_t \setminus \mathbf{x}_t$ while the main network input is defined as $\mathbf{x}_t$. Note that $\mathbf{h}_t$ grows (as sequential data) as the agent interacts with the MDP $\mathbb{M}$, motivating the usage of a RNN as neuromodulatory network. A graphical comparison between both architectures is shown on Figure 8.2.

To be as similar as possible to the neuronal model proposed by Drion, O'Leary, & Marder (2015), the main network is a fully-connected neural network built using saturated rectified linear unit (sReLU) activation functions $\sigma(x) = \min(1, \max(-1, x))$, except for the final layer (also neuromodulated), for which $\sigma(x) = x$. In Section 8.4, we also report results obtained with sigmoidal activation functions which are often appreciably inferior to those obtained with sReLUs, further encouraging their use.

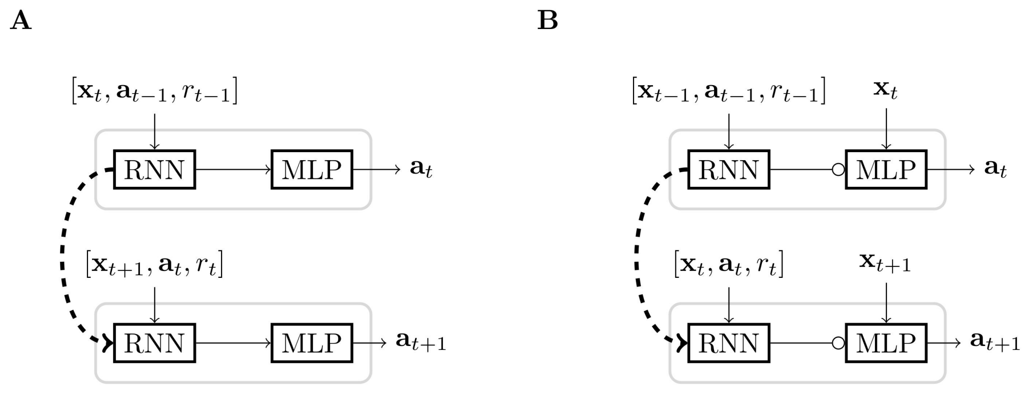

A
B
$[\mathbf{x}_t, \mathbf{a}_{t-1}, r_{t-1}]$
RNN
MLP
$\mathbf{a}_t$
$[\mathbf{x}_{t+1}, \mathbf{a}_t, r_t]$
RNN
MLP
$\mathbf{a}_{t+1}$
$[\mathbf{x}_{t-1}, \mathbf{a}_{t-1}, r_{t-1}]$
$\mathbf{x}_t$
RNN
MLP
$\mathbf{a}_t$
$[\mathbf{x}_t, \mathbf{a}_t, r_t]$
$\mathbf{x}_{t+1}$
RNN
MLP
$\mathbf{a}_{t+1}$

Figure 8.2: Sketch of a standard recurrent network (**A**) and of an NMN (**B**) in a meta-RL framework. $\rightarrow$ represent standard connections, $\multimap$ represent a neuromodulatory connection, $\dashrightarrow$ represent temporal connections and MLP stands for Multi-Layer Perceptron (standard feed-forward network).

The models are built such that both standard RNN and NMN architectures have the same number of recurrent layers/units and a relative difference between the numbers of parameters that is lower than 2%. Both models are trained using an A2C algorithm with generalized advantage estimation (Schulman, Moritz, et al., 2015) and proximal policy updates (Schulman et al., 2017). Finally, no parameter is shared between the actor and the critic. This choice is motivated by noting that the neuromodulatory signal might need to be different for the actor and the critic. For completeness and reproducibility, a formal description of the algorithms used is provided as supplementary material in Appendix C. This Appendix also provides the exact neural architectures used for each benchmark.

8.3.2 Benchmarks description

Experiments are carried on three custom benchmarks: a simple toy problem and two navigation problems with sparse rewards. These benchmarks were built to evaluate our architecture in environments with continuous action spaces. For conciseness, a mathematical definition is only provided for the first benchmark. The two other benchmarks are briefly textually depicted and further details are available in Appendix C. Figures 8.3, 8.4 and 8.5 are a graphical representation of each of the benchmarks.

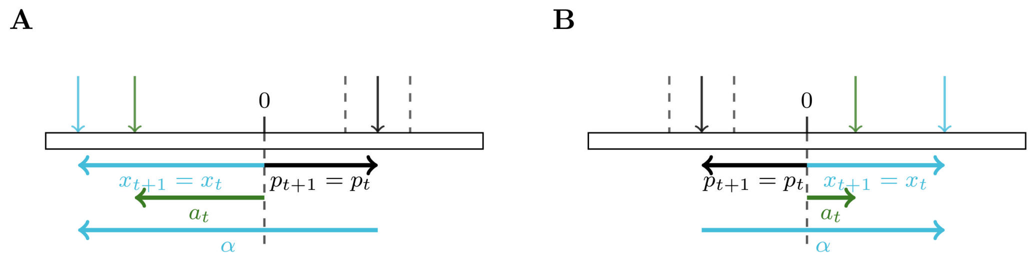


Figure 8.3: Sketch of a time-step interaction between an agent and two different tasks $\mathbb{M}$ (**A** and **B**) sampled in η for the first benchmark. Each task is defined by the bias α on the target's position p_t observed by the agent. x_t is the observation made by the agent at time-step t and a_t its action. For these examples, a_t falls outside the target area (the zone delimited by the dashed lines), and thus the reward r_t received by the agent is equal to $-|a_t - p_t|$ and $p_{t+1} = p_t$. If the agent had taken an action near the target, then it would have received a reward equal to 10 and the position of the target would have been re-sampled uniformly in $[-5 - \alpha, 5 - \alpha]$.

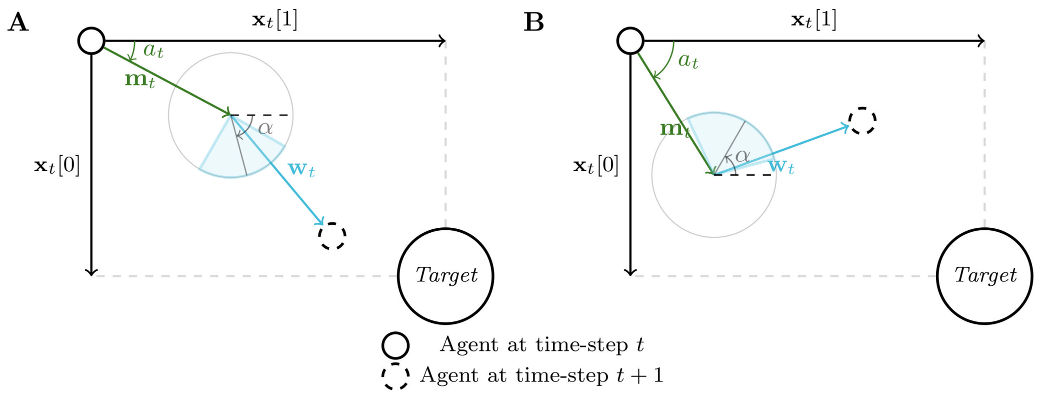


Figure 8.4: Sketch of a time-step interaction between an agent and two different tasks $\mathbb{M}$ (**A** and **B**) sampled in η for the second benchmark. Each task is defined by the main direction α of a wind cone from which a perturbation vector $\mathbf{w}_t$ is sampled at each time-step. This perturbation vector is then applied to the movement m_t of the agent, whose direction is given by the action a_t. If the agent reaches the target, it receives a reward of 100, otherwise a reward of -2.

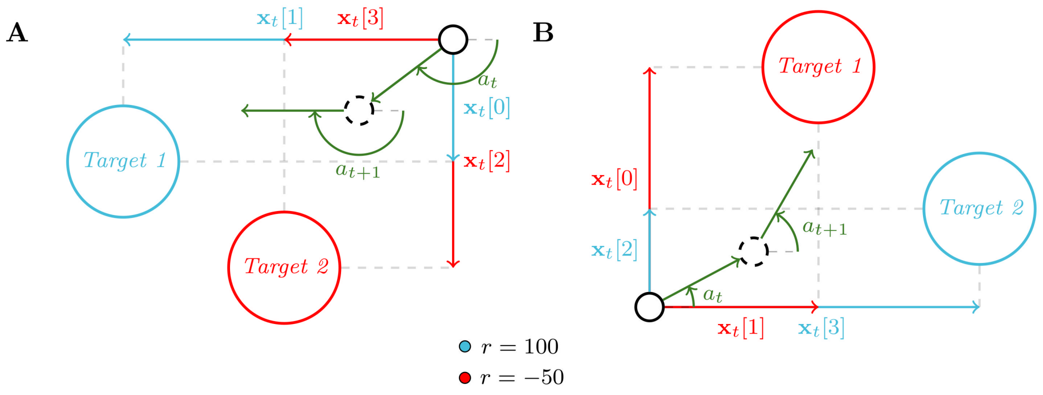


Figure 8.5: Sketch of a time-step interaction between an agent for the two different tasks $\mathbb{M}$ (**A** and **B**) sampled in η for the third benchmark. Each task is defined by the attribution of a positive reward to one of the two targets (in blue) and a negative reward to the other (in red). At each time-step the agent outputs an action a_t which drives the direction of its next move. If the agent reaches a target, it receives the corresponding reward.

Benchmark 1. The first benchmark (made of a 1-D state space and action space) is defined through a random variable α, informative enough to distinguish all different MDPs in η. With this definition, α represents the current task and drawing α at the beginning of each episode amounts to sampling a new task in η. At each time-step, the agent observes a biased version $x_t = p_t + \alpha$ of the exact position of a target p_t belonging to the interval $[-5-\alpha, 5-\alpha]$, with $\alpha \sim \mathbb{U}[-10, 10]$. The agent outputs an action $a_t \in [-20, 20]$ and receives a reward r_t which is equal to 10 if $|a_t - p_t| < 1$ and $-|a_t - p_t|$ otherwise. In case of positive reward, p_{t+1} is re-sampled uniformly in its domain, else $p_{t+1} = p_t$. This benchmark is represented on Figure 8.3.

Benchmark 2. The second benchmark consists of navigating towards a target in a 2-D space with noisy movements. Similarly to the first benchmark, all different MDPs in η can be distinguished through a three-dimensional random vector of variables $\boldsymbol{\alpha}$. The target is placed at $(\boldsymbol{\alpha}[1], \boldsymbol{\alpha}[2])$ in the 2-D space. At each time-step, the agent observes its relative position to the target and outputs the direction of a move vector $\mathbf{m}_t$. A perturbation vector $\mathbf{w}_t$ is then sampled uniformly in a cone, whose main direction $\boldsymbol{\alpha}[3] \sim \mathbb{U}[-\pi, \pi[$, together with the target's position, define the current task in η. Finally the agent is moved following $\mathbf{m}_t + \mathbf{w}_t$ and receives a reward ($r_t = -0.2$). If the agent reaches the target, it instead receives a high reward ($r_t = 100$) and is moved to a position sampled uniformly in the 2-D space. This benchmark is represented on Fig 8.4

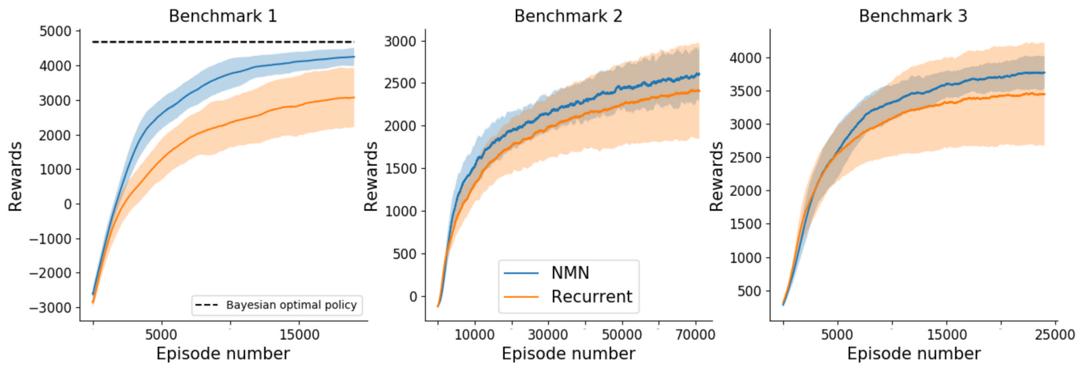


Figure 8.6: Mean ($\pm$ std in shaded) sum of rewards obtained over 15 training runs with different random seeds with respect to the episode number. Results of benchmark 1,2 and 3 are displayed from left to right. The plots are smoothed thanks to a running mean over 1000 episodes.

Benchmark 3. The third benchmark also involves navigating in a 2-D space, but which contains two targets. As for the two previous benchmarks, all different MDPs in η are distinguished through a five-dimensional random vector of variables $\boldsymbol{\alpha}$. The targets are placed at positions $(\boldsymbol{\alpha}[1], \boldsymbol{\alpha}[2])$ and $(\boldsymbol{\alpha}[3], \boldsymbol{\alpha}[4])$. At each time-step, the agent observes its relative position to the two targets and is moved along a direction given by its action. One target, defined by the task in η through $\boldsymbol{\alpha}[5]$, is attributed a positive reward (100) and the other a negative reward (-50). In other words, $\boldsymbol{\alpha}[5]$ is a Bernoulli variable that determines which target is attributed the positive reward and which is attributed the negative one. As for benchmark 2, once the agent reaches a target, it receives the corresponding reward and is moved to a position sampled uniformly in the 2-D space. This benchmark is represented on Figure 8.5.

8.4 Results

Learning. From a learning perspective, a comparison of the sum of rewards obtained per episode by NMNs and RNNs on the three benchmarks is shown in Figure 8.6. Results show that, on average, NMNs learn faster (with respect to the number of episodes) and converge towards better policies than RNNs (i.e., higher rewards for the last episodes). It is worth mentioning that, NMNs show very stable results, with small variances over different random seeds, as opposed to RNNs. To put the performance of the NMN in perspective, we note that an optimal Bayesian policy would achieve an expected sum of rewards of 4679 on benchmark 1 (see Appendix C for a formal proof) whereas NMNs reach, after 20000 episodes, an expected sum of rewards of 4534. For this simple benchmark, NMNs manage to learn near-optimal Bayesian policies.

Adaptation. From an adaptation perspective, Figure 8.7 shows the temporal evolution of the neuromodulatory signal $\mathbf{z}$ (part **A**), of the scale factor (for each neuron of a hidden layer, part **B**) and of the rewards (part **C**) obtained with respect to α for 1000 episodes played on benchmark 1. For small values of t, the agent has little information on the current task, leading to a non-optimal behaviour (as it can be seen from the low rewards). Of greatest interest, the signal $\mathbf{z}$ for the first time-steps exhibits little dependence on α, highlighting the agent uncertainty on the current task and translating to noisy scale factors. Said otherwise, for small t, the agent learned to play a (nearly) task-independent strategy. As time passes, the agent gathers further information about the current task and approaches a near-optimal policy. This is reflected in the convergence of $\mathbf{z}$ (and thus scale factors) with a clear dependency on α and also in wider-spread values of $\mathbf{z}$. For a large value of t, $\mathbf{z}$ holding constant between time-steps shows that the neuromodulatory signal is almost state-independent and serves only for adaptation. It is noted that the value of $\mathbf{z}$ in each of its dimensions varies continuously with α, meaning that for two similar tasks, the signal will converge towards similar values. Finally, it is interesting to look at the neurons scale factor variation with respect to α (**B**). Indeed, for some neurons, one can see that the scale factors vary between negative and positive values, effectively inverting the slope of the activation function. Furthermore, it is interesting to see that some neurons are inactive (scale factor almost equal to 0, leading to a constant activation function) for some values of α.

For benchmark 2, it is first noted that $\mathbf{z}$ seems to code exclusively for $\boldsymbol{\alpha}[3]$. Indeed, $\mathbf{z}$ converges slowly with time with respect to $\boldsymbol{\alpha}[3]$, whatever the value of $\boldsymbol{\alpha}[1]$ and $\boldsymbol{\alpha}[2]$ (Figure 8.8). This, could potentially be explained by the fact that one does not need the values of $\boldsymbol{\alpha}[1]$ and $\boldsymbol{\alpha}[2]$ to compute an optimal move. The graphs on Figure 8.8 are projected on the dimension $\boldsymbol{\alpha}[3]$, allowing the same analysis as for benchmark 1.

The results obtained for benchmark 2 (Figure 8.8) show similar characteristics. Indeed, despite the agent receiving only noisy information on $\boldsymbol{\alpha}[3]$ at each time-step (as perturbation vectors are sampled uniformly in a cone centered on $\boldsymbol{\alpha}[3]$), $\mathbf{z}$ quasi-converges slowly with time (part **A**). The value of $\mathbf{z}$ in each of its dimensions also varies continuously with $\boldsymbol{\alpha}[3]$ (as for the first benchmark) resulting also in continuous scale factors variations. This is clearly highlighted at time-step 100 on Figure 8.8 where the scale factors of some neurons appear highly asymmetric, but with smooth variations with respect to $\boldsymbol{\alpha}[3]$. Finally, it is highlighted that for this benchmark, the agent continues to adapt even when it is already performing well. Indeed, one can see that after 40 time-steps the agent is already achieving good results (part **C**), even though $\mathbf{z}$ has not yet converged (part **A**), which is due to the stochasticity of the environment. Indeed, the agent only

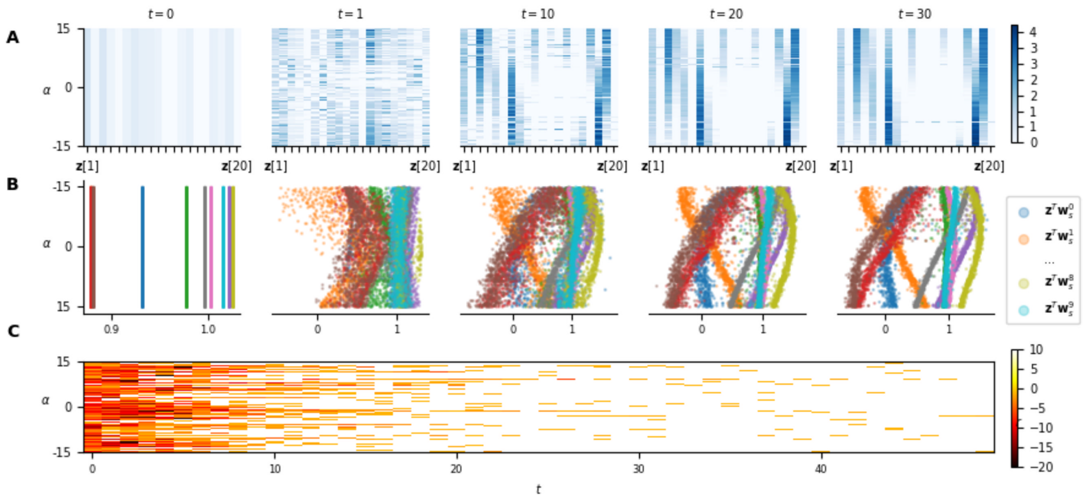


Figure 8.7: Adaptation capabilities of the NMN architecture on benchmark 1. **A.** Temporal evolution of the neuromodulatory signal **z** with respect to α, gathered on 1000 different episodes. Note that the neuromodulatory signals go from uniform distributions over all possible α values (i.e., the different contexts) to non-uniform and adapted (w.r.t. α) distributions along with an increase in the rewards. **B.** The value of the scale factors with respect to α for each neuron of a hidden layer in the main network. **C.** Rewards obtained at each time-step by the agent during those episodes. Note that light colours represent high rewards and correspond to adapated neuromodulatory signals.

receives noised information on α and thus after 40 time-steps it has gathered sufficient information to act well on the environment, but insufficient information to deduce a near-exact value of $\boldsymbol{\alpha}[3]$. This shows that the agent can perform well, even while it is still gathering relevant information on the current task.

It is harder to interpret the neuromodulatory signal for benchmark 3. In fact, for that benchmark, it is shown that the signal seems to code not only for the task in η but also for the state of the agent in some sense. As $\boldsymbol{\alpha}$ is five-dimensional, it would be very difficult to look at its impact on **z** as a whole. Rather, one can fix the position of the two references in the 2-D space and look at the behaviour of **z** with respect to $\boldsymbol{\alpha}[5]$. In Figure 8.9 adaptation is clearly visible in the rewards obtained by the agent (part **C**) with very few negative rewards after 30 time-steps. It is noted that for later time-steps, **z** tends to partially converge (**A**) and :

- some dimensions of **z** are constant with respect to $\boldsymbol{\alpha}[5]$, indicating that they might be coding for features related to $\boldsymbol{\alpha}[1, 2, 3, 4]$.
- Some other dimensions are well correlated to $\boldsymbol{\alpha}[5]$, for which similar observations than for the two other benchmarks can be made. For example, one can see that some neurons have a very different scale factors for the two possible different values of $\boldsymbol{\alpha}[5]$ (**B**).

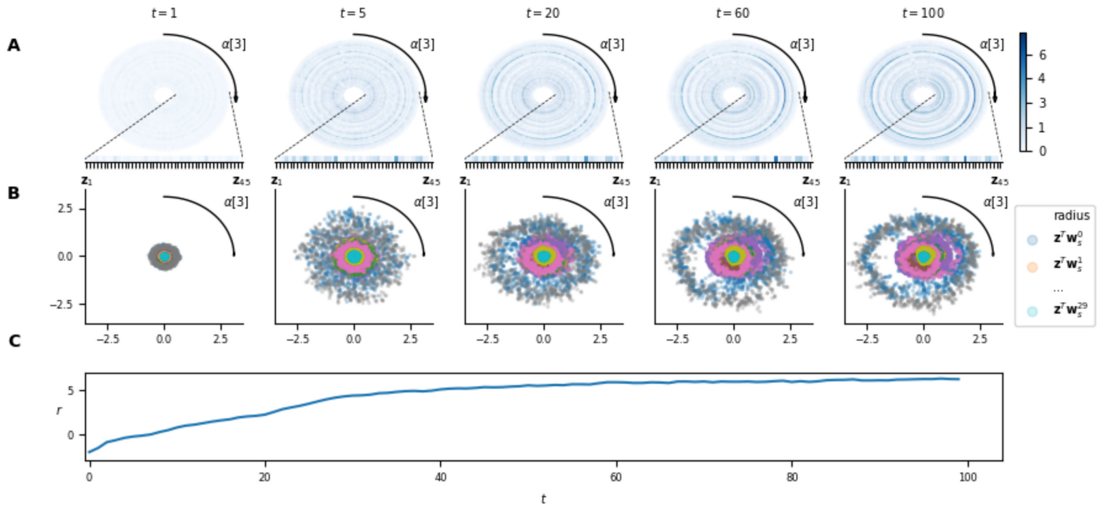


Figure 8.8: Adaptation capabilities of the NMN architecture on benchmark 2. **A.** Temporal evolution of the neuromodulatory signal $\mathbf{z}$ with respect to $\boldsymbol{\alpha}[3]$, gathered on 1000 different episodes. As $\boldsymbol{\alpha}[3]$ is an angle, the plot is projected in polar coordinates for a better interpretability of the results. Each dimension of $\mathbf{z}$ is corresponds to a different radius. **B.** The value of the scale factors with respect to $\boldsymbol{\alpha}[3]$ for each neuron of a hidden layer in the main network. Again, the plot is projected in polar coordinates. For a given $\boldsymbol{\alpha}[3]$, the values of the neurons' scale factor are given thanks to the radius. **c.** Average reward obtained at each time-step by the agent during those episodes. Note that after an average of 40 time-steps, the agent is already achieving decent performances even though $\mathbf{z}$ has not yet converged.

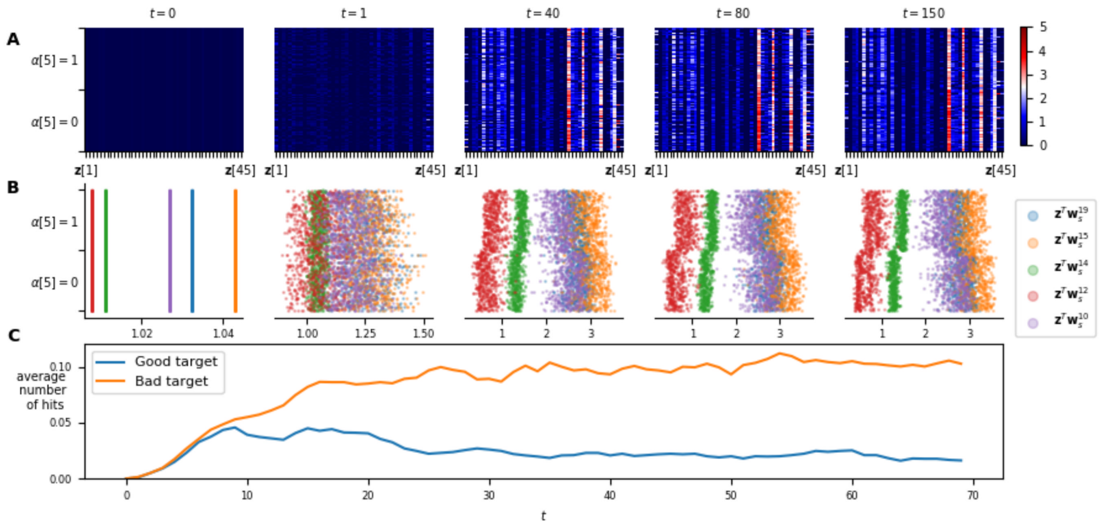


Figure 8.9: Adaptation capabilities of the NMN architecture on benchmark 3. **A.** Temporal evolution of the neuromodulatory signal $\mathbf{z}$ with respect to $\boldsymbol{\alpha}[5]$, gathered on 1000 different episodes. Note that the neuromodulatory signals go from uniform distributions over all possible alpha values (i.e., the different contexts) to non-uniform and adapted (w.r.t. alpha) distributions along with an increase of the rewards. **B.** The value of the scale factors with respect to $\boldsymbol{\alpha}[5]$ for the 5 neurons of a hidden layer in the main network, for which the scale factor is the most correlated to $\boldsymbol{\alpha}[5]$. **C.** Average number of good and bad target hits at each time-step during those episodes. On average, after 15 time-steps, the agent starts navigating towards the correct target while avoiding the wrong one.

- The remaining dimensions do not converge at all, implying that these are not related to $\boldsymbol{\alpha}$, but rather to the state of the agent.

These results suggest that in this case, the neuromodulation network is used to code more complex information than simply that required to differentiate tasks, making $\mathbf{z}$ harder to interpret. Despite $\mathbf{z}$ not converging on some of its dimensions, it should be stressed that freezing $\mathbf{z}$ after adaptation will not strongly degrade the agent's performance. That is, the features coded in $\mathbf{z}$ that do not depend on $\boldsymbol{\alpha}$ are not critical to the performance of the agent. To illustrate this, we will analyse the behaviour of the agent within an episode when freezing and unfreezing the neuromodulation signal and when changing task. This behaviour is shown on Figure 8.10, for which:

(a) Shows the behaviour of the agent when $\mathbf{z}$ is locked to its initial value. This plot thus shows the initial "exploration" strategy used by the agent; that is, the strategy played by the agent when it has not gathered any information on the current task.

(b) Shows the behaviour of the agent after unlocking $\mathbf{z}$, that is when the agent is able to adapt freely to the current task by updating $\mathbf{z}$ at each time-step.

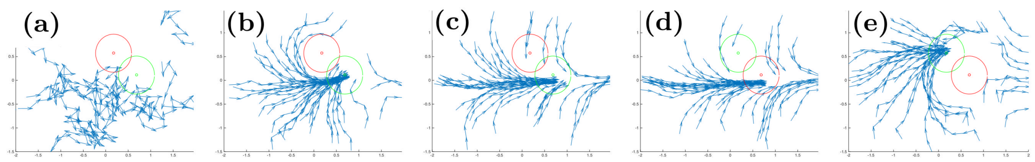


Figure 8.10: Analysis of the agent's behaviour when freezing and unfreezing the neuromodulation signal and when changing task within an episode. The green reference is attributed a reward of 100 while the red one is attributed a reward of −50. Each blue arrow represents the movement of the agent for a given time-step. **(a)** Shows the behaviour with $\mathbf{z}$ fixed at its initial value. In **(b)** we unlock $\mathbf{z}$. Then, in **(c)** we lock $\mathbf{z}$ with its current value. Finally in **(d)** we switch the references before unlocking $\mathbf{z}$ once again in **(e)**.

(c) Shows the behaviour of the agent when locking $\mathbf{z}$ at a random time-step after adaptation. $\mathbf{z}$ is thus fixed at a value which fits well the current task. As one can see, the agent continues to navigate towards the correct target. The performance is however a slightly degraded as the agent seems to lose some capacity to avoid the wrong target. This further suggests that, in this benchmark (as opposed to the two others), the neuromodulation signal does not only code for the current task but also for the current state, in some sense, that is hard to interpret.

(d) Shows the same behaviour as in **(c)** as $\mathbf{z}$ is still locked to the same value, but the references are now switched. As there is no adaptation without updating $\mathbf{z}$; the agent is now always moving towards to wrong target.

(e) Shows the behaviour of the agent when unlocking $\mathbf{z}$ once again. As one can see, the agent is now able to adapt correctly by updating $\mathbf{z}$ at each time-step, and thus it navigates towards the correct target once again.

Robustness study. Even though results are quite promising for the NMN, it is interesting to see how it holds up with another type of activation function as well as analysing its robustness to different main networks' architectures.

Sigmoid activation functions. Figure 8.11 shows the comparison between having sigmoids as the main network's activation function instead of sReLUs. As one can see, sigmoid activation functions lead to worse or equivalent results to sReLUs, be it for RNNs or NMNs. In particular, the NMN architecture seems more robust to the change of activation function as opposed to RNNs, as the difference between sReLUS and sigmoids is often far inferior for NMNs than RNNs (especially for benchmark 2).

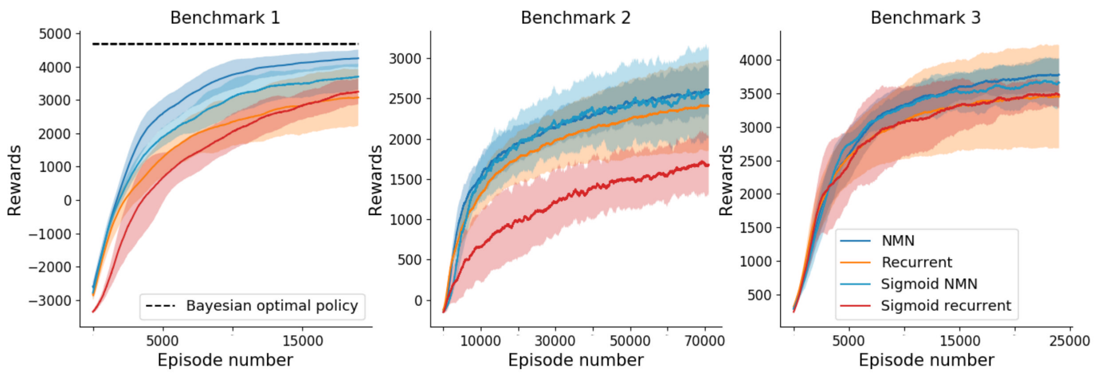


Figure 8.11: Mean (± std in shaded) sum of rewards obtained over 15 training runs with different random seeds with respect to the episode number. Results of benchmark 1,2 and 3 are displayed from left to right. The plots are smoothed thanks to a running mean over 1000 episodes.

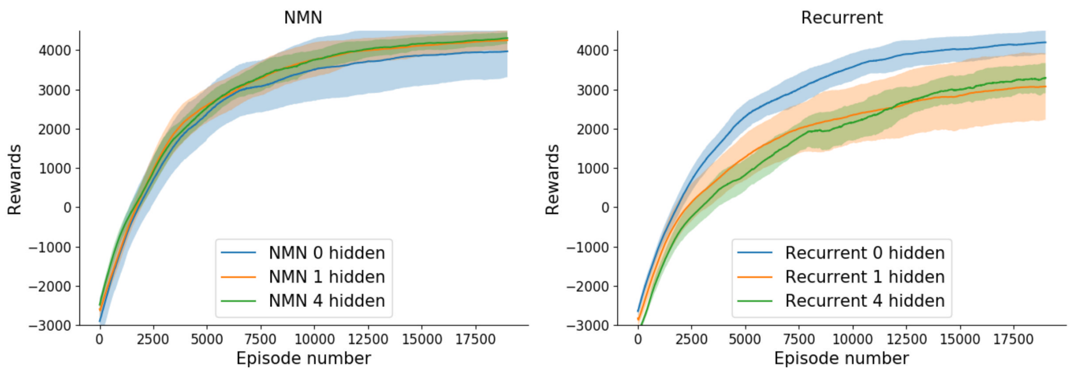


Figure 8.12: Mean (± std in shaded) sum of rewards obtained on benchmark 1 over 15 training runs with different random seeds with respect to the episode number. The plots are smoothed thanks to a running mean over 1000 episodes.

Architecture impact. Figure 8.12 shows the learning curve, on benchmark 1, for different main network architectures (0, 1 and 4 hidden layers in the main network respectively). As one can see, RNNs can, in fact, reach NMNs' performances for a given architecture (no hidden layer in this case), but seem relatively dependant on the architecture. On the contrary, NMNs seem surprisingly consistent with respect to the number of hidden layers composing the main network.

Summary

This Chapter studied a high-level view of a nervous system mechanism called cellular neuromodulation in order to improve the adaptive capabilities of artificial neural networks. After giving an overview of the actual state of the field, a specific neuromodulatory architecture is discussed. The results obtained for three meta-RL benchmark problems showed that this new architecture was able to perform better than classical RNN. The architecture discussed in this Chapter could be extended along several lines.

First and most importantly, this architecture could very easily be extended to a more probabilistic framework. Indeed, it was shown by Rakelly et al. (2019) that using a latent variable (encoded through a CNP) to capture the context was of great interest for sample efficiency in meta-RL. In their work, Rakelly et al. (2019) then use that variable, in addition to the current state, as inputs of a standard neural network (representing the policy). In such a context it would be immediate to rather use the context to neuromodulate the policy network, which might be very promising. Indeed, seeing the way $\mathbf{z}$ codes for the context in the experiments of Section 8.4 hints that it might be beneficial to let the network code for uncertainty in its context by rather using a latent probabilistic variable.

Second, it would make sense to explore other types of machine-learning problems where adaptation is required. Supervised meta-learning would be an interesting track to follow as, for example, it is easy to see how the architecture could be applied to few-shot learning. In such a framework, the context fed to the neuromodulatory network would be a set composed of a few samples and their associated ground-truth. It would be of great interest to compare the performance of this architecture to that of conditional neural processes or other few-shot learning algorithms. Furthermore, it would also be highly interesting to test other neuromodulatory architectures than standard RNNs.

Third, research work could also be carried out to further improve the NMN introduced here. For instance, one could introduce new types of parametric activation functions which are not linear (as was done by Geadah et al. (2020)), or even spiking neurons. This would amount to designing a brand-new parametric activation functions, the parameters of which could thus be more powerful than simple slope and bias. It would also be of interest to look at sharing activation function parameters per layer, especially in convolution layers, as this would essentially result in scaling the filters. One could also build a neuromodulatory signal per-layer rather than for the whole network, allowing for more complex forms of modulation (similarly to what is proposed by Tsuda et al. (2021)). Furthermore, it would be interesting to see if, with such a scheme, continuity in the neuromodulatory signal (with respect to the task) would be preserved.

Fourth, it would be a logical progression to tackle other benchmarks to see if the observations made here hold true. More generally, analysing the neuromodulatory signal to a greater depth (and its impact on activation functions) with respect to different more complex tasks would be worthwhile. An interesting point raised in this work is that, for some tasks, neurons have been shown to have a scaling factor of zero, making their activation constant with respect to the input. Generally, any neuron that has a constant output can be pruned if the corresponding offset is added to its connected neurons. This has two interesting implications. First, some neurons have a scale factor of zero for all of the tasks and thus, by using this information, one could prune the main network without losing performance. One could think of decomposition based methods for feature selection, which would attribute a null importance to such neurons. Second, neurons having a zero-scale factor for some tasks essentially leads to only a sub-network being used for the given task. It would be interesting to discover if very different sub-networks would emerge when an NMN is trained on tasks with fewer similarities than those used in this work.

Fifth, it is also worth noting that some similar neuromodulatory concepts could be used in different ways. For example, it would be interesting to try the MAML for which only neuromodulatory parameters would be modified at test-time. Finally, we should emphasize that even if the results obtained by the NMN architecture are good and also rather robust with respect to a large choice of parameters, further research is certainly still needed to better characterise the NMN performances.

PART IV

Conclusion

Chapter 9

Conclusion and future works

This Part concludes the thesis and tries to provide future works ideas. In particular, this thesis was articulated along two main parts:

- while Part II focused on the analysis of RNN dynamics for long-term memory,
- Part III looked at neuromodulatory principles in ANNs, and more specifically for adaptation purposes.

As such, the main findings for both parts will be respectively presented in Section 9.1 and 9.2. Concerning future works, taking inspiration from the works presented in Part II, some potential ideas revolving around RNNs will be presented in Section 9.1.

Finally, more global comments on this thesis will be made and in particular, it will be argued that Part II and Part III are more linked than it would first appear. Therefore, to end this thesis, Section 9.3 provides a discussion about the connections that exist between both parts, from which potential future works concerning neuromodulation in ANNs will be proposed, aiming to further close this gap.

9.1 Fixed points and long-term memory in RNNs

In the first Part of this thesis, particular attention was given to RNNs and the difficulties of training such networks. Notably, it was first discussed that problems can frequently arise when dealing with longer data sequences. It was also discussed that control theory could well be used to analyse the dynamics of trained RNNs, providing valuable insights on the inner workings of such networks. Following these discussions, a new recurrent bistable cell (called BRC, or nBRC for its neuromodulated version) was introduced. The proposed BRC was biologically inspired and built through control theory to promote bi-stability. Finally, Part II introduced a technique for warming up RNNs and improve their ability to learn long-term dependancies, while allowing for the usage of usual recurrent cells such as GRUs and LSTMs. The main findings of these works are presented hereunder in Subsection 9.1.1 before the description of potential related future works in Subsection 9.1.2.

9.1.1 Main findings

Thanks to multiple recent works, it is known that initial (before training) dynamics of RNNs are important (Tallec & Ollivier, 2018; Jing et al., 2019; Pascanu et al., 2013). Nevertheless, how one should initialise RNNs to obtain optimal training performance remains an open question. More precisely, parameters play different roles for each type of recurrent cell, and different benchmarks might benefit from different dynamics. Due to the non-linearity of such cells and to the complex nature of RNNs, it is extremely difficult to fully grasp the effect of different initialisations for a given cell, let alone to come up with generic well-working initialisation rules.

Furthermore, it was also recently shown that trained neural networks rely heavily on fixed points to compute their predictions (Sussillo & Barak, 2013; Ceni et al., 2020; Maheswaranathan et al., 2019).

Both works presented in Part II focused on the fixed points characteristics of RNNs at initialisation, aiming to improve training of RNNs on long-term memory benchmarks. In particular it was highlighted that, for such problems, a decrease in loss was highly correlated with an increase in the number of attractors possessed by the network. It was also shown that, with standard initialisation, RNNs built with usual recurrent cells (such as GRUs) are mono-stable. Throughout Chapter 5 and 6, it was shown that RNNs with multi-stability properties at initialisation were able to easily learn much longer time-dependencies. As such, one of the main findings of Part II lies in the importance of building RNNs such that they possess multiple attractors at initialisation. This can specifically be done by building new recurrent cells (as done with BRCs in Chapter 5) or by using a different initialisations for more usual cells. Chapter 6 focused on the latter and showed that it is possible to build a proxy measure which allows to maximise the number of attractors in an RNN, independently of the recurrent cells used.

9.1.2 Future works

Both bistable cells and warm up provide interesting avenues of research in their own rights and these have been respectively discussed in Chapter 5 and 6. One such important avenue is that it would be highly interesting to use the long-term memory characteristics of BRCs and warm-up to tackle partially observable reinforcement learning with sparse observations. In such setting, relevant information should be remembered for many time-steps by the agent. If the agent is modelled as an RNN and initialised to be multi-stable, the experiments of Part II suggest that it should be easier for it to learn how to remember this information. As such, this framework might be extremely relevant when it comes to further testing and using BRCs as well as warm-up.

More general directions, following similar lines of work than those of Chapter 5 and 6, will now be discussed.

In particular, the interest of linking non-linear control theory and RNNs was highlighted and it might be extremely beneficial to pursue such line of research further. Indeed, while this work focused solely on attractors, other dynamical properties of RNNs could surely be of interest. Amongst others, transient dynamics and saddle points are known to be highly relevant for RNNs' predictions as well. Studying these properties more thoroughly could lead to the development of new recurrent cells or to the further improvements of existing ones.

Finally, the variability amongst attractors (presented in Chapter 6) suggested a very interesting observation. Naturally, the deeper an RNN becomes (that is, the more recurrent layers are stacked), the slower its convergence in phase space might become (convergence of layer n requires convergence of layer $n-1$). As such, of particular interest would be a deeper analysis through control theory on the effect of stacking recurrent layers for RNN dynamics. This could be compared to the effect of widening such layers (i.e. increasing their number of neurons). This could also lead to a very interesting study on the effect of parallelising recurrent layers, like was done with the double-layer architecture presented in Chapter 6. Indeed, it was shown that it can be of interest to build multiple "sub layers" with different dynamical properties at the same depth of a network. This hints that it could also be of interest to study the dynamics of RNNs built with parallel layers, sequential layers and more specifically, a mix of both.

9.2 Neuromodulation in neural networks

Neuromodulation in artificial neural networks was the centrepiece of the second Part of this thesis. From a biological viewpoint, neuromodulation is often seen as an important factor for adaptative behaviours. It is thus logical that neuromodulatory principles in neural networks are often used for similar purposes. As such, a small overview of different ways to incorporate neuromodulation in ANNs for adaptation was presented in Chapter 8, from which it appeared that neuromodulation has often been associated to meta-learning and meta-reinforcement learning. This part focused on the latter setting and introduced a new neuromodulated architecture, called NMN. Subsection 9.2.1 will now highlight the main findings of the experiments carried with NMNs.

9.2.1 Main findings

In Chapter 8, it was shown that NMNs allow for much improved adaptive capabilities over more standard networks. In particular, they were shown to greatly reduce the variance of the training process on some meta-RL navigation benchmarks. This is of great interest as, due to the complexity of the setting, results of different runs on these benchmarks were shown to be vastly different when using more usual recurrent neural networks in the RL2 framework. Furthermore, for some of the benchmarks, NMNs were also shown to achieve better average rewards in addition to the reduced variance.

It is also interesting to note that, thanks to their structure, NMNs separate contextual inputs and “instantaneous” observations. As such, the neuromodulatory networks were shown to implicitly learn to encode the context of the task at hand through the contextual inputs. Interestingly, the encoding of the neuromodulatory network was continuous with respect to the different tasks. That is, similar tasks got mapped to similar neuromodulatory signals, and consequently, similar neuromodulatory parameters of the predictive network. Effectively, this highlighted some sort of continuity in the predictive network’s activation functions’ properties.

Finally, it was also discussed that for simple enough tasks, these parameters converge as the number of interactions with the environment increases. After convergence, these parameters could be frozen while maintaining good performance of the agent, highlighting that the neuromodulatory network is exclusively used for adaptation. All these findings lead to think that using architectures specifically designed for adaptation provide great benefits in such settings. The interests of designing such architectures are also supported by more recent works, such as that of Beaulieu et al. (2020).

For follow-up ideas immediately related to the neuromodulatory architecture proposed in Chapter 8, the reader is referred to the summary of that chapter. The next Section aims at linking both parts of this thesis by taking a more global view on neuromodulation in ANNs and consequently, providing more general avenues of research for using neuromodulatory principles in ANNs.

9.3 Closing the gap

Contextualising similarities Both parts of this thesis can be linked through Chapter 5 and 8, for which activation functions were tackled as the main subject of interest. Indeed,

- in Chapter 5, BRCs were proposed as a new recurrent cell. Consequently, the chapter focused on the study of recurrent update rules, which can be described as the composition of multiple sequential activation functions.

- On the other hand, Chapter 8 introduced parameterised versions of usual activation functions, such that they could be adapted dynamically.

To close the loop and make the link between both of these works, one should look at the BRCs' update rule as a single, parameterised activation function. As a reminder, the rule writes (Equation 5.2),

$$\mathbf{h}_t = \mathbf{c}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{c}_t) \odot \tanh(U\mathbf{x}_t + \mathbf{a}_t \odot \mathbf{h}_{t-1}) \quad .$$

As such, **c** and **a** can be considered as being the two parameters of the update (or equivalently, activation function). In Chapter 5, two methods for computing these parameters were discussed.

- The first, leading to the BRC, did so by keeping the cellular memory constraint,
- while the other, leading to the nBRC, did so by relaxing this constraint and introducing recurrent neuromodulation.

This neuromodulation is analogous to that discussed in Chapter 8 of this thesis, in that, both methods use the outputs of neurons to modulate activation functions of some other neurons. As such, it is interesting to note that usual gated cells can well be seen as particular cases of neuromodulation (although more difficult to interpret due to their non-cellular memory).

Future lines for research Now that this link has been highlighted, it is interesting to discuss the potential research topics that could result from it.

One obvious line would be to target the parameters of BRCs using a similar neuromodulatory architecture than that of the NMN and ANML. That is, a totally separate neuromodulatory network could be used to compute gates parameters, which could consequently lead to new types of gated recurrent architectures with more usual cells, for which gates' activations would be computed through a separate network.

Other types of recurrent activation functions could also be thought of. In particular, Geadah et al. (2020) introduced such kind of parameterised cells. In their work, parameters are kept static, and it would be very interesting to see if benefits would arise from making them depend on a dynamic neuromodulatory signal.

Furthermore, in the same work, the authors analyse the different behaviours of those recurrent cells with respect to their parameters. In a similar vein, it would be interesting to use some of the control tools discussed in Chapter 6 to understand the impact of neuromodulation on the dynamics of such networks. In particular, this kind of analysis could well be performed for networks where BRCs' parameters would be the target of a neuromodulatory network. For example, it would be expected that for different neuromodulatory signals, attractors of such networks would vary widely.

Finally, many other neuromodulatory schemes could be thought of, notably that proposed by Tsuda et al. (2021). In that work, the authors show that using a single neuromodulating signal value on different subnetworks within an ANN, depending on the task, can be more beneficial than using different values for the whole network. As such, it could be particularly interesting to study the effect of computing different neuromodulatory signals, each of which would impact different subparts of the predictive network. Going even one step further, one could even potentially link this to parallel recurrent architectures, for which each parallel sub-layer would be neuromodulated to have different dynamical properties.

A final word

Overall, this thesis tried to propose some novel methods for ANNs by taking inspiration from important biological concepts. In particular, very high level abstraction of neuromodulatory principles, as well as neuronal behaviours (such as bi-stability) were introduced and proved to exhibit different benefits.

There currently exists a rise in interest for the biological plausibility of artificial networks. Examples of this are the different works about neuromodulation cited in Part III, the quick increase in popularity of spiking neural networks and the multiple works aiming specifically at reconciling usual models with biological concepts (Bengio et al., 2015; Miconi, 2017; Bellec et al., 2019).

In this vein, using control theory to extract high-level abstractions of important biological concepts is highly relevant. Indeed this thesis showed that integrating these abstractions in deep networks could be pertinent. As such, further pursuing this approach for other topics might also be beneficial and is well in concordance with recent works in the literature.

Integration of these high-level abstractions in usual machine learning models and frameworks remains the most arduous task. Obviously, there are often many different interpretations to those abstractions and consequently, multiple ways to pursue such integration. This can be attested by the multiple different ways neuromodulation has recently been used in deep neural networks. While some of these methods share similarities, they all remain quite different however, despite still being associated to the same biological concept. In this thesis, and other works towards biological plausibility as well, recurrent neural networks have proven to be an extremely powerful and versatile tool to integrate these abstractions.

As such and as a final note, due to the well defined link between control theory, biological neurons and recurrent neural networks (which could, theoretically, allow for the discrete simulation of biological neurons), mixing all three topics could provide extremely new valuable lines of research and this should certainly continue to be explored.

Hopefully, this was successfully highlighted throughout this thesis.

PART V
Supplementary material

Appendix A

Supplementary material for Chapter 3

A.1 Modelling a multivariate Gaussian with ANN and computing KL divergence

**This explanation comes from Vecoven et al. (2020).*

We now detail how to compute the KL divergence used in the PPO loss described in Chapter 3. First, let us stress that we have chosen to work with multi-variate Gaussian policies for the actor. This choice is particularly well suited for MDPs with continuous action spaces. The approximation architecture of the actor will therefore not directly output an action, but the means and standard deviations of an m-dimensional multi-variate Gaussian from which the actor's policy can be defined in a straightforward way. For each dimension, we bound the multi-variate Gaussian to the support, $\mathcal{A}$, by playing the action that is clipped to the bounds of $\mathcal{A}$ whenever the multi-variate Gaussian is sampled outside of $\mathcal{A}$. In the remaining of this section, we will sometimes abusively use the terms "output of the actor at time t of episode i" to refer to the means vector $\mu_{i,t}^{\theta_k}$ and the standard deviations vector $\sigma_{i,t}^{\theta_k}$ that the actor uses to define its probabilistic policy at time-step t of episode i. Note that we have chosen to work with a diagonal covariance matrix for the multi-variate Gaussian distribution. Its diagonal elements correspond to those of the vector $\sigma_{i,t}^{\theta_k}$. We can then compute the KL divergence in each pair $[i, t]$ following the well-established formula:

$$
\begin{aligned}
&KL(\pi_{\theta_k}(\cdot|s_{i,t}), \pi_\theta(\cdot|s_{i,t})) = \\
&\qquad \frac{1}{2}\{tr(\Sigma_{\theta,i,t}^{-1}\Sigma_{\theta_k,i,t}) + (\mu_{i,t}^{\theta} - \mu_{i,t}^{\theta_k})^T \Sigma_{\theta,i,t}^{-1}(\mu_{i,t}^{\theta} - \mu_{i,t}^{\theta_k}) - k + \ln(\frac{|\Sigma_{\theta,i,t}|}{|\Sigma_{\theta_k,i,t}|})\} \quad \text{(A.1)}
\end{aligned}
$$

where $\Sigma_{\theta_k,i,t}, \Sigma_{\theta,i,t}$ are the diagonal covariance matrices of the two multi-variate Gaussian distributions $\pi_{\theta_k}(\cdot|s_{i,t}), \pi_\theta(\cdot|s_{i,t})$ that can be derived from $\sigma_{i,t}^{\theta_k}$ and $\sigma_{i,t}^{\theta}$. The loss function $L_{vanilla}$ can be expressed as a function of $\Sigma_{\theta_k,i,t}$, $\Sigma_{\theta,i,t}$, $\mu_{i,t}^{\theta}$ and $\mu_{i,t}^{\theta_k}$ when working with a multi-variate Gaussian. To this end, we use the log-likelihood function

$\ln(\pi_\theta(a_{i,t}|s_{i,t}))$, which gives the log-likelihood of having taken action $a_{i,t}$ given a state $s_{i,t}$. In the case of a multi-variate Gaussian, $\ln(\pi_\theta(a_{i,t}|s_{i,t}))$ is defined as:

$$\ln(\pi_\theta(a_{i,t}|s_{i,t})) = -\frac{1}{2}(\ln(|\Sigma_{\theta,i,t}|)+(a_{i,t}-\mu^{\theta}_{i,t})^T*\Sigma^{-1}_{\theta,i,t}*(a_{i,t}-\mu^{\theta}_{i,t})+m*\ln(2*\pi)) \quad (A.2)$$

where m is the dimension of the action space and where $|\Sigma_{\theta,i,t}|$ represents the determinant of the matrix. From this definition, one can rewrite $L_{vanilla}$ as:

$$L_{vanilla}(h_{i,t};\theta) = -e^{\ln(\pi_\theta(a_{i,t}|s_{i,t}))-\ln(\pi_{\theta_k}(a_{i,t}|s_{i,t}))} * GAE^i_t \quad . \quad (A.3)$$

By merging equation (A.3), (A.2) and equation (3.5), one gets a loss L that depends only on $\Sigma_{\theta_k,i,t}$, $\Sigma_{\theta,i,t}$, $\mu^{\theta}_{i,t}$ and $\mu^{\theta_k}_{i,t}$.

Appendix B

Supplementary material for Chapter 5

B.1 Proof of bistability for BRC and nBRC for $a_t > 1$

Theorem B.1.1. *The system defined by the equation*

$$h_t = ch_{t-1} + (1-c)\tanh(Ux_t + ah_{t-1}) = F(h_{t-1}) \tag{B.1}$$

with $c \in [0,1]$ *is monostable for* $a \in [0,1[$ *and bistable for* $a > 1$ *in some finite range of* Ux_t *centered around* $x_t = 0$.

Proof. We can show that the system undergoes a supercritical pitchfork bifurcation at the equilibrium point $(x_0, h_0) = (0,0)$ for $a = a_{pf} = 1$ by verifying the conditions

$$G(h_0)\big|_{a_{pf}} = \tfrac{dG(h_t)}{dh_t}\big|_{h_0,a_{pf}} = \tfrac{d^2G(h_t)}{dh_t^2}\big|_{h_0,a_{pf}} = \tfrac{dG(h_t)}{da}\big|_{h_0,a_{pf}} = 0 \tag{B.2}$$

$$\tfrac{d^3G(h_t)}{dh_t^3}\big|_{h_0,a_{pf}} > 0, \tfrac{d^2G(h_t)}{dh_t da}\big|_{h_0,a_{pf}} < 0 \tag{B.3}$$

where $G(h_t) = h_t - F(h_t)$ (Golubitsky & Schaeffer (2012)). This gives

$$\begin{aligned}
G(h_0)\Big|_{a_{pf}} &= (1-c)(h_0 - \tanh(a_{pf}h_0)) = 0, && \text{(B.4)}\\
\frac{dG(h_t)}{dh_t}\Bigg|_{h_0,a_{pf}} &= (1-c)(a_{pf}(\tanh^2(a_{pf}h_0)-1)+1) = (1-c)(1-a_{pf}) = 0, && \text{(B.5)}\\
\frac{d^2G(h_t)}{dh_t^2}\Bigg|_{h_0,a_{pf}} &= (1-c)2a_{pf}^2\tanh(a_{pf}h_0)(1-\tanh^2(a_{pf}h_0)) = 0, && \text{(B.6)}\\
\frac{dG(h_t)}{da}\Bigg|_{h_0,a_{pf}} &= (1-c)h_0(\tanh(a_{pf}h_0)^2 - 1) = 0, && \text{(B.7)}\\
\frac{d^3G(h_t)}{dh_t^3}\Bigg|_{h_0,a_{pf}} &= (1-c)*(2a^3(\tanh^2(a_{pf}h_0)-1)^2 + 4a_{pf}^3\tanh^2(a_{pf}h_0)(\tanh^2(a_{pf}h_0)-1))\\
&= 2(1-c) > 0, && \text{(B.8)}\\
\frac{d^2G(h_t)}{dh_t da}\Bigg|_{h_0,a_{pf}} &= (1-c)((\tanh^2(a_{pf}h_0)-1) + 2a_{pf}h_0\tanh(a_{pf}h_0)(1-\tanh^2(a_{pf}h_0)))\\
&= c-1 < 0. && \text{(B.9)}
\end{aligned}$$

The stability of (x_0, h_0) for $a \neq 1$ can be assessed by studying the linearized system

$$h_t = \frac{dF(h_t)}{dh_t}\Bigg|_{h_0} h_{t-1}. \tag{B.10}$$

The equilibrium point is stable if $dF(h_t)/dh_t \in [0,1[$, singular if $dF(h_t)/dh_t = 1$, and unstable if $dF(h_t)/dh_t \in]1,+\infty[$. We have

$$\begin{aligned}
\frac{dF(h_t)}{dh_t}\Bigg|_{h_0} &= c + (1-c)a(1-\tanh^2(a_t h_0)) && \text{(B.11)}\\
&= c + (1-c)a, && \text{(B.12)}
\end{aligned}$$

which shows that (x_0, h_0) is stable for $a \in [0,1[$ and unstable for $a > 1$.

It follows that for $a < 1$, the system has a unique stable equilibrium point at (x_0, h_0), whose uniqueness is verified by the monotonicity of $G(h_t)$ $(dG(h_t)/dh_t > 0 \forall h_t)$.

For $a > 1$, the point (x_0, h_0) is unstable, and there exist two stable points $(x_0, \pm h_1)$ whose basins of attraction are defined by $h_t \in]-\infty, h_0[$ for $-h_1$ and $h_t \in]h_0, +\infty[$ for h_1. □

Appendix C

Supplementary material for Chapter 8

C.1 Detailed description of benchmark 2 and 3

Before defining the three benchmark problems, let us remind that for each benchmark, the MDPs that belong to the support of η, which generates the different tasks, have transition probabilities and reward functions that differ only according to the value of a scalar α. Drawing an MDP according to η will amount for all the benchmark problems to draw a value of α according to a probability distribution $P_\alpha(\cdot)$ and to determine the transition function and the reward function that correspond to this value. Let us also denote by $\mathcal{X}$ and $\mathcal{A}$ the state and action spaces respectively.

C.1.1 Benchmark 2

State space and action space:

$$\mathcal{X} = [-3.0, 3.0]^2$$

$$\mathcal{A} = \mathcal{R}$$

Probability distribution of α:

$$\boldsymbol{\alpha}[\boldsymbol{i}] \sim U[-1.0, 1.0], \ \forall i \in [1, 2]$$

$$\boldsymbol{\alpha}[\mathbf{3}] \sim U[-\pi, \pi[$$

where $U[a, b]$ stands for a uniform distribution between a and b.

Initial state distribution:
The initial state x_0 is drawn through 2 auxiliary random variables that represent the x and y initial coordinates of the agent and are denoted u_0^x, u_0^y. At the beginning of an episode, those variables are drawn as follows:

$$u_0^k \sim U[-1.5 * \pi, 1.5 * \pi] \ \forall k \in \{x, y\}$$

From those four auxiliary variables, we define x_0 as:

$$x_0 = [\boldsymbol{\alpha}[\mathbf{1}] - a_0^x, \boldsymbol{\alpha}[\mathbf{2}] - a_0^y]$$

The distribution $P_{x_0}(\cdot)$ is thus fully given by the distributions over the auxiliary variables.

Transition function:
First, let *target* be the set of points $(x, y) \in \mathcal{R}^2$ such that

$$(x, y) \in target \Leftrightarrow \sqrt{(x - \boldsymbol{\alpha}[\mathbf{1}])^2 + (y - \boldsymbol{\alpha}[\mathbf{2}])^2} \leq 0.4 \quad .$$

When taking action a_t in state x_t drawing the state x_{t+1} from the transition function amounts to first compute u_{t+1}^x and u_{t+1}^y according to the following procedure:

1. If $(u_t^x, u_t^y) \in target$ then $u_{t+1}^k \sim U[-1.5, 1.5] \ \forall k \in \{x, y\}$.
2. If the preceding condition is not met, an auxiliary variable $n_t \sim U[\frac{-\pi}{4}, \frac{\pi}{4}]$ is drawn to compute u_{t+1}^x and u_{t+1}^y through the following sub-procedure:
   (a) Step one:

$$u_{t+1}^x = u_t^x + 0.25 * (\sin(a_t) + \sin(\boldsymbol{\alpha}[\mathbf{3}] + n_t))$$

$$u_{t+1}^y = u_t^x + 0.25 * (\cos(a_t) + \cos(\boldsymbol{\alpha}[\mathbf{3}] + n_t)) \quad .$$

   One can see that taking an action a_t moves the agent in a direction which is the vectoral sum of the intended move $\mathbf{m}_t$ of direction a_t and of a perturbation vector $\mathbf{p}_t$ of direction $\alpha + n_t$ sampled through the distribution over n_t.

(b) Step two: In the case where the coordinates computed by step one lay outside $S[-2;2]^2$, they are corrected so as to model the fact that when the agent reaches an edge of the 2D space, it is moved to the opposite edge from which it continues its move. More specifically, $\forall k \in \{x, y\}$:

$$u^k_{t+1} \leftarrow \begin{cases} u^k_{t+1} - 4 & \text{if } u^k_{t+1} > 2 \\ u^k_{t+1} + 4 & \text{if } u^k_{t+1} < -2 \\ u^k_{t+1} & \text{otherwise} \quad . \end{cases}$$

Once u^x_{t+1} and u^y_{t+1} have been computed, x_{t+1} is set equal to $[\boldsymbol{\alpha}[\mathbf{1}] - u^x_{t+1}, \boldsymbol{\alpha}[\mathbf{2}] - u^y_{t+1}]$.

Reward function:
The reward function can be expressed as follows:

$$\rho(a_t, x_t, x_{t+1}) = \begin{cases} 100 & \text{if } (u^x_t, u^y_t) \in target \\ -2 & \text{otherwise} \quad . \end{cases}$$

C.1.2 Benchmark 3

State space and action space:

$$\mathcal{X} = [-2.5, 2.5]^4$$

$$\mathcal{A} = \mathcal{R}$$

Probability distribution of α:

$$\boldsymbol{\alpha}[\boldsymbol{i}] \sim U[-1.0, 1.0], \ \forall i \in [1, 2, 3, 4]$$

$$\boldsymbol{\alpha}[\mathbf{5}] \sim U\{-1, 1\}$$

Note that $\boldsymbol{\alpha}[1, 2, 3, 4]$ define the 2-D positions of two targets. For clarity, we will refer to these values respectively by α^{x_1},α^{y_1},α^{x_2} and α^{y_2}.

Initial state distribution:
The initial state x_0 is drawn through two auxiliary random variables that represent the x and y initial coordinates of the agent and are denoted u_0^x, u_0^y. At the beginning of an episode, those variables are drawn as follows:

$$u_0^k \sim U[-1.5, 1.5]\ \forall k \in \{x, y\} \quad .$$

From those six auxiliary variables, we define x_0 as:

$$x_0 = [\alpha^{x_1} - u_0^x, \alpha^{y_1} - u_0^y, \alpha^{x_2} - u_0^x, \alpha^{y_2} - u_0^y] \quad .$$

Transition function:
For all $i \in \{1, 2\}$ let $target_i$ be the set of points $(x, y) \in \mathcal{R}^2$ such that

$$\sqrt{(x - \alpha^{x_i})^2 + (y - \alpha^{y_i})^2} \leq 0.4 \quad .$$

When taking action a_t in state x_t, drawing the state x_{t+1} from the transition function amounts to first compute u_{t+1}^x and u_{t+1}^y according to the following procedure:

1. If $\exists i \in \{1, 2\} : (u_t^x, u_t^y) \in target_i$, which means that the agent is in one of the two targets, then $u_{t+1}^k \sim U[-1.5, 1.5]\ \forall k \in \{x, y\}$
2. If the preceding condition is not met, u_{t+1}^x and u_{t+1}^y are computed by the following sub-procedure:
   (a) Step one:
   $$u_{t+1}^x = u_t^x + \sin(a_t * \pi) * 0.25$$
   $$u_{t+1}^y = u_t^y + \cos(a_t * \pi) * 0.25 \quad .$$
   This step moves the agent in the direction it has chosen.
   (b) Step two: In the case where the coordinates computed by step one lay outside $[-2; 2]^2$, they are corrected so as to model the fact that when the agent reaches an edge of the 2D space, it is moved to the opposite edge from which it continues its move. More specifically, $\forall k \in \{x, y\}$:
   $$u_{t+1}^k \leftarrow \begin{cases} u_{t+1}^k - 4.0 & \text{if } u_{t+1}^k > 2 \\ u_{t+1}^k + 4.0 & \text{if } u_{t+1}^k < -2 \\ u_{t+1}^k & \text{otherwise} \quad . \end{cases}$$

Once u_{t+1}^x and u_{t+1}^y have been computed, x_{t+1} is set equal to $[\alpha^{x_1} - u_{t+1}^x, \alpha^{y_1} - u_{t+1}^y, \alpha^{x_2} - u_{t+1}^x, \alpha^{y_2} - u_{t+1}^y]$.

Reward function:
In the case where (u_t^x, u_t^y) either belongs to only $target_1$, only $target_2$ or none of them, the reward function can be expressed as follows:

$$\rho(a_t, x_t, x_{t+1}) = \begin{cases} 100 * \boldsymbol{\alpha}[5] & \text{if } (u_t^x, u_t^y) \in target_1 \wedge (u_t^x, u_t^y) \notin target_2 \\ -50 * \boldsymbol{\alpha}[5] & \text{if } (u_t^x, u_t^y) \in target_2 \wedge (u_t^x, u_t^y) \notin target_1 \\ 0 & \text{if } (u_t^x, u_t^y) \notin target_1 \wedge (u_t^x, u_t^y) \notin target_2 \end{cases} .$$

In the case where (u_t^x, u_t^y) belongs to both $target_1$ and $target_2$, that is $(u_t^x, u_t^y) \in target_1 \wedge (u_t^x, u_t^y) \in target_2$, the reward function can be expressed as follows:

$$\rho(a_t, x_t, x_{t+1}) = \begin{cases} 100 * \boldsymbol{\alpha}[5] & \text{if } \sqrt{(u_t^x - p^{x_1})^2 + (u_t^y - p^{y_1})^2} \leq \sqrt{(u_t^x - p^{x_2})^2 + (u_t^y - p^{y_2})^2} \\ -50 * \boldsymbol{\alpha}[5] & \text{otherwise} \end{cases} .$$

That is, we consider that the agent belongs to the target to which it is closer to the centre.

C.2 Architecture details

For conciseness, let us denote by f_n a hidden layer of n neurons with activation functions f, by $\rightarrow$ a connection between two fully-connected layers and by $\multimap$ () a neuromodulatory connection (as described in Section 8.2).

Benchmark 1. The architectures used for this benchmark were as follows:

- RNN : $GRU_{50} \rightarrow ReLU_{20} \rightarrow ReLU_{10} \rightarrow I_1$
- NMN : $GRU_{50} \rightarrow ReLU_{20} \multimap (SReLU_{10} \rightarrow I_1)$

Benchmark 2 and 3. The architectures used for benchmark 2 and 3 were the same and as follows:

- RNN : $GRU_{100} \rightarrow GRU_{75} \rightarrow ReLU_{45} \rightarrow ReLU_{30} \rightarrow ReLU_{10} \rightarrow I_1$
- NMN : $GRU_{100} \rightarrow GRU_{75} \rightarrow ReLU_{45} \multimap (ReLU_{30} \rightarrow ReLU_{10} \rightarrow I_1)$

C.3 Bayes optimal policy for benchmark 1

A Bayes optimal policy is a policy that maximises the expected sum of rewards it obtains when playing an MDP drawn from a known distribution η. That is, a Bayes optimal policy π^*_{bayes} belongs to the following set:

$$\pi^*_{bayes} \in \arg\max_{\pi\in\Pi} \underset{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a_\cdot\sim\pi(\cdot) \\ x_\cdot\sim P_{\mathbb{M}}(\cdot,\cdot)}}{\mathbb{E}} R^{\pi}_{\mathbb{M}} \quad ,$$

with $P_{\mathbb{M}}$ being the state-transition function of the MDP $\mathbb{M}$ and $R^{\pi}_{\mathbb{M}}$ the discounted sum of rewardsobtained when playing policy π on $\mathbb{M}$.

In the first benchmark, the MDPs only differ by a bias, which we denote α. Drawing an MDP according to η amounts to draw a value of α according to a uniform distribution of α over $[-\alpha_{max}, \alpha_{max}]$, denoted by U_α, and to determine the transition function and the reward function that correspond to this value. Therefore, we can write the previous equation as:

$$\pi^*_{bayes} \in \arg\max_{\pi\in\Pi} \underset{\substack{\alpha\sim U_\alpha \\ x_0\sim P_{x_0} \\ a_\cdot\sim\pi(\cdot) \\ x_\cdot\sim P_{\mathbb{M}(\alpha)}(\cdot,\cdot)}}{\mathbb{E}} R^{\pi}_{\mathbb{M}} \quad ,$$

with $\mathbb{M}(\alpha)$ being a function giving as output the MDP corresponding to α and Π the set of all possible policies.

We now prove the following theorem.

Theorem C.3.1. *The policy that selects:*

1. *at time-step $t = 0$ the action $a_0 = x_0 + \frac{\gamma*(\alpha_{max}+4.5)}{1+\gamma}$*
2. *at time-step $t = 1$*
   a) *if $r_0 = 10$, the action $a_1 = x_1 + a_0 - x_0$*
   b) *else if $|r_0| > \alpha_{max} - (a_0 - x_0) \quad \wedge \quad a_0 - x_0 > 0$, the action $a_1 = a_0 + r_0$*
   c) *else if $|r_0| > \alpha_{max} - (x_0 - a_0) \quad \wedge \quad a_0 - x_0 < 0$, the action $a_1 = a_0 - r_0$*
   d) *and otherwise the action $a_1 = a_0 + r_0 + 1$*
3. *for the remaining time-steps:*
   a) *if $r_0 = 10$, the action $a_t = x_t + a_0 - x_0$*
   b) *else if $r_1 = 10$, the action $a_t = x_t + a_1 - x_1$*
   c) *and otherwise the action $a_t = x_t + i_t$ where i_t is the unique element of the set $\{a_0 - x_0 + r_0; a_0 - x_0 - r_0\} \cap \{a_1 - x_1 + r_1; a_1 - x_1 - r_1\}$*

is Bayes optimal for benchmark 1.

Proof. Let us denote by $\pi^*_{theorem1}$ the policy described in this theorem. To prove this theorem, we first prove that in the set of all possible policies Π there are no policy π which leads to a higher value of

$$\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a.\sim\pi(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (r_0 + \gamma * r_1) \tag{C.1}$$

than $\pi^*_{theorem1}$. Or equivalently:

$$\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a.\sim\pi^*_{theorem1}(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (r_0 + \gamma * r_1) \geq \mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a.\sim\pi(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (r_0 + \gamma * r_1)\ \forall \pi \in \Pi \quad . \tag{C.2}$$

Afterwards, we prove that the policy $\pi^*_{theorem1}$ generates for each time-step $t \geq 2$ a reward equal to R_{max} which is the maximum reward achievable, or written alternatively as:

$$\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a.\sim\pi^*_{bayes}(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (\sum_{t=2}^{\infty} \gamma^t * r_t) = \sum_{t=2}^{\infty} \gamma^t * R_{max} \geq \mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a.\sim\pi(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (\sum_{t=2}^{\infty} \gamma^t * r_t)\ \forall \pi \in \Pi \quad . \tag{C.3}$$

By merging (C.2) and (C.3), we have that

$$\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0}(\cdot) \\ a.\sim\pi_{theorem1}(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (\sum_{t=0}^{\infty} \gamma^t * r_t) \geq \mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0}(\cdot) \\ a.\sim\pi(\cdot) \\ x.\sim P_{\mathbb{M}}(\cdot,\cdot)}} (\sum_{t=0}^{\infty} \gamma^t * r_t)\ \forall \pi \in \Pi$$

which proves the theorem.

$\triangleright$ *Part 1.* Let us now prove inequality (C.2). The first thing to notice is that, for a policy to maximise expression (C.1), it only needs to satisfy two conditions for all x_0. The first one: to select an action a_1, which knowing the value of (x_0, a_0, r_0, x_1), maximises the expected value of r_1. We denote by $V_1(x_0, a_0, r_0, x_1)$ the maximum expected value of r_1 that can be obtained knowing the value of (x_0, a_0, r_0, x_1). The second one: to select an action a_0 knowing the value of x_0 that maximises the expected value of the sum $r_0 + \gamma V_1(x_0, a_0, r_0, x_1)$. We now show that the policy $\pi_{theorem1}$ satisfies these two conditions.

Let us start with the first condition that we check by analysing four cases, which correspond to the four cases a), b), c), d) of policy $\pi_{theorem1}$ for time step $t = 1$.

a) If $r_0 = 10$, the maximum reward that can be obtained, we are in a context where a_0 belongs to the target interval. It is easy to see that, by playing $a_1 = x_1 + a_0 - x_0$, we will obtain r_1 equal to 10. This shows that in case a) for time step $t = 1$, $\pi_{theorem1}$ maximises this expected value of r_1.

b) If $|r_0| > \alpha_{max} - (a_0 - x_0) \quad \wedge \quad a_0 - x_0 > 0$ and $r_0 \neq 10$ it is easy to see that the value of α to which the MDP corresponds can be inferred from (x_0, a_0, r_0) and that the action $a_1 = a_0 + r_0$ will fall in the middle of the target interval, leading to a reward of 10. Hence, in this case also, the policy $\pi_{theorem1}$ maximises the expected value of r_1.

c) If $|r_0| > \alpha_{max} - (x_0 - a_0) \quad \wedge \quad a_0 - x_0 < 0$ and $r_0 \neq 10$, we are also in a context where the value of α can be inferred directly from (x_0, a_0, r_0) and the action $a_1 = a_0 - r_0$ targets the centre of the target interval, leading to a reward of 10. Here again, $\pi_{theorem1}$ maximises the expected value of r_1.

d) When none of the three previous conditions is satisfied, a is not satisfied and so $x_1 = x_0$, we need to consider two cases: $(a_0 - x_0) \geq 0$ and $(a_0 - x_0) < 0$. Let us first start with $(a_0 - x_0) \geq 0$. In such a context, $\alpha \in \{a_0 - x_0 + r_0; a_0 - x_0 - r_0\} = \{a_0 - x_0 - |a_0 - x_0 - \alpha|, a_0 - x_0 + |a_0 - x_0 - \alpha|\}$ and where:

   1) $P(\alpha = a_0 - x_0 - |a_0 - x_0 - \alpha||x_0, a_0, r_0, x_1) = 0.5$

   2) $P(\alpha = a_0 - x_0 + |a_0 - x_0 - \alpha||x_0, a_0, r_0, x_1) = 0.5$.

   Let us now determine the action a_1 that maximises $\hat{r}_1$, the expected value of r_1 according to $P(\alpha|x_0, a_0, r_0, x_1)$. Five cases, represented on Figure C.1, have to be considered:

   1) $a_1 < a_0 - |a_0 - x_0 - \alpha| - 1$. Here $\hat{r}_1 = a_1 - a_0$ and the maximum of $\hat{r}_1$ is equal to $-|a_0 - x_0 - \alpha| - 1$.

   2) $a_1 \in [a_0 - |a_0 - x_0 - \alpha| - 1, a_0 - |a_0 - x_0 - \alpha| + 1]$. Here we have $\hat{r}_1 = \frac{1}{2}(10 + a_0 - |a_0 - x_0 - \alpha| - a_1)$ whose maximum over the interval is $5.5 - |a_0 - x_0 - \alpha|$ which is reached for $a_1 = a_0 + |a_0 - x_0 - \alpha| - 1$.

   3) $a_1 \in [a_0 - |a_0 - x_0 - \alpha| + 1, a_0 + |a_0 - x_0 - \alpha| - 1]$. In this case $\hat{r}_1 = -|a_0 - x_0 - \alpha|$ and is independent from a_1.

   4) $a_1 \in [a_0 + |a_0 - x_0 - \alpha| - 1, a_0 + |a_0 - x_0 - \alpha| + 1]$. The expected reward is $\hat{r}_1 = \frac{1}{2}(10 + a_0 - |a_0 - x_0 - \alpha| - a_1)$ whose maximum over the interval is $5.5 - |a_0 - x_0 - \alpha|$ which is reached for $a_1 = a_0 + |a_0 - x_0 - \alpha| + 1$.

   5) $a_1 > a_0 + |a_0 - x_0 - \alpha| + 1$. In this case the expected reward is $\hat{r}_1 = a_0 - a_1$ and the maximum of $\hat{r}_1$ is equal to $-|a_0 - x_0 - \alpha| - 1$.

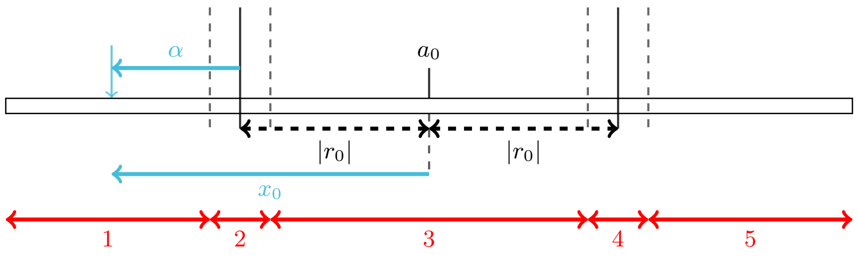


Figure C.1: Graphical representation of the 5 different cases when playing a_1.

From 1), 2), 3), 4) and 5) one can see that, given the conditions considered here, an optimal policy can either play $a_1 = a_0 + |a_0 - x_0 - \alpha| - 1$ or $a_1 = a_0 - |a_0 - x_0 - \alpha| + 1$. In the following we will fix a_1 to $a_0 + |a_0 - x_0 - \alpha| + 1$ when $a_0 - x_0 \geq 0$. Let us also observe that the expected value of r_1 is equal to $5.5 - |a_0 - x_0 - \alpha|$. Up to now in this item d), we have only considered the case where $(a_0 - x_0) > 0$. When $(a_0 - x_0) \leq 0$, using the same reasoning we reach the exact same expression for the optimal action to be played and for the maximum expected return of r_1. This is due to the symmetry that exists between both cases. Since $\pi_{theorem1}$ plays the action $a_1 = a_0 + r_0 + 1 = a_0 - |a_0 - x_0 - \alpha| + 1$ in the case d) at time step 1, it is straightforward to conclude that, in this case, it also plays an action that maximises the expected value of r_1.

Now that the first condition for $\pi_{theorem1}$ to maximise expression (C.1) has been proved, let us turn our attention to the second one. To this end, we will compute for each $x_0 \in \mathcal{X}$, the action $a_0 \in \mathcal{A}$ that maximises:

$$\mathop{\mathbb{E}}_{\substack{\alpha \sim U_\alpha \\ x_1 \sim P_{\mathbb{M}(\alpha)}(x_0, a_0)}} \left(r_0 + \gamma * V_1(x_0, a_0, r_0, x_1) \right) \tag{C.4}$$

and show that this action coincides with the action taken by $\pi_{theorem1}$ for time step $t = 0$. First let us observe that for this optimisation problem, one can reduce the search space $\mathcal{A}$ to $[x_0 - \alpha_{max} + 1, x_0 + \alpha_{max} - 1] \subset \mathcal{A}$. Indeed, an action a_0 that does not belong to this latter interval would not give more information about α than playing $a_0 = x_0 - \alpha_{max} + 1$ or $x_0 + \alpha_{max} - 1$ and lead to a worse expected r_0. This reduction of the search space will be exploited in the developments that follow.

However, we should first remember that $U_\alpha = U[-\alpha_{max}, \alpha_{max}]$ and that the function $V_1(x_0, a_0, r_0, x_1)$ can be written as follows:

1. if $r_0 = 10$, V_1 is equal to $R_{max} = 10$
2. else if $|r_0| > \alpha_{max} - (a_0 - x_0) \quad \wedge \quad a_0 - x_0 > 0$ and $r_0 \neq 10$, then V_1 is equal to $R_{max} = 10$

3. else if $|r_0| > \alpha_{max} - (x_0 - a_0) \quad \wedge \quad a_0 - x_0 < 0$ and $r_0 \neq 10$, then V_1 is equal to $R_{max} = 10$
4. and otherwise V_1 is equal to $5.5 - |a_0 - x_0 - \alpha|$.

We note that the value of $V_1(x_0, a_0, r_0, x_1)$ does not depend on the state x_1, which allows us to rewrite expression (C.4) as follows:

$$\mathop{\mathbb{E}}_{\alpha \sim U_\alpha} (r_0 + \gamma * V_1(x_0, a_0, r_0, x_1)) \tag{C.5}$$

and since the expectation is a linear operator:

$$(C.5) = \mathop{\mathbb{E}}_{\alpha \sim U_\alpha} (r_0) + \gamma * \mathop{\mathbb{E}}_{\alpha \sim U_\alpha} (V_1(x_0, a_0, r_0, x_1)) \quad . \tag{C.6}$$

Let us now focus on the second term of this sum:

$$\mathop{\mathbb{E}}_{\alpha \sim U_\alpha} (V_1(x_0, a_0, r_0, x_1)) \quad . \tag{C.7}$$

We note that when $a_0 - x_0 \geq 0$ the function V_1 can be rewritten under the following form:

1. if $\alpha \in [-\alpha_{max}, 2 * (a_0 - x_0) - \alpha_{max}[$, V_1 is equal to 10
2. else if $\alpha \in [2 * (a_0 - x_0) - \alpha_{max}, a_0 - x_0 - 1]$, v_1 is equal to $5.5 + \alpha - (a_0 - x_0)$
3. else if $\alpha \in [a_0 - x_0 - 1, a_0 - x_0 + 1]$, V_1 is equal to 10
4. else if $\alpha \in]a_0 - x_0 + 1, \alpha_{max}]$, V_1 is equal to $5.5 - \alpha + (a_0 - x_0)$.

From here, we can compute the value of expression (C.7) when $a_0 - x_0 \geq 0$. We note that due to the symmetry that exists between the case $a_0 - x_0 \geq 0$ and $a_0 - x_0 \leq 0$, expression (C.7) will have the same value for both cases. Since we have:

$$(C.7) = \int_{-\infty}^{\infty} V_1 * p_\alpha * d\alpha$$

where p_α is the probability density function of α, we can write:

$$\begin{aligned}
(C.7) &= \int_{-\alpha_{max}}^{\alpha_{max}} V_1 * \frac{1}{2 * \alpha_{max}} d\alpha \\
&= \int_{-\alpha_{max}}^{2*(a_0 - x_0) - \alpha_{max}} \frac{10}{2 * \alpha_{max}} d\alpha + \int_{2*(a_0 - x_0) - \alpha_{max}}^{a_0 - x_0 - 1} \frac{5.5 + \alpha - (a_0 - x_0)}{2 * \alpha_{max}} d\alpha \\
&+ \int_{a_0 - x_0 - 1}^{a_0 - x_0 + 1} \frac{10}{2 * \alpha_{max}} d\alpha + \int_{a_0 - x_0 + 1}^{\alpha_{max}} \frac{5.5 - \alpha + (a_0 - x_0)}{2 * \alpha_{max}} d\alpha \quad .
\end{aligned}$$

And thus, by computing the integrals, we have:

$$\underset{\alpha\sim U_\alpha}{\mathbb{E}}(V_1) = -\frac{1}{2*\alpha_{max}}(a_0 - x_0)^2 + \frac{1}{\alpha_{max}}(\alpha_{max} + 4.5) * (a_0 - x_0) + \frac{1}{\alpha_{max}}(5 + 5.5 * \alpha_{max} - \frac{\alpha_{max}^2}{2}) \quad .$$

Let us now analyse the first term of the sum in equation (C.6), namely $\underset{\alpha\sim U_\alpha}{\mathbb{E}}(r_0)$. We have that:

$$\underset{\alpha\sim U_\alpha}{\mathbb{E}}(r_0) = \int_{-\infty}^{\infty}(r_0|x_0, a_0, \alpha) * p_\alpha * d\alpha$$

which can be rewritten as:

$$\underset{\alpha\sim U_\alpha}{\mathbb{E}}(r_0) = \int_{-\alpha_{max}}^{\alpha_{max}}(r_0|x_0, a_0, \alpha) * \frac{1}{2*\alpha_{max}}d\alpha \quad .$$

Due to the reduction of the search space, we can assume that a_0 belongs to $[x_0 - \alpha_{max} + 1, x_0 + \alpha_{max} - 1]$, we can write:

$$\int_{-\alpha_{max}}^{\alpha_{max}}(r_0|x_0, a_0, \alpha) * \frac{1}{2*\alpha_{max}}d\alpha = \int_{-\alpha_{max}}^{a_0 - x_0 - 1}\frac{\alpha - (a_0 - x_0)}{2*\alpha_{max}}d\alpha + \int_{a_0 - x_0 - 1}^{a_0 - x_0 + 1}\frac{10}{2*\alpha_{max}}d\alpha + \int_{a_0 - x_0 + 1}^{\alpha_{max}}\frac{(a_0 - x_0) - \alpha}{2*\alpha_{max}}d\alpha \quad .$$

Given that $R_{max} = 10$, we have:

$$\underset{\alpha\sim U_\alpha}{\mathbb{E}}(r_0) = \frac{-(a_0 - x_0)^2 + 21 - \alpha_{max}^2}{2*\alpha_{max}}$$

and therefore:

$$(C.6) = -\frac{1+\gamma}{2*\alpha_{max}} * (a_0 - x_0)^2 + \frac{\gamma}{\alpha_{max}}(\alpha_{max} + 4.5) * (a_0 - x_0) + \frac{1}{2*\alpha_{max}}(21 - \alpha_{max}^2 + \gamma * (10 + 11 * \alpha_{max} - \alpha_{max}^2)) \quad .$$

To find the action a_0 that maximises (C.4), one can differentiate (C.6) with respect to a_0:

$$\frac{d(C.6)}{d(a_0)} = -\frac{1}{\alpha_{max}} * (1+\gamma)(a_0 - x_0) + \frac{\gamma}{\alpha_{max}}(\alpha_{max} + 4.5) \quad .$$

This derivative has a single zero value equal to:

$$a_0 = \frac{\gamma * (\alpha_{max} + 4.5)}{1+\gamma} + x_0 \quad .$$

It can be easily checked that it corresponds to a maximum of expression (C.4) and since it also belongs to the reduced search space $[x_0 - \alpha_{max} + 1, x_0 + \alpha_{max} - 1]$, it is indeed the solution to our optimisation problem. Since $\pi_{theorem1}$ plays this action at time $t = 0$, *Part 1* of this proof is now fully completed.

▷ *Part 2.* Let us now prove that the policy $\pi^*_{theorem1}$ generates for every $t \geq 2$ rewards equal to $R_{max} = 10$. We will analyse three different cases, corresponding to the three cases a), b) and c) of policy $\pi_{theorem1}$ for time step $t \geq 2$.

a) If $r_0 = 10$, we are in a context where a_0 belong to the target interval. It is straightforward to see that, by playing $a_t = x_t + a_0 - x_0$, the action played by $\pi_{theorem1}$ in this case, we will get a reward r_t equal to 10.

b) If $r_1 = 10$ and $r_0 \neq 10$, one can easily see that playing action $a_t = x_t + a_1 - x_1$, the action played by $\pi_{theorem1}$, will always generate rewards equal to 10.

c) If $r_0 \neq 10$ and $r_1 \neq 10$, it is possible to deduce from the first action a_0 that the MDP played corresponds necessarily to one of these two values for α: $\{a_0 - x_0 + r_0; a_0 - x_0 - r_0\}$. Similarly, from the second action played, one knows that α must also stand in $\{a_1 - x_1 + r_1; a_1 - x_1 - r_1\}$. It can be proved that because $a_0 \neq a_1$ (a property of our policy $\pi_{theorem1}$), the two sets have only one element in common. Indeed if these two sets had all their elements in common, either this pair of equalities would be valid:

$$a_0 - x_0 + r_0 = a_1 - x_1 + r_1$$
$$a_0 - x_0 - r_0 = a_1 - x_1 - r_1$$

or this pair of equalities would be valid:

$$a_0 - x_0 + r_0 = a_1 - x_1 - r_1$$
$$a_0 - x_0 - r_0 = a_1 - x_1 + r_1 \quad .$$

By summing member by member the two equations of the first pair, we have:

$$a_0 - x_0 = a_1 - x_1 \quad .$$

Taking into account that $x_0 = x_1$ because none of the two actions yielded a positive reward, it implies that $a_0 = a_1$, which results in a contradiction. It can be shown in a similar way that another contradiction appears with the second pair. As a result the intersection of these two sets is unique and equal to α. From here, it is straightforward to see that in this case c), the policy $\pi_{theorem1}$ will always generate rewards equal to R_{max}.

□

From Theorem C.3.1, one can easily prove the following theorem.

Theorem C.3.2. *The value of expected return of a Bayes optimal policy for benchmark 1 is equal to* $\frac{3*\gamma^2*(\alpha_{max}+4.5)^2}{2*\alpha_{max}*(1+\gamma)} + \frac{21+\alpha_{max}^2+\gamma*(10+11*\alpha_{max}-\alpha_{max}^2)}{2*\alpha_{max}} + \frac{\gamma^2}{1-\gamma} * 10$.

Proof. The expected return of a Bayes optimal policy can be written as follows:

$$\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a_.\sim\pi^*_{bayes}(\cdot) \\ x_.\sim P_{\mathbb{M}}(\cdot,\cdot)}} \sum_{t=0}^{1} \gamma^t * r_t + \mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a_.\sim\pi^*_{bayes}(\cdot) \\ x_.\sim P_{\mathbb{M}}(\cdot,\cdot)}} \sum_{t=2}^{\infty} \gamma^t * r_t \quad .$$

From the proof of Theorem C.3.1, it is easy to see that:

1. $\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a_.\sim\pi^*_{bayes}(\cdot) \\ x_.\sim P_{\mathbb{M}}(\cdot,\cdot)}} \sum_{t=0}^{1} \gamma^t * r_t = \frac{3*\gamma^2*(\alpha_{max}+4.5)^2}{2*\alpha_{max}*(1+\gamma)} + \frac{21+\alpha_{max}^2+\gamma*(10+11*\alpha_{max}-\alpha_{max}^2)}{2*\alpha_{max}}$

2. $\mathop{\mathbb{E}}_{\substack{\mathbb{M}\sim\eta \\ x_0\sim P_{x_0} \\ a_.\sim\pi^*_{bayes}(\cdot) \\ x_.\sim P_{\mathbb{M}}(\cdot,\cdot)}} \sum_{t=2}^{\infty} \gamma^t * r_t = \frac{\gamma^2}{1-\gamma}10$

which proves Theorem C.3.2. □

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... Zheng, X. (2015). *TensorFlow: Large-scale machine learning on heterogeneous systems.* Retrieved from `http://tensorflow.org/` (Software available from tensorflow.org)

Abbott, L. F., & Nelson, S. B. (2000). Synaptic plasticity: taming the beast. *Nature neuroscience*, *3*(11), 1178–1183.

Agostinelli, F., Hoffman, M., Sadowski, P., & Baldi, P. (2014). Learning activation functions to improve deep neural networks. *arXiv preprint arXiv:1412.6830*.

Arjovsky, M., Shah, A., & Bengio, Y. (2016). Unitary evolution recurrent neural networks. In *International conference on machine learning* (pp. 1120–1128).

Barak, O. (2017). Recurrent neural networks as versatile tools of neuroscience research. *Current opinion in neurobiology*, *46*, 1–6.

Bargmann, C. I., & Marder, E. (2013). From the connectome to brain function. *Nature methods*, *10*(6), 483–490.

Beaulieu, S., Frati, L., Miconi, T., Lehman, J., Stanley, K. O., Clune, J., & Cheney, N. (2020). Learning to continually learn. *arXiv preprint arXiv:2002.09571*.

Bellec, G., Salaj, D., Subramoney, A., Legenstein, R., & Maass, W. (2018). Long short-term memory and learning-to-learn in networks of spiking neurons. In *Advances in neural information processing systems* (pp. 787–797).

Bellec, G., Scherr, F., Hajek, E., Salaj, D., Legenstein, R., & Maass, W. (2019). Biologically inspired alternatives to backpropagation through time for learning in recurrent neural nets. *arXiv preprint arXiv:1901.09049*.

Bengio, Y., Frasconi, P., & Simard, P. (1993). The problem of learning long-term dependencies in recurrent networks. In *Ieee international conference on neural networks* (pp. 1183–1188).

Bengio, Y., Lee, D.-H., Bornschein, J., Mesnard, T., & Lin, Z. (2015). Towards biologically plausible deep learning. *arXiv preprint arXiv:1502.04156*.

Bromley, J., Bentz, J. W., Bottou, L., Guyon, I., LeCun, Y., Moore, C., ... Shah, R. (1993). Signature verification using a "siamese" time delay neural network. *International Journal of Pattern Recognition and Artificial Intelligence*, *7*(04), 669–688.

Capogrosso, M., Milekovic, T., Borton, D., Wagner, F., Moraud, E., Mignardot, J.-B., ... Courtine, G. (2016, 11). A brain–spinal interface alleviating gait deficits after spinal cord injury in primates. *Nature*, *539*, 284-288. doi: 10.1038/nature20118

Casas, N. (2017). Deep deterministic policy gradient for urban traffic light control. *arXiv preprint arXiv:1703.09035*.

Ceni, A., Ashwin, P., & Livi, L. (2020). Interpreting recurrent neural networks behaviour via excitable network attractors. *Cognitive Computation*, *12*(2), 330–356.

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). *Empirical evaluation of gated recurrent neural networks on sequence modeling*.

De Geeter, F., & Drion, G. (2021). Using multistability to solve fading memory problems in reinforcement learning.

Dey, R., & Salemt, F. M. (2017). Gate-variants of gated recurrent unit (gru) neural networks. In *2017 ieee 60th international midwest symposium on circuits and systems (mwscas)* (pp. 1597–1600).

Doya, K. (1993). Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on neural networks*, *1*(75), 164.

Drion, G., O'Leary, T., Dethier, J., Franci, A., & Sepulchre, R. (2015). Neuronal behaviors: A control perspective. In *2015 54th ieee conference on decision and control (cdc)* (pp. 1923–1944).

Drion, G., O'Leary, T., & Marder, E. (2015). Ion channel degeneracy enables robust and tunable neuronal firing rates. *Proceedings of the National Academy of Sciences*, *112*(38), E5361–E5370.

Ernst, D., Geurts, P., & Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, *6*, 503–556.

Finn, C., Abbeel, P., & Levine, S. (2017). Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning* (pp. 1126–1135).

Franci, A., Drion, G., Seutin, V., & Sepulchre, R. (2013a). A balance equation determines a switch in neuronal excitability. *PLoS computational biology*, *9*(5).

Franci, A., Drion, G., Seutin, V., & Sepulchre, R. (2013b). A balance equation determines a switch in neuronal excitability. *PLoS computational biology*, *9*(5), e1003040.

Garnelo, M., Rosenbaum, D., Maddison, C., Ramalho, T., Saxton, D., Shanahan, M., ... Eslami, S. A. (2018). Conditional neural processes. In *International conference on machine learning* (pp. 1704–1713).

Geadah, V., Kerg, G., Horoi, S., Wolf, G., & Lajoie, G. (2020). Advantages of biologically-inspired adaptive neural activation in rnns during learning. *arXiv preprint arXiv:2006.12253*.

Geurts, P. (2002). *Contributions to decision tree induction: bias/variance tradeoff and time series classification* (Unpublished doctoral dissertation). University of Liège Belgium.

Golubitsky, M., & Schaeffer, D. G. (2012). *Singularities and groups in bifurcation theory* (Vol. 1). Springer Science & Business Media.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial networks. *arXiv preprint arXiv:1406.2661*.

Gu, S., Lillicrap, T., Sutskever, I., & Levine, S. (2016). Continuous deep q-learning with model-based acceleration. In *International conference on machine learning* (pp. 2829–2838).

Ha, D., Dai, A., & Le, Q. V. (2016a). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Ha, D., Dai, A., & Le, Q. V. (2016b). Hypernetworks. *arXiv preprint arXiv:1609.09106*.

Ha, D., & Schmidhuber, J. (2018). World models. *arXiv preprint arXiv:1803.10122*.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The elements of statistical learning: data mining, inference, and prediction.* Springer Science & Business Media.

Henaff, M., Szlam, A., & LeCun, Y. (2016). Recurrent orthogonal networks and long-memory tasks. In *International conference on machine learning* (pp. 2034–2042).

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hospedales, T., Antoniou, A., Micaelli, P., & Storkey, A. (2020). Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*.

Jing, L., Gulcehre, C., Peurifoy, J., Shen, Y., Tegmark, M., Soljacic, M., & Bengio, Y. (2019). Gated orthogonal recurrent units: On learning to forget. *Neural computation*, *31*(4), 765–783.

Jozefowicz, R., Zaremba, W., & Sutskever, I. (2015). An empirical exploration of recurrent network architectures. In *International conference on machine learning* (pp. 2342–2350).

Kakade, S. M. (2001). A natural policy gradient. *Advances in neural information processing systems*, *14*.

Kalashnikov, D., Irpan, A., Pastor, P., Ibarz, J., Herzog, A., Jang, E., ... Levine, S. (2018). Qt-opt: Scalable deep reinforcement learning for vision-based robotic manipulation. *arXiv preprint arXiv:1806.10293*.

Katz, G. E., & Reggia, J. A. (2017). Using directional fibers to locate fixed points of recurrent neural networks. *IEEE transactions on neural networks and learning systems*, *29*(8), 3636–3646.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koch, G., Zemel, R., & Salakhutdinov, R. (2015). Siamese neural networks for one-shot image recognition. In *Icml deep learning workshop* (Vol. 2).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, *25*, 1097–1105.

Lambrechts, G., & Ernst, D. (2021). Bistable recurrent cells and belief filtering for q-learning in partially observable markov decision processes.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541–551.

LeCun, Y., & Cortes, C. (2010). MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/. Retrieved 2016-01-14 14:24:11, from `http://yann.lecun.com/exdb/mnist/`

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

Liu, X., Yu, H.-F., Dhillon, I., & Hsieh, C.-J. (2020). Learning to encode position for transformer with continuous dynamical model. In *International conference on machine learning* (pp. 6327–6335).

Maheswaranathan, N., Williams, A. H., Golub, M. D., Ganguli, S., & Sussillo, D. (2019). Reverse engineering recurrent networks for sentiment classification reveals line attractor dynamics. *Advances in neural information processing systems*, *32*, 15696.

Marder, E., Abbott, L., Turrigiano, G. G., Liu, Z., & Golowasch, J. (1996). Memory from the dynamics of intrinsic membrane currents. *Proceedings of the national academy of sciences*, *93*(24), 13481–13486.

Marder, E., O'Leary, T., & Shruti, S. (2014). Neuromodulation of circuits with variable parameters: single neurons and small circuits reveal principles of state-dependent and robust neuromodulation. *Annual review of neuroscience*, *37*, 329–346.

Marder, E., et al. (1996). Principles of rhythmic motor pattern generation. *Physiological reviews*, *76*(3), 687–717.

Marichal, R., PiÑeiro, J., González, E., & Torres, J. (2009). New approach of recurrent neural network weight initialization. In *Advances in computational algorithms and data analysis* (pp. 537–548). Springer.

Markram, H., Muller, E., Ramaswamy, S., Reimann, M. W., Abdellah, M., Sanchez, C. A., … others (2015). Reconstruction and simulation of neocortical microcircuitry. *Cell*, *163*(2), 456–492.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Miconi, T. (2017). Biologically plausible learning in recurrent neural networks reproduces neural dynamics observed during cognitive tasks. *Elife*, *6*, e20899.

Miconi, T., Rawal, A., Clune, J., & Stanley, K. O. (2020). Backpropamine: training self-modifying neural networks with differentiable neuromodulated plasticity. *arXiv preprint arXiv:2002.10585*.

Miconi, T., Stanley, K., & Clune, J. (2018). Differentiable plasticity: training plastic neural networks with backpropagation. In *International conference on machine learning* (pp. 3559–3568).

Mikolov, T., Sutskever, I., Deoras, A., Le, H.-S., Kombrink, S., & Cernocky, J. (2012). Subword language modeling with neural networks. *preprint (http://www. fit. vutbr. cz/imikolov/rnnlm/char. pdf)*, *8*, 67.

Mitchell, T. M., & McGraw, H. (1997). Machine learning.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., . . . Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning* (pp. 1928–1937).

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Moerland, T. M., Broekens, J., & Jonker, C. M. (2020). Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*.

Murphy, K. P. (2022). *Probabilistic machine learning: An introduction*. MIT Press. Retrieved from `probml.ai`

O'Leary, T., Williams, A. H., Caplan, J. S., & Marder, E. (2013). Correlations in ion channel expression emerge from homeostatic tuning rules. *Proceedings of the National Academy of Sciences*, *110*(28), E2645–E2654.

O'Leary, T., Williams, A. H., Franci, A., & Marder, E. (2014). Cell types, network homeostasis, and pathological compensation from a biologically plausible ion channel expression model. *Neuron*, *82*(4), 809–821.

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *International conference on machine learning* (pp. 1310–1318).

Pfeiffer, M., & Pfeil, T. (2018). Deep learning with spiking neurons: opportunities and challenges. *Frontiers in neuroscience*, *12*, 774.

Potdar, K., Pardawala, T. S., & Pai, C. D. (2017). A comparative study of categorical variable encoding techniques for neural network classifiers. *International journal of computer applications*, *175*(4), 7–9.

Rakelly, K., Zhou, A., Quillen, D., Finn, C., & Levine, S. (2019). Efficient off-policy meta-reinforcement learning via probabilistic context variables. *arXiv preprint arXiv:1903.08254*.

Ribar, L., & Sepulchre, R. (2019). Neuromodulation of neuromorphic circuits. *IEEE Transactions on Circuits and Systems I: Regular Papers*, *66*(8), 3028–3040.

Rokach, L., & Maimon, O. (2005). Clustering methods. In *Data mining and knowledge discovery handbook* (pp. 321–352). Springer.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1985). *Learning internal representations by error propagation* (Tech. Rep.). California Univ San Diego La Jolla Inst for Cognitive Science.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2015). Trust region policy optimization. In *International conference on machine learning* (pp. 1889–1897).

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *International conference on machine learning* (pp. 387–395).

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929–1958.

Sussillo, D., & Barak, O. (2013). Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, *25*(3), 626–649.

Tallec, C., & Ollivier, Y. (2018). Can recurrent neural networks warp time? *arXiv preprint arXiv:1804.11188*.

Tavanaei, A., Ghodrati, M., Kheradpisheh, S. R., Masquelier, T., & Maida, A. (2019). Deep learning in spiking neural networks. *Neural Networks*, *111*, 47–63.

Tsuda, B., Pate, S. C., Tye, K. M., Siegelmann, H. T., & Sejnowski, T. J. (2021). Neuromodulators enable overlapping synaptic memory regimes and nonlinear transition dynamics in recurrent neural networks. *bioRxiv*.

Van den Brand, R., Heutschi, J., Barraud, Q., DiGiovanna, J., Bartholdi, K., Huerlimann, M., ... Courtine, G. (2012). Restoring voluntary control of locomotion after paralyzing spinal cord injury. *science*, *336*(6085), 1182–1185.

Van Der Westhuizen, J., & Lasenby, J. (2018). The unreasonable effectiveness of the forget gate. *arXiv preprint arXiv:1804.04849*.

Van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 30).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Vecoven, N., Ernst, D., & Drion, G. (2021a). A bio-inspired bistable recurrent cell allows for long-lasting memory. *Plos one*, *16*(6), e0252676.

Vecoven, N., Ernst, D., & Drion, G. (2021b). Warming-up recurrent neural networks to maximize reachable multi-stability greatly improves learning. *arXiv preprint arXiv:2106.01001*.

Vecoven, N., Ernst, D., Wehenkel, A., & Drion, G. (2020). Introducing neuromodulation in deep neural networks to learn adaptive behaviours. *PloS one*, *15*(1), e0227922.

Voelker, A., Kajić, I., & Eliasmith, C. (2019). Legendre memory units: Continuous-time representation in recurrent neural networks. In *Advances in neural information processing systems* (pp. 15570–15579).

Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., … Botvinick, M. (2016). Learning to reinforcement learn. *CoRR*, *abs/1611.05763*. Retrieved from `http://arxiv.org/abs/1611.05763`

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995–2003).

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wiering, M. A. (2005). Qv (lambda)-learning: A new on-policy reinforcement learning algrithm. In *Proceedings of the 7th european workshop on reinforcement learning* (pp. 17–18).

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, *8*(3-4), 229–256.

Wilson, D. G., Cussat-Blanc, S., Luga, H., & Harrington, K. (2018). Neuromodulated learning in deep neural networks. *arXiv preprint arXiv:1812.03365*.

Zhou, G.-B., Wu, J., Zhang, C.-L., & Zhou, Z.-H. (2016). Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, *13*(3), 226–234.