Introducing neuromodulation in deep neural networks to learn adaptive behaviours

Nicolas Vecoven
Damien Ernst
Antoine Wehenkel
Guillaume Drion

Department of Electrical Engineering and Computer Science
Montefiore Institute, University of Liège
B-4000 Liège, Belgium

Abstract

In this paper, we propose a new deep neural network architecture, called NMD net, that has been specifically designed to learn adaptive behaviours. This architecture exploits a biological mechanism called neuromodulation that sustains adaptation in biological organisms. This architecture has been introduced in a deep-reinforcement learning architecture for interacting with Markov decision processes in a meta-reinforcement learning setting where the action space is continuous. The deep-reinforcement learning architecture is trained using an advantage actor-critic algorithm. Experiments are carried on several test problems. Results show that the neural network architecture with neuromodulation provides significantly better results than state-of-the-art recurrent neural networks which do not exploit this mechanism.

1. INTRODUCTION

The field of reinforcement learning (RL), and machine learning in general, have made tremendous progress during the past few years, predominantly owing to the improvement of deep neural network (DNN) algorithms, combined with an increase in data availability and computer power (Hinton et al. (2012), Mnih et al. (2015)). We are now seeing the emergence of highly efficient algorithms that are capable of learning complex behaviours in specific environments (Mnih et al. (2015), Silver et al. (2016)). However, even the most advanced learning algorithms still lack the ability to generalise and adapt their behaviour to a wide variety of tasks in a constantly changing environment, which calls for the development of novel approaches on this topic.

Adaptation can be described as the ability to efficiently tackle a new, unforeseen environment based on past experience. In a supervised learning setting, adaptation can be associated to meta-learning. Simple meta-learning problems have been successfully tackled in the recent past using Recurrent Neural Networks (RNN) (Hochreiter et al. (2001), Santoro et al. (2016)). Recently, a new framework has been proposed to address a similar kind of problem in a RL setting (Wang et al. (2016)). RL is a learning framework where an agent learns to perform actions while interacting with specific environments, aiming at maximising the sum of rewards obtained from the said environments. In meta-RL, the environments
are potentially highly variable and can change between episodes. In this framework, RNNs, which are trained following an advantage actor-critic framework (A2C, see e.g. Mnih et al. (2016)), are used as a way of adapting to new, similar environments. However, this approach is still in its infancy and could benefit from the development of novel network architectures that are specifically designed to tackle the adaptation problem.

In this paper, we propose such a new network architecture inspired from a biological mechanism that underlies adaptation in biological systems. This mechanism, called neuro-modulation, provides the possibility to continuously tune neuron input/output properties to shape their response to external inputs in different contexts, generally in response to an external signal carried by biochemistry called neuromodulators (Bargmann and Marder (2013), Marder et al. (2014)). Neuromodulation regulates many critical nervous system properties that cannot be achieved solely through synaptic plasticity (Marder and Calabrese (1996), Marder and Bucher (2001)), which represents the ability for neurons to tune their connectivity within a network during learning and has inspired the construction of DNN algorithms (McCulloch and Pitts (1943)). Neuromodulation has been shown to be critical to the adaptive control of continuous behaviours, such as in motor control, among others (Marder and Calabrese (1996), Marder and Bucher (2001)). Here, we propose an abstract version of neuromodulation that is specifically designed for DNN.

The integration of neuromodulation in deep-RL algorithms is performed in two steps: (i) by making neuron activation functions of an action network modulable (an action network is defined as a network that outputs an action in response to past and current inputs received from the environment), and (ii) by creating a neuromodulation network whose neuromodulatory signals specifically target and modulate the activation functions of the action network neurons, mimicking the role of neuromodulators in nervous systems. The proposed neuromodulation-based architecture, which we call NMD net, is sketched in Figure 1. The action network is a feedforward neural network that takes inputs from the environment and outputs actions that are applied to the said environment (Figure 1A, bottom). The activation functions of the neurons of the action network are defined by parameterised saturated rectified linear units (ReLUs). Parameterised saturated ReLUs are conditioned by two parameters that regulate function slope and offset. Formally, with \( s \) and \( b \) the modulated parameters, the activation function is defined as \( g(x) = \min(1, \max(-1, s \cdot x + b)) \). The higher the slope parameter(s), the more sensitive the neuron (Figure 1B, bottom). As such, the action network is the sole network that interacts with the environments in the proposed architecture. The neuromodulation network is an RNN that takes as inputs “input-output pairs” from the action network, as well as the rewards obtained from the environment for each input-output pair (Figure 1A, top). It outputs neuromodulatory signals that dynamically modify the slope and bias of the activation functions of the action network neurons (Figure 1B - the neuromodulatory connection is sketched as a line with a rounded tip). The neuromodulation network therefore has all the information required to identify the current environment and adapt the behaviour of the action network to this specific environment.
Figure 1: Sketch of the NMD net architecture. A. The architecture is composed of the interaction of a recurrent neural network (top) and a feedforward neural network (bottom). The feedforward neural network takes inputs from the environment and outputs actions that are applied to the said environment. The RNN takes as inputs “input-output pairs” from the action network as well as the rewards obtained from the environment for each input-output pair, and outputs a neuromodulatory signal that modulates the activation functions of the feedforward network neurons. B. Sketch of a neuromodulatory connection in two neuromodulation states (left and right). The top neurons represent neuromodulatory neurons, the bottom neurons represent action neurons that are being neuromodulated, and the line with a rounded edge represent a neuromodulatory connection. As the neuromodulation state of the neuromodulation neuron changes, it affects the slope and bias of the action neuron activation function.

In order to move further towards adaptive RL policies, we propose to use the NMD net in a deep RL setting. The goal is to learn a policy on an ensemble of environments and to ensure that this policy is still efficient on unseen but similar environments. This means that the policy has to find and understand what the current characteristics of the new environment are and exploit them as efficiently as possible. This constitutes a meta-RL setting, as studied in Wang et al. (2016). In this latter paper, the authors use a deep-RL architecture trained with an advantage actor-critic algorithm that relies on the sampling of sequences of trajectories. Despite the fact that the trajectories are generated on different environments (sharing similarities), the authors show that the algorithm is able to learn a policy that tends to perform better and better over time on new environments to which it is exposed and, hence, it can be concluded that it has adaptation capabilities.

Since meta-RL is still in its infancy and has rarely been formalised, we first propose a definition of a specific meta-RL problem that is tackled in this paper (Section 2). In a few words, we assume that a meta-RL agent is facing a sequence of new environments and that it can generate a single trajectory on any new environment using any policy it wants. The environments are formally described by Markov decision processes (MDPs) and are drawn independently from an unknown distribution over MDPs. The goal of the meta-RL agent is to maximise the expected return that it has collected over a given number of trajectories. This problem shares similarities with a classical Bayesian RL setting, except that here the
After having formalised the meta-RL problem, we detail the RL algorithms that are used to solve this meta-RL problem in Section 3. As for the meta RL setting studied in Wang et al. (2016), we have chosen an advantage actor-critic algorithm (Mnih et al. (2016)) that works with a parametric approximation architecture. The main difference between our approach and the approach described in Wang et al. (2016) stems from the different type of neural nets used as approximation architecture. In the same section, we also highlight how to specify the generic actor-critic algorithm to DNNs. Section 4 then introduces the NMD net and discusses the differences between this architecture and a classic RNN. The performances of the NMD net are evaluated in Section 5 and compared to those obtained with a classical RNN net.

Finally, we emphasise that we work with MDPs having continuous action spaces, in order to make the approach as general as possible and easily transferable to a physical context. This differs from earlier works on meta-RL, where discrete-action problems were used to evaluate the performance of the algorithms. To this end, we create custom benchmarks with continuous action spaces and use them for comparison. These are briefly described in Section 5 and fully detailed in Appendix A.1.

2. META REINFORCEMENT LEARNING: A FORMALISATION

In this section, we formalise the meta-reinforcement learning (meta-RL) problem that is addressed in this paper. Variants of this formalisation can also be thought of as relevant instances of the generic meta-RL problem.

In our meta-RL setting, an agent has to interact through a sequence of episodes with MDPs drawn from a distribution $\eta$. An MDP is a discrete time control process. Let $t = 0, 1, \ldots$ denote the discrete time, $s_t$ the state of the MDP at time $t$, $u_t$ the action taken at time $t$ and $r_t$ the reward obtained at the subsequent time-step. An MDP is defined through two main elements: a state transition function $P(s_{t+1} | s_t, u_t)$ that sets the probability of observing state $s_{t+1}$ while taking action $u_t$ in state $s_t$, and a reward function $\rho(s_t, u_t, s_{t+1})$ that gives the reward $r_t$ obtained by transitioning from state $s_t$ to state $s_{t+1}$ through the action $u_t$. All MDPs belonging to the support of $\eta$ are assumed to have the same state space $S \subseteq \mathbb{R}^n$ and the same action space $U \subseteq \mathbb{R}^m$. All the reward functions are also assumed to be bounded, i.e. $\rho(s_t, u_t, s_{t+1}) \in [R_{\min}, R_{\max}] \forall s_t, s_{t+1} \in S, u_t \in U$ and with $R_{\min}, R_{\max} \in \mathbb{R}$.

At the beginning of a new episode $i$, we draw an element from $\eta$ to define an MDP, referred to by $MDP_i$, with which the meta-RL agent interacts afterwards. The interaction process between the meta-RL agent and $MDP_i$ is defined as follows:

1. The initial state $s_0$ is drawn according to the distribution over the initial states $P_{s_0} (\cdot)$.
2. At each time step, the agent selects any action $u_t \in U$, for which it transitions from state $s_t$ to a new state $s_{t+1}$ and receives a reward $r_t$.
3. The agent can observe the different states encountered and the rewards obtained.
4. The interaction lasts an infinite number of time steps.
The only information that the agent collects on the transition function of $\text{MDP}_i$ and its reward function is through observing the states crossed and the rewards obtained at each time-step.

Let $s_{i,t}$, $u_{i,t}$ and $r_{i,t}$ refer to the values of $s_t$, $u_t$ and $r_t$ in episode $i$. Let $h_{i,t} = \{s_{i,0}, u_{i,0}, r_{i,0}, s_{i,1}, \ldots, u_{i,t-1}, r_{i,t-1}, s_{i,t}\}$ be the history of the interaction of the meta-RL agent with $\text{MDP}_i$ up to time step $t$. Let $\mathcal{H}$ denote the set of all possible histories. Let $\pi^i : [\mathcal{H} \times U] \rightarrow [0, 1]$ be the policy played by the RL agent during episode $i$. $\Pi$ denotes the set of all such policies. Without loss of generality, we assume that it is a (possibly degenerated) probabilistic policy that selects the action $u_{i,t}$ to be played at time $t$ based on the knowledge of $h_{i,t} : u_{i,t} \sim \pi^i(h_{i,t})$. We will see later in this paper why we choose to work with policies dependent on the whole history.

The return of policy $\pi^i$ during episode $i$ called $R_{\text{MDP}_i}^{\pi^i}$ is defined as follows:

$$R_{\text{MDP}_i}^{\pi^i} = \lim_{T \rightarrow \infty} \sum_{t=0}^{T} \gamma^t r_{i,t}$$

(1)

where $\gamma \in [0, 1]$ is called the discount factor. The goal of the meta-learning agent is to maximise the expected value of the sum of returns it can obtain over a budget $E \in \mathbb{N}$ of episodes. As we will see later in this document, the policy $\pi^i$ computed by our algorithm will implicitly, at time $t$, use the history $h_{t-1}$, to adapt its behaviour to $\text{MDP}_i$.

3. REINFORCEMENT LEARNING PROCEDURE

For solving the meta-RL problem introduced in Section 2, we propose to use a strategy based on actor-critic (AC) algorithms. Subsection 3.1 details how the actor-critic algorithm interacts with this problem. Subsection 3.2 presents the specific types of loss functions we use for updating the approximation architectures of the AC algorithm in this meta-RL context through the use of gradient descent techniques. Subsection 3.3 discusses the strategy that we have adopted for computing the gradient of these loss functions in the particular context where approximation architectures are made of RNNs.

3.1 ADVANTAGE ACTOR-CRITIC FOR SOLVING META-RL PROBLEMS: AN INTERACTION

An actor-critic framework revolves around two distinct functions: the actor and the critic. The actor represents the policy currently used to interact with the MDPs, while the critic is a function that rates the performance of the agent’s policy. All actor-critic algorithms follow an iterative procedure that often consists of three steps. The first step involves using the policy to interact with the environment; the second step involves using the critic ratings to update the actors parameters, while the last involves updating the critic to approximate a value function. This iterative procedure provides the core loop of actor-critic algorithms (note that the order of the two last steps can be inverted).

In our meta-RL setting, both the actor and the critic are parameterised functions that are defined on the trajectories’ histories. With $\theta \in \Theta$ and $\psi \in \Psi$ the parameters of the actor and critic ($\Theta$ and $\Psi$ are the actor and critic parameters spaces), respectively, we define $\pi_\theta$ and $c_\psi$ as the policy and critic functions. It is current practice to make the critic and actor
share parameters. However, in this paper, we have chosen to move away from that choice and never share any parameters between critics and actors. Let $\pi_{\theta_k}$ and $c_{\psi_k}$ be the models for the policy and the critic after $k$ updates of the parameters $\theta$ and $\psi$, respectively. To update from $\theta_k$ to $\theta_{k+1}$ and $\psi_k$ to $\psi_{k+1}$, the actor-critic algorithm uses the policy $\pi_{\theta_k}$ to select actions during $B$ MDPs drawn sequentially from $\eta$, where $B \in \mathbb{N}_0$ is a parameter of the actor-critic approach. Note that we have: $\pi^i = \pi_{\theta_{F(i)}}$ where $F(i) = \lfloor \frac{i}{B} \rfloor$. Note that $F(i)$ is a function which allows to go from an episode number to the corresponding iteration of the actor-critic algorithm (that is, the number of updates of the actor and critic parameters done at a given episode). This interaction between the actor-critic algorithm and the meta-RL problem is presented in a tabular version in Algorithm 1 of Appendix B.

Using the $L \in \mathbb{N}_0$ first elements of each trajectory generated from the interaction with the $B$ MDPs and the values of $\theta_k$ and $\psi_k$, the algorithm computes $\theta_{k+1}$ and $\psi_{k+1}$. To this end, the algorithm exploits the set $[h_{I(k),L}, h_{I(k)+1,L}, \ldots, h_{I(k+1)-1,L}]$, which we denote as $H_k$ (as well as several previous sets $H_{k-1}, H_{k-2}$, etc. for $\psi_{k+1}$). The function $I : \mathbb{N} \to \mathbb{N}$ is defined as: $I(k) = B \times k$ and gives the number of the first episode played for a given iteration (update) of the actor-critic algorithm. One will note that once $I(k) \geq E$, the budget has been consumed and the algorithm stops. A tabular version of the algorithm that details how MDPs are drawn and played, as well as how the set $H_k$ is built, is presented in Algorithm 2 of Appendix B. Note that Algorithm 2 only runs the trajectories of the system for $L$ time-steps and not an infinite number of time-steps, something that would be computationally impossible. However, to have a properly performing actor-critic algorithm to work well, the value chosen for $L$ has to be chosen sufficiently large to produce an accurate estimation of the returns $R^\pi_{\theta_k}$, $\forall i \in [I(k), \ldots, I(k+1) - 1]$ obtained by the policy $\pi_{\theta_k}$.

When used in a classical RL setting, an AC algorithm should interact with its environment to find the value of $\theta$ that leads to high values of the expected return given a probability distribution over the initial states. This expected return is written as:

$$E_{s_0 \sim P_{s_0}} R^\pi_{MDP}$$

where $MDP$ denotes the Markov Decision Process with which the AC algorithm interacts. When working well, actor critic algorithms produce a sequence of policies $\pi_{\theta_1}, \pi_{\theta_2}, \pi_{\theta_3}, \ldots$ whose expected returns increase as the iterative process evolves and eventually reaches values close to those obtained by $\pi^*_{\theta_{MDP}}$ with $\theta^*_{MDP} = \arg \max_{\theta \in \Theta} E_{s_0 \sim P_{s_0}} R^\pi_{MDP}$, which, if $\pi^\theta$ is flexible enough, are themselves close to those obtained by an optimal policy $\pi^*_{MDP}$ defined as:

$$\pi^*_{MDP} \in \arg \max_{\pi \in \Pi} E_{s_0 \sim P_{s_0}} R^\pi_{MDP}$$

where $\Pi$ is the set of all admissible policies.

Let $h_t = \{s_0, u_0, r_0, \ldots, s_t\}$ be a trajectory generated by policy $\pi_\theta$ on this $MDP$ and let $J^\pi_{MDP}(h_t)$ be the expected sum of discounted rewards that can be obtained while starting from $h_t$ and playing the policy $\pi_\theta$ in this environment, that is:

$$J^\pi_{MDP}(h_t) = \sum_{j=t}^{\infty} \gamma^{j-t} \rho(s_j, u_j \sim \pi_\theta(h_j), s_{j+1})$$.
In a classical RL setting, and again for an efficient AC algorithm, the value of the critic for \( h, c_\psi(h_t) \), also converges to \( J^*_{\pi_{MDP}}(h_t) \). We also note that in such a setting, the critic is updated at iteration \( k + 1 \) in a direction that provides a better approximation of \( J^*_{\pi_{MDP}}(\cdot) \). Now, let us go back to our meta-RL problem and let \( V^\pi \) denote the expected sum of returns that policy \( \pi \) can obtain on this problem, that is, using the notations introduced in Section 2:

\[
V^\pi = \mathbb{E}_{MDP \sim \eta} R^\pi_{MDP}.
\]  

Let \( \theta^* \in \text{arg max}_{\theta \in \Theta} V_{\pi_{\theta}} \). When interacting with our meta-RL problem, a performant AC algorithm should, in principle, converge towards a policy \( \pi_{\theta^*} \) when \( E \) increases, leading to a value of \( V^*_{\pi_{\theta^*}} \) close to \( V^\pi \) and that is itself close to \( \max_{\pi \in \Pi} V^\pi \). A policy \( \pi^* \) such that \( \pi^* \in \text{arg max}_{\pi \in \Pi} V^\pi \) is called a Bayes optimal policy in a Bayesian RL setting where the distribution \( \eta \) is assumed to be known. If we are working with policies that are, indeed, able to quickly adapt to the environment, we may also expect that the policy \( \pi_{\theta^*} \) learned by the algorithm is such that, when applied on an MDP belonging to the support of \( \eta \), it leads to a value of \( J^*_{\pi_{\theta^*}}(h_t) \) close to \( \max_{\pi \in \Pi} J^*_{\pi_{MDP}}(h_t) \) as \( t \) increases. In other words, once the agent has gathered enough information to adapt to the current MDP, it should start behaving (almost) optimally. This is the essence of meta-RL.

We may also expect that, in this context, the value of the critic for \( h_t \) when the budget is exhausted, namely \( c_{\psi_{F(E)}}(h_t) \), closely estimates the expected value of the future discounted rewards that can be obtained when using policy \( \pi_{\theta^*} \) and after having already observed a trajectory \( h_t \). Therefore, we may also expect that \( c_{\psi_{F(E)}}(h_t) \):

1. will be close to \( \mathbb{E}_{MDP \sim \eta} J^*_{\pi_{\theta^*}}(h_t) \approx \mathbb{E}_{MDP \sim \eta} \max_{\pi \in \Pi} J^*_{\pi_{MDP}}(h_t) \) if \( h_t = \{s_0\} \);

2. will, as \( t \) increases, tend to get closer to \( \max_{\pi \in \Pi} J^*_{\pi_{MDP}}(h_t) \approx J^*_{\pi_{MDP}}(h_t) \) where MDP can be any environment belonging to the support of \( \eta \) used to generate \( h_t \).

### 3.2 ADVANTAGE ACTOR-CRITIC FOR SOLVING META-RL PROBLEMS: USING GENERALISED ADVANTAGE ESTIMATION AND PROXIMAL POLICY OPTIMIZATION

Existing actor-critic algorithms mainly differ from each other by the way the actor is 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 Williams (1992)), now it is more common to use an advantage function. This function represents the advantage in terms of return of selecting specific actions given a trajectory history (or simply a state when AC algorithms are used in a standard setting) over selecting them following the policy used to generate the trajectories.

AC algorithms using an advantage function have been successfully applied to meta-RL problems, which includes the so-called A2C algorithm introduced in Mnih et al. (2016) (see for instance Wang et al. (2016)). The AC algorithm used here differs from the A2C algorithm by the way the actor is updated.
First, we use generalised advantage estimations (GAE), as introduced in Schulman et al. (2015b), rather than classical advantage estimations. Second, 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, we update here the actor’s parameters \( \theta \) by minimising a loss function that represents a surrogate objective. We have selected as surrogate function one that is similar to the one introduced in Schulman et al. (2017) that describes the proximal policy optimisation (PPO) algorithm. Our algorithm is derived from the PPO algorithm, but differs in how actor and critic are updated, following a recommendation from Coady. The benchmarks studied in this paper indeed showed that the PPO algorithm performed better than the A2C algorithm, and that the recommendation of Coady still improved (albeit slightly) the performances of PPO in all cases.

Both the actor and the critic updates are performed following gradient descent on the losses. In Section 3.2.1 we describe how the parameters of the actor are updated (i.e. how \( \theta_{k+1} \) is computed at iteration \( k \)). To this end, we first describe how GAEs are computed and how they are used to build the loss function. Then, we introduce the algorithm used to minimise that loss. In Section 3.2.2 we describe how the parameters of the critic are updated (i.e. how \( \psi_{k+1} \) is computed at iteration \( k \)) following the same structure as in Section 3.2.1.

### 3.2.1 ACTOR UPDATE

**Loss definition**

We start by explaining how the actor is updated based on a loss function. First, we define the temporal error difference term for any two consecutive time-steps of any trajectory:

\[
TD_{i,j} = r_{i,j} + \gamma \cdot c_{\psi_{F(i)}}(h_{i,j+1}) - c_{\psi_{F(i)}}(h_{i,j}), \forall i \in [1, \ldots, E], j \in [0, \ldots, L]
\]

This temporal difference term represents, in some sense, the (immediate) advantage obtained, after having played action \( u_{i,j} \) over what was expected by the critic. If \( c_{\psi_{F(i)}}(\cdot) \) was the true estimate of \( J_{\pi_{\theta_{F(i)}}}^{MDP}(\cdot) \) and if the policy played was \( \pi_{\theta_{F(i)}} \), the expected value of these temporal differences would be equal to zero. We now define the GAE’s terms that will be used later in our loss functions:

\[
GAE_{i,j} = \sum_{t=j}^{L} (\gamma \cdot \lambda)^{t-j} \cdot TD_{i,j}, \forall i \in [1, \ldots, E], j \in [0, \ldots, L']
\]  

where \( \lambda \in [0,1] \) is a discount factor used for computing GAEs, and where \( L' \) is another hyper-parameter of the algorithm, chosen in combination with \( L \) in order to have a value of \( GAE_{i,j} \) that accurately approximates \( \sum_{t=j}^{\infty} (\gamma \cdot \lambda)^{k-j} \cdot TD_{i,j} \) \( \forall i, j \). Note that the value chosen for \( L' \) also has to be sufficiently large to provide the loss function with a sufficient number of GAE terms. These GAE terms, introduced in Schulman et al. (2015b), 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
In the loss function, we will actually not use the advantage terms as defined by Equation 6, but normalised versions in order to have advantages that remain in a similar range regardless of rewards magnitude. Thanks to this normalisation, the policy learning rate does not have to be tuned according to the loss magnitude. However, this normalisation does not mask actions that have led to higher or lower returns than average. The normalised terms, referred to as $\text{GAE}'_{i,j}$, are defined as follows

$$
\mu_{gae} = \sum_{i=0}^{B-1} \sum_{j=0}^{L'-1} \text{GAE}_{I(k)+i,j}
$$

$$
\sigma_{gae} = \sqrt{\sum_{i=0}^{B-1} \sum_{j=0}^{L'-1} (\mu_{gae} - \text{GAE}_{I(k)+i,j})^2}
$$

$$
\text{GAE}'_{i,j} = \frac{\text{GAE}_{I(k)+i,j} - \mu_{gae}}{\sigma_{gae}} \quad \forall i \in [0, \ldots, B-1], j \in [0, \ldots, L'-1]
$$

where $\sum$ is the symbol we use to represent the average sum operator (i.e. $\sum_{x=1}^{m} f(x) = \sum_{x=1}^{m} f(x) / m$). To define the loss functions used to compute $\theta_{k+1}$ and $\psi_{k+1}$, only the GAE terms corresponding to time-steps $[0, \ldots, L']$ of episodes $[I(k), I(k)+1, \ldots, I(k+1)-1]$ are computed. A tabular version of the algorithm used to compute these terms is given in Algorithm 3 of Appendix B.

Once advantages have been computed, the values of $\theta_{k+1}$ are computed using updates that are strongly related to PPO updates with a Kullback Leibler (KL) divergence implementation Schulman et al. (2017). Before explaining the PPO updates, let us first emphasise that in paper Wang et al. (2016) (where a meta-RL setting was considered), they update the policies through the following approximation of the gradient on the policy’s return proposed in Mnih et al. (2016):

$$
\sum_{[i,t] \in B_k} \nabla_{\theta} \log \pi_a(a_{i,t} | h_{i,t}) \ast \text{GAE}'_{i,t}
$$

where $B_k$ is the set of all pairs $[i, t]$ for which $i \in [I(k), \ldots, I(k+1)-1]$ and for which $t \in [0, \ldots, L']$, that is, the set containing the first $L'$ time-steps of the $B$ trajectories played for iteration $k$ of the actor-critic algorithm. With the PPO approach, rather than directly computing gradients, a surrogate objective, which is a loss function to be minimised, is used. This loss function is called a surrogate objective function because it is not directly

1. Although not explicitly written in the text for clarity, we use a normalisation technique when computing discounted sums for the AC algorithm update. In fact, when carrying an update of the AC algorithm, if rewards appear in discounted sums, they are multiplied by $(1 - \gamma)$. This has for effect that the discounted sum values remain of the same magnitude regardless of $\gamma$. The implications of this normalization are two-fold. (i) The critic does not directly approximate $J_{MDP}(\cdot)$ but rather $(1 - \gamma) \ast J_{MDP}(\cdot)$. (ii) Second, for the temporal differences to remain coherent with this normalisation, $r_{i,j}$ must also be multiplied by $(1 - \gamma)$ when computing $TD_{i,j}$. Those two small changes are explicated in Algorithm 3.
related to the expected return of a policy $\pi_\theta$. The loss function proposed in Schulman et al. (2017) is as follows:

$$L_{\text{vanilla}}(\theta) = - \sum_{[i,t] \in B_k} \frac{\pi_\theta(u_{i,t}|h_{i,t})}{\pi_\theta^k(u_{i,t}|h_{i,t})} \ast GAE'_{i,t}. \quad (7)$$

One can easily become intuitive about Equation 7 as, given a history $h_{i,t}$, 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. $GAE'_{i,t} < 0$). It has been found that to obtain good performances with this above-written loss function, it was important to have a policy that does not change too rapidly from one iteration to the other. Before explaining how this can be achieved, let us first give an explanation on why it may be important to have slow updates of the policy. Let us go back to the loss function given by Equation 7. Minimising this loss function will give a value for $\theta_{k+1}$ 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 $\pi_{\theta_k}$ but are instead related to the advantages of policy $\pi_{\theta_k}$ over $\pi_{\theta_{k-1}}$. Indeed, the advantages $GAE'_{i,t}$ are computed using the value function $c_{\psi_k}$, whose parameters have been updated from $\psi_{k-1}$ in order to better approximate the sum of discounted rewards obtained during the episodes $[I(k-1), \ldots, I(k-1)]$. It clearly appears that $\psi_k$ 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 $\pi_{\theta_k}$. We may reasonably suppose that the advantage function used in (7) still correctly reflects the real advantage function of $\pi_{\theta_{k+1}}$ over $\pi_{\theta_k}$. To achieve this, we add a penalisation term $P(\theta)$ to the loss function. In the PPO approach, the penalisation term is $P_{\text{ppo}}(\theta) = \beta_k \ast d(\theta)$, where:

i) $\beta_k$ is an adaptive weight

ii) $d(\theta) = \sum_{[i,t] \in B_k} [KL(\pi_{\theta_k}(.|h_{i,t}), \pi_{\theta_k}(.|h_{i,t}))]$, where $KL$ is the Kullback-Leibler divergence, detailed later on. This term penalises policies that are too different from $\pi_{\theta_k}$.

We note that the $\beta_k$ dynamical updates use a hyper-parameter $d_{\text{targ}} \in \mathbb{N}_0$ called the divergence target. The update is done through the following procedure (note that, unlike updates of $\beta$ proposed in Schulman et al. (2017), we constrain $\beta$ to remain in the range $[\beta_{\text{min}}, \beta_{\text{max}}]$; we explain later why):

$$\beta_{k+1} = \begin{cases} \max(\beta_{\text{min}}, \frac{\beta_k}{1.5}) & \text{if } d(\theta) < \frac{d_{\text{targ}}}{2} \\ \min(\beta_{\text{max}}, \beta_k \ast 1.5) & \text{if } d(\theta) > \frac{d_{\text{targ}}}{2} \\ \beta_k & \text{otherwise} \end{cases} \quad (8)$$

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 beyond $d_{\text{targ}}$ without having to add an explicit constraint on $d$, as was the case in Trust Region Policy Optimization (TRPO) updates Schulman et al. (2015a), which is more cumbersome to implement.
As suggested in Coady, adding another penalisation term (squared hinge loss) to $\mathcal{P}_{PPO}$ to further penalise the KL divergence, in cases where it surpasses $2 \cdot d_{\text{larg}}$, improved algorithm performance. The final expression of the penalisation term is:

$$
\mathcal{P}(\theta) = \beta_k \cdot d(\theta) + \delta \cdot \max(0, d(\theta) - 2 \cdot d_{\text{larg}})^2
$$

where $\delta$ is a hyper-parameter that weights the third loss term. The loss function $\mathcal{L}_{\text{policy}}$ that we minimise as a surrogate objective becomes:

$$
\mathcal{L}_{\text{policy}}(\theta) = \mathcal{L}_{\text{vanilla}}(\theta) + \mathcal{P}(\theta)
$$

We now detail how to compute the KL divergence. 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 actors policy can be defined in a straightforward way. For each dimension, we bound the multi-variate Gaussian to the support, $\mathcal{U}$, by playing the action that is clipped to the bounds of $\mathcal{U}$ whenever the multi-variate Gaussian is sampled outside of $\mathcal{U}$. In the remaining of this paper, we will sometimes abusively use the terms "output of the actor at time $t$" 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:

$$
KL(\pi_{\theta_k}(|h_{i,t})), \pi_{\theta}(|h_{i,t})) = \frac{1}{2} \{ tr(\Sigma_{\theta,i,t}^{-1} \Sigma_{\theta_k,i,t}) + \mu_{i,t}^{\theta_k} - \mu_{i,t}^{\theta} \}^T \Sigma_{\theta,i,t}^{-1} \mu_{i,t}^{\theta} - \mu_{i,t}^{\theta_k} \} - k + \ln \left( \frac{1}{|\Sigma_{\theta,i,t}|} \right) \right)
$$

where $\Sigma_{\theta_k,i,t}, \Sigma_{\theta,i,t}$ are the diagonal covariance matrices of the two multi-variate Gaussian distributions $\pi_{\theta_k}(|h_{i,t}), \pi_{\theta}(|h_{i,t})$ that can be derived from $\sigma_{i,t}^{\theta_k}$ and $\sigma_{i,t}^{\theta}$. The loss function $\mathcal{L}_{\text{vanilla}}$ can be expressed as a function of $\Sigma_{\theta_k,i,t}, \Sigma_{\theta,i,t}, \mu_{i,t}^{\theta_k}$ and $\mu_{i,t}^{\theta}$ when working with a multi-variate Gaussian. To this end, we use the log-likelihood function $\ln (\pi_{\theta}(u_{i,t}|h_{i,t})), \Sigma_{\theta,i,t}$, which gives the log-likelihood of having taken action $u_{i,t}$ given a trajectory history $h_{i,t}$. In the case of a multi-variate Gaussian, $\ln (\pi_{\theta}(u_{i,t}|h_{i,t}))$ is defined as:

$$
\ln (\pi_{\theta}(u_{i,t}|h_{i,t})) = -\frac{1}{2} \{ tr(\Sigma_{\theta,i,t}) + (u_{i,t} - \mu_{i,t}^{\theta})^T \Sigma_{\theta,i,t}^{-1} (u_{i,t} - \mu_{i,t}^{\theta}) + m \cdot \ln (2 \cdot \pi) \}
$$

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 $\mathcal{L}_{\text{vanilla}}$ as:

$$
\mathcal{L}_{\text{vanilla}} = - \sum_{[i,t] \in B_k} e^{\ln (\pi_{\theta}(u_{i,t}|h_{i,t})) - \ln (\pi_{\theta_k}(u_{i,t}|h_{i,t}))} \cdot GAE_{t,t}^{\prime}.
$$

By merging equation (12), (11) and equation (9), one gets a loss $\mathcal{L}_{\text{policy}}$ that depends only on $\Sigma_{\theta_k,i,t}, \Sigma_{\theta,i,t}, \mu_{i,t}^{\theta_k}$ and $\mu_{i,t}^{\theta}$. 

11
Algorithmic loss updates

For minimising the loss function, we will apply a gradient-based strategy. We use the ADAM optimisation algorithm from Kingma and Ba (2014), due to its general good performances. This algorithm introduces a momentum in the gradient updates through three hyper-parameters, denoted by $\omega_1$, $\omega_2$ and $\epsilon$, and works as follows.

After initialising $\theta'_0$ to $\theta_k$, $m$ to 0, the following procedure is repeated $AE \in \mathbb{N}_0$ number of times ($AE$ stands for "Actor Epochs"):

1. $lr_{Actor} k^{*} AE + m = actor \, lr_k \ast \sqrt{\frac{1-\omega_1^{k*} AE + m}{1-\omega_2^{k*} AE + m}}$ where $actor \, lr_k$ is an adaptive learning rate and $actor \, lr_0$ is a hyper-parameter.
2. $z_{Actor} k^{*} AE + m = \omega_1 \ast z_{Actor} k^{*} AE + m - 1 + (1 - \omega_1) \ast \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m)$ where $z_{Actor} k^{*} AE + m$ is a term that depends on the first gradient estimates.
3. $v_{Actor} k^{*} AE + m = \omega_2 \ast v_{Actor} k^{*} AE + m - 1 + (1 - \omega_2) \ast \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m) \odot \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m)$, where $\odot$ is the element-wise product and $v_{Actor} k^{*} AE + m$ is a term which depends on the second gradient estimates.
4. $\theta_{m+1} = \theta'_m - lr_{Actor} k^{*} AE + m \ast v_{Actor} k^{*} AE + m + \epsilon$
5. $m \leftarrow m + 1$

with $z_{Actor_0}$ and $v_{Actor_0}$ chosen equal to 0 and where $\nabla_{\theta} \mathcal{L}_{policy}$ is an approximate of the gradient of the loss. We explain the way our gradient approximate is computed in Subsection 3.3.

Once we have iterated $AE$ times over the above written procedure, we use $\theta'_{AE}$ as value for $\theta_{k+1}$. Note that the value of the learning rate $actor \, lr_k$ is updated when going from iteration $k$ to iteration $k + 1$ as follows:

- $actor \, lr_{k+1} = \frac{actor \, lr_k}{1.5}$ if $\beta_k > \beta_{max} \ast 0.85$
- $actor \, lr_{k+1} = actor \, lr_k \ast 1.5$ if $\beta_k < 1.15 \ast \beta_{min}$

where $\beta_k$ is computed according to Equation (8). Since $\beta_k$ is bounded, instead of further increasing/decreasing the penalisation $\mathcal{P}$, the algorithm will rather decrease/increase the learning rate, which results in smaller/bigger policy updates without changing the ratio of $\mathcal{P}$ with respect to $\mathcal{L}_{vanilla}$. In addition, the momentum is not reinitialised in between actor updates ($\theta_0 \rightarrow \theta_1 \rightarrow \ldots \rightarrow \theta_{F(E)}$) in order to keep the momentum going. We note however that there will not be much gradient momentum between policy updates in our simulations. This is related to the fact that parameters $\omega_1$ and $\omega_2$ were chosen close to 1, as detailed in Appendix C, which has the consequence that the previous gradients quickly lose their impact over the newer ones.

In cases where $d(\theta'_m) > d_{\text{threshold}} \ast d_{\text{targ}}$ with $d_{\text{threshold}} \in \mathbb{R}_+$, we stop the above iterative procedure and set $\theta_{k+1}$ to $\theta_k$. This helps with stabilising the learning in cases where updates diverge despite $\mathcal{P}$. A similar condition has been used in other work as an early-stop
criterion for the research of a policy leading to high expected returns Coady.

Finally, we stress that the average sum hidden in Equation 9 is computed over the set of all pairs in $B_k$, that is, the actor updates its parameters in a "full-batch learning" fashion. Indeed, each gradient update ($\theta'_m \rightarrow \theta'_{m+1}$) is computed on the same batch, which contains all the pairs in the set $B_k$ that we may consider as being our training set for iteration $k$. As a reminder, $B_k$ contains the $L'$ first time-steps of the $B$ trajectories played for the actor-critic algorithm’s $k$th iteration. Using all available pairs provides a better estimate of the policy distribution, which in turns provides a better estimate of the divergence between the two policies.

A tabular version of the policy update is given in Appendix B, see Algorithm 4.

### 3.2.2 CRITIC UPDATE

#### Loss definition

We now explain how the critic’s parameters $\psi$ are updated. The critic is updated at iteration $k$ in a way to better approximate the expected return obtained when following the policy $\pi_{\theta_k}$, starting from a given trajectory history. To this end, we use a mean-square error loss as a surrogate objective for optimizing $\psi$. First, we define $\hat{R}_{i,j} = \sum_{t=j}^{L} \gamma^{t-j} \cdot r_{i,t}$ $\forall i,j \in [I(k),\ldots,I(k+1) - 1],[0,\ldots,L]$. From the definition of $\hat{R}_{i,j}$ we express the loss as:

$$L_{\text{critic}}(\psi) = \sum_{[i,t] \in B_{k-CRB}} [(c_\psi(h_{i,t}) - \hat{R}_{i,t})^2]$$  \hspace{1cm} (13)

where (i) $CRB \in \mathbb{N}_0$ is a hyper-parameter; (ii) $B_{k-CRB}$ is the set of all pairs $[i,t]$ for which $i \in [I(k-CRB),\ldots,I(k+1) - 1]$ and for which $t \in [0,\ldots,L']$. The set $B_{k-CRB}$ used in (13) contains all the pairs from the current trajectory batch and from the $CRB$ previous trajectory batches. We call this a replay buffer whose length is controlled by $CRB$ which stands for "CriticReplayBuffer". Minimising $L_{\text{critic}}$ does not lead to updates such that $c_\psi$ directly approximates the average expected return of the policy $\pi_{\theta_k}$. Rather, the updates are such that $c_\psi$ directly approximates the average expected return obtained by the last $CRB + 1$ policies played. We found out that using a replay buffer for the critic smoothed the critic’s updates and improved algorithm performances.

Note that the loss (13) is only computed on the $L' << L$ first time-steps of each episode, as was the case for the actor. The reason behind this choice is simple. The value function $c_\psi$ should approximate $R_{i,j} = \sum_{t=j}^{\infty} \gamma^{t-j} \cdot r_{i,t}$ for every $h_{i,j}$, where $R_{i,j}$ the infinite sum of discounted rewards that are attainable when "starting" from $h_{i,j}$. However, this approximation can become less accurate when $j$ becomes close to $L$ since we can only guarantee $\hat{R}_{i,j}$ to stand in the interval: $[R_{i,j} - \frac{\gamma^{L-j}}{1-\gamma} R_{max}, R_{i,j} - \frac{\gamma^{L-j}}{1-\gamma} R_{min}]$. Hence this choice of $L'$.

#### Algorithmic loss updates

As for the actor, we use a stochastic gradient descent procedure based on the ADAM optimizer. This algorithm depends on a few hyper-parameters, namely: (i) $CMB \in \mathbb{N}_0$, which stands for "Critic MiniBatch" and is used to determine the number of pairs on which the
loss gradient is computed; (ii) $T \in \mathbb{N}_0$, which is a hyper-parameter of our gradient estimate, and which will be discussed afterwards; (iii) $CE' \in \mathbb{N}_0$, which stands for “Critic Epoch” and which drives the number of iterations carried by the ADAM algorithm. From these three hyper-parameters, we define $CE$ as $CE = CE' \cdot \left[ \frac{\|\theta_{k-CRB}\|}{CMB + T} \right]$. 

The algorithm works as follows. First, we set $\psi_0'$ to $\psi_k$, $\mathcal{X}_0$ to $\emptyset$, $m$ to 0, $v_{Critic0}$ to 0 and $z_{Critic0}$ to 0. Then, we repeat the following procedure $CE$ number of times.

1. One draws a set $\mathcal{Y}_m$ of $CMB \times T$ different pairs in $\mathcal{B}_{k-CRB} \setminus \mathcal{X}_m$ and defines the following surrogate loss function:

$$L_{sur}(\psi, \mathcal{Y}) = \sum_{[i,t] \in \mathcal{Y}} [(c_{\psi}(h_{i,t}) - \hat{R}_{i,t})^2] . \tag{14}$$

2. $lr_{Critic_{k+CE+m}} = \text{critic}_{lr} \cdot \sqrt{1 - \omega_{1+CE+m} - \frac{1}{1 - \omega_{1+CE+m}}}$, where $\text{critic}_{lr}$ is a fixed learning rate that does not depend on $k$ and that is a hyper-parameter.

3. $z_{Critic_{k+CE+m}} = \omega_1 \cdot z_{Critic_{k+CE+m-1}} + (1 - \omega_1) \cdot \nabla_{\psi} L_{sur}(\psi_m')$, where $z_{Critic_{k+CE+m}}$ is a term that depends on the first gradient estimates.

4. $v_{Critic_{k+CE+m}} = \omega_2 \cdot v_{Critic_{k+CE+m-1}} + (1 - \omega_2) \cdot \nabla_{\psi} L_{sur}(\psi_m') \odot \nabla_{\psi} L_{sur}(\psi_m')$, where $\odot$ is the element-wise product and $v_{Critic_{k+CE+m}}$ is a term that depends on the second gradient estimates.

5. $\psi_{m+1}' = \psi_m' - \frac{lr_{Critic_{k+CE+m}} \cdot z_{Critic_{k+CE+m}}}{\sqrt{v_{Critic_{k+CE+m}} + \epsilon}}$

6. We then update the set $\mathcal{X}_m$ through $\mathcal{X}_{m+1} \leftarrow \mathcal{X}_m \cup \mathcal{Y}_m$. If $\mathcal{B}_{k-CRB} \setminus \mathcal{X}_{m+1} = \emptyset$, then we reset $\mathcal{X}_{m+1}$ to $\emptyset$.

7. $m$ through $m \leftarrow m + 1$.

After completion of the iterative procedure, we set $\psi_{k+1}$ to $\psi_{CE'}'$. Note that we have not detailed how the pairs of $\mathcal{Y}_m$ are drawn to keep this subsection general. The drawing process is in fact constrained by the way we compute our gradient estimate $\nabla_{\psi}$, as it will be explained in next Subsection. If $\mathcal{B}_{k-CRB} \setminus \mathcal{X}_m$ does not contain $CMB \times T$ different pairs, then $\mathcal{Y}_m = \mathcal{B}_{k-CRB} \setminus \mathcal{X}_m$. Note also that as for the actor, the gradient momentum is carried in between critic updates. We also emphasise that the loss is always updated on a different batch of pairs, a strategy that could be associated with a ”mini batch learning” strategy in a standard supervised learning setting. Mini-batch learning introduces noise in the optimization process, which often results in better performances. As opposed to the actor, which requires lots of samples to have a decent approximation on the true policy encoded, the critic only needs to approximate expected returns. Thus, we have chosen to train the critic using mini-batches. As a side note, we emphasise that the ADAM hyper-parameters are, in our simulation reported later in this paper, chosen to be the same for the critic and for the actor.

A tabular version of the critic update algorithm is given in Algorithm 6 of Appendix B
3.3 APPROXIMATION OF THE GRADIENTS FOR RECURRENT NEURAL NETWORKS

In this subsection, we explain how to compute the gradients of the loss functions used to update the critic and the actor. The main difficulty for computing the gradients of these loss functions is related to the computation of the gradients of \( \psi_0 \) and \( c_\psi \) with respect to \( \theta \) and \( \psi \), respectively. Computing the gradients of these functions is indeed not straightforward due to the fact that they are defined by recurrent neural nets. Since the procedure for computing the gradients is similar for the actor and for the critic, we will focus our explanations on the actor network.

The actor being represented by an RNN, it makes use, for a given episode \( i \), of a recurrent function for evaluating its output given an input \( h_{i,t} = \{s_{i,0}, u_{i,0}, r_{i,0}, \ldots, s_{i,t}\} \). At each time-step, a recurrent network has two distinct outputs. The first one is the network output, which we have been referring to until now (\( \sigma_{i,t}^{\theta_F(i)} \) and \( \mu_{i,t}^{\theta_F(i)} \) at time-step \( t \) of episode \( i \)). The second one, is a state carry, which we call internal state, denoted by \( x_{i,t}^{\theta_F(i)} \) and used by the network to encode temporal features. Let us consider the case where we want to evaluate \( \pi_{\theta_F(i)}(h_{i,t}) \), which amounts to evaluate \( \sigma_{i,t}^{\theta_F(i)}(h_{i,t}) \) and \( \mu_{i,t}^{\theta_F(i)}(h_{i,t}) \).

The procedure for carrying out this evaluation is the following. First, the actor computes \( \mu_{i,0}^{\theta_F(i)}(h_{i,0}), \sigma_{i,0}^{\theta_F(i)}(h_{i,0}) \) and \( x_{i,1}^{\theta_F(i)}(h_{i,0}) \) using the recurrent neural net, with an input \( s_0 \) and an internal state \( x_{i,0} \) initialised to a 0-vector. Afterwards, to evaluate \( \mu_{i,1}^{\theta_F(i)}(h_{i,1}), \sigma_{i,1}^{\theta_F(i)}(h_{i,1}) \) and \( x_{i,2}^{\theta_F(i)}(h_{i,1}) \), it will use as inputs of the recurrent network \( (u_{i,0}, r_{i,0}, s_{i,1}, x_{i,1}^{\theta_F(i)}) \). And so on until computing \( \mu_{i,t}^{\theta_F(i)}(h_{i,t}), \sigma_{i,t}^{\theta_F(i)}(h_{i,t}) \) and \( x_{i,t+1}^{\theta_F(i)}(h_{i,t}) \) with the recurrent net using \( (u_{i,t-1}, r_{i,t-1}, s_{i,t}, x_{i,t}^{\theta_F(i)}) \) as inputs. We stress that, for our NMD nets, the network takes as inputs \( (s_{t-1}, u_{t-1}, r_{t-1}, s_t) \); this will be further discussed in Section 4. We will consider for the remainder of this section that the neural net is a standard RNN. However, we stress that all the explanations here under are valid for the NMD net; one just has to adjust the input vector of the network. The overall procedure for a classic RNN is sketched on Figure 2. We note that \( \pi_{\theta_F(i)}(h_{i,t}) \) will actually depend only on \( x_{i,t}^{\theta_F(i)}(h_{i,t-1}) \) and \( (u_{i,t-1}, r_{i,t-1}, s_{i,t}) \). In a well-working recurrent neural network \( x_{i,t}^{\theta_F(i)}(h_{i,t-1}) \) actually “encodes” all the necessary information about \( h_{i,t-1} \). Furthermore, in a standard RL setting, only the state \( s_{i,t} \) is used as input at time-step \( t \) of episode \( i \), whereas here, in addition to the internal state \( x_{i,t}^{\theta_F(i)} \), the network also takes as inputs the previous action \( u_{i,t-1} \) and the obtained reward \( r_{i,t-1} \). This change is necessary due to our meta-RL setting. Indeed, it is precisely the sequence of states observed, actions performed, and rewards obtained that let the agent gather knowledge about \( P_{MDP}(s_{t+1}|s_t, u_t) \) and \( P_{MDP}(s_t|u_t, s_{t+1}) \), which is essential for adapting its behaviour to the new environment.

As a reminder, our goal is to compute a gradient estimate so as to minimise a loss function \( L_{\text{policy}}(\theta) \), defined by Equation 9, this amounts to differentiate the function \( L_{\text{policy}} \) with respect to each variable \( v \in \theta \). In the fully developed version of \( L_{\text{policy}}(\theta) \), only the terms \( \mu_{\theta}^v \) and \( \sigma_{\theta}^v \) depend on \( \theta \). As a consequence, the main difficulty associated with the computation of the gradient of \( L_{\text{policy}} \) is related to computing the gradient of the output
of the RNN with respect to all \( v \in \theta \). Indeed, from the gradient of the outputs, we can compute the gradient of \( L_{\text{policy}} \) by applying standard differentiation rules as \( L_{\text{policy}} \) can be expressed by using only additions, subtractions, multiplications, divisions, exponentials and logarithms of \( \mu^\theta_i \) and \( \sigma^\theta_i \), which are all easily differentiable. There is however a single exception due to our custom penalisation term \( P(\theta) \), which introduces a \( \max(0, f(\theta)) \). Indeed, the latter function is not differentiable in 0. To address this problem, we fix the value of the derivative in 0 to be equal to 0. This means that we effectively have the following equation:

\[
\frac{d}{d\theta} \max(0, f(\theta)) = \begin{cases} 
0 & \text{if } f(\theta) \leq 0 \\
\frac{d}{d\theta} f(\theta) & \text{otherwise}
\end{cases}
\]

Thus, using this definition, it is very easy to compute the derivative of \( L_{\text{policy}} \) on the basis of the derivative of \( \mu^\theta_i \) and \( \sigma^\theta_i \). However, the expressions of \( \mu^\theta_{i,j} \) and \( \sigma^\theta_{i,j} \) grow in complexity as \( j \) increases due to the recurrence implied. Indeed, \( \mu^\theta_{i,j} \) and \( \sigma^\theta_{i,j} \) are functions of \( x^\theta_{i,j}(h_{i,j-1}) \) and one can make the increasing complexity appear, thanks to the following rewriting:

\[
x^\theta_{i,j}(h_{i,j-1}) = x^\theta_{i,j}(s_{i,j-1}, u_{i,j-2}, r_{i,j-2}, x^\theta_{i,j-1}(h_{i,j-2})) \\
= x^\theta_{i,j}(s_{i,j-1}, u_{i,j-2}, r_{i,j-2}, x^\theta_{i,j-1}(s_{i,j-2}, u_{i,j-3}, r_{i,j-3}, x^\theta_{i,j-2}(h_{i,j-3}))) \\
= \ldots
\]

One can continue the preceding enumeration until one reaches the point where \( x^\theta_{i,j}(h_{i,j-1}) \) is fully expressed by the elements of \( h_{i,t} \). Thus, to have an exact gradient, the computational time and memory required to differentiate \( L_{\text{policy}} \) grows linearly with the size of the history \( h_{i,t} \), up to a point where it becomes too complex to be computed and this despite \( L_{\text{policy}} \) being defined only on the \( L' \) first time-steps of each episode. To address this problem, we use, at most, the last \( T \) deployments of the recurrent neural nets for the computation of the derivative of \( \mu^\theta_{i,j} \) and \( \sigma^\theta_{i,j} \), forgetting about the previous ones, as illustrated in Figure 3. Put another way, \( T \) is a hyper-parameter that limits the number of time-steps for

![Figure 2: Graph of the recurrence flow in recurrent neural networks.](image-url)
The thick red line on the left represents the fact that $x_{i,j-T+1}^{θ_F(i)}$ is taken as a constant and not as a function of the previous recurrent network occurrences.

which the gradient is able to propagate through the internal states. That is, when differentiating $μ_{i,j}^θ$ and $σ_{i,j}^θ$ with $j > T$, we consider $x_{i,j-T+1}^{θ_F(i)}$ as a constant rather than as a function of $θ$, which effectively stops the differentiation process as the derivative of a constant is equal to 0. Instead of having an exact gradient which must be expressed as a function of all the elements in $h_{i,j}$, by stopping the differentiation at time-step $j - T$, the gradient estimate is expressed as a function of the elements in $[u_{i,j-T}, r_{i,j-T}, s_{i,j-T+1}, \ldots, s_{i,j}]$ and as a function of the constant value taken for $x_{i,j-T+1}^{θ_F(i)}$. However, when limiting the number of deployments of the neural net to maximum $T$, one must be very careful about two aspects. First, the value chosen for $T$ has to be sufficiently large to capture the temporal features contained in the data. For example, in the extreme case where $T = 1$, the derivative of the loss will be expressed using only the inputs of the current time-step and the current internal state (treated as a constant). This means that the gradient will never move $θ$ in a direction that improves the encoding of the internal state and thus, the network will never be able to compute temporal features. Second, when differentiating $μ_{i,j}^θ$ and $σ_{i,j}^θ$ with $j > T$ and considering $x_{i,j-T+1}^{θ_F(i)}$ as a constant, we found out that it is very important to use its true value when evaluating the gradient. Intuitively, $x_{i,j-T+1}^{θ_F(i)}$ appears in the derivative as a constant and can thus be seen as another network input that encodes all the past information of that episode, allowing the gradient to move $θ$ in a direction which also depends (although implicitly) on all the information gathered since the beginning of the episode. Our implementation of the gradient estimate respects the latter condition, that is, internal states are set to their true values when considered as constants.
In a meta-RL setting, the temporal correlation can be quite long as, for example, high rewards obtained at a specific instant may be the result of actions taken many steps previously. As the network needs to encode implicitly relevant information about \( P_{MDP}(s_{t+1}|s_t, u_t) \) and \( \rho_{MDP}(s_t, u_t, s_{t+1}) \), it may be necessary to work with relatively large values of \( T \). As a result, from a computational viewpoint, the previously detailed gradient estimate may remain quite cumbersome. To address this problem, we further modify our estimate of the gradient such as to make it more computationally efficient, but at the cost of accuracy in the gradients computation.

The modification is based on the following observation. When computing the gradient of \( \theta_i^{\rho_{i,j}} \) and \( \sigma_i^{\rho_{i,j}} \) with the above detailed procedure, we need to compute the derivative of \( x_i^{\rho_{i,k}}(k) \forall k \in [j−T+1, j] \) under the assumption that in the computational process, \( x_i^{\rho_{i,j}} \) is considered as being a constant. Now assume that computing the gradient of the loss function implies not only the computation of the gradients of \( \mu_i^{\rho_{i,j}} \) and \( \sigma_i^{\rho_{i,j}} \), but also the computation of the gradients of \( \mu_i^{\rho_{i,j'}} \) and \( \sigma_i^{\rho_{i,j'}} \) with \( j' \in [j−T+1, j] \). Actually, in such a situation, when computing those gradients, one has the possibility to recycle for the sake of computational efficiency the derivatives of \( x_i^{\rho_{i,j'}} \) obtained when computing the derivatives of \( \mu_i^{\rho_{i,j}} \) and \( \sigma_i^{\rho_{i,j}} \). This has the consequence that, in such a context, additional inaccuracies may occur since the gradient will not be anymore propagated over \( T \) time-steps but well over \( j'−j + T \) time-steps for each \( \mu_i^{\rho_{i,j'}} \) and \( \sigma_i^{\rho_{i,j'}} \) with \( j' \in [j−T+1, j] \). This computational trick will be used to compute the gradient of the surrogate loss for the actor \( (\nabla_\theta \mathcal{L}_{policy}(\theta)) \) and of the surrogate loss for the critic \( (\nabla_\psi \mathcal{L}_{sur}(\psi)) \). Let us now detail, precisely, how this is be done for update \( k \). For the actor, when differentiating \( \mathcal{L}_{policy} \), one needs in fact to differentiate \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) for all \( i \in [I(k), \ldots, I(k+1)−1] \) and \( j \in [0, \ldots, L'] \). One can thus use the computational trick for first evaluating simultaneously the gradient of \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) for all \( j' \) in \([0, \ldots, T−1] \). Then one uses the trick again to compute \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) for all \( j' \) in \([T, \ldots, 2*T−1] \) and so on, until evaluating the gradient of \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) for all \( j' \) in \([L, \ldots, L'] \) (note that, for this latter computation, we might evaluate the gradient on less than \( T \) pairs simultaneously). Afterwards, one proceeds to the computation of \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) for all \( j' \) in \([0, \ldots, T−1] \) and so on, until all required \( \sigma_i^{\theta_k} \) and \( \mu_i^{\theta_k} \) gradients have been evaluated. Let us now detail how we apply the same principle for computing \( \mathcal{L}_{sur} \). We stress that we simplify a bit the process here-below for ease of understanding. Indeed, in the following explanation, we consider that the sets of pairs \( \mathcal{Y}_m \) are always drawn from \( \mathcal{B}_{k-CRB} \) whereas they are in fact drawn from \( \mathcal{B}_{k-CRB} \setminus \mathcal{X}_m \). One can easily change the algorithm down below to address this aspect.

Let \( c_{i,j}^{\psi_k} = c_{\psi_k}(h_{i,j}) \) denote the output of the critic’s network at time-step \( j \) of episode \( i \). When differentiating \( \mathcal{L}_{sur} \), one needs to differentiate \( c_{i,j}^{\psi_k} \) for all \( [i, j] \) in a set \( \mathcal{Y} \) of \( CMB*T \) different pairs drawn from \([ [I(k−CRB), 0], \ldots, [I(k−CRB), L'], [I(k−CRB) + 1, 0] \ldots, [I(k + 1) − 1, L'] \)\) to exploit this computational trick, the set \( \mathcal{Y} \) is in fact not drawn randomly but rather so that we get \( CMB \) different batches of \( T \) consecutive pairs. To achieve this we first define the sets \( \mathcal{Z}_{i,x}^k \) such that \( \mathcal{Z}_{i,x}^k = [[i, x*T], \ldots, [i, \max((x + 1) *
\[ T - 1, L' \) for all \( i \in [I(k - CRB), \ldots, I(k + 1) - 1] \) and for all \( x \in [0, \ldots, \lfloor \frac{L'}{T} \rfloor] \). Then, for evaluating the loss' gradient, one uses the following procedure \( CMB \) times. First, one draws a set \( Z \) (which has to be different at each iteration) in \( [Z_{I(k - CRB)}^k, 0, \ldots, Z_{I(k + 1) - 1}^k, 0, \ldots, Z_{I(k + 1) - 1}^k, \lfloor \frac{L'}{T} \rfloor] \), afterwards one uses the computation trick to simultaneously evaluate the gradient for all \( c_{i,j}^k \forall [i', j'] \in Z \).

4. NMD NET

In Subsection 4.1, we first introduce the notations that will be used in this paper to describe neural networks. To this end, we also formalise the key mechanisms of artificial neural networks. Afterwards, in Subsection 4.2, we detail the NMD net architecture using the previously introduced notations. Finally, in Subsection 4.3 we highlight the main differences between an NMD net and a standard RNN.

4.1 ARCHITECTURE

4.1.1 Artificial neural network: a formalisation

A neural network (net) is a sequence of layers, each of which is composed of one or multiple neurons. Each neuron is represented by a function called an activation function, which has one or multiple inputs as well as one or multiple outputs. In the context of this paper, we will focus on neurons having a single output. The output of a layer can be seen as the union of its neurons outputs. Using the outputs of a layer of neurons as inputs of another layer is the core principle behind artificial neural nets. The strength of connections between neurons are defined by weights, which are the main parameters of a neural net. Let \( f^M_L \) denote the layer \( l \) of \( M \) neurons with an activation function \( f \). Let \( f^M_{L,i} \) denote the output of neuron \( i \) of layer \( l \). If neuron \( i \) of layer \( l \) has no input, we have \( f^M_{L,i} = f(b^i_l) \) where \( b^i_l \) is the bias of neuron \( i \) of layer \( l \). In the context of this paper we will mainly use the standard form of neural nets with fully connected layers (except for "Gated recurrent units" (GRU Cho et al. (2014)) layers and our neuromodulatory connections, which will be described below). When \( f^M_L \) is said to be fully connected to \( g^N_{L'} \), something we denote by \( f^M_L \rightarrow g^N_{L'} \), we can compute the output of the neurons of the layer \( g^N_{L'} \) as follows:

\[ g_{L',j}^N = g(b^j_{L'} + \sum_{i=1}^{M} w_{i,j}^{L',L} * f^M_{L,i}) \forall j \in [1, \ldots, N] \]

where \( w_{i,j}^{L,L'} \) is the weight that connects neuron \( i \) of layer \( L \) to neuron \( j \) of layer \( L' \). For recurrent networks, we also introduce the notation \( f^M_{L,i,t} \) to denote the output of neuron \( i \) of layer \( L \), containing \( M \) neurons, at time-step \( t \). Layer \( f^M_L \) can also be connected with itself through time, something we denote by \( f^M_L \odot \), in which case the output of a neuron of this layer at time \( t + 1 \) is given by:

\[ f_{L,i,t+1}^M = f(b^j_t + \sum_{i=1}^{M} w_{i,j}^{L,L} * f^M_{L,i,t}) \forall j \in [1, \ldots, M] \].
We emphasise that a layer can be connected to itself and to another layer. Using our now-extended notation, we write $f_l^M \rightarrow (g_l^K)_{j,t}$ to emphasize that the layer $l$ (having $M$ neurons and an activation function $f$) is connected to layer $l'$ (having $K$ neurons and an activation function $g$) which is also connected to itself through time. In such a context, the output of layer $l'$ at time $t$ is given by:

$$g_{l',j,t}^K = g(b_j^l + \sum_{i=1}^M w_{i,j}^l f_{l,i,t}^M + \sum_{i=1}^K w_{i,j}^l g_{l',i,t-1}^K) \forall j \in [1, \ldots, K].$$

The activation functions ($\mathbb{R} \rightarrow \mathbb{R}$) we use can be of different types, which include:

1. $ReLU(x) = \max(0, x)$. ReLU stands for "Rectified linear unit" and it is one of the most commonly used activation function Nair and Hinton (2010).
2. $sReLU(x) = \min(1, \max(-1, x))$, where sReLU stands for "saturated ReLU".
3. $sigmoid(x) = \frac{1}{1+e^{-x}}$
4. $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
5. $I(x) = x$.

Our proposed architecture will also be composed of GRU neurons (Cho et al. (2014)). GRU neurons will be the only type of recurrent neurons used in our networks. GRU neurons have more complex connections that require an extension of our notation. Let $GRU_l^M$ denote the layer $l'$ made of $M$ GRU neurons. All GRU neurons must be connected to themselves so as to encode temporal features. Let $f_l^N$ be a layer $l$ of $N$ neurons that is fully connected to a layer $l'$ of $M$ GRU neurons, something we denote by: $f_l^N \rightarrow GRU_l^M$.

Similar to the notations introduced before, $f_l^N \rightarrow GRU_l^M$ will also express the fact that the GRU layer is connected to itself through time. In order to compute the output of the GRU layer, we can implicitly build three intermediate layers $l'_1, l'_2, l'_3$, whose outputs will be used when computing the GRU layer output at time-step $t$ by the following four-step procedure:

1. $sigmoid_{l'_1,j,t}^M = sigmoid(b_j^{l'_1} + \sum_{i=1}^N w_{i,j}^{l'_1} f_{l,i,t}^N + \sum_{i=1}^M w_{i,j}^{l'_1} GRU_{l'_i,t-1}^M) \forall j \in [1, \ldots, M]$
2. $sigmoid_{l'_2,j,t}^M = sigmoid(b_j^{l'_2} + \sum_{i=1}^N w_{i,j}^{l'_2} f_{l,i,t}^N + \sum_{i=1}^M w_{i,j}^{l'_2} GRU_{l'_i,t-1}^M) \forall j \in [1, \ldots, M]$
3. $tanh_{l'_3,j,t}^M = tanh(\sum_{i=1}^N w_{i,j}^{l'_3} f_{l,i,t}^N + \sum_{i=1}^M w_{i,j}^{l'_3} GRU_{l'_i,t-1}^M \ast sigmoid_{l'_2,j,t}^M) \forall j \in [1, \ldots, M]$
4. $GRU_{l'_3,j,t}^M = (1.0 - sigmoid_{l'_1,j,t}^M) \ast tanh_{l'_3,j,t}^M + sigmoid_{l'_1,j,t}^M \ast GRU_{l'_3,j,t-1}^M \forall j \in [1, \ldots, M].$

As a final note, one needs to be reminded that artificial neural networks are a type of parametric function. In previous sections, we referred to the parameters of these parametric functions through the symbol $\theta$ for the actor and $\psi$ for the critic. Sufficient details regarding neural networks have now been given to provide the exact composition of the parameter vectors. In fact, the parameter vector $\alpha$ of any network as detailed above is the union of all
Weights $w$ and biases $b$. In Section 3.3, we introduced the concept of an internal state for a recurrent neural network. In this work, the only neurons having an internal state will be GRU neurons. As a result, the internal state of our networks at time-step $t$ will be equal to the union of all GRU neurons.

### 4.2 NMD net: a formalisation

An NMD net is made of two different parts: a recurrent network and a feedforward network. Both of these networks take dedicated inputs, interacting together through neuromodulatory connections. The goal of the recurrent network is to output features that are used to compute the parameters of the feedforward network activation functions. This requires the introduction of a parametric activation function ($\mathbb{R} \to \mathbb{R}$) for the feedforward network. We call this function $PSR$ (standing for "Parametric Saturated ReLU") and define it as follows:

$$PSR(x) = sReLU(m \cdot x + a)$$

where $m \in \mathbb{R}$ and $a \in \mathbb{R}$ are parameters that drive the slope and the offset of a saturated ReLU, respectively. All the neurons of the feedforward network will have this type of parametric activation function except for the output layer. For this layer, we use another, unbounded parametric activation function in order to avoid constraining actor and critic output to the range $[1, 1]$. The parametric activation function chosen for the output layer is called $PI(x)$, where the acronym $PI$ stands for “Parametric Identity” and is defined as follows:

$$PI(x) = (m \cdot x + a) .$$

The feedforward network of our NMD net will be composed of multiple $PSR$ layers followed by a $PI$ layer. Neurons of all layers are being "neuromodulated" by the recurrent network. This neuromodulatory connection forms the core of our NMD net. To formalise this type of connection, let us denote by $PSR^M_l$ a layer $l$ of $M$ neurons with an activation function $PSR$ and by $f^N_{l'}$ a layer $l'$ of $N$ neurons with activation function $f$. When $f^N_{l'}$ neuromodulates $PSR^M_l$, which we denote by $f^N_{l'} \circ PSR^M_l$, the output of every neuron $j \in [1, \ldots, M]$ in layer $PSR^M_l$ at time $t$ can be computed using the following three-step procedure:

1. $I^M_{l,m,j,t} = 1.0 + \sum_{i=1}^{N} w^{'N}_{i,j} \cdot f^N_{l',i,t}$. The goal of this layer is to compute the value of the parameter $m$ at time $t$ of each neuron in $PSR^M_l$.

2. $I^M_{l,a,j,t} = \sum_{i=1}^{N} w^{'N}_{i,j} \cdot f^N_{l',i,t}$. The goal of this layer is to compute the value of the parameter $a$ at time $t$ of each neuron in $PSR^M_l$.

3. $PSR^M_{l,i,j,t} = sReLU(b^l_j + I^M_{l',j,t} + I^M_{l,m,j,t} \cdot x_{j,t})$. Here, $x_{j,t}$ represents standard input of neuron $j$ at time-step $t$ computed from the output of any other layer. For example, if we have $g^K_l \to PSR^M_l$, then $x_{j,t} = \sum_{i=1}^{K} w^{'N}_{i,j} \cdot g^N_{l',i,t}$.
Using all the previous notations, we can now give a readable view of the NMD net architecture. This architecture depends on hyper-parameters, such as the number of layers in the feedforward network or the recurrent network, or the number of neurons in a layer.

To fully describe this architecture, let us first denote by $I_{n+m+1}^n \in \mathbb{R}^{n+m+1}$ the input layer of the recurrent part where $n$ is the state space dimension and $m$ the action space dimension. The dimension of the input layer is equal to $n+m+1$ as, in the context of meta-RL, the recurrent part of the NMD nets take, at time-step $t$ of episode $i$, the tuple $(s_{t-1}, u_{t-1}, r_{t-1})$ as inputs. Using our notations, this means that for a given episode $i$: $I_{n+m+1}^n = [s_{i,t-1}, u_{i,t-1}, r_{i,t-1}]_j \forall j \in [1, \ldots, n+m+1]$ where $[s_{i,t-1}, u_{i,t-1}, r_{i,t-1}]_j$ denotes the $j$th index of the vector $[s_{i,t-1}, u_{i,t-1}, r_{i,t-1}]$. In a standard RNN used in a meta-RL setting, the recurrent part would normally take $(s_{i,t}, u_{i,t-1}, r_{i,t-1})$ as inputs, as is the case for example in Wang et al. (2016). This choice is discussed in next subsection. Let $I_{0}^n$ denote the input layer of the feedforward part. In the context of meta-RL, the feedforward part of our NMD net takes $s_{i,t}$ as input at time-step $t$ of episode $i$. From here, we define our NMD net through multiple sequences of layers: (i) the recurrent part that outputs the NMD features; (ii) the connections between the NMD features and the feedforward part; and (iii) the feedforward part that outputs the NMD net’s output.

(i) The recurrent part is defined by the following sequence:

$$I_{n+m+1}^n \xrightarrow{gru_{n+1}} (GRU_{r_1}^{R_1} \circ ) \xrightarrow{gru_{m+1}} (GRU_{r_2}^{R_2} \circ ) \xrightarrow{gru_{m+1}} \ldots \xrightarrow{gru_{m+1}} (GRU_{r_{KR}}^{R_{KR}} \circ ).$$

where $KR \in \mathbb{N}_0$ is a hyper-parameter that drives the number of recurrent layers and where $R_i \forall i \in [1, \ldots, KR]$ are hyper-parameters that determine the number of neurons in the $KR$ layers.

(ii) The NMD connections, which are at the core of the proposed architecture, can be expressed as multiple small sequences:

$$\left\{ (GRU_{r_{KR}}^{R_{KR}} \circ ) \rightarrow ReLU_{T_{1_i}} \xrightarrow{o} PSR_{F_{1_i}} \right\}$$

$$\left\{ (GRU_{r_{KR}}^{R_{KR}} \circ ) \rightarrow ReLU_{T_{2_i}} \xrightarrow{o} PSR_{F_{2_i}} \right\}$$

$$\ldots$$

$$\left\{ (GRU_{r_{KR}}^{R_{KR}} \circ ) \rightarrow ReLU_{T_{KF_i}} \xrightarrow{o} PSR_{F_{KF_i}} \right\}$$

$$\left\{ (GRU_{r_{KF}}^{R_{KF}} \circ ) \rightarrow ReLU_{T_{KF+1_i}} \xrightarrow{o} PSR_{F_{KF+1_i}} \right\}$$

where $KF \in \mathbb{N}_0$ is a hyper-parameter that drives the number of feedforward layers; $T_i \forall i \in [1, \ldots, KF + 1]$ are hyper-parameters that provide the number of NMD features used per corresponding feedforward layer; and $F_i \forall i \in [1, \ldots, KF]$ are hyper-parameters that give the size of the feedforward layers. Note that $F_{KF+1}$ is not a hyper-parameter and is rather problem-dependant, as explained below in (iii). Please note that we chose to have the same number of NMD features per feedforward layer, i.e. $T_i = C \forall i \in [1, \ldots, KF]$ with $C \in \mathbb{N}_0$.

(iii) Finally, the feedforward part is defined by the following sequence:

$$...
Figure 4: Sketch of a policy modelled as a NMD net in the context of our meta-RL setting. Neuromodulation connections are denoted by $\rightarrow$ symbols.

\[
I^n f_0 \rightarrow PSR^{F_1} f_1 \rightarrow PSR^{F_2} f_2 \rightarrow \ldots \rightarrow PSR^{F_{K+1}} f_{K+1} \rightarrow PI^{F_{K+1}} f_{K+1}.
\]

The output of the layer $PI^{F_{K+1}} f_{K+1}$ is the output of the NMD net, which is why $F_{K+1}$ is in fact not a hyper-parameter. Indeed, for the critic we always have $F_{K+1} = 1$ as the critic only outputs a scalar rating ($c(\cdot)$) and for the actor we have $F_{K+1} = 2 \cdot m$ where $m$ is the action space’s dimension. $F_{K+1} = 2 \cdot m$ stands for the fact that the network outputs both a mean vector ($\mu(\cdot)$) and a standard deviation vector ($\sigma(\cdot)$) both of dimension $m$. Figure 4 sketches a NMD net used to approximate a policy in our meta-RL setting.

4.3 NMD net and RNN: a comparison

**RNN definition.** Let us now define a standard RNN (as the one used in Wang et al. (2016) for solving a meta-RL problem) using our previously introduced notations. Let us first introduce the input layer $I^n_{f_0}$. In a meta-RL context, at time-step $t$ of episode $i$, standard RNNs take the tuple $(s_{i,t}, u_{i,t-1}, r_{i,t-1})$ as input. That is, for a given episode $i$,
In this paper, we define our standard RNNs thanks to the following sequence of layers:

\[ I_{t_0 + m + 1}^{n + m + 1} \rightarrow (GRU_{i_1}^{R_i}) \rightarrow \ldots \rightarrow (GRU_{i_{K'}^{R'}}) \rightarrow sReLU_{i_1}^{F_1} \rightarrow \ldots \rightarrow sReLU_{K'}^{F_{K'}} \rightarrow I_{K' + 1}^{F_{K' + 1}} \]

The symbol ”′” is used to differentiate the architecture hyper-parameters of the RNN to those of the NMD net, allowing them to be tuned independently. This will allow to provide the fairest possible comparison. Furthermore, we stress that we use \(sReLU\) activation functions for the non-recurrent neurons whereas in the literature \(ReLU\) or \(sigmoid\) are mainly used. We ran multiple tests with these two latter activation functions and found the results to be either similar or worse than when using \(sReLU\) functions. As \(PSR\) and \(sReLU\) activation functions are rather similar, we decided to continue using \(sReLU\) activation function, again to have the fairest possible comparison.

There are two significant differences between the NMD nets and the standard RNNs.

First, in the NMD architecture, at time-step \(t\) of episode \(i\), the NMD net has two distinct input vectors. One is composed of \((s_{i,t-1}, u_{i,t-1}, r_{i,t-1})\) and the other is solely made of \((s_{i,t})\). As a comparison, in classic RNN, there is only a single input vector comprising \((s_{i,t}, u_{i,t-1}, r_{i,t-1})\) Wang et al. (2016). The presence of two input vectors in the NMD architecture comes from the extra degree of freedom provided by the neuromodulatory connections. Having both input vectors allows for a more intuitive and biologically motivated split in the inputs, which allows each part of the network to play a clearly distinct role. On the one hand, the recurrent part of the network is only given past information, which can be used by the network to adapt to the current environment by coding the current MDP’s characteristics. On the other hand, the feedforward network is given the current observations and thus represents the agent’s current ”behaviour” (which is adapted through time by the NMD process). The feedforward part can, thus, be seen as a network of its own, which is adapted by the NMD process to optimally perform on the current MDP. Using \((s_{i,t}, u_{i,t-1}, r_{i,t-1})\) as inputs for the recurrent part instead of \((s_{i,t-1}, u_{i,t-1}, r_{i,t-1})\) also works, but often led to inferior performance in our test examples. This mainly led to similar results, but sometimes worse. Seeing the results and how intuitive the previously mentioned split is from a neuroscience perspective, we decided to select the input vector \([s_{i,t-1}, u_{i,t-1}, r_{i,t-1}]\).

Second, and most importantly, the way the recurrent part is connected to the feedforward part is vastly different in both architectures. This really is where our contribution lies. In standard RNNs, the features of the final GRU layer are fed in a fully connected fashion to the feedforward part. In an NMD network, the features output by the GRU layer are used to compute activation function parameters that drive each neuron of each layer in the feedforward part.

5. RESULTS

In this section, our algorithms are tested on three different benchmark problems. In Subsection 5.1, we start by detailing our validation methodology and discuss the choice of our
hyper-parameters, especially those concerning the networks’ architecture. In Subsection 5.2, we describe our first benchmark, which is a toy problem where an agent simply needs to select, at every time step, an action that belongs to a state-dependent target interval to maximise its return. This subsection also reports simulation results obtained on the benchmark problem. In Subsection 5.3 and 5.4, we study the two other benchmarks, which involves navigating a 2D map with varying conditions. For clarity, non-exhaustive descriptions of the benchmarks are given in this section. We refer the reader to Appendices A.1, A.2 and A.3 for a fully detailed and structured description of the three benchmarks. We also refer the reader to "https://github.com/nvecoven/nmd_net" for a full implementation of the benchmarks and algorithms.

5.1 VALIDATION METHODOLOGY

Our actor-critic algorithm approach for solving our meta-RL problem will be tested on every benchmark problem with two types of neural nets: the NMD net and classical RNN, as described in the previous section. To this end, we have adopted an experimental protocol that will be used for each benchmark problem. This protocol includes the choice of the hyper-parameters of the algorithm, which we first discuss in this subsection. Afterwards, we discuss the different measures that will be used to assess the performance of the different algorithms.

**Choice of hyper-parameters.** In order to keep the number of tests within a reasonable range and to limit the tuning of hyper-parameters, we use the exact same actor-critic algorithm parameters for each benchmark, except for $L$, $L'$ and the architecture-related hyper-parameters. To select the value of these hyper-parameters, we proceed as follows.

Let us start by discussing the strategy used for selecting an appropriate value for $L'$. We established that for our algorithm to perform well, $L'$ needs to be significantly greater than the time needed for a Bayes optimal policy to start behaving similarly to an optimal policy $\pi^*_\text{MDP}$ \(^2\) This can be explained as follows. First, $L'$ bounds the size of the trajectories used for updating the parameters of the actor and the critic. For the update to be performed in the right direction, the trajectories need, as a minimum, to contain information both about the adaptation phase (that we loosely relate to the phase needed for a (near-) Bayes optimal policy to output actions similar to those of an optimal policy) and the phase that follows during which the agent follows (near-)optimal trajectories. Second, during the first episodes, the actor-critic algorithm, during the first episodes, will play policies that are far away from Bayes optimal ones, and should be able to exploit a sufficiently long piece of trajectory to learn to adapt, hence another reason for selecting a large value of $L'$. We note, however, that the computing times are increasing with $L'$. For the first benchmark, for which it is possible to rapidly generate relevant information about the MDP drawn by observing the trajectories, we have chosen a value of $L' = 400$. For the two others, $L' = 2000$

\(^2\) In this section, when referring to an optimal policy, we refer to a policy which is optimal when played on an MDP which is fully known. When referring to a Bayes optimal policy we will do so explicitly. We note that the performance of any meta-RL algorithm will be worse on our meta-RL problem than the performances of a Bayes optimal policy.
has been chosen.

Now, concerning the choice of $L$, we first need to remember that $L$ must be large enough in comparison to $L'$ so as to have a good estimate $\hat{R}_{i,j} = \sum_{t=j}^{\infty} \gamma^t * r_{i,t}$ for all $j \in [0, \ldots, L']$ where $\hat{R}_{i,j} = \sum_{t=j}^{L-1} \gamma^t * r_{i,t}$. Since we can prove that $\hat{R}_{i,j}$ stands in the interval $[R_{i,j} − \frac{\gamma}{1-\gamma} R_{max}, R_{i,j} − \frac{\gamma}{1-\gamma} R_{min}]$, we will therefore choose $L$ so as to have $\frac{\gamma(L-L')}{1-\gamma}(R_{max} - R_{min})$, the maximum size of the interval which is relatively small. The value of $L$ chosen for a benchmark will be a value of $L$ that leads to a size of this interval that is below 1000 (a small interval regarding the magnitude of $\hat{R}_{i,j}$ for each of the three benchmarks). In this paper, we chose $L = 1400$ for the first benchmark, $L = 4000$ for the second benchmark and $L = 5000$ for the third one.

Concerning the architecture-related parameters, we emphasise that, as mentioned in Subsection 4.3, we always choose them so that both networks are as similar as possible. To this end, we always make the following choices. (i) The number of GRU layers and GRU neurons chosen are equal in both RNN and NMD nets, that is, $KR = KR'$ and $R_i = R'_i \forall i \in [1, \ldots, KR]$. (ii) The NMD connections and the feed-forward part of the NMD net are taken such that $KF = KF' + 1$, $F_i = F'_i + 1 \forall i \in [0, \ldots, KF]$ and $\sum_{i=1}^{KF+1} t_i = F'_0$. The exact values of these architecture-related hyper-parameters will however be problem dependent and given later on. We note that with the choices made here above, the number of parameters $\omega$, $\beta$, and $\sum_{i=1}^{KF+1} t_i$ to be learned by the actor-critic algorithm will vary by at most 0.3% between the two types of neural nets.

Finally, for the other hyper-parameters of the actor-critic algorithms, we have set them by trial and error or by borrowing default values that were reported/suggested in other papers, such as in the case of the parameters: $\omega_1$, $\omega_2$, $\epsilon$, $\beta_0$ and $d_{targ}$. The values used for these other hyper-parameters are given in Appendix C, Table 4.

**Results reported.** The objective in our meta-RL setting is to maximise the expected value of the sum of returns the agent can obtain over $E$ episodes, namely:

$$
\sum_{E-1}^{E-1} \sum_{i=0}^{E-1} R_{i,j}^{\pi_{i,j}}
$$

where $R_{i,j}^{\pi_{i,j}} = \sum_{i=0}^{\infty} \gamma^t * r_{i,t}$. To estimate this sum of expected returns, we will run our actor-critic algorithm $A$ times over $E$ episodes, with the infinite horizon of each episode truncated to $L$, and average the results obtained. To formalise this, let $\hat{R}_{i,0}^k$ and $r_{i,t}^k \forall k \in [1, \ldots, A], \forall i \in [0, \ldots, E-1]$ denote the value of $\hat{R}_{i,0}$ and the value of $r_{i,t}$ for the $k$th run, respectively. With these notations, we define $R_A$, which we will use as the estimate of (15), as:

$$
R_A = \sum_{k=1}^{A} \sum_{i=0}^{E-1} \hat{R}_{i,0}^k
$$

26
We stress that the quality of this estimate will depend on $L$ and $A$. Given the choice of $L$ made (see previous paragraph on the choice of the hyper-parameters), the truncation of the time horizon should only bias the estimate of the expected return in a very minor way (see the bounds on $\hat{R}_{i,0}$ given $L$ provided in previous paragraph). As for the choice of $A$, this parameter has to be chosen sufficiently large to lead to an accurate estimate of the expectation of the sum of returns over $E$ episodes. We emphasise that the sum of returns over $E$ episodes varies from one run to the other of the algorithm owing to four sources of variance: (i) the stochasticity of the agent policy; (ii) the sampling of MDPs from $\eta$; (iii) the transition function stochasticity; (iv) the sampling of the initial value of the networks parameters from a given distribution before each run of the actor-critic algorithm (sampling this distribution amounts to sample independently each $w_{i,*}$ and $b_{i}$ from a truncated normal distribution). Information about the variance in the results will also be provided through the computation of the following values:

1. $R_A^{\text{max}} = \max_{k \in [1, \ldots, A]} \sum_{i=0}^{E-1} \hat{R}_{i,0}^{k}$ which gives the meta-RL algorithm performance for the best run.
2. $R_A^{\text{min}} = \min_{k \in [1, \ldots, A]} \sum_{i=0}^{E-1} \hat{R}_{i,0}^{k}$ which gives the meta-RL algorithm performance for the worst run.
3. $R_A^{\text{std}} = \sqrt{\sum_{k=1}^{A} (\sum_{i=0}^{E-1} \hat{R}_{i,0}^{k} - R_A)^2}$ which is the estimation of the variance of the performances of the meta-RL algorithm over the $A$ runs.

For each benchmark, we will report the values of $R_A$, $R_A^{\text{std}}$, $R_A^{\text{max}}$ and $R_A^{\text{min}}$ in a table for both NMD net and RNN architectures as detailed in Section 4. For every benchmark, we will also report the following quantities on a figure, again both for a NMD architecture and for a classical RNN architecture and for every value of $i$:

1. The value of the average return for episode $i$, namely: $\sum_{k=1}^{A} \hat{R}_{i,0}^{k}$
2. The maximum return observed over $A$ runs for episode $i$, namely $\max_{j \in [1, \ldots, A]} \hat{R}_{i,0}^{j}$
3. The minimum return observed over $A$ runs for episode $i$, namely $\min_{j \in [1, \ldots, A]} \hat{R}_{i,0}^{j}$

Finally, to illustrate the adaptive performances of the policy learned by our actor-critic algorithm, we will, for the first benchmark, more carefully evaluate the characteristics of the policy $\pi_{\theta_{F}(E)}$. For this, we will carry out additional simulations. First, we will evaluate the expected return of the policy $\pi_{\theta_{F}(E)}$ when it has to play a MDP drawn at random from $\eta$. To this end, we will play $A$ episodes with policy $\pi_{\theta_{F}(E)}$, learned by a typical run of the algorithm, on $A$ different MDPs sampled from $\eta$ and compute

$$\sum_{i=1}^{A} \sum_{t=1}^{L} \gamma^t * r'_{i,t}$$

where $r'_{i,t}$ is the reward obtained at time-step $t$ of episode $i$ played with the policy $\theta_{F(E)}$. Afterwards, we will compare this value with the value of the expected return that can be
obtained by (i) playing a Bayes optimal policy (ii) playing for each of the $A$ episodes a policy which is optimal with respect to the MDP corresponding to this episode. As we will see, when choosing a sufficiently large value of $E$, the expected return of the policy $\pi_{\theta_{F(E)}}$ will be close to that of a Bayes optimal one. We note however, that, as mentioned earlier in Section 3, after a certain number of time-steps corresponding to the end of the adaptation phase, the learned policy $\pi_{\theta_{F(E)}}$, if close to a Bayes optimal one, should behave in a similar way to a policy that is optimal with respect to the MDP played. To illustrate this, all the time-steps for which the action taken by $\pi_{\theta_{F(E)}}$ differs from an optimal action will be displayed in a figure and for a few MDPs drawn from $\eta$.

5.2 STATE-DEPENDENT TARGET INTERVAL BENCHMARK

Benchmark details. For this relatively simple benchmark, both the state and action spaces are one-dimensional. If the agent outputs an action that belongs to a state-dependent target interval, it obtains a reward of 10 and the next state is drawn at random. If the interval is not reached, then a negative reward proportional to the distance between the agent action and the target interval centre is received and the state does not change. Before moving any further, we note that in the context of this benchmark, drawing $MDP_i$ from $\eta$ amounts to drawing a scalar $\alpha_i$ in $[-10, 10]$ and that all the $MDP_i$ in the support of $\eta$ are fully identifiable from the value of $\alpha_i$ to which they correspond. Concerning the transition function, when being in the state $s_{i,t}$ and taking action $u_{i,t}$, the next state is computed as follows. If $u_{i,t}$ is in the target interval $[s_{i,t} + \alpha_i - 1, s_{i,t} + \alpha_i + 1]$, that we refer to as target, then $s_{i,t+1}$ is drawn from $\mathcal{U}[-5, 5]$. Otherwise, $s_{i,t+1} = s_{i,t}$. The reward function is defined as follows:

$$\rho_{MDP_i}(s_{i,t}, u_{i,t}, s_{i,t+1}) = \begin{cases} 10 & \text{if } u_{i,t} \in \text{target} \\ -|u_{i,t} - (s_{i,t} + \alpha_i)| & \text{otherwise} \end{cases}$$

When playing $MDP_i$, any policy that plays an action $u_{i,t} \in \text{target}$ at time-step $t$ of episode $i$ is an optimal one. Of course, in the context of our meta-RL problem, the MDP to be played is unknown and one can expect our meta-RL algorithm to converge to a Bayes optimal policy at best, as well as its expected return. In the particular context of this benchmark, it is possible to analytically compute a Bayes optimal policy. We refer the reader to Appendix A.1 for more information about this. We now discuss the performances of our algorithm.

Architecture. This benchmark is relatively simple and a small architecture should therefore produce good performances. The results of this benchmark reported below were obtained with the following architecture hyper-parameters:

<table>
<thead>
<tr>
<th>RNN</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$KR$</td>
<td>$KR$</td>
</tr>
<tr>
<td>$r_1'$</td>
<td>$r_1$</td>
</tr>
<tr>
<td>$KF'$</td>
<td>$KF$</td>
</tr>
<tr>
<td>$f_1'$</td>
<td>$t_1$</td>
</tr>
<tr>
<td>$f_2'$</td>
<td>$f_1$</td>
</tr>
<tr>
<td></td>
<td>$t_2$</td>
</tr>
</tbody>
</table>
Figure 5: State-dependent target interval benchmark. The agent has to play an action that belongs to the target. The value of $\alpha$ changes for every MDP belonging to the support of $\eta$. Only positive rewards are observed if the agent selects an action that falls within the target interval.

**Results.** Let us first discuss Table 1 that gathers the values of $R_A$, $R_A^{\text{min}}$, $R_A^{\text{max}}$ and $R_A^{\text{std}}$ for both the RNN and the NMD nets. On one hand, we see that runs are more efficient with NMD net architectures than with classical recurrent architectures. Indeed, we have that $R_A$ is equal to 9.46E+07 for the classic recurrent architecture and equal to 1.35E+08 for the NMD net, which highlights that the average performance of the algorithm is far greater for NMD net architectures than for classic architectures. On the other hand, the performances of the NMD net are much less run dependent than with classical RNNs. This is shown by the fact that $R_A^{\text{std}}$ is equal to 2.69E+07 for the classic recurrent architecture and only equal to 9.39E+06 for the NMD net. Let us now discuss Figure 6 that plots, for both architectures, a running mean (over 100 episodes) of the average return (averaged over $A$ runs), the minimum return and the maximum return for each episode $i$. This figure highlights that the best run with the classic recurrent architecture only provides the performance of the average NMD net run and that the worst NMD net run still performs better than the average classic recurrent net run.
Table 1: (Expected) returns of the actor-critic algorithm for both architectures and variance of the returns.

<table>
<thead>
<tr>
<th></th>
<th>classic recurrent</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A$</td>
<td>9.46E+07</td>
<td>1.35E+08</td>
</tr>
<tr>
<td>$R_A^{\text{min}}$</td>
<td>1.25E+07</td>
<td>1.05E+08</td>
</tr>
<tr>
<td>$R_A^{\text{max}}$</td>
<td>1.27E+08</td>
<td>1.46E+08</td>
</tr>
<tr>
<td>$R_A^{\text{std}}$</td>
<td>2.69E+07</td>
<td>9.39E+06</td>
</tr>
</tbody>
</table>

Figure 6: Evolution of the expected, maximum and minimum value of $\hat{R}_{i,0}$ with respect to $i$ for both architectures. The plot is smoothed out thanks to a running mean over 100 episodes and made out of 15 different runs for each architecture.

For completeness, Figure 7 shows the curves of Figure 6 without applying a running mean. In this case we see that there is a reasonably high variance in between episodes themselves for a same run. This variance is even larger for the classical RNNs.
Comparison to a Bayes optimal policy. In order to estimate the adaptation capabilities of the NMD net, we now compare it to an optimal Bayes policy. For this benchmark, such a policy, and the expected reward it provides, can be computed and are given in Theorem 1 and Theorem 2 of Appendix D, respectively. Let $\pi_{\text{final}}$ denote a policy $\pi_{\theta_F(E)}$ sampled at random from the $A$ runs previously used to evaluate the performance of the algorithm. In order to compare the performances of $\pi_{\text{final}}$ to the performances of a Bayes optimal policy, we play $\pi_{\text{final}}$ over $A$ different episodes (corresponding to $A$ different MDPs) and average the sum of discounted rewards obtained over these $A$ episodes. This leads to an average sum of discounted rewards equal to 4534, which can be compared to the expected sum of discounted rewards of a Bayes optimal policy, which is equal to 4679.1 when playing benchmark 1 with $L = 1400$. From these numbers, it appears that the policy learned when using NMD nets as function approximators reaches near-optimal performances on the state-dependent target interval benchmark. Figure 8 displays for multiple episodes the time-steps for which the agent does not play optimally. As one can see in Figure 8.a, the agent only needs very few time-steps to adapt and start playing optimally. Furthermore, one can see
that once the agent has adapted, it always plays either in the target interval (dark-blue colour on the figure), or very close (slightly lighter colour).

5.3 NAVIGATION WITH A WIND CONE BENCHMARK

Benchmark details. For this benchmark, the agent has to navigate in a bounded two-dimensional space in which lies a single target (represented by the green circle on Figure 9). Whenever the agent reaches the target interval, its next position is drawn randomly in the two-dimensional space and it receives a high reward. We note that the main characteristic of this benchmark is that the agent’s movement from one time-step to another time-step is affected by a “wind cone”. As shown on Figure 9, at each-time step, the agent outputs an intended move (red vector on Figure 9), then a wind vector is drawn uniformly at random in the cone defined (purple vector on Figure 9) and the resulting move of the agent is the vectorial sum of the intended move and the wind vector (orange vector on Figure 9). For this benchmark, the MDPs in $\eta$ differ by the wind cone direction and by the position of the target interval on the map. Here, the state space is two-dimensional (the dimensions are denoted $s_1$ and $s_2$ on Figure 9) and the state gives the relative position of the agent to the target interval. An action $u$ gives the direction (in radians) of the agent’s intended move.

Architecture. This benchmark is more complex than the previous one. The state-space is larger and in terms of adaptation, the task is also significantly more complex. Indeed, for a correct adaptation, the agent needs to be able to identify the direction of the wind cone, which is not a straightforward process since it does not get direct information about the wind cone itself. Rather, the agent only gets access at each time-step, and even in an indirect way, to a wind vector which is sampled in this cone. Consequently, we have reported results here that correspond to a bigger architecture, corresponding to the following parameters:

<table>
<thead>
<tr>
<th>RNN</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$KR'$</td>
<td>2</td>
</tr>
<tr>
<td>$r'_1$</td>
<td>100</td>
</tr>
<tr>
<td>$r'_2$</td>
<td>75</td>
</tr>
<tr>
<td>$KF'$</td>
<td>3</td>
</tr>
<tr>
<td>$f'_1$</td>
<td>45</td>
</tr>
<tr>
<td>$f'_2$</td>
<td>30</td>
</tr>
<tr>
<td>$f'_3$</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results. Let us first analyse Table 2 that provides information about the expected returns. As we can see, $R_A$ is equal to 1.35E+08 for the classic recurrent architecture and equal to 1.47E+08 for the NMD net, which shows again that the average performance of the algorithm is greater for NMD net architectures than for classic architectures. As for the state-dependent target interval benchmark, Table 2 also shows that results are much less run-dependent with our proposed architecture as $R^{std}_A$ is only equal to 1.54E+07 for NMD net architecture whereas it is equal to 2.62E+07 for classic recurrent networks. These observations are further supported by Figure 10 that, for both architectures, plots a running mean (over 1000 episodes) of the average return (averaged over 15 runs), the minimum
(a) Plot of the distance from the target interval for the actions taken by $\pi_{\text{final}}$ on the 40 first time-steps of 20 episodes.

(b) Plot of the distance from the target interval for the actions taken by $\pi_{\text{final}}$ on the 1400 first time-steps of 20 episodes.

Figure 8: Display of the adaptation capabilities of the NMD net. On (a) and (b) are displayed the distances of the agent action from the closest bound of the target interval for each time-step of 20 different episodes (and consequently MDPs). Dark blue corresponds to an action that stands in the target interval and which is therefore an optimal action. The lighter the colour of blue, the further the action is from the target interval and the more suboptimal it is.
Figure 9: An illustration of the navigation with a wind cone benchmark. At each time-step a wind vector is drawn uniformly in the wind cone and impacts the agent’s move.

return and the maximum return for each episode $i$. Let us now point out an interesting feature of the NMD net architecture by discussing Figure 11 which displays only the first 6000 time-steps of Figure 10. As one can see, on this benchmark, the NMD net shows poorer performances than the classical RNN during the first episodes. However, it still manages to outperform recurrent networks after about 3500 episodes. This suggests that, although more complicated to train, NMD nets are better suited for adaptation.

<table>
<thead>
<tr>
<th></th>
<th>classic recurrent</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A$</td>
<td>1.35E+08</td>
<td>1.47E+08</td>
</tr>
<tr>
<td>$R_A^{\min}$</td>
<td>8.74E+07</td>
<td>1.19E+08</td>
</tr>
<tr>
<td>$R_A^{\max}$</td>
<td>1.70E+08</td>
<td>1.74E+08</td>
</tr>
<tr>
<td>$R_A^{std}$</td>
<td>2.62E+07</td>
<td>1.54E+07</td>
</tr>
</tbody>
</table>

Table 2: (Expected) returns of the actor-critic algorithm for both architectures, and variance of the returns.
Figure 10: Evolution of the expected, maximum and minimum value of $\hat{R}_{i,0}$ with respect to $i$ for both architectures. The plot is smoothed out thanks to a running mean over 1000 episodes and made out of 15 different runs for each architecture.
5.4 NAVIGATION TOWARDS THE CORRECT TARGET BENCHMARK

**Benchmark details.** In this benchmark, the agent has to navigate within a bounded two-dimensional space in which lie two targets (represented by target\(_1\) and target\(_2\) on Figure 12). One of these targets is associated with a high reward (green circle on Figure 12) and the other with a low reward (red circle on Figure 12). Whenever the agent reaches a target, it receives either a positive or negative reward, depending on the target, and its next position is randomly sampled in the two-dimensional space outside of both targets. The MDPs belonging to the support of \(\eta\) differ by the positions of the targets and the rewards with which they are associated. The state space is four-dimensional (the dimensions are denoted \(s_1, s_2, s_3\) and \(s_4\) on Figure 12) and the action space is one-dimensional (denoted \(u\) on Figure 12). The four components of a state give the relative position of the agent to the two target centres and the action determines the direction in which the agent moves (red vector on Figure 12) at each time-step.

**Architecture.** In this section, we report results that correspond to the architecture used in the previous benchmark. As a reminder, this architecture is defined as:
Figure 12: Navigation towards the correct target benchmark. A target associated with a red colour gives a negative reward whereas a green target gives a positive reward.

<table>
<thead>
<tr>
<th>RNN</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$KR'$</td>
<td>2</td>
</tr>
<tr>
<td>$r'_1$</td>
<td>100</td>
</tr>
<tr>
<td>$r'_2$</td>
<td>75</td>
</tr>
<tr>
<td>$KF'$</td>
<td>3</td>
</tr>
<tr>
<td>$f'_1$</td>
<td>45</td>
</tr>
<tr>
<td>$f'_2$</td>
<td>30</td>
</tr>
<tr>
<td>$f'_3$</td>
<td>10</td>
</tr>
</tbody>
</table>

**Results.** Multiple observations deserve to be made on this benchmark. Let us start by analysing Table 3. First, we see that the best run of classic RNN architectures ($R^{\text{max}}_A = 8.94\text{E+07}$) outperforms that of NMD nets ($R^{\text{max}}_A = 8.29\text{E+07}$). Second, despite this observation, the average performance of NMD nets ($R_A = 7.6\text{E+07}$) still outperforms by a decently large margin that of classic RNNs ($R_A = 7.13\text{E+07}$). Third, and most importantly, performances are notably less run-dependent using NMD nets than using classic RNNs. Indeed, we have $R^{\text{std}}_A = 1.31\text{E+07}$ for classic RNNs and $R^{\text{std}}_A = 4.41\text{E+06}$ for NMD nets. Furthermore, $R^{\text{min}}_A$ can even go as low as 4.07E+07 for RNNs whereas $R^{\text{min}}_A$ only goes as low as 6.57E+07 with NMD nets. These observations are further supported by Figure 13 that, for both architectures, plots a running mean (over 1000 episodes) of the average return (averaged over 15 runs), the minimum return and the maximum return for each episode $i$. We underline that recurrent networks often have very poor performances and are very susceptible to the run stochasticity, whereas NMD nets remain very consistent throughout
the 15 runs. As a final note, we see that in the very early stages of learning, NMD nets tend to perform worse than RNNs and catch up after around 4500 episodes. This is the same observation as for the previous benchmark.

<table>
<thead>
<tr>
<th></th>
<th>classic recurrent</th>
<th>NMD net</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_A$</td>
<td>7.13E+07</td>
<td>7.60E+07</td>
</tr>
<tr>
<td>$R_A^{\min}$</td>
<td>4.07E+07</td>
<td>6.57E+07</td>
</tr>
<tr>
<td>$R_A^{\max}$</td>
<td>8.94E+07</td>
<td>8.29E+07</td>
</tr>
<tr>
<td>$R_A^{\text{std}}$</td>
<td>1.31E+07</td>
<td>4.41E+06</td>
</tr>
</tbody>
</table>

Table 3: (Expected) returns of the actor-critic algorithm for both architectures, and variance of the returns.

Figure 13: Evolution of the expected, maximum and minimum value of $\hat{R}_{i,0}$ with respect to $i$ for both architectures. The plot is smoothed out thanks to a running mean over 1000 episodes and made out of 15 different runs for each architecture.
As a further demonstration of the NMD net adaptation capabilities, Figure 14 shows the number of times the agent reaches a target per episode. From this figure, it is straightforward to see that as learning progresses, the agent learns to dodge the bad target while still reaching the correct target.

Figure 14: Evolution of the number of times the agent hits a target per episode. On the left, the number of times the agent hits the correct target is plotted. On the right, the number of times the agent hits the wrong target is plotted. The plot is smoothed out thanks to a running mean over 1000 episodes and made out of 15 different runs for each architecture.

5.5 A SUMMARY OF THE OBSERVATIONS THAT CAN BE DRAWN FROM OUR EXPERIMENTS.

• **Average performances.** For each of the three benchmarks, the average performances of the NMD nets and the classic RNN have been computed by running 15 simulations. The results show that the average performances of the NMD nets are always significantly better than those of the classic RNN, except in some cases where the value of \( E \) that defines the number of episodes in our problem statement is very low. In such a context, where the quality of the strategy learned is still very poor for both architectures, classic RNNs may perform slightly better.

• **Variance of the performances.** Results on all the benchmarks show that classic RNNs lead to a significantly higher variance than NMD nets. In particular, we have observed that it was quite common with several runs of the AC algorithm with classical RNNs to have a result where the policy learned was, even after large number of episodes, exhibiting performances that were extremely poor, e.g. close to those obtained at the very early stages of the learning. Such a situation did not occur with NMD nets.

• **Best and worst performances.** For all three benchmarks, the worst run is always obtained with a classical RNN and is often very far from average performances.
(whereas the worst run with NMD nets is often relatively close to the average performances). The best run is obtained with NMD nets on benchmark 1 and on benchmark 2 (albeit the best RNN run and NMD run are very close for this benchmark) while it is obtained with a RNN on benchmark 3 (for that particular benchmark, it outperforms the best NMD net run by a relatively significant margin).

6. CONCLUSION

In this work, we have used a high level view of a nervous system mechanism called neuromodulation to improve artificial neural networks adaptive capacities. The new neural architecture, called NMD net, has been embedded in an actor-critic algorithm to solve meta-RL problems in continuous-state and action spaces. The results obtained on three benchmark problems showed that this new architecture was able to perform much better than a classical recurrent deep neural net that does not rely on this NMD mechanism.

The work reported in this paper could be extended along several lines. First, it would be interesting to explore how the work could be extended to other types of machine-learning problems, where a fast adaptive behaviour of the algorithms is required. In particular, given the strong similarities between our meta-RL problem and Bayesian RL problems, it would be interesting to study whether the algorithms proposed in this paper could be competitive to state-of-the-art Bayesian RL techniques. We note that in the context of problems with continuous action spaces, Bayesian algorithms relying on tree-search techniques (that usually perform very well on problems with discrete action spaces) suffer from the fact that they have to come up with a discretisation of the action space, which may significantly degrade their performances. We may, therefore, suspect that it is in such a context that the techniques developed in this paper may be particularly interesting. Second, research work could also be carried out to further improve the NMD net introduced in this paper. For example, we could look at different ways of neuromodulating the feed-forward layers. Currently, we use multiple ReLU layers, each layer having the same number of neurons and neuromodulating a different feed-forward layer. Even though it proved to have worked well, this solution is certainly not optimal and further investigation could certainly highlight better deep net architecture exploiting this NMD mechanism. There is another key aspect of the NMD net that should certainly deserve further research: the parametrised activation functions of the feed-forward layer. They are currently driven by only two parameters, which may not be particularly meaningful from a neuroscientific perspective, and new types of parametrised activation functions could certainly be thought of. Finally, let us emphasise that even if the results obtained by our NMD net were good, and also rather robust with respect to a large choice of parameters, further research is certainly still needed to better characterise their performances. While it would be extremely interesting to have a theoretical characterization of their performances, we believe that achieving this may be difficult given the rather complex algorithm for fitting the parameters of the NMD net and the relative complexity of the NMD architecture itself.
Acknowledgments

Nicolas Vecoven and Antoine Wehenkel gratefully acknowledge the financial support of the Belgian National Fund of Scientific Research.
Appendix A. Detailed description of the benchmarks

Before defining the three benchmark problems, let us note that for each benchmark, the MDPs that belong to the support of \( \eta \), which generates the different environments (see Section 2), have transition probabilities and reward functions that differ only according to the value of a parameter vector \( \alpha \). Drawing an MDP according to \( \eta \) will amount for all the benchmark problems to draw a value of \( \alpha \) according to a probability distribution \( P_\alpha(\cdot) \) and to determine the transition function and the reward function that correspond to this value. Therefore, when describing hereafter our benchmark, we will not talk further about \( \eta \) but well about \( P_\alpha(\cdot) \).

A.1 State-dependent target interval benchmark

State space and action space:
\[
\mathcal{S} = [-5.0, 5.0] \\
\mathcal{U} = [-20.0, 20.0]
\]

Discount factor:
\[ \gamma = 0.998 \]

Probability distribution of \( \alpha \):
\[ P_\alpha(\cdot) = \mathbb{U}[-\alpha_{\max}, \alpha_{\max}] \]
where \( \mathbb{U}[a,b] \) stands for a uniform distribution between \( a \) and \( b \).

Initial state distribution:
\[ P_{s_0}(\cdot) = \mathbb{U}[-5.0, 5.0] \]

Transition function:
First, let us define the target interval \( \text{target}_t \) as the interval \( [s_t + \alpha - 1, s_t + \alpha + 1] \). When being in a state \( s_t \) and taking action \( u_t \), the next state is computed as follows. If \( u_t \not\in \text{target}_t \) then \( s_{t+1} = s_t \), that is if the agent does not output an action in the target interval, the state does not change. If the previous condition is not met, \( s_{t+1} \) is drawn from \( \mathbb{U}[-5.0, 5.0] \).

Reward function:
\[ \rho(s_t, u_t, s_{t+1}) = \begin{cases} 
10 & \text{if } u_t \in \text{target}_t \\
-|u_t - (s_t + \alpha)| & \text{otherwise}
\end{cases} \]

One will note that in this problem, the reward does not depend on the next state’s value.

A.2 Navigation with a wind cone benchmark

State space and action space:
\[
\mathcal{S} = [-3.0, 3.0]^2 \\
\mathcal{U} = \mathbb{R}
\]

42
Discount factor: \[ \gamma = 0.998 \]

Probability distribution of \( \alpha \):
\[ \alpha \sim \mathbb{U}[-\pi, \pi] \]

Initial state distribution:
The initial state \( s_0 \) is drawn through 4 auxiliary random variables. Two of them represent the \( x \) and \( y \) coordinates of a target interval and are denoted \( p_x, p_y \). The other two represent the \( x \) and \( y \) initial coordinates of the agent and are denoted \( a_{x,0}, a_{y,0} \). At the beginning of an episode, those variables are drawn as follows:
\[
\begin{align*}
p_k &\sim \mathbb{U}[-1.0, 1.0] \quad \forall k \in \{x, y\} \\
 a_{k,0} &\sim \mathbb{U}[-1.5, 1.5] \quad \forall k \in \{x, y\}
\end{align*}
\]
From those four auxiliary variables, we define \( s_0 \) as:
\[
 s_0 = [p_x - a_{x,0}, p_y - a_{y,0}]
\]
As in Subsection A.2, we note that the distribution \( P_{s_0}(\cdot) \) is fully given by the distributions over the auxiliary variables.

Transition function:
First, let \( \text{target} \) be the set of points \((x, y)\in\mathbb{R}^2\) such that \((x, y)\in\text{target} \Leftrightarrow \sqrt{(x - p_x)^2 + (y - p_y)^2} \leq 0.4\). When taking action \( u_t \) in state \( s_t \) drawing the state \( s_{t+1} \) from the transition function amounts to first compute \( a_{x,t+1} \) and \( a_{y,t+1} \) according to the following procedure:

1. If \((a_{x,t}, a_{y,t}) \in \text{target}\) then \(a_{k,t+1} \sim \mathbb{U}[-1.5, 1.5] \quad \forall k \in \{x, y\} \).

2. If the preceding condition is not met, an auxiliary variable \( d_t \sim \mathbb{U}[-\frac{\pi}{5}, \frac{\pi}{5}] \) is drawn to compute \( a_{x,t+1} \) and \( a_{y,t+1} \) through the following sub-procedure:
   
   (a) Step one:
   \[
   \begin{align*}
   a_{x,t+1} &= a_{x,t} + 0.25 \times (\sin(u_t \cdot \pi) + \sin(\alpha + d_t)) \\
   a_{y,t+1} &= a_{y,t} + 0.25 \times (\cos(u_t \cdot \pi) + \cos(\alpha + d_t))
   \end{align*}
   \]
   One can see that taking an action \( u_t \) moves the agent in a direction which is the vectoral sum of the intended move direction \( u_t \cdot \pi \) and of the wind direction. This wind direction is sampled (through the auxiliary variable \( d_t \)) from a wind cone with a \( 2 \times \frac{\pi}{5} \) angle and which has a direction \( \alpha \).

(b) Step two: In the case where the coordinates computed by step one lie outside the square \( S = [-3; 3]^2 \), they are corrected so as to model the fact that when the agent reaches an edge of the square, it is moved to the opposite edge from which it continues its move. More specifically, \( \forall k \in \{x, y\} \):
\[
 a_{k,t+1} \left\{ \begin{array}{ll}
  a_{k,t+1} - 3.0 & \text{if } a_{k,t+1} > 1.5 \\
  a_{k,t+1} + 3.0 & \text{if } a_{k,t+1} < -1.5 \\
  a_{k,t+1} & \text{otherwise}
\end{array} \right.
\]

Once \( a_{x,t+1} \) and \( a_{y,t+1} \) have been computed, \( s_{t+1} \) is set equal to \([p_x - a_{x,t+1}, p_y - a_{y,t+1}]\).
Reward function:
The reward function can be expressed as follows:
\[
\rho(u_t, s_t, s_{t+1}) = \begin{cases} 
100 & \text{if } (a_{x,t}, a_{y,t}) \in \text{target} \\
-2 & \text{otherwise} 
\end{cases}.
\]

A.3 Navigation towards the correct target benchmark
State space and action space:
\[
S = [-2.5, 2.5]^4 \\
\mathcal{U} = \mathbb{R}
\]

Discount factor:
\[
\gamma = 0.998
\]

Probability distribution of \(\alpha\):
\[
\alpha \sim \mathbb{U}\{-1,1\}
\]

Initial state distribution:
The initial state \(s_0\) is drawn through six auxiliary random variables. Four of them represent the \(x\) and \(y\) coordinates of the two target intervals and are denoted \(p^1_x, p^1_y, p^2_x, p^2_y\). Two of them represent the \(x\) and \(y\) initial coordinates of the agent and are denoted \(a_{x,0}, a_{y,0}\). At the beginning of an episode, those variables are drawn as follows:
\[
p^b_k \sim \mathbb{U}[{-1.0}, {1.0}] \forall b \in \{1, 2\}, k \in \{x, y\} \\
a_{k,0} \sim \mathbb{U}[{-1.5}, {1.5}] \forall k \in \{x, y\}
\]
From those six auxiliary variables, we define \(s_0\) as:
\[
s_0 = [p^b_x - a_{x,0}, p^b_y - a_{y,0}, p^1_x - a_{x,0}, p^2_y - a_{y,0}] 
\]

We note that the distribution \(P_{s_0}(\cdot)\) is defined through the distributions over the auxiliary variables.

Transition function:
First, let \(\text{target}_1\) be the set of points \((x, y)\) such that \(\sqrt{(x - p^1_x)^2 + (y - p^1_y)^2} \leq 0.4\).
Second, let \(\text{target}_2\) be the set of points \((x, y)\) such that \(\sqrt{(x - p^2_x)^2 + (y - p^2_y)^2} \leq 0.4\).

When taking action \(u_t\) in state \(s_t\), drawing the state \(s_{t+1}\) from the transition function amounts to first compute \(a_{x,t+1}\) and \(a_{y,t+1}\) according to the following procedure:

1. If \(\exists k \in \{1, 2\} : (a_{x,t}, a_{y,t}) \in \text{target}_k\), which means that the agent is in one of the two targets, then \(a_{k,t+1} \sim \mathbb{U}[-1.5, 1.5] \forall k \in \{x, y\}\)

2. If the preceding condition is not met, \(a_{x,t+1}\) and \(a_{y,t+1}\) are computed by the following sub-procedure:
(a) Step one:

\[ a_{x,t+1} = a_{x,t} + \sin(u_t \pi) \times 0.25 \]

\[ a_{y,t+1} = a_{y,t} + \cos(u_t \pi) \times 0.25 \]

This step moves the agent in the direction it has selected.

(b) Step two: In the case where the coordinates computed by step one lay outside the square \( S = [-1.5; 1.5]^2 \), they are corrected so as to model the fact that when the agent reaches an edge of the square, it is moved to the opposite edge from which it continues its move. More specifically, \( \forall k \in \{x, y\} \):

\[ a_{k,t+1} \leftarrow \begin{cases} 
    a_{k,t+1} - 3.0 & \text{if } a_{k,t+1} > 1.5 \\
    a_{k,t+1} + 3.0 & \text{if } a_{k,t+1} < -1.5 \\
    a_{k,t+1} & \text{otherwise}
  \end{cases} \]

Once \( a_{x,t+1} \) and \( a_{y,t+1} \) have been computed, \( s_{t+1} \) is set equal to \( [p_{1x}^1 - a_{x,t+1}, p_{1y}^1 - a_{y,t+1}, p_{2x}^2 - a_{x,t+1}, p_{2y}^2 - a_{y,t+1}] \).

**Reward function:**

In the case where \( (a_{x,t}, a_{y,t}) \) either belongs to only \( target_1 \), only \( target_2 \) or none of them, the reward function can be expressed as follows:

\[
\rho(u_t, s_t, s_{t+1}) = \begin{cases} 
    100 \times \alpha & \text{if } (a_{x,t}, a_{y,t}) \in target_1 \land (a_{x,t}, a_{y,t}) \notin target_2 \\
    -50 \times \alpha & \text{if } (a_{x,t}, a_{y,t}) \in target_2 \land (a_{x,t}, a_{y,t}) \notin target_1 \\
    0 & \text{if } (a_{x,t}, a_{y,t}) \notin target_1 \land (a_{x,t}, a_{y,t}) \notin target_2
  \end{cases}
\]

In the case where \( (a_{x,t}, a_{y,t}) \) belongs to both \( target_1 \) and \( target_2 \), that is \( (a_{x,t}, a_{y,t}) \in target_1 \land (a_{x,t}, a_{y,t}) \in target_2 \), the reward function can be expressed as follows:

\[
\rho(u_t, s_t, s_{t+1}) = \begin{cases} 
    100 \times \alpha & \text{if } \sqrt{(a_{x,t} - p_{1x}^1)^2 + (a_{y,t} - p_{1y}^1)^2} \leq \sqrt{(a_{x,t} - p_{2x}^2)^2 + (a_{y,t} - p_{2y}^2)^2} \\
    -100 \times \alpha & \text{otherwise}
  \end{cases}
\]

That is, we consider that the agent belongs to the target to which it is closer to the centre.
Algorithm 1 Advantage actor-critic with generalised advantage estimate for solving the meta-RL problem

1: Run$(\eta, E, \text{hyperparameters}_0)$

2: Inputs:

   [1] $\eta$: The distribution over MDPs.
   [3] $\text{hyperparameters}_0$: The set of hyper-parameters that contains the following elements:

   • $B$: Number of episodes played between updates.
   • $P_{\theta_0}$ and $P_{\psi_0}$: The distributions for initialising actor and critic’s parameters. Those distributions are intrinsically tied to the models used as function approximators.
   • $A$: The architecture of the actor and critic parametrised functions.
   • $\lambda \in [0, 1]$: The discount factor for computing GAE.
   • $L$: Number of time steps played per episode.
   • $L'$: Number of time steps per episode used to compute gradients.
   • $AE$: The number of epochs per critic update.
   • $\eta$: The squared hinge loss weight.
   • $d_{\text{arg}}$: The KL divergence target.
   • $d_{\text{threshold}}$: The threshold used for early stopping.
   • $\beta_{\text{min}}$ and $\beta_{\text{max}}$: The minimum and maximum $\beta$ values.
   • $\beta_0$: The initial value of $\beta_k$ for penalising the KL divergence.
   • $\text{actor}\_lr_0$: The initial value of the policy learning rate $\text{actor}\_lr_k$.
   • $\text{V}_{\text{critic}0}$, $\text{z}_{\text{critic}0}$, $\text{v}_{\text{actor}0}$ and $\text{z}_{\text{actor}0}$: The initial value for the ADAM optimiser moments $\text{v}_{\text{critic}k}$, $\text{z}_{\text{critic}k}$, $\text{v}_{\text{actor}k}$ and $\text{z}_{\text{actor}k}$.
   • $\epsilon$, $\omega_1$, $\omega_2$: The three ADAM optimiser hyper-parameters.
   • $\text{critic}\_lr$: The critic learning rate.
   • $\text{CriticEpochs}$: The number of epochs per critic update.
   • $\text{CMB}$: The mini-batch size used for computing the critic’s gradient.
   • $\text{CRB}$: The number of previous trajectory batches used in the replay buffer for the critic.

3: $hp_0 \leftarrow \text{hyperparameter}_0$

4: $k \leftarrow 0$

5: $\theta_0 \sim P_{\theta_0}(.)$ $\triangleright$ Random initialisation
6: $\psi_0 \sim P_{\psi_0}(.)$ $\triangleright$ Random initialisation

7: while $I(k) \leq E$ do

8: \hspace{1cm} $H_k = \text{run episodes}(k, \theta_k, \eta, hp_k)$

9: \hspace{1cm} $\theta_{k+1}, \psi_{k+1} = \text{update ac}(H_{\text{max}(0,k-CRB)}, \ldots, H_k, \theta_k, \psi_k, hp_k)$

10: \hspace{1cm} $k \leftarrow k + 1$

We note that some of the hyper-parameters are adaptive. These are $\beta_k$, $\text{actor}\_lr_k$, $\text{V}_{\text{critic}k}$, $\text{z}_{\text{critic}k}$, $\text{v}_{\text{actor}k}$ and $\text{z}_{\text{actor}k}$. Thus the hyper-parameter vector may have to change in between iterations. For this reason we introduce the notation $hp_k$ which represents the hyper-parameter vector with the values of the adaptive parameters at iteration $k$. 

3: $hp_0 \leftarrow \text{hyperparameter}_0$

4: \hspace{1cm} $k \leftarrow 0$

5: \hspace{1cm} $\theta_0 \sim P_{\theta_0}(.)$ $\triangleright$ Random initialisation
6: \hspace{1cm} $\psi_0 \sim P_{\psi_0}(.)$ $\triangleright$ Random initialisation

7: while $I(k) \leq E$ do

8: \hspace{1cm} $H_k = \text{run episodes}(k, \theta_k, \eta, hp_k)$

9: \hspace{1cm} $\theta_{k+1}, \psi_{k+1} = \text{update ac}(H_{\text{max}(0,k-CRB)}, \ldots, H_k, \theta_k, \psi_k, hp_k)$

10: \hspace{1cm} $k \leftarrow k + 1$
Algorithm 2 Kth episodes run

1: run episodes$(k, \theta_k, \eta, hp_k)$

2: Inputs:

   [1] $\theta_k$ : The parameters of the policy at iteration $k$.
   [2] $\eta$ : The distribution from which the MDPs are sampled.
   [3] $hp_k$ : In this procedure, we use as hyper-parameters:
      • $B$ : The number of episodes to be played.
      • $L$ : The number of time steps played by episode.
      • $A$ : The architecture of the networks.

3: Output:

   [1] $H_k$ : The set of trajectories $[h_{I(k),L}, h_{I(k)+1,L}, \ldots, h_{I(k)+B-1,L}]$

4: $i \leftarrow I(k)$

5: while $i < I(k) + B$ do

6:   $t \leftarrow 0$

7:   $MDP_i \sim \eta$

8:   $s_{i,t} \sim P_{s_0}(\cdot)$

9:   $h_{i,t} = [s_{i,t}]$

10: while $t < L$ do

11:   $u_{i,t} \sim \pi_{\theta_k}(h_{i,t})$

12:   $s_{i,t+1} \sim P_{MDP_i}(s_{i,t} s_{i,t+1} | s_{i,t}, u_{i,t})$ ▷ The right-side refers to $P(s_{t+1} | s_t, u_t)$ of $MDP_i$. 

13:   $r_{i,t} = \rho_{MDP_i}(s_{i,t} u_{i,t}, s_{i,t+1})$ ▷ The right-side refers to $\rho(s_{t+1}, u_t, s_t)$ of $MDP_i$.

14:   $h_{i,t} = [s_{i,0}, u_{i,0}, r_{i,0}, \ldots, s_{i,t}]$

15:   $t \leftarrow t + 1$

16: $i \leftarrow i + 1$

17: Return $H_k = [h_{I(k),L}, \ldots, h_{I(k+1)-1,L}]$
Algorithm 3 Kth update of the actor critic models

1: **update ac**(\(H_{k-CRB}, \ldots, H_k, \theta_k, \psi_k, \beta_k, \text{actor lr}_k, hp_k\))

2: **Inputs:**
   
   [1] \(H_{k-CRB}, \ldots, H_k\): The CRB + 1 last sets of trajectories of length \(L\).
   
   [2] \(\theta_k\) and \(\psi_k\): The parameters of the actor and critic.
   
   [3] \(hp_k\): In this procedure, we use as hyper-parameter:
      
      • \(\lambda \in [0, \ldots, 1]\) : The discount factor for computing GAE.

3: **Output:**

   [1] \(\theta_{k+1}, \psi_{k+1}\): The updated actor and critic parameters.

4: \(Disc_{i,j} = \sum_{t=j}^{L} \gamma^{t-j} * r_{i,j} * (1 - \gamma), \forall i \in [I(k), \ldots, I(k + 1) - 1], j \in [0, \ldots, L - 1]\)

5: \(TD_{i,j} = (1 - \gamma) * r_{i,j} - c_{\psi_k}(h_{i,j}) + c_{\psi_k}(h_{i,j+1}), \forall i \in [I(k), \ldots, I(k+1) - 1], j \in [0, \ldots, L-1]\)

6: \(GAE_{i,j} = \sum_{t=j}^{L} (\gamma * \lambda)^{t-j} * TD_{i,j}, \forall i \in [I(k), \ldots, I(k + 1) - 1], j \in [0, \ldots, L - 1]\)

7: \(\mu_{gae} = \frac{1}{I(k+1)-1} \sum_{j=0}^{L-1} GAE_{i,j}\)

8: \(\sigma_{gae} = \sqrt{\frac{1}{I(k+1)-1} \sum_{j=0}^{L-1} (\mu_{gae} - GAE_{i,j})^2}\)

9: \(GAE'_{i,j} = \frac{GAE_{i,j} - \mu_{gae}}{\sigma_{gae}}, \forall i \in [I(k), \ldots, I(k + 1) - 1], j \in [0, \ldots, L - 1]\)

10: \(Advantages = [GAE'_{I(k),0}, \ldots, GAE'_{I(k),L}, GAE'_{I(k)+1,0}, \ldots, GAE'_{I(k+1)-1,L}]\)

11: \(Discounted\_sum = [Disc_{I(k-CRB),0}, \ldots, Disc_{I(k-CRB),L}, Disc_{I(k-CRB)+1,0}, \ldots, Disc_{I(k+1)-1,L}]\)

12: \(\theta_{k+1} = \text{update policy parameters}(H_k, Advantages, \theta_k, hp_k)\)

13: \(\psi_{k+1} = \text{update critic parameters}(H_{k-CRB}, \ldots, H_k, Discounted\_sum, \psi_k, hp_k)\)

14: **Return** \(\theta_{k+1}, \psi_{k+1}\)
Algorithm 4 Update from $\theta_k$ to $\theta_{k+1}$

1: **update policy parameters**($H_k$, Advantages,$\theta_k$,hp$_k$)

2: **Inputs:**

   [2] $\theta_k$: The actor’s parameters.
   [3] $\epsilon$, $\omega_1$ and $\omega_2$: The three ADAM optimizer hyper-parameters.
   [4] hp$_k$: In this procedure, we use as hyper-parameters:
   - $AE$: The number of epochs per actor update.
   - $\eta$: The squared hinge loss weight.
   - $d_{targ}$: The KL divergence target.
   - $d_{threshold}$: The threshold used for early stopping.
   - $L'$: The number of time-steps per trajectory used for computing gradients.
   - $\beta_k$: The KL penalisation weight.
   - $\text{actor}_{lr_k}$: The actor learning rate.
   - $vActor_k^i$ and $zActor_k^i$: The last ADAM moments computed at iteration $k - 1$.

3: **Output:**

   [1] $\theta_{k+1}$: The updated actor parameters.

4: $m \leftarrow 0$

5: $B_k \leftarrow [[I(k),0],\ldots,[I(k),L'],\ldots,[I(k+1)-1,0],\ldots,[I(k+1)-1,L']]$

6: $\theta'_m \leftarrow \theta_k$

7: $vActor_{k*AE-1} \leftarrow vActor_k$

8: $zActor_{k*AE-1} \leftarrow zActor_k$

9: while $m < AE$ do

10:    $\mathcal{L}_{vanilla} = - \sum_{[i,t] \in B_k} \pi_{\theta_k}(u_{i,t} | h_{i,t}) + GAE'_{i,t}$

11:    $d = \sum_{[i,t] \in B_k} KL(\pi_{\theta_k}(. | h_{i,j}), \pi_{\theta}(., h_{i,j}))$

12:    $shl = \max(0,(d-2*d_{targ}))^2$

13:    $\mathcal{L}_{policy} = \mathcal{L}_{vanilla} + \beta_k * d + \eta * shl$

14:    $\nabla_{\theta} \mathcal{L}_{policy}(\theta'_m) = \text{compute gradients}(\mathcal{L}_{policy},B_k,\theta'_m)$

15:    $lr_{Actor_{k*AE+m}} = \text{actor}_{lr_k} \ast \sqrt{\frac{1-\omega}{1-\omega^{k*AE+m}}}$

16:    $zActor_{k*AE+m} = \omega_1 * zActor_{k*AE+m} + (1-\omega_1) * \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m)$

17:    $vActor_{k*AE+m} = \omega_2 * vActor_{k*AE+m} + (1-\omega_2) * \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m) \odot \nabla_{\theta} \mathcal{L}_{policy}(\theta'_m)$

18:    $\theta'_{m+1} \leftarrow \theta'_m - \frac{lr_{Actor_{k*AE+m}} * zActor_{k*AE+m}}{\sqrt{vActor_{k*AE+m} + \epsilon}}$

19:    $m \leftarrow m + 1$

20: if $d > d_{threshold} * d_{targ}$ then

21:    $\theta'_{AE} \leftarrow \theta_k$

22:    $m \leftarrow AE$

23: **update auxiliary parameters**($d$,hp$_k$)

24: $vActor_{k+1}^i \leftarrow vActor_{(k+1)*AE-1}$

25: $zActor_{k+1}^i \leftarrow zActor_{(k+1)*AE-1}$

26: $\theta_{k+1} \leftarrow \theta'_{AE}$

27: **Return** $\theta_{k+1}$
Algorithm 5 Actor auxiliary parameters update

1: update auxiliary parameters($d, hp_k$)

2: Inputs:

   [1] $d$: The KL divergence between $\pi_{\theta_k}$ and $\pi_{\theta_{k+1}}$ empirically averaged.

   [2] $hp_k$: In this procedure, we use as hyper-parameters:

   - $d_{\text{targ}}$: The KL divergence target.
   - $\beta_{\min}$ and $\beta_{\max}$: The minimum and maximum $\beta$ values.
   - $\beta_k$: The current KL penalisation weight.
   - actor lr$_k$: The current actor learning rate.

3: if $d > 2 \ast d_{\text{targ}}$ then

4:     $\beta_{k+1} \leftarrow \min(\beta_{\max}, \beta_k \ast 1.5)$

5:   if $\beta_k > 0.85 \ast \beta_{\max}$ then

6:     actor lr$_{k+1} \leftarrow \frac{\text{actor lr}_k}{1.5}$

7: else if $d < \frac{d_{\text{targ}}}{2}$ then

8:     $\beta_{k+1} \leftarrow \max(\beta_{\min}, \frac{\beta_k}{1.5})$

9:   if $\beta_k < 1.15 \ast \beta_{\min}$ then

10:    actor lr$_{k+1} \leftarrow \text{actor lr}_k \ast 1.5$
Algorithm 6 Update from $\psi_k$ to $\psi_{k+1}$

1: **update critic parameters**($H_{k-CRB}, \ldots, H_k, Discounted\_sum, \psi_k, hp_k$)

2: **Inputs:**

   [1] $H_{k-CRB}, \ldots, H_k$: The CRB + 1 last sets of trajectories of length $L$.
   [2] $\psi_k$: The critic’s parameters.
   [3] $hp_k$: In this procedure, we use as hyper-parameters:

   • $CE'$: The number of epochs per critic update.
   • $CMB$: The mini-batch size used for computing the critic’s gradient.
   • $T$: A hyper-parameter of our gradient estimate.
   • $CRB$: The replay buffer size.
   • $critic\_lr$: The critic learning rate.
   • $L'$: The number of time-steps per trajectory used for computing gradients.
   • $vCritic_k$ and $zCritic_k$: The last ADAM moments computed at iteration $k - 1$.

3: **Output:**

   [1] $\psi_{k+1}$: The updated critic parameters.

4: $m \leftarrow 0$

5: $vActor_{k+AE-1} \leftarrow vActor_k'$

6: $zActor_{k+AE-1} \leftarrow zActor_k'$

7: $T_{i,t} = [[i, t \times T], \ldots, [i, \max((t + 1)\times T - 1, L')]] \forall i \in [I(k - CRB), \ldots, I(k + 1) - 1], t \in [0, \ldots, \left\lfloor \frac{T'}{T} \right\rfloor]$

8: $BT = [(I(k - CRB), 0], \ldots, [I(k - CRB), \left\lfloor \frac{T'}{T} \right\rfloor], \ldots, [I(k) + B - 1, 0], \ldots, [I(k) + B - 1, \left\lfloor \frac{T'}{T} \right\rfloor]]$

9: $\psi_0 \leftarrow \psi_k$

10: $CE \leftarrow CE' \times \frac{|B_{k-CRB}|}{CMB+T}$

11: $X \leftarrow \emptyset$

12: **while** $m < CE$ **do**

13:   $p \leftarrow 0$, $Y_m \leftarrow \emptyset$

14:   **while** $p < CMB$ and $BT \setminus X \neq \emptyset$ **do**

15:     $[i_{cur}, t_{cur}] \sim BT \setminus X$

16:     $X \leftarrow X \cup [i_{cur}, t_{cur}]$

17:     $Y_m \leftarrow Y_m \cup T_{i_{cur}, t_{cur}}$

18:     $p \leftarrow p + 1$

19: **if** $BT \setminus X = \emptyset$ **then**

20:     $X \leftarrow \emptyset$

21:     $L_{sur}(\psi) = \sum_{[i,t] \in Y_m} (c_{\psi}(h_{i,t}) - Disc_{i,t})^2$

22:     $\nabla_{\psi} L_{sur}(\psi_m', Y_m) = compute\_gradients(L_{sur}, Y_m, \psi_m')$

23:     $lrCritic_{k+AE+m} = critic\_lr \times \sqrt{\frac{1 - \omega_k^{CE+m}}{1 - \omega_1^{CE+m}}}$

24:     $zCritic_{k+AE+m} = \omega_2 \ast zCritic_{k+AE+m} + (1 - \omega_2) \ast \nabla_{\psi} L_{sur}(\psi_m', Y_m)$

25:     $vCritic_{k+AE+m} = \omega_2 \ast vCritic_{k+AE+m} + (1 - \omega_2) \ast \nabla_{\psi} L_{sur}(\psi_m', Y_m) \cup \nabla_{\psi} L_{sur}(\psi_m', Y_m)$

26:     $\psi_{m+1} \leftarrow \psi_m' - \frac{lrCritic_{k+AE+m}}{\sqrt{vCritic_{k+AE+m} + \epsilon}}$

27:     $m \leftarrow m + 1$

28: $\psi_{k+1} \leftarrow \psi_{CE}$

29: **Return** $\psi_{k+1}$
Algorithm 7 Gradient computing with BPTT

1: `compute gradients(L(α), Z, α', hp_k)`

2: **Inputs:**

   [1] $L(α)$ : A loss function which is dependent on a function approximate $v(α)$.

   [2] $Z$ : The set of pairs $[i, j]$ such that $v(α)_{i,j}$ appears in $L(α)$. ▷ We emphasise that from the way $Z$ is built (see Algorithms 4 and 6), most of the time $Z$ contains $x$ batches of $T$ consecutive pairs. Note that very rarely, batches may have fewer than $T$ consecutive pairs (whenever a batch contains the last pairs of an episode which does not contain a multiple of $T$ pairs), although, the same gradient descent algorithm as the one presented in Section 3.3 can still be applied.

   [3] $α'$ : The element for which the estimate gradient of $L(α)$ needs to be evaluated.

   [4] $hp_k$ : In this procedure, we use as hyper-parameter:

      • $T$ : The number of time-steps for which the gradient can propagate.

3: **Output:**

   [1] $\nabla_α L(α')$ : The gradient estimate of the function $L(α)$ evaluated in $α'$.

   ▷ See Section 3.3 for a description of the algorithm. The reader can also refer to the source code which also available on Github ([https://github.com/nvecoven/nmd_net](https://github.com/nvecoven/nmd_net)). We note that giving a full tabular version of the algorithm here would not constitute valuable information to the reader, due to its complexity/length.
Appendix C. Hyper-parameter values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$</td>
<td>50</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>0.98</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.998</td>
</tr>
<tr>
<td>$\beta_0$</td>
<td>1</td>
</tr>
<tr>
<td>$\beta_{\min}$</td>
<td>$1/30$</td>
</tr>
<tr>
<td>$\beta_{\max}$</td>
<td>30</td>
</tr>
<tr>
<td>$d_{\text{targ}}$</td>
<td>0.003</td>
</tr>
<tr>
<td>$\delta$</td>
<td>50</td>
</tr>
<tr>
<td>$\text{actor lr}_0$</td>
<td>$2 \times 10^{-4}$</td>
</tr>
<tr>
<td>$\omega_1$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\omega_2$</td>
<td>0.999</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$10^{-8}$</td>
</tr>
<tr>
<td>$AE$</td>
<td>20</td>
</tr>
<tr>
<td>$CRB$</td>
<td>2</td>
</tr>
<tr>
<td>$CMB$</td>
<td>25</td>
</tr>
<tr>
<td>$\text{critic lr}$</td>
<td>$6 \times 10^{-3}$</td>
</tr>
<tr>
<td>$T$</td>
<td>200</td>
</tr>
<tr>
<td>$CE'$</td>
<td>10</td>
</tr>
<tr>
<td>$\omega_1$</td>
<td>0.9</td>
</tr>
<tr>
<td>$\omega_2$</td>
<td>0.999</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>$10^{-8}$</td>
</tr>
<tr>
<td>$A$</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 4: Value of the hyper-parameters that are kept constant for every benchmark in this paper.
Appendix D. Bayes optimal policy for the state-dependent target interval benchmark

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 $\eta$. That is, a Bayes optimal policy $\pi_{\text{bayes}}^*$ belongs to the following set:

$$\pi_{\text{bayes}}^* \in \arg\max_{\pi \in \Pi} \mathbb{E}_{\text{MDP} \sim \eta} R_{\text{MDP}}^\pi,$$

with $P_{\text{MDP}}$ being the state-transition function of $\text{MDP}$.

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

$$\pi_{\text{bayes}}^* \in \arg\max_{\pi \in \Pi} \mathbb{E}_{\alpha \sim U_\alpha} \mathbb{E}_{\text{MDP} \sim \eta} R_{\alpha}^\pi,$$

with $\text{MDP}(\alpha)$ being a function giving as output the MDP corresponding to $\alpha$.

We now prove the following theorem.

**Theorem 1** The policy that selects:

1. at time-step $t = 0$ the action $u_0 = s_0 + \frac{\gamma \times (\alpha_{\text{max}} + 4.5)}{1 + \gamma}$
2. at time-step $t = 1$
   a) if $r_0 = 10$, the action $u_1 = s_1 + u_0 - s_0$
   b) else if $|r_0| > \alpha_{\text{max}} - (u_0 - s_0)$ and $u_0 - s_0 > 0$, the action $u_1 = u_0 + r_0$
   c) else if $|r_0| > \alpha_{\text{max}} - (s_0 - u_0)$ and $u_0 - s_0 < 0$, the action $u_1 = u_0 - r_0$
   d) and otherwise the action $u_1 = u_0 + r_0 + 1$
3. for the remaining time-steps:
   a) if $r_0 = 10$, the action $u_t = s_t + u_0 - s_0$
   b) else if $r_1 = 10$, the action $u_t = s_t + u_1 - s_1$
   c) and otherwise the action $u_t = s_t + i_t$ where $i_t$ is the unique element of the set
      \[ \{ u_0 - s_0 + r_0; u_0 - s_0 - r_0 \} \cap \{ u_1 - s_1 + r_1; u_1 - s_1 - r_1 \} \]
      is Bayes optimal for the state-dependent target interval benchmark defined in Appendix A.1.
Proof Let us denote by $\pi^*_{\text{theorem1}}$ the policy described in this theorem. To prove this theorem, we first prove that in the set of all possible policies $\Pi$ there are no policy $\pi$ which leads to a higher value of

$$
\mathbb{E}_{\text{MDP}\sim \eta} \left( r_0 + \gamma \pi \right) \quad \text{(17)}
$$

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

$$
\mathbb{E}_{\text{MDP}\sim \eta} \left( r_0 + \gamma \pi \right) \geq \mathbb{E}_{\text{MDP}\sim \eta} \left( r_0 + \gamma \pi \right) \quad \forall \pi \in \Pi \quad \text{(18)}
$$

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

$$
\mathbb{E}_{\text{MDP}\sim \eta} \left( \sum_{t=2}^{\infty} \gamma^t r_t \right) = \sum_{t=2}^{\infty} \gamma^t R_{\text{max}} \quad \geq \quad \mathbb{E}_{\text{MDP}\sim \eta} \left( \sum_{t=2}^{\infty} \gamma^t r_t \right) \quad \forall \pi \in \Pi \quad \text{(19)}
$$

By merging (18) and (19), we have that

$$
\mathbb{E}_{\text{MDP}\sim \eta} \left( \sum_{t=0}^{\infty} \gamma^t r_t \right) \geq \mathbb{E}_{\text{MDP}\sim \eta} \left( \sum_{t=0}^{\infty} \gamma^t r_t \right) \quad \forall \pi \in \Pi
$$

which proves the theorem.

> Part 1. Let us now prove inequality (18). The first thing to notice is that for a policy to maximise expression (17), it only needs to satisfy two conditions for all $s_0$. The first one: to select an action $u_1$, which knowing the value of $(s_0, u_0, r_0, s_1)$, maximises the expected value of $r_1$. We denote by $V_1(s_0, u_0, r_0, s_1)$ the maximum expected value of $r_1$ that can be obtained knowing the value of $(s_0, u_0, r_0, s_1)$. The second one: to select an action $u_0$ knowing the value of $s_0$ that maximises the expected value of the sum $r_0 + \gamma V_1(s_0, u_0, r_0, s_1)$. We now show that the policy $\pi^*_{\text{theorem1}}$ 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 $\pi^*_{\text{theorem1}}$ for time step $t = 1$.

a) If $r_0 = 10$, the maximum reward that can be obtained, we are in a context where $u_0$ belongs to the target interval. It is easy to see that, by playing $u_1 = s_1 + u_0 - s_0$, we will obtain $r_1$ equal to 10. This shows that in case a) for time step $t = 1$, $\pi^*_{\text{theorem1}}$ maximises this expected value of $r_1$.

b) If $|r_0| > \alpha_{\text{max}} - (u_0 - s_0)$ and $u_0 - s_0 > 0$ and $r_0 \neq 10$ it is easy to see that the value of $\alpha$ to which the MDP corresponds can be inferred from $(s_0, u_0, r_0)$ and that the action $u_1 = u_0 + r_0$ will fall in the middle of the target interval, leading to a reward
of 10. Hence, in this case also, the policy $\pi_{\text{theorem}}$ maximises the expected value of $r_1$.

c) If $|r_0| > \alpha_{\text{max}} - (s_0 - u_0)$ and $u_0 - s_0 < 0$ and $r_0 \neq 10$, we are also in a context where the value of $\alpha$ can be inferred directly from $(s_0, u_0, r_0)$ and the action $u_1 = u_0 - r_0$ targets the centre of the target interval, leading to a reward of 10. Here again, $\pi_{\text{theorem}}$ maximises the expected value of $r_1$.

d) When none of the three previous conditions is satisfied, $\alpha$ is not satisfied and so $s_1 = s_0$, we need to consider two cases: $(u_0 - s_0) \geq 0$ and $(u_0 - s_0) < 0$. Let us first start with $(u_0 - s_0) \geq 0$. In such a context, $\alpha \in \{u_0 - s_0 + r_0; u_0 - s_0 - r_0\} = \{u_0 - s_0 - |u_0 - s_0 - \alpha|, u_0 - s_0 + |u_0 - s_0 - \alpha|\}$ and where:

\begin{enumerate}
  \item $P(\alpha = u_0 - s_0 - |u_0 - s_0 - \alpha|; s_0, u_0, r_0, s_1) = 0.5$
  \item $P(\alpha = u_0 - s_0 + |u_0 - s_0 - \alpha|; s_0, u_0, r_0, s_1) = 0.5$
\end{enumerate}

Let us now determine the action $u_1$ that maximises $\hat{r}_1$, the expected value of $r_1$ according to $P(\alpha; s_0, u_0, r_0, s_1)$. Five cases, represented on Figure 15, have to be considered:

\begin{enumerate}
  \item $u_1 < u_0 - |u_0 - s_0 - \alpha| - 1$. Here $\hat{r}_1 = u_1 - u_0$ and the maximum of $\hat{r}_1$ is equal to $-|u_0 - s_0 - \alpha| - 1$.
  \item $u_1 \in [u_0 - |u_0 - s_0 - \alpha| - 1, u_0 - |u_0 - s_0 - \alpha| + 1]$. Here we have $\hat{r}_1 = \frac{1}{2}(10 + u_0 - |u_0 - s_0 - \alpha| - u_1)$ whose maximum over the interval is $5.5 - |u_0 - s_0 - \alpha|$ which is reached for $u_1 = u_0 + |u_0 - s_0 - \alpha| - 1$.
  \item $u_1 \in [u_0 - |u_0 - s_0 - \alpha| + 1, u_0 + |u_0 - s_0 - \alpha| - 1]$. In this case $\hat{r}_1 = -|u_0 - s_0 - \alpha|$ and is independent from $u_1$.
  \item $u_1 \in [u_0 + |u_0 - s_0 - \alpha| - 1, u_0 + |u_0 - s_0 - \alpha| + 1]$. The expected reward is $\hat{r}_1 = \frac{1}{2}(10 + u_0 - |u_0 - s_0 - \alpha| - 1)$ whose maximum over the interval is $5.5 - |u_0 - s_0 - \alpha|$ which is reached for $u_1 = u_0 + |u_0 - s_0 - \alpha| + 1$.
  \item $u_1 > u_0 + |u_0 - s_0 - \alpha| + 1$. In this case the expected reward is $\hat{r}_1 = u_0 - u_1$ and the maximum of $\hat{r}_1$ is equal to $-|u_0 - s_0 - \alpha| - 1$.
\end{enumerate}

![Figure 15: Graphical representation of the 5 different cases when playing $u_1$. The length of a dashed black line with two arrowheads is equal the absolute value of the reward received when taking action $u_0$.](image-url)
From 1), 2), 3), 4) and 5) one can see that, given the conditions considered here, an optimal policy can either play \( u_1 = u_0 + |u_0 - s_0 - \alpha| - 1 \) or \( u_1 = u_0 - |u_0 - s_0 - \alpha| + 1 \). In the following we will fix \( u_1 \) to \( u_0 + |u_0 - s_0 - \alpha| + 1 \) when \( u_0 - s_0 \geq 0 \). Let us also observe that the expected value of \( r_1 \) is equal to \( 5.5 - |u_0 - s_0 - \alpha| \). Up to now in this item d), we have only considered the case where \((u_0 - s_0) > 0\). When \((u_0 - s_0) \leq 0\), 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 \( \pi_{\text{theorem}1} \) plays the action \( u_1 = u_0 + r_0 + 1 = u_0 - |u_0 - s_0 - \alpha| + 1 \) 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 \( \pi_{\text{theorem}1} \) to maximise expression (17) has been proved, let us turn our attention to the second one. To this end, we will compute for each \( s_0 \in \mathcal{S} \), the action \( u_0 \in \mathcal{U} \) that maximises:

\[
\mathbb{E}_{\alpha \sim \mathcal{U}_\alpha} (r_0 + \gamma * V_1(s_0, u_0, r_0, s_1)) \tag{20}
\]

and show that this action coincide with the action taken by \( \pi_{\text{theorem}1} \) for time step \( t = 0 \). First let us observe that for this optimisation problem, one can reduce the search space \( \mathcal{U} \) to \([s_0 - \alpha_{\text{max}} + 1, s_0 + \alpha_{\text{max}} - 1] \subset \mathcal{U} \). Indeed, an action \( u_0 \) that does not belong to this latter interval would not give more information about \( \alpha \) than playing \( u_0 = s_0 - \alpha_{\text{max}} + 1 \) or \( s_0 + \alpha_{\text{max}} - 1 \) 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 \( \mathcal{U}_\alpha = \mathbb{U}[-\alpha_{\text{max}}, \alpha_{\text{max}}] \) and that the function \( V_1(s_0, u_0, r_0, s_1) \) can be written as follows:

1. If \( r_0 = 10 \), \( V_1 \) is equal to \( R_{\text{max}} = 10 \)
2. Else if \(|r_0| > \alpha_{\text{max}} - (u_0 - s_0) \land u_0 - s_0 > 0 \) and \( r_0 \neq 10 \), then \( V_1 \) is equal to \( R_{\text{max}} = 10 \)
3. Else if \(|r_0| > \alpha_{\text{max}} - (s_0 - u_0) \land u_0 - s_0 < 0 \) and \( r_0 \neq 10 \), then \( V_1 \) is equal to \( R_{\text{max}} = 10 \)
4. And otherwise \( V_1 \) is equal to \( 5.5 - |u_0 - s_0 - \alpha| \).

We note that the value of \( V_1(s_0, u_0, r_0, s_1) \) does not depend on the state \( s_1 \), which allows us to rewrite expression (20) as follows:

\[
\mathbb{E}_{\alpha \sim \mathcal{U}_\alpha} (r_0 + \gamma * V_1(s_0, u_0, r_0, s_1)) \tag{21}
\]

and since the expectation is a linear operator:

\[
(21) = \mathbb{E}_{\alpha \sim \mathcal{U}_\alpha} (r_0) + \gamma * \mathbb{E}_{\alpha \sim \mathcal{U}_\alpha} (V_1(s_0, u_0, r_0, s_1)) \tag{22}
\]

Let us now focus on the second term of this sum:
\[ E_{\alpha \sim U_{\alpha}}(V_1(s_0, u_0, r_0, s_1)) \] \hspace{1cm} (23)

We note that when \( u_0 - s_0 \geq 0 \) the function \( V_1 \) can be rewritten under the following form:

1. if \( \alpha \in [-\alpha_{\text{max}}, 2 \cdot (u_0 - s_0) - \alpha_{\text{max}}] \), \( V_1 \) is equal to 10
2. else if \( \alpha \in [2 \cdot (u_0 - s_0) - \alpha_{\text{max}}, u_0 - s_0 - 1] \), \( v_1 \) is equal to \( 5.5 + \alpha - (u_0 - s_0) \)
3. else if \( \alpha \in [u_0 - s_0 - 1, u_0 - s_0 + 1] \), \( V_1 \) is equal to 10
4. else if \( \alpha \in [u_0 - s_0 + 1, \alpha_{\text{max}}] \), \( V_1 \) is equal to \( 5.5 - \alpha + (u_0 - s_0) \).

From here, we can compute the value of expression (23) when \( u_0 - s_0 \geq 0 \). We note that due to the symmetry that exists between the case \( u_0 - s_0 \geq 0 \) and \( u_0 - s_0 \leq 0 \), expression (23) will have the same value for both cases. Since we have:

\[
(23) = \int_{-\infty}^{\infty} V_1 \cdot p_{\alpha} \cdot d\alpha
\]

where \( p_{\alpha} \) is the probability density function of \( \alpha \), we can write:

\[
(23) = \int_{-\alpha_{\text{max}}}^{\alpha_{\text{max}}} V_1 \cdot \frac{1}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha = \int_{-\alpha_{\text{max}}}^{2 \cdot (u_0 - s_0) - \alpha_{\text{max}}} \frac{10}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha + \int_{2 \cdot (u_0 - s_0) - \alpha_{\text{max}}}^{u_0 - s_0 - 1} \frac{5.5 + \alpha - (u_0 - s_0)}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha \\
+ \int_{u_0 - s_0 - 1}^{u_0 - s_0 + 1} \frac{10}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha + \int_{u_0 - s_0 + 1}^{\alpha_{\text{max}}} \frac{5.5 - \alpha + (u_0 - s_0)}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha
\]

And thus, by computing the integrals, we have:

\[
E_{\alpha \sim U_{\alpha}}(V_1) = - \frac{1}{2 \cdot \alpha_{\text{max}}} (u_0 - s_0)^2 + \frac{1}{\alpha_{\text{max}}} (\alpha_{\text{max}} + 4.5) \cdot (u_0 - s_0) \\
+ \frac{1}{\alpha_{\text{max}}} \left( 5 + 5 \cdot \alpha_{\text{max}} - \frac{\alpha_{\text{max}}^2}{2} \right)
\]

Let us now analyse the first term of the sum in equation (22), namely \( E_{\alpha \sim U_{\alpha}}(r_0) \).

We have that:

\[
E_{\alpha \sim U_{\alpha}}(r_0) = \int_{-\infty}^{\infty} (r_0 | s_0, u_0, \alpha) \cdot p_{\alpha} \cdot d\alpha
\]

which can be rewritten as:

\[
E_{\alpha \sim U_{\alpha}}(r_0) = \int_{-\alpha_{\text{max}}}^{\alpha_{\text{max}}} (r_0 | s_0, u_0, \alpha) \cdot \frac{1}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha
\]

Due to the reduction of the search space, we can assume that \( u_0 \) belongs to \( [s_0 - \alpha_{\text{max}} + 1, s_0 + \alpha_{\text{max}} - 1] \), we can write:

\[
\int_{-\alpha_{\text{max}}}^{\alpha_{\text{max}}} (r_0 | s_0, u_0, \alpha) \cdot \frac{1}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha = \int_{-\alpha_{\text{max}}}^{u_0 - s_0 - 1} \frac{\alpha - (u_0 - s_0)}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha \\
+ \int_{u_0 - s_0 - 1}^{u_0 - s_0 + 1} \frac{10}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha + \int_{u_0 - s_0 + 1}^{\alpha_{\text{max}}} \frac{(u_0 - s_0) - \alpha}{2 \cdot \alpha_{\text{max}}} \cdot d\alpha
\]
Given that $R_{\text{max}} = 10$, we have:

$$
\mathbb{E}_{\alpha \sim U_{\alpha}}(r_0) = \frac{-(u_0 - s_0)^2 + 21 - \alpha^2_{\text{max}}}{2 \alpha_{\text{max}}}
$$

and therefore:

$$(22) = \frac{-1 + \gamma}{2 \alpha_{\text{max}}} \cdot (u_0 - s_0)^2 + \frac{\gamma}{\alpha_{\text{max}}} (\alpha_{\text{max}} + 4.5) \cdot (u_0 - s_0)
$$

$$+ \frac{1}{2 \alpha_{\text{max}}} \cdot (21 - \alpha^2_{\text{max}} + \gamma \cdot (10 + 11 \cdot \alpha_{\text{max}} - \alpha^2_{\text{max}})).$$

To find the action $u_0$ that maximises (20), one can differentiate (22) with respect to $u_0$:

$$
\frac{d(22)}{d(u_0)} = -\frac{1}{\alpha_{\text{max}}} \cdot (1 + \gamma)(u_0 - s_0) + \frac{\gamma}{\alpha_{\text{max}}} (\alpha_{\text{max}} + 4.5).
$$

This derivative has a single zero value equal to:

$$u_0 = \frac{\gamma \cdot (\alpha_{\text{max}} + 4.5)}{1 + \gamma} + s_0.$$

It can be easily checked that it corresponds to a maximum of expression (20) and since it also belongs to the reduced search space $[s_0 - \alpha_{\text{max}} + 1, s_0 + \alpha_{\text{max}} - 1]$, it is indeed the solution to our optimisation problem. Since $\pi_{\text{theorem}1}$ plays this action at time $t = 0$, Part 1 of this proof is now fully completed.

$\triangleright$ Part 2. Let us now prove that the policy $\pi^*_{\text{theorem}1}$ generates for every $t \geq 2$ rewards equal to $R_{\text{max}} = 10$. We will analyse three different cases, corresponding to the three cases a), b) and c) of policy $\pi_{\text{theorem}1}$ for time step $t \geq 2$.

a) If $r_0 = 10$, we are in a context where $u_0$ belong to the target interval. It is straightforward to see that, by playing $u_t = s_t + u_0 - s_0$, the action played by $\pi_{\text{theorem}1}$ 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 $u_t = s_t + u_1 - s_1$, the action played by $\pi_{\text{theorem}1}$, 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 $u_0$ that the MDP played corresponds necessarily to one of these two values for $\alpha$: $\{u_0 - s_0 + r_0; u_0 - s_0 - r_0\}$. Similarly, from the second action played, one knows that $\alpha$ must also stand in $\{u_1 - s_1 + r_1; u_1 - s_1 - r_1\}$. It can be proved that because $u_0 \neq u_1$ (a property of our policy $\pi_{\text{theorem}1}$), 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:

$$u_0 - s_0 + r_0 = u_1 - s_1 + r_1$$
$$u_0 - s_0 - r_0 = u_1 - s_1 - r_1$$

59
or this pair of equalities would be valid:

\[ u_0 - s_0 + r_0 = u_1 - s_1 - r_1 \]
\[ u_0 - s_0 - r_0 = u_1 - s_1 + r_1 \]

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

\[ u_0 - s_0 = u_1 - s_1 \]

Taking into account that \( s_0 = s_1 \) because none of the two actions yielded a positive reward, it implies that \( u_0 = u_1 \), 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 \( \alpha \). From here, it is straightforward to see that in this case c), the policy \( \pi_{\text{theorem1}} \) will always generate rewards equal to \( R_{\text{max}} \).

From Theorem 1, one can easily prove the following theorem.

**Theorem 2** The value of expected return of a Bayes optimal policy for the state-dependent target interval benchmark is equal to

\[
\frac{3 \gamma^2 s(\alpha_{\text{max}} + 4.5)^2}{2 s \alpha_{\text{max}} (1 + \gamma)} + \frac{21 + \alpha_{\text{max}}^2 + \gamma s(10 + 11 s \alpha_{\text{max}} - \alpha_{\text{max}}^2)}{2 s \alpha_{\text{max}}} + \frac{\gamma^2}{1 - \gamma} * \frac{1}{10}.
\]

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

\[
\mathbb{E}_n_{\text{MDP}} \sum_{t=0}^{1} \gamma^t * r_t + \mathbb{E}_n_{\text{MDP}} \sum_{t=2}^{\infty} \gamma^t * r_t.
\]

From the proof of Theorem 1, it is easy to see that:

1. \[
\mathbb{E}_n_{\text{MDP}} \sum_{t=0}^{1} \gamma^t * r_t = \frac{3 \gamma^2 s(\alpha_{\text{max}} + 4.5)^2}{2 s \alpha_{\text{max}} (1 + \gamma)} + \frac{21 + \alpha_{\text{max}}^2 + \gamma s(10 + 11 s \alpha_{\text{max}} - \alpha_{\text{max}}^2)}{2 s \alpha_{\text{max}}} + \frac{\gamma^2}{1 - \gamma} * \frac{1}{10}
\]

2. \[
\mathbb{E}_n_{\text{MDP}} \sum_{t=2}^{\infty} \gamma^t * r_t = \frac{\gamma^2}{1 - \gamma} * 10
\]

which proves Theorem 2.
References


