

# NYILATKOZAT

**Név:** Szabó Kristóf

**ELTE Természettudományi Kar, szak:** Matematika BSc

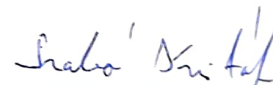
**NEPTUN azonosító:** QD3FMF

**Szakedolgozat címe:**

Theorem Proving with Deep Learning

A **szakedolgozat** szerzőjeként fegyelmi felelősségem tudatában kijelentem, hogy a dolgozatom önálló szellemi alkotásom, abban a hivatkozások és idézések standard szabályait következetesen alkalmaztam, mások által írt részeket a megfelelő idézés nélkül nem használtam fel.

Budapest, 2022.05.30.



---

*a hallgató aláírása*

EÖTVÖS LORÁND UNIVERSITY  
FACULTY OF SCIENCE

---

Szabó Kristóf  
BSc in Mathematics

# Theorem Proving with Deep Learning

Thesis

External Supervisor:

Zsolt Zombori, research fellow  
*Alfréd Rényi Institute of Mathematics*

Internal Consultant:

András Lukács, senior lecturer  
*ELTE, Department of Computer Science*



Budapest, 2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Notations . . . . .	4
1.2	Preliminaries . . . . .	5
<b>2</b>	<b>Reinforcement learning</b>	<b>8</b>
2.1	Foundation . . . . .	8
2.2	Value-based methods . . . . .	10
2.3	Temporal difference learning . . . . .	14
<b>3</b>	<b>Planning</b>	<b>18</b>
3.1	Multi-armed bandits . . . . .	18
3.2	Monte Carlo Tree Search . . . . .	24
<b>4</b>	<b>Deep learning</b>	<b>26</b>
4.1	Neural architectures . . . . .	26
4.2	Deep reinforcement learning . . . . .	27
4.3	Autoencoders . . . . .	31
<b>5</b>	<b>Dreaming to Prove</b>	<b>33</b>
5.1	Dreamer algorithm . . . . .	33
5.2	Dreaming to Prove . . . . .	35

## Acknowledgement

I would like to express my special thanks to Zsolt Zombori for introducing me to the topic of Automated Theorem Proving. I believe that I learned a lot from the Dreaming to Project that would not have been possible without the help and guidance of Zsolt. He was always available if I had a question and always thoroughly assisted my work on the Dreaming to Prove project. Furthermore, I truly appreciate that he undertook the supervision of my thesis, despite spending the year abroad. Moreover, I am thankful to András Lukács for taking the internal consultant role. Finally, I would like to thank Beátrix Benkő for revisioning my thesis and providing helpful feedback.

# 1 Introduction

In recent years, Artificial Intelligence research made breakthroughs that weren't anticipated before. Now, computers can generate pictures (DALL-E) and texts (GTP-2) indistinguishable from real ones. Other examples are AlphaZero which achieved super-human level in chess, go and shogi or the AlphaFold, which is capable of reliably predicting protein structures which was a slow and expensive procedure.

This thesis introduces an automated theorem proving system titled Dreaming to Prove that builds upon the theories of Reinforcement learning, Deep learning, Planning methods and several results of novel research directions in deep learning.

Automated Theorem Proving is a subfield of mathematical logic and computer science focusing on proving mathematical theorems. Many researchers believe that the large-scale semantics process and computer assistance of mathematics and science is our inevitable future. Unfortunately, theorem proving is a complex task even for mathematicians, so building systems that are able to solve mathematical problems is a real challenge. In Chapter 2, we introduce the Markov Decision Process, which is the mathematical formulation of interaction with an object or environment that responds to our actions. They are not only beneficial because we can consider theorem proving as a particular type of Markov Decision Process, but there is also a theory built upon them, called Reinforcement Learning, which introduces algorithms that are able to find an action sequence that maximises the reward. In Chapter 3, we introduce the one-armed bandits, an optimisation problem within reinforcement learning that emphasises the importance of a balanced exploration-exploitation trade-off, which briefly means that we cannot play the best action each time and try new ones that might perform the current best one. However, as one might expect, theorem proving is far more complicated to be tackled with simple algorithms; therefore, ideas of deep learning are used to improve upon the results of Chapter 2 as shown in Chapter 4. Finally, Chapter 5 introduces the Dreaming to Prove system and explains how the mechanism of each component uses algorithms, methods or ideas discussed in previous chapters.

## 1.1 Notations

In this section we establish some notations.

- $\mathcal{P}(\mathcal{X})$ , space of probability measures over an  $X$  measurable space.
- $L^p(\mathcal{X})$ , space of measurable functions on an  $X$  measurable space with finite  $p$ -norm. (We mostly use  $L^\infty$  which contains the almost surely bounded functions.)

We often write  $\mathbb{E}_{x \sim p(\cdot)} [f(x)]$  to emphasise that  $x$  is a variable drawn from the  $p$  distribution. This is especially useful, when we have conditional probabilities:  $\mathbb{E}_{x \sim p(\cdot|C)} [f(x)]$  where  $C$  is some event.

## 1.2 Preliminaries

The Kullback-Leibler divergence plays a crucial role in statistics and in machine learning.

**Definition 1.1** (KL-divergence). Let  $P, Q \in \mathcal{P}(\mathcal{X})$  be two probability measures such that  $P$  is absolutely continuous with respect to  $Q$ , then the KL-divergence from  $Q$  to  $P$  is defined as

$$D_{KL}(P \parallel Q) = \int_{\mathcal{X}} \log \left( \frac{dP}{dQ} \right) dP$$

Many papers interpret KL-divergence as measurement of some distance between two probability distribution, although it violates the symmetric property of metrics. Gibbs' inequality states the following.

**Theorem 1.2.** 1. For every  $P, Q$  probability measures  $D_{KL}(P \parallel Q) \geq 0$ .

2. If  $D_{KL}(P \parallel Q) = 0$ , then  $P = Q$ .

*Proof.* The Gibbs' inequality immediately follows from the Jensen's inequality: Let  $(\Omega, \mathcal{A}, \mu)$  be a probability space,  $f : \Omega \rightarrow \mathbb{R}$  be a  $\mu$ -measurable function and  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  a convex function, then:

$$\int_{\Omega} \varphi \circ f d\mu \geq \varphi \left( \int_{\Omega} f d\mu \right)$$

Since  $-\log$  is a convex function, we can apply the above inequality:

$$D_{KL}(P \parallel Q) = \int_{\Omega} -\log \left( \frac{dQ}{dP} \right) dP \geq -\log \left( \int_{\Omega} \frac{dQ}{dP} dP \right) = 0$$

The last integrand is  $\int_{\Omega} dQ = 1$  by definition, hence  $D_{KL}(P \parallel Q) \geq 0$ .

For the second part of the theorem, note that  $-\log$  is strictly convex, therefore equality occurs only if  $\frac{dQ}{dP}$  is constant with 1-probability, which holds precisely when  $P = Q$ . □

In deep learning, the primary optimisation method is gradient descent which updates the model using the objective gradient. Usually, we only have an approximation of the objective; therefore, the following lemma is of great use.

**Theorem 1.3** (Leibniz rule). Let  $I$  be an open subset of  $\mathbb{R}$ , and  $(\mathcal{X}, \mathcal{A}, \mu)$  be a measure space. Suppose  $f : I \times \mathcal{X} \rightarrow \mathbb{R}$  satisfies the following conditions:

1. For every  $t \in I$  the  $f(t, \cdot)$  function is  $\mathcal{A}$ -measurable.
2. For almost all  $\omega \in \mathcal{X}$ , the  $\partial_t f(t, \omega)$  derivative exists for all  $t \in I$ .
3. There is an integrable function  $g : \mathcal{X} \rightarrow \mathbb{R}$  such that  $|\partial_t f(t, \omega)| \leq g(\omega)$  for all  $t \in I$  and almost every  $\omega \in \mathcal{X}$ .

Then, for all  $t \in I$ ,

$$\partial_t \int_{\mathcal{X}} f(t, \omega) d\mu(\omega) = \int_{\mathcal{X}} \partial_t f(t, \omega) d\mu(\omega)$$

The proof relies on the dominant convergence theorem:

**Theorem 1.4** (Dominated convergence theorem). *Let  $f_n$  be a sequence of measurable functions on a  $(\mathcal{X}, \mathcal{F}, \mu)$  measure space. Suppose that  $f_n$  converges pointwise to  $f$  and there is an integrable function  $g$  that dominates  $f_n$ , i.e.,  $|f_n| \leq g$ . Then  $f$  is integrable and*

$$\lim_{n \rightarrow \infty} \int_{\mathcal{X}} f_n d\mu = \int_{\mathcal{X}} f d\mu$$

*Proof of Theorem 1.3.* Let  $F(t) = \int_{\mathcal{X}} f(t, \omega) d\mu(\omega)$ , then the derivative is

$$\begin{aligned} \partial_t F(t) &= \lim_{h \rightarrow 0} \frac{F(t+h) - F(t)}{h} \\ &= \lim_{h \rightarrow 0} \int_{\mathcal{X}} \frac{f(t+h, \omega) - f(t, \omega)}{h} d\mu(\omega) \end{aligned}$$

Due to the mean value theorem there is a  $c \in [t, T+h]$  such that  $f(c, \omega) = \frac{f(t+h, \omega) - f(t, \omega)}{h}$ . The latter one is dominated by  $g$  because of the assumption in the statement. Therefore  $\frac{f(t+h, \omega) - f(t, \omega)}{h}$  is dominated, so we can apply the dominated convergence theorem and swap the limit and the integral and obtain that

$$\begin{aligned} \partial_t F(t) &= \int_{\mathcal{X}} \lim_{h \rightarrow 0} \frac{f(t+h, \omega) - f(t, \omega)}{h} d\mu(\omega) \\ &= \int_{\mathcal{X}} \partial_t f(t, \omega) d\mu(\omega) \end{aligned}$$

This is exactly what we wanted to prove. □

The following definitions and theorems are intended only for an audience that is interested in the precise mathematical definition of the Markov Decision Process.

**Definition 1.5** (Probability kernel). Let  $(\mathcal{X}, \mathcal{F}), (\mathcal{Y}, \mathcal{G})$  be measurable spaces. A  $\kappa : \mathcal{X} \times \mathcal{G}$  is probability kernel (or Markov kernel) such that

1.  $\kappa(x, \cdot)$  is a measure for all  $x \in \mathcal{X}$
2.  $\kappa(\cdot, G)$  is  $\mathcal{F}$ -measurable for all  $A \in \mathcal{G}$

The natural interpretation of the above definition is that we have a distribution over  $\mathcal{Y}$  depending on some event in  $\mathcal{X}$ . Therefore, we will use the notion  $\kappa(A|x) = \kappa(x, A)$  to emphasise the dependence on  $x$ .

**Definition 1.6** (Composition of probability kernels). Let  $\kappa_1$  be a  $(\mathcal{X}, \mathcal{F}) \rightarrow (\mathcal{Y}, \mathcal{G})$  probability kernel and  $\kappa_2$  be another  $(\mathcal{Y}, \mathcal{G}) \rightarrow (\mathcal{Z}, \mathcal{H})$  probability kernel, then the product kernel  $\kappa_1 \otimes \kappa_2$  is the probability kernel from  $(\mathcal{X}, \mathcal{F}) \rightarrow (\mathcal{Y} \times \mathcal{Z}, \mathcal{G} \otimes \mathcal{H})$  defined by

$$(\kappa_1 \otimes \kappa_2)(A|x) = \int_{\mathcal{Y}} \int_{\mathcal{Z}} \mathbb{1}_A((x, y)) \kappa_2(dz|y) \kappa_1(dy, x) \quad \forall A \in \mathcal{G} \otimes \mathcal{F}$$

When  $P$  is a measure on  $(\mathcal{X}, \mathcal{F})$  and  $\kappa$  is a  $(\mathcal{X}, \mathcal{A}) \rightarrow (\mathcal{Y}, \mathcal{G})$  probability kernel, we define the  $P \otimes \kappa$  composition as a measure on  $(\mathcal{X} \times \mathcal{G}, \mathcal{F} \otimes \mathcal{G})$  defined by

$$(P \otimes \kappa)(A) = \int_{\mathcal{X}} \int_{\mathcal{Y}} \mathbb{1}_A((x, y)) \kappa(dy|x) dP(x)$$

**Theorem 1.7** (Ionescu-Tulcea theorem). *Let  $(\Omega_0, \mathcal{A}_0, P_0)$  be a probability space and  $(\Omega_i, \mathcal{A}_i)$  for  $i > 0$  a sequence of measurable spaces. For each  $i > 0$  let  $\kappa_i$  be a probability kernel from  $(\Omega^{i-1}, \mathcal{A}^{i-1}) \rightarrow (\Omega_i, \mathcal{A}_i)$ , where*

$$\Omega^i := \prod_{j=0}^i \Omega_j \quad \text{and} \quad \mathcal{A}^i := \bigotimes_{j=0}^i \mathcal{A}_j$$

*Then there exists a sequence of probability measures  $P_i := P_0 \otimes \bigotimes_{j=1}^i \kappa_j$  defined on  $(\Omega^i, \mathcal{A}^i)$ . Moreover, there exists a unique  $P$  probability measure on  $(\prod_{j=0}^{\infty} \Omega_j, \bigotimes_{j=0}^{\infty} \mathcal{A}_j)$ , so that*

$$P_i(A) = P(A \times \prod_{j=i+1}^{\infty} \Omega_j) \quad \forall A \in \mathcal{A}^i$$

The Ionescu-Tulcea plays a crucial role in the mathematical formulation of the Markov Decision Process, as we will see later. Unfortunately, the proof is out of the scope of this thesis, but it can be found in [1].



## 2 Reinforcement learning

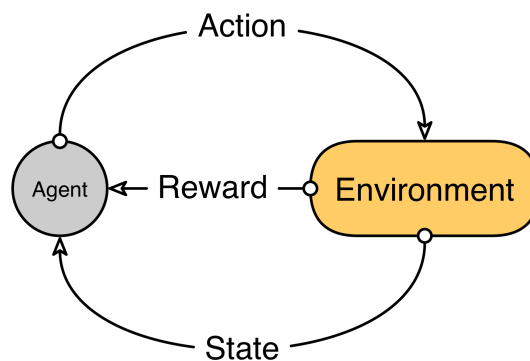
### 2.1 Foundation

In reinforcement learning, the primary goal is to find an agent (an intelligent actor) which interacts with the environment in a way that the accumulated reward is maximized. This chapter is based on [2, 3, 4]. The mathematical formulation of the "environment" is called Markov decision process (MDP).

**Definition 2.1.** An MDP is a 4-tuple  $(\mathcal{S}, \mathcal{A}, T, R)$  where:

- The  $\mathcal{S}$  measurable space is the state space,
- The  $\mathcal{A}$  measurable space is the action space,
- The  $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{P}(\mathcal{S})$  probability kernel is the transition function. It assigns a distribution over states to each state action pair.
- The  $R : L^\infty(\mathcal{S} \times \mathcal{A} \times \mathcal{S})$  is the reward function which assigns a reward for going from a state with an action to an other state. (*The reward function is often defined as a distribution over  $\mathbb{R}$  in order to introduce additional randomness.* )

The elements of  $\mathcal{S}$  are considered the inputs of the agent. So, the agent chooses an  $a \in \mathcal{A}$  action at an  $s \in \mathcal{S}$  state and then the MDP's transition function determines the next state. An MDP is the interaction of an agent that plays an action in each state that the environment processes by giving back the next state and reward. Note that the agent is not part of the MDP.



An MDP is called **episodic** if for every sequence of actions a terminal state is reached. The next step is to define the policy (the agent), so that we can mathematically formulate the interaction with an MDP.

**Definition 2.2.** The policy is a  $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$  probability kernel, which is a distribution over actions in every state.

**Definition 2.3.** Let  $\mathcal{M}$  be an MDP and  $\pi$  a policy, then the  $(S_0, A_0, R_1, S_1, A_1, R_2, \dots)$  sequence is called a **trajectory**, where  $S_i, A_i, R_{i+1}$   $i \geq 0$  are random variables such that.  $S_0$  is the starting position (or distribution in some cases) and  $A_i, S_i, R_{i+1}$  are defined by  $A_i \sim \pi(\cdot | S_i)$ ,  $S_{i+1} \sim T(\cdot | S_i, A_i)$ ,  $R_i = R(S_i, A_i, S_{i+1})$ . The measure defined on the trajectories is denoted by  $\tau$ .

Note that according to Theorem 1.7, the above  $\tau$  measure is well defined. Now, that we have established that  $\tau$  exists, we can define further variables. If it is not specified, the expectations are always defined over the  $\tau$  probability measure. From now on, a  $0 < \gamma < 1$  discount factor is fixed. Intuitively, a reward that is obtained after many steps is worth less than an equal but immediate reward. As the following definition shows, the discount is used to down-weight rewards obtained later.

**Definition 2.4** (expected return). Given an  $\mathcal{M}$  MDP and  $\pi$  policy we define the expected return  $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$  such that

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid s_t = s \right]$$

One of the most useful properties of the value-function is that we can rewrite the above equation by substituting back the definition of  $V^\pi$

$$V^\pi(s) = \mathbb{E}_\pi \left[ R_t + \sum_{k=1}^{\infty} \gamma^k R_{t+k} \right] \tag{2.1}$$

$$= \mathbb{E}_\pi R(s, a, s') + \gamma \mathbb{E}_\pi V(s') \tag{2.2}$$

The  $V^\pi$  value function has finite  $\infty$ -norm, because

$$\|V^\pi\| \leq \sup_{s \in \mathcal{S}} \mathbb{E}_\pi \left( \sum_{k=0}^{\infty} \gamma^k |R_{t+k}| \mid s_t = s \right) \leq \|R\|_\infty / (1 - \gamma)$$

The optimal expected return can be defined as:

**Definition 2.5** (optimal expected return).

$$V^*(s) = \max_{\pi \in \Pi} V^\pi(s)$$

It is also in our interest to define the expected value for actions.

**Definition 2.6** ( $Q$ -value). Given a  $\pi$  policy we define the  $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$   $Q$ -value function such that

$$Q^\pi(s, a) = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k} \mid s_t = s, a_t = a, \pi \right]$$

Similarly the optimal version is

**Definition 2.7** (Optimal  $Q$ -value).

$$Q^*(s, a) = \max_{\pi \in \Pi} Q^\pi(s, a)$$

**Definition 2.8** (Optimal policy).

$$\pi^*(s) = \arg \max_{a \in \mathcal{A}} Q^*(s, a)$$

The functions  $Q$  and  $V$  can be approximated by enrolling traces and their returns. However, we can approximate  $Q, V$  recursively which is computationally more efficient. The later one utilises the following reformulation of  $Q, V$ :

$$Q^\pi(s, a) = \mathbb{E}_{s' \sim T(\cdot|a,s)} (R(s, a, s') + \gamma V^\pi(s')) \quad (2.3)$$

$$V^\pi(s) = \mathbb{E}_{a \sim \pi(\cdot|s)} Q^\pi(s, a) \quad (2.4)$$

It is convenient to also introduce the advantage function:

**Definition 2.9.** The advantage function is defined by  $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ .

## 2.2 Value-based methods

In this section, we show some fundamental value-learning results, but we need the Banach fixed point theorem before that.

**Definition 2.10** (Contraction mapping). Let  $(X, d)$  be a complete metric space. Then a map  $T : X \rightarrow X$  is called a contraction mapping on  $C$  if there exists  $q \in [0, 1)$  such that  $d(T(x), T(y)) \leq qd(x, y)$  for all  $x, y \in X$ .

**Theorem 2.11** (Banach fixed point theorem). *Let  $(X, d)$  be a non-empty complete metric space with a contraction mapping  $T : X \rightarrow X$ . Then  $T$  admits a unique fixed-point  $x^*$  in  $X$ . Furthermore, for an arbitrary  $x_0 \in X$  starting point, the  $(x_n)_{n \in \mathbb{N}}$  sequence defined as  $x_n = T(x_{n-1})$  for  $n \geq 1$  converges to  $x^*$ , i.e.,  $\lim_{n \rightarrow \infty} x_n = x^*$ .*

The Bellman operator leverages the reformulation in Eq 2.1 of the value function to iteratively update the value-function. As we will see later, this method converges to the optimal value-function and an optimal-policy.

**Definition 2.12.** Let  $\pi$  be an arbitrary policy. The  $\mathcal{B}^\pi : L^\infty(\mathcal{S}) \rightarrow L^\infty(\mathcal{S})$  Bellman operator with respect to the  $\pi$  policy is defined such that

$$(\mathcal{B}^\pi V)(s) = \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')]$$

Moreover, the  $\mathcal{B}^* : L^\infty(\mathcal{S}) \rightarrow L^\infty(\mathcal{S})$  Bellman optimality operator is defined such that

$$(\mathcal{B}^* V)(s) = \sup_{a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')]$$

First of all, we show that the above two operators are well-defined, i.e., for any  $V \in L^\infty(\mathcal{S})$  the  $\mathcal{B}V$  is in  $L^\infty(\mathcal{S})$ .

*Proof.* Let  $C = \|V\|_\infty$ , then we have

$$\begin{aligned} \|\mathcal{B}^\pi V\|_\infty &= \sup_{s \in \mathcal{S}} |\mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')]| \\ &\leq \sup_{s \in \mathcal{S}} \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} [ |R(s, a, s')| + \gamma |V(s')| ] \\ &\leq \sup_{s, a, s'} |R(s, a, s')| + \gamma C \end{aligned}$$

Hence,  $\mathcal{B}^\pi V \in L^\infty(\mathcal{S})$ . Similarly follows that  $\mathcal{B}^*$  is also well defined:

$$\begin{aligned} \|\mathcal{B}^* V\|_\infty &= \sup_{s \in \mathcal{S}} \sup_{a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')] \\ &\leq \sup_{s, a, s'} |R(s, a, s')| + \gamma C \end{aligned}$$

□

We list some basic properties of both Bellman-operators which will be of great use later.

**Lemma 2.13.** *For the Bellman-operator it holds that*

1. *For any  $\pi$  policy  $\mathcal{B}^\pi \leq \mathcal{B}^*$  where  $\leq$  means that for every  $V \in L^\infty(\mathcal{S})$  value-function we have  $\mathcal{B}^\pi V \leq \mathcal{B}^* V$ .*
2. *Let  $U, V \in L^\infty(\mathcal{S})$  be two value-function such that  $U \leq V$ , then  $\mathcal{B}^* U \leq \mathcal{B}^* V$ .*

*Proof.* The first point is an immediate consequence of the definition.

Once again, the second point follows from the definition of  $\mathcal{B}^*$ :

$$\begin{aligned} \mathcal{B}^* U &= \sup_{a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma U(s')] \\ &\leq \sup_{a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')] \\ &= \mathcal{B}^* V \end{aligned}$$

□

**Definition 2.14** (Greedy policy). We say that a  $\pi$  policy is greedy with respect to a  $V$  value-function if

$$\mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s')]$$

is maximalized for every  $s \in \mathcal{S}$ . Or equivalently  $\mathcal{B}^\pi V = \mathcal{B}^* V$ .

*In the finite MDP case the greedy policy always exists, but in general, the supremum is not necessarily achieved by a policy.*

**Theorem 2.15.** *The only fix point of  $\mathcal{B}^\pi$  is  $V^\pi$  and for any  $V_0 \in L^\infty(\mathcal{S})$*

$$\lim_{n \rightarrow \infty} (\mathcal{B}^\pi)^n V_0 = V^\pi$$

where the limit is taken according the  $\infty$ -norm.

Similarly, the only fix point of  $\mathcal{B}^*$  is  $V^*$  and

$$\lim_{n \rightarrow \infty} (\mathcal{B}^*)^n V_0 = V^*$$

Furthermore, the  $\pi$  greedy policy with respect to  $V^*$  is an optimal policy.

*Proof.* First of all, it follows from Equation 2.1 that  $V^\pi$  is a fixpoint of  $\mathcal{B}^\pi$ . Our goal is to show that  $V^\pi$  is a unique fixpoint and that the limit of iterations by  $\mathcal{B}^\pi$  is always  $V^\pi$ , to do so we want to use Banach's fixpoint theorem for what it is sufficient to prove that  $\mathcal{B}^\pi$  is a contraction.

$$\begin{aligned} \|\mathcal{B}^\pi U - \mathcal{B}^\pi V\|_\infty &= \gamma \sup_{s \in \mathcal{S}} \left| \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} [U(s') - V(s')] \right| \\ &\leq \gamma \sup_{s \in \mathcal{S}} \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} |U(s') - V(s')| \\ &\leq \gamma \sup_{s \in \mathcal{S}} \mathbb{E}_{a \sim \pi(\cdot|s)} \mathbb{E}_{s' \sim T(\cdot|s,a)} \|U - V\|_\infty \\ &\leq \gamma \|U - V\|_\infty \end{aligned}$$

The above calculation shows that  $\mathcal{B}^\pi$  is a contraction, hence we proved the first part of theorem.

The second part is slightly more tricky because the fact that  $V^*$  is a fix point does not follow immediately. From a similar calculation as the previous one it follows that  $\mathcal{B}^*$  is a contraction. By using the definition and the fact that

$$\left| \sup_{x \in H} f(x) - \sup_{x \in H} g(x) \right| \leq \sup_{x \in H} |f(x) - g(x)|$$

we obtain that

$$\begin{aligned} \|\mathcal{B}^* U - \mathcal{B}^* V\|_\infty &\leq \gamma \sup_{s \in \mathcal{S}, a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} |U(s') - V(s')| \\ &\leq \gamma \sup_{s \in \mathcal{S}, a \in \mathcal{A}} \mathbb{E}_{s' \sim T(\cdot|s,a)} \|U - V\|_\infty \\ &\leq \gamma \|U - V\|_\infty \end{aligned}$$

So,  $\mathcal{B}^*$  is contraction, hence has a  $V$  fixpoint.

Let  $\pi$  be a greedy policy with respect to  $V$ . By definition  $\mathcal{B}^\pi V = \mathcal{B}^* V$ . Since  $V$  is the fixpoint of  $\mathcal{B}^*$ , we have  $\mathcal{B}^\pi V = V$ . Fortunately, we know that  $\mathcal{B}^\pi$  has a unique fix point,  $V^\pi$ , thus  $V^\pi = V$ .

Now, let  $\pi$  be an arbitrary  $\pi$  policy. According 2.13.1, we know that

$$V^\pi = \mathcal{B}^\pi V^\pi \leq \mathcal{B}^* V^\pi$$

Applying 2.13.2 to  $V^\pi$  and  $\mathcal{B}^*V^\pi$  we obtain the following chain of inequalities

$$V^\pi \leq \mathcal{B}^*V^\pi \leq (\mathcal{B}^*)^2V^\pi \leq \dots \leq \lim_{n \rightarrow \infty} (\mathcal{B}^*)^n V^\pi = V$$

where  $V$  is the unique fixpoint of  $\mathcal{B}^*$ . So,  $V^\pi \leq V$ . Since  $\pi$  was arbitrary,  $V$  is the optimal value-function ( $V = V^*$ ) and  $\pi$  is an optimal policy. □

The last theorem let us find the best policy by iteratively applying the  $\mathcal{B}^*$  operator and then taking the greed policy.

**Theorem 2.16.** *Let  $\pi_0$  be an arbitrary policy and  $\pi$  be the greedy policy with respect to  $V^{\pi_0}$ . Then  $V^\pi \geq V^{\pi_0}$ , in other words,  $\pi$  is an improved policy compared to  $\pi_0$*

*Proof.* We have  $\mathcal{B}^\pi V^{\pi_0} = \mathcal{B}^*V^{\pi_0} \geq \mathcal{B}^{\pi_0}V^{\pi_0} = V^{\pi_0}$ . Applying 2.13.2 to both sides infinitely many times we obtain that  $V^\pi \geq V^{\pi_0}$ . □

That means that we can get an optimal policy by estimating the value function for the current policy and then improving upon it by taking the greedy policy with respect to the estimated value-function. According to 2.15, the latter method will converge to an optimal policy. There are several value-based methods with different advantages that relies on the  $V$  function. However, we could do the same for  $Q$ -functions:

**Definition 2.17** (Bellman operator). The  $\mathcal{B}$  Bellman operator is an operator that is a mapping between  $Q \in L^\infty(\mathcal{S} \times \mathcal{A})$   $Q$ -functions such that

$$\mathcal{B}Q(s, a) = \mathbb{E}_{s' \sim T(s'|s, a)} \left( R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a') \right)$$

Other similar theorems can be obtained for  $Q$ -functions.

## 2.3 Temporal difference learning

Temporal difference learning (TD for short) uses the idea of *bootstrapping* which handles predictions as targets. In this section, we introduce the most basic TD algorithm and compare it to a Monte-Carlo based method. Finally, we generalize the two algorithms giving rise to a family of TD algorithms.

### Basic TD algorithm

Let  $\mathcal{M}$  be a finite MDP and fix a  $\pi$  policy. We wish to estimate the value-function associated with  $\pi$ . Let  $\hat{V}_t$  denote the estimate of  $V^\pi$  in the  $t$ th time step. Previously, we saw that the Bellman operator can be used to approximate  $V^\pi$ . The TD algorithm performs the following calculation at each time step:

$$\begin{aligned}\delta_{t+1} &= R_{t+1} + \gamma \hat{V}_t(S_{t+1}) - \hat{V}_t(S_t) \\ \hat{V}_{t+1}(s) &= \hat{V}_t(s) + \alpha_t \delta_{t+1} \mathbb{1}_{S_t=s}\end{aligned}$$

Where  $(\alpha_t)_{t=0}^\infty$  is the step-size sequence chosen by the user. Later, we show that if the  $(\alpha_t)_{t=0}^\infty$  meets some criteria then convergence is always ensured.

The temporal difference name comes from the  $\delta$  variable which captures the difference between values of states corresponding to successive time steps. Let  $\mathcal{T}_0 : L^\infty(\mathcal{S}) \rightarrow L^\infty(\mathcal{S})$  be an operator over value-functions such that

$$\mathcal{T}_0 V(s) = \mathbb{E}_{a \sim \pi(\cdot|s), s' \sim T(\cdot|s,a)} [R(s, a, s') + \gamma V(s') - V(s)]$$

The above operator calculates the  $\delta$  difference and can be decomposed as  $\mathcal{T}_0 V = \mathcal{B}^\pi V - V$ . Recall that  $\mathcal{B}^\pi$  has a unique solution  $V^\pi$ , thus the unique solution of  $\mathcal{T}_0 V = 0$  is  $V^\pi$ .

---

**Algorithm 1** Temporal difference learning

---

```
1: while V converges do
2:    $S, R, S' \leftarrow \text{SampleStep}()$ 
3:    $\delta \leftarrow R + \gamma \cdot V[S'] - V[S]$ 
4:    $V[S] \leftarrow V[S] + \alpha \cdot \delta$ 
```

---

### Every-visit Monte-Carlo

In Every-visit Monte-Carlo we omit bootstrapping and instead approximate value-functions by enrolling full trajectories. Let  $(S_t, R_{t+1}, S'_{t+1})_{t=0}^\infty$  a continual sampling from the  $\mathcal{M}$  MDP following the  $\pi$  policy such that  $S_{t+1} = S'_{t+1}$  if  $S'_{t+1}$  is not a terminal state and  $S_{t+1}$  be the starting state otherwise. Furthermore, let  $T_t$  denote the next time step, when  $\mathcal{M}$  was restarted. We define  $\mathcal{R}_t$  as:

$$\mathcal{R}_t = \sum_{k=t}^{T_t-1} \gamma^{k-t} R_{k+1}$$

Clearly,  $V^\pi(S_t) = \mathbb{E}(\mathcal{R}_t|S_t)$ . Finally, we define the update rule as

$$\hat{V}_{t+1}(s) = \hat{V}_t(s) + \alpha_t(\mathcal{R}_t - \hat{V}_t(s))\mathbb{1}_{S_t=s}$$

---

**Algorithm 2** Every-visit Monte-Carlo

---

```

1: while V converges do
2:    $sum \leftarrow 0$ 
3:    $S_0, R_1, S_1, \dots, S_{T-1}, R_T \leftarrow \text{SampleEpisode}()$ 
4:   for  $t = T - 1$  to 0 do
5:      $sum \leftarrow R_{t+1} + \gamma \cdot sum$ 
6:      $V[S_t] \leftarrow (1 - \alpha) \cdot V[S_t] + \alpha \cdot sum$ 

```

---

### General TD algorithm

The main difference between the two previous algorithm is that we depend differently on our previous estimate. In case of the basic TD, if the value of successor states differs from their true value by a significant amount, then the convergence of the root value will be slower compared to the Monte-Carlo method. The Monte-Carlo variant does not depend on an accurate estimation of successor state values, hence the increased speed of convergence. At the same time, we have to enroll full trajectories to estimate the value of a state which has higher variance and therefore can slow down the convergence in some cases.

Fortunately, we can generalise the two previous algorithms by introducing a  $\lambda \in [0, 1]$  control variable that, in essence, determines how much we take advantage of each algorithm. We define  $\mathcal{R}_{t:k}$  as

$$\mathcal{R}_{t:r}^\lambda = \gamma^{r+1}\hat{V}(S_{t+r+1}) + \sum_{k=t}^{t+r} \gamma^{k-t} R_{k+1}$$

and the  $\mathcal{R}_t^\lambda$  target is defined as

$$\mathcal{R}_t^\lambda = \sum_{k=0}^{\infty} (1 - \lambda)\lambda^k \mathcal{R}_{t:k}$$

**Lemma 2.18.** *When  $\lambda = 0, 1$  the  $\mathcal{R}_t^\lambda$  target is the same as in the two previous algorithm.*

- If  $\lambda = 0$ , then  $\mathcal{R}_t^\lambda = R_{t+1} + \gamma\hat{V}(S_{t+1})$ .
- The limit of  $\mathcal{R}_t^\lambda$  in  $\lambda \rightarrow 1$  exists and

$$\lim_{\lambda \rightarrow 1} \mathcal{R}_t^\lambda = \sum_{r=t}^{\infty} \gamma^{r-t} R_{k+1}$$



*Proof.* The first claim immediately follows from  $\mathcal{R}_{t:0} = R_{t+1} + \gamma\hat{V}(S_{t+1})$  and that the weight of every other term is zero.

For the second one, we need to write out the equation and then swap the sums.

$$\begin{aligned}
\lim_{\lambda \rightarrow 1} \mathcal{R}_t^\lambda &= \lim_{\lambda \rightarrow 1} \left[ \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \left( \gamma^{r+1}\hat{V}(S_{t+r+1}) + \sum_{k=t}^{t+r} \gamma^{k-t} R_{k+1} \right) \right] \\
&= \lim_{\lambda \rightarrow 1} \left[ \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \gamma^{r+1}\hat{V}(S_{t+r+1}) + \sum_{k=t}^{\infty} \gamma^{k-t} R_{k+1} \sum_{r=k-t}^{\infty} (1-\lambda)\lambda^r \right] \\
&= \lim_{\lambda \rightarrow 1} \left[ \sum_{k=t}^{\infty} \gamma^{k-t} \lambda^{k-t} R_{k+1} \right] \\
&= \sum_{r=t}^{\infty} \gamma^{r-t} R_{k+1}
\end{aligned}$$

where we used that in the second line the limit of the first sum is zero, because

$$\begin{aligned}
\left| \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \gamma^{r+1}\hat{V}(S_{t+r+1}) \right| &\leq \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \gamma^{r+1} \|\hat{V}\|_{\infty} \\
&= (1-\lambda)\gamma \|\hat{V}\|_{\infty} (1-\lambda\gamma)^{-1}
\end{aligned}$$

which clearly converges to zero. □

**Lemma 2.19.** *The  $\mathcal{R}_t^\lambda - \hat{V}(S_t)$  can be decomposed as*

$$\mathcal{R}_t^\lambda - \hat{V}(S_t) = \sum_{k=t}^{\infty} (\gamma\delta)^{k-t} \delta_k$$

where  $\delta_k$  is the temporal difference in the  $k$ th step:  $\delta_k = R_{k+1} + \gamma\hat{V}(S_{k+1}) - \hat{V}(S_k)$ .

*Proof.* The proof is a simple but long calculation:

$$\begin{aligned}
\mathcal{R}_t^\lambda - \hat{V}(S_t) &= \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \left( \gamma^{r+1}\hat{V}(S_{t+r+1}) + \sum_{k=t}^{t+r} \gamma^{k-t} R_{k+1} \right) - \hat{V}(S_t) \\
&= \sum_{r=0}^{\infty} (1-\lambda)\lambda^r \gamma^{r+1}\hat{V}(S_{t+r+1}) + \sum_{k=t}^{\infty} \gamma^{k-t} R_{k+1} \sum_{r=k-t}^{\infty} (1-\lambda)\lambda^r - \hat{V}(S_t) \\
&= \sum_{k=t}^{\infty} \gamma^{k-t} \lambda^{k-t} \gamma \hat{V}(S_{k+1}) - \sum_{k=t+1}^{\infty} \gamma^{k-t} \lambda^{k-t} \hat{V}(S_k) + \sum_{k=t}^{\infty} \gamma^{k-t} \lambda^{k-t} R_{k+1} - \hat{V}(S_t) \\
&= \sum_{k=t}^{\infty} \gamma^{k-t} \lambda^{k-t} \left( R_{k+1} + \gamma\hat{V}(S_{k+1}) - \hat{V}(S_k) \right) \\
&= \sum_{k=t}^{\infty} (\gamma\delta)^{k-t} \delta_k
\end{aligned}$$

□

The above decomposition allows us to update  $\hat{V}$  by the following rule:

$$\begin{aligned}\delta_{t+1} &= R_{t+1} + \gamma \hat{V}_t(C_{t+1}) - \hat{V}_t(S_t) \\ z_{t+1} &= \gamma \lambda z_t + \mathbb{1}_{S_t = s} \\ \hat{V}_{t+1} &= \hat{V}_t + \alpha_t \delta_{t+1} z_{t+1}\end{aligned}$$

where  $z$  is often called the eligibility trace. Note that Lemma 2.19 works only if  $\alpha_t$  is constant throughout the trajectory. This means we can change  $\alpha$  only when a new trajectory is started. Fortunately, the following theorem states that even with this restriction, convergence is still easily ensured.

**Theorem 2.20** (Robbins-Monro condition). *Let  $(\alpha_t)_{t=0}^{\infty} \subset \mathbb{R}_+$  be a step-size sequence such that the Robbins-Monro condition is satisfied:*

$$\sum_{t=0}^{\infty} \alpha_t = \infty, \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty$$

*Then the TD( $\lambda$ ) algorithm converges.*

The proof of this theorem is surprisingly long [5], so we do not include it in the thesis.

---

**Algorithm 3** Lambda-temporal difference learning

---

- 1: **while** V converges **do**
  - 2:    $S, R, S' \leftarrow \text{SampleStep}()$
  - 3:    $\delta \leftarrow R + \gamma \cdot V[S'] - V[S]$
  - 4:    $z \leftarrow \gamma \cdot \lambda \cdot z + \mathbb{1}_{S=s}$
  - 5:    $V \leftarrow V + \alpha \cdot \delta \cdot z$
-

### 3 Planning

This chapter is based on [6]. We introduce the most fundamental ideas of reinforcement learning and some more advanced ideas. Before we dive deeper into reinforcement learning, we will consider one of the most basic settings of MDPs called the multi-armed bandit problem. We have introduced learning algorithms capable of finding the best policy by repeatedly simulating trajectories from the environment. How efficient are these algorithms, and how can we find the best ones? In reinforcement learning, there is the problem of explore-exploit trade-off which briefly means that we cannot play the best action each time and try new ones that might perform the current best one. In the previous chapter, this problem was not relevant because we only focused on finding the best policy regardless of the number of queries and updates we have to take, but once we try to optimise the convergence time, this explore-exploit problem naturally arises.

#### 3.1 Multi-armed bandits

In the  $k$ -armed bandits learning problem, an actor is faced with a choice of  $k$  different actions, resulting in a reward paid off from stationary probability distribution that depends on the selected action. Equivalently, the  $k$ -armed bandits problem is an MDP with one start state and one end state. The goal is to maximise the expected total reward over some time period.

**Definition 3.1.** Let  $\mathcal{A}$  be a set of actions and  $R(\cdot|a) \rightarrow \mathcal{P}(\mathbb{R})$  a probability distribution for each  $a \in \mathcal{A}$  action. We seek a series of actions  $a_t \in \mathcal{A}$  such that  $a_t$  depends only on the previously observed rewards  $r_k \sim R(\cdot|a_k)$  ( $1 \leq k < t$ ) and

$$\mathbb{E} \left[ \frac{1}{T} \sum_{t=1}^T r_t \right]$$

is maximalized.

**Definition 3.2.** An  $\mathcal{E}$  class of bandits is a set of bandits (MDP) and it is unstructured if there exist  $\mathcal{M}_a$  sets of distributions for every  $a \in \mathcal{A}$  such that

$$\mathcal{E} = \prod_{a \in \mathcal{A}} \mathcal{M}_a$$

Intuitively it means that we cannot deduce any information about other actions that we played. Even when the  $\mathcal{E}$  environment class is unstructured, we still have a rather large family. Fortunately, we can narrow down our interest to the gaussian environments.

**Definition 3.3.** The Gaussian environment class is defined as

$$\mathcal{E}_{\mathcal{N}}^k = \prod_{a \in \mathcal{A}} \{\mathcal{N}(\mu, \sigma)\}$$

where  $k = |\mathcal{A}|$  is the number of arms.

Through out this chapter  $\nu$  denotes an element of  $\mathcal{E}_{\mathcal{N}}^k$  and  $\mu_a(\nu) = \mathbb{E}_{x \sim P_a}(x)$  the expected return of action  $a$ . Furthermore, we will follow the notion of previous chapters and let  $\mu^*(\nu) = \max_{a \in \mathcal{A}} \mu_a(\nu)$  and  $\Delta_a(\nu) = \mu^*(\nu) - \mu_a(\nu)$ . Let  $T_a(n)$  denote the number of times  $a$  was played during the first  $n$  step and  $\hat{\mu}_a(n) = \frac{1}{T_a(n)} \sum_{t=1}^n x_t \mathbb{1}_{A_t=a}$  the average reward received playing the  $a$ th arm during the first  $n$  step. We also use the  $*$  =  $\arg \max_{a \in \mathcal{A}} \mu_a(\nu)$  notion, i.e.  $\hat{\mu}_*$  is the estimated reward for the best arm.

**Definition 3.4.** A policy  $\pi$  is a sequence  $(\pi_t)_{t=1}^n$  of probability distributions such that  $\pi_t$  is a probability kernel from  $\Omega^{t-1}$  to  $\mathcal{A}$ , i.e.

$$A_t \sim \pi_t(\cdot | a_1, x_1, \dots, a_{t-1}, x_{t-1})$$

where  $\Omega^{t-1}$  is the measure space defined in Theorem 1.7 or alternatively the measure space where  $(a_1, x_1, \dots, x_{t-1})$  is defined.

**Definition 3.5.** For every  $\pi$  policy and  $\nu \in \mathcal{E}$  we define the regret as

$$R(\pi, \nu) = n\mu^*(\nu) - \mathbb{E} \left( \sum_{i=1}^n X_i \right) = \mathbb{E} \left( \sum_{i=1}^n \Delta_{A_i} \right)$$

In the bandit problem, we seek to achieve as much accumulated reward as just we can. So far, this is not a well-defined problem, but we would like to say something about the followings

- Whether it is true that

$$\forall \nu \in \mathcal{E}, \lim_{n \rightarrow \infty} \frac{R(\pi, \nu)}{n} = 0$$

- How small can the regret be asymptotically? I.e, for what  $C : \mathcal{E} \rightarrow [0, \infty)$  and  $f : \mathbb{N} \rightarrow [0, \infty]$  functions satisfy the following

$$\forall n \in \mathbb{N}, \forall \nu \in \mathcal{E}, \quad R_n(\pi, \nu) \leq C(\nu)f(n)$$

There are many families of bandits, and each has its own merit to be thoroughly studied; however, in the thesis, we only consider the Gaussian environment and restrict our focus to those where the deviation is one. At first glance, it seems complicated to determine what kind of  $(\pi_t)_{t=1}^{\infty}$  policies are optimal because the dependence on the whole history gives rise to an overwhelmingly large family of functions. However, the following lemma reassures us that we can consider  $(\pi_t)_{t=1}^{\infty}$  policies in a less-complicated form.

**Theorem 3.6.** *Let  $\mathcal{E}$  be the 1-gaussian bandit class and  $(\pi_t)_{t=1}^{\infty}$  be a policy, then there is a  $\pi'(\cdot | \hat{\mu}, T)$  policy which achieves the same regret as  $\pi$ .*

*Proof.* The main idea is that  $\hat{\mu}$  and  $T$  captures every information from the  $(a_1, x_1, \dots, x_n)$  history. We want to show that  $\hat{\mu}, T$  is a sufficient statistic for the  $\mathcal{E}$  bandit class. To prove this we will use the Fisher-Neyman factorization theorem which states that that

an  $F(x)$  statistic is sufficient for the underlying  $\Theta$  parameter precisely if the density function can be factorized as

$$f_{\Theta}(x) = h(x)g_{\Theta}(F(x))$$

In our case  $\mu$  is the underlying parameter and  $F(\hat{\mu}, T)$  is the statistic. The density function:

$$\begin{aligned} \rho_{\mu}(a_1, x_1, \dots, x_n) &= \prod_{i=1}^n \pi_i(a_i | a_1, x_1, \dots, x_{i-1}) \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(x_i - \mu_{a_i})^2} \\ &= \left( \prod_{i=1}^n \pi_i(a_i | a_1, x_1, \dots, x_{i-1}) \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x_i^2} \right) \exp \left( \sum_{i=1}^n x_i \mu_{a_i} - \frac{1}{2} \sum_{i=1}^n \mu_{a_i}^2 \right) \end{aligned}$$

The left part of the last equation is a function independent from the  $\Theta$  parameter, while the right part is function in  $\sum_{i=1}^n x_i \mu_{a_i} - \frac{1}{2} \sum_{i=1}^n \mu_{a_i}^2$ . The latter one can be rewritten as  $\sum_{a \in \mathcal{A}} \mu_a \hat{\mu}_a T_a(n) - \frac{1}{2} \sum_{a \in \mathcal{A}} T_a(n) \mu_a^2$  which is a  $\mu$ -parametrized function in  $\hat{\mu}$  and  $T$ . Since we showed that it is impossible to gain more information from the  $(a_1, x_1, \dots, x_n)$  history, than it is possibly from  $\hat{\mu}, T$ , we can deduce that  $\pi'$  can be constructed such that we take

$$\pi'(a | \nu, \tau) = \sum_{(a_1, \dots, x_n)} \mathbb{P}(a_1, \dots, x_n | \nu, \tau) \pi_{\sum_i \tau_i}(a | a_1, \dots, x_n)$$

where  $\mathbb{P}(a_1, \dots, x_n | \nu, \tau)$  does not depend on  $\mu$ , since  $\hat{\mu}, T$  is a sufficient statistic, therefore the above sum can be calculated regardless of the  $\mu$  parameter.  $\square$

### The Explore-Then-Commit algorithm

The first algorithm we will examien is Explore-Then-Commit (ETC), where the first few steps are made regardless of the previous outcomes, and then we commit to the action which is best according to estimation.

**Definition 3.7.** Let  $m \in \mathbb{Z}_+$  be a fixed positive integer. The ETC algorithm explores in the first  $mk$  steps, i.e,  $A_t = (t \bmod k)$  if  $t \leq mk$  and chooses the best afterwards:  $A_t = \arg \max_a \hat{\mu}_a(mk)$ .

**Theorem 3.8.** Let  $n$  be the number of steps taken and  $m \leq n/k$ . The following upper bound for the regret holds when ETC interacts with a 1-subgaussian environment.

$$R_n \leq m \sum_{a=1}^k \Delta_a + (n - mk) \sum_{a=1}^k \Delta_a e^{-\frac{m\Delta_a^2}{4}}$$

we will need the following lemma:

**Lemma 3.9.** Let  $X \sim \mathcal{N}(\mu, \sigma)$  a normal variable, then for any  $\varepsilon \geq 0$ ,

$$\mathbb{P}(X \geq \varepsilon) \leq e^{-\frac{\varepsilon^2}{2\sigma^2}}$$

*Proof.* The following calculation is a generic method to obtain estimations, called Cramér-Chernoff method. Let  $\lambda = \varepsilon/\sigma^2 > 0$ .

$$\begin{aligned}\mathbb{P}(X \geq \varepsilon) &= \mathbb{P}(e^{\lambda X} \geq e^{\lambda\varepsilon}) \\ &\leq \mathbb{E}(e^{\lambda X})e^{-\lambda\varepsilon} \\ &= e^{\frac{\lambda^2\sigma^2}{2}-\lambda\varepsilon}\end{aligned}$$

where the first implication is the Markov's inequality and second one comes from the momentum generating function of the normal distribution. Substituting back  $\lambda = \varepsilon/\sigma^2 > 0$  gives us the upper bound stated in the lemma.  $\square$

*Proof of Theorem 3.8.* First of all,  $R_n$  can be decomposed as  $R_n = \sum_a \Delta_a \mathbb{E}(T_a(n))$ . We have

$$\mathbb{E}(T_a(n)) = m + (n - mk)\mathbb{P}\left(\hat{\mu}_a(mk) \geq \max_{b \neq a} \hat{\mu}_b\right)$$

We estimate the following probability on the right-hand side:

$$\begin{aligned}\mathbb{P}\left(\hat{\mu}_a(mk) \geq \max_{b \neq a} \hat{\mu}_b\right) &= \mathbb{P}(\mu_i(mk) \geq \hat{\mu}^*(mk)) \\ &= \mathbb{P}(\mu_i(mk) - \mu_a - (\hat{\mu}^*(mk) - \mu^*) \geq \Delta_a)\end{aligned}$$

Since  $\mathcal{E}$  is a 1-subgaussian environment we know that the probability variable is normal distribution with zero mean and  $\sqrt{2/m}$  deviation. It is known that if  $X$  is a  $\sigma$ -subgaussian, then for any  $\epsilon \geq 0$

$$\mathbb{P}(X \geq \epsilon) \leq e^{-\frac{\epsilon^2}{2\sigma^2}}$$

Applying the latter result we get that

$$\mathbb{E}(T_a(n)) \leq m + (n - mk)e^{-\frac{m\Delta_a^2}{4}}$$

Adding together these terms yields the upper-bound stated in the theorem.  $\square$

When  $k = 2$ , then the upper-bound is minimal if  $m = \frac{4\log(n\Delta_a^2/4)}{\Delta_a^2}$ , where  $a$  is the non-optimal action. Substituting the latter one back, we get that  $R_n \leq 1 + C\sqrt{n}$ .

## The Upper Confidence Bound

In the ETC algorithm we explore each arm  $m$  times and then stuck to the best one. Intuitively, there are two weakness. First, it might be obvious after fewer exploration that an arm is not the best one. Second, we may achieve a better performance by gradually reducing the amount of exploration instead of cutting it off. In the Upper Confidence Bound algorithm (UCB), we use an upper-bound, as the name indicates, to determine how much better an action might be. Before that we need the following technical lemma.

**Lemma 3.10.** Let  $X_1, X_2, \dots, X_n \sim \mathcal{N}(\mu, 1)$  be independent normal variables with  $\mu$  mean and  $\hat{\mu} = \frac{1}{n} \sum_{i=1}^n X_i$ . Then for all  $\delta \in (0, 1)$  we have

$$\mathbb{P} \left( \mu \geq \hat{\mu} + \sqrt{\frac{2 \log(1/\delta)}{n}} \right) \leq \delta$$

*Proof.* It immediately follows from Lemma 3.9 because  $\mu \sim \mathcal{N}(\hat{\mu}, 1/\sqrt{n})$  and then substituting  $\varepsilon = \sqrt{2 \log(1/\delta)/n}$  yields the theorem.  $\square$

**Definition 3.11.** Let  $\delta > 0$  be the confident level. The UCB formula defined as

$$UCB_a(t, \delta) = \begin{cases} \infty & \text{if } T_a(t) = 0 \\ \hat{\mu}_a(t) + \sqrt{\frac{2 \log(1/\delta)}{T_a(t)}} & \text{otherwise} \end{cases}$$

**Definition 3.12.** Fix a  $\delta \in \mathbb{R}_+$  confidence level series. The UCB algorithm always chooses the step which maximalise the UCB formula:

$$A_{t+1} = \arg \max_a UCB_a(t, \delta)$$

As we mentioned, there are two ways to improve on ETC. The UCB formula allows us to explore the most 'interesting' arms, hence we gave a better alternative for the first one. As we can see, we still need to determine the best  $\delta$  confidence level. If the time horizon is known in advance, then  $\delta = n^{-2}$  is a reasonable choice as the following theorem shows.

**Theorem 3.13.** If  $\delta = 1/n^2$ , the regret of UCB is bounded by

$$R_n \leq 8\sqrt{nk \log(n)} + 3 \sum_{i=1}^k \Delta_i$$

*Proof.* Fix an  $a$  non-optimal arm and let  $t_u^a$  denote the time step when the  $a$ th arm was pulled for the  $u$ th time. Note that  $t_u^a$  is a probability variable. Fix a  $u_a > 0$  integer which we determine later. For every arm let us consider the following event:

$$G_a = \left\{ \mu^* < \min_{t \leq n} UCB_*(t, \delta) \right\} \cap \left\{ \hat{\mu}_a(t_{u_a}^a) + \sqrt{\frac{2}{u_a} \log(1/\delta)} < \mu^* \right\}$$

First, we show that if  $G_a$  occurs, then  $T_a(n) \leq u_a$ . Assume the contrary and let  $t$  be a time step such that  $T_a(t-1) = u_a$  and  $A_t = a$ . We have the following

$$\begin{aligned} UCB_a(t-1, \delta) &= \hat{\mu}_a(t-1) + \sqrt{\frac{2 \log(1/\delta)}{T_a(t-1)}} \\ &= \hat{\mu}_a(t-1) + \sqrt{\frac{2 \log(1/\delta)}{u_a}} \\ &< \mu^* < UCB_*(t-1, \delta) \end{aligned}$$

Where we used that  $T_a(t-1) = u_a$  and that  $G_a$  occurs implying the last two inequality. Since the UCB score is higher for  $*$ , we have  $A_t = *$  which is a contradiction. Secondly we show that  $\mathbb{P}(G_a^c)$  has low probability.

$$G_a^c = \left\{ \mu^* \geq \min_{t \leq n} UCB_*(t, \delta) \right\} \cup \left\{ \hat{\mu}_a(t_{u_a}^a) + \sqrt{\frac{2}{u_a} \log(1/\delta)} \geq \mu^* \right\}$$

The probability of the first event can be bounded the following way

$$\begin{aligned} \mathbb{P} \left( \mu^* \geq \min_{t \leq n} UCB_*(t, \delta) \right) &= \mathbb{P} \left( \mu^* \geq \min_{s \leq T_*(n)} \left\{ \hat{\mu}_*(t_s^*) + \sqrt{\frac{2 \log(1/\delta)}{s}} \right\} \right) \\ &\leq \mathbb{P} \left( \bigcup_{s \leq T_*(n)} \left\{ \mu^* \geq \hat{\mu}_*(t_s^*) + \sqrt{\frac{2 \log(1/\delta)}{s}} \right\} \right) \\ &\leq \sum_{s \leq T_*(n)} \mathbb{P} \left( \mu^* \geq \hat{\mu}_*(t_s^*) + \sqrt{\frac{2 \log(1/\delta)}{s}} \right) \\ &\leq T_*(n) \delta \leq n \delta \end{aligned}$$

For the second event we have the following estimation.

$$\mathbb{P} \left( \hat{\mu}_a(t_{u_a}^a) + \sqrt{\frac{2}{u_a} \log(1/\delta)} \geq \mu^* \right) = \mathbb{P} \left( \hat{\mu}_a(t_{u_a}^a) - \mu_a \geq \Delta_a - \sqrt{\frac{2}{u_a} \log(1/\delta)} \right)$$

Using Lemma 3.9 for the  $\hat{\mu}_a(t_{u_a}^a) - \mu_a$  normal distribution with 0 mean and  $\sqrt{1/u_a}$  deviation, we get

$$\mathbb{P} \left( \hat{\mu}_a(t_{u_a}^a) + \sqrt{\frac{2}{u_a} \log(1/\delta)} \geq \mu^* \right) \leq \exp \left( -\frac{u_a}{2} \left( \Delta_a - \sqrt{\frac{2 \log(1/\delta)}{u_a}} \right)^2 \right)$$

Choosing  $u_a = \frac{8 \log(1/\delta)}{\Delta_a^2}$ , gives us

$$\mathbb{P} \left( \hat{\mu}_a(t_{u_a}^a) + \sqrt{\frac{2}{u_a} \log(1/\delta)} \geq \mu^* \right) \leq \exp \left( -\frac{u_a}{8} \Delta_a^2 \right)$$

Together these two inequalities yield the following estimation

$$\begin{aligned} \mathbb{E} [T_a(n)] &= \mathbb{E} [\mathbb{1}_{G_a} T_a(n)] + \mathbb{E} [\mathbb{1}_{G_a^c} T_a(n)] \\ &\leq u_a + n^2 \delta + n \exp \left( -\frac{u_a}{8} \Delta_a^2 \right) \\ &= \frac{8 \log(1/\delta)}{\Delta_a^2} + n^2 \delta + n \delta \end{aligned}$$

The above equation shows that roughly  $\delta = n^{-2}$  is the best choice to minimise the upper bound. By substituting back  $\delta = n^{-2}$  and using that  $\frac{1}{n} \leq 1$ , we get

$$\mathbb{E} [T_a(n)] \leq 3 + \frac{16 \log(n)}{\Delta_a^2}$$



The other  $+1$  comes from the fact that  $u_a$  should have been an integer. Let  $\Delta > 0$  be some constant to be determined later. We decompose the regret and then separate the cases when  $\Delta_a \geq \Delta$ .

$$\begin{aligned}
R_n &= \sum_a \Delta_a \mathbb{E}[T_a(n)] \\
&= \sum_{a|\Delta_a < \Delta} \Delta_a \mathbb{E}[T_a(n)] + \sum_{a|\Delta_a \geq \Delta} \Delta_a \mathbb{E}[T_a(n)] \\
&\leq n\Delta + \sum_{a|\Delta_a \geq \Delta} \left( 3\Delta_a + \frac{16 \log(n)}{\Delta_a} \right) \\
&\leq n\Delta + \frac{16k \log(n)}{\Delta} + 3 \sum_a \Delta
\end{aligned}$$

We want to choose  $\Delta$  such that the above upper-bound is minimized. The arithmetic mean - geometric mean inequality shows that it is minimized when  $\Delta = \sqrt{16k \log(n)/n}$ . After substituting  $\Delta$  back, we get that

$$R_n \leq 8\sqrt{nk \log(n)} + 3 \sum_a \Delta_a$$

□

### Asymptotical UCB

The last theorem shows that  $\delta = \frac{1}{n^2}$  is a good choice when we know  $n$  in advance. In the asymptotical scenario we do not know  $n$  in advance.

**Definition 3.14.** Fix an  $f : \mathbb{Z}_+ \rightarrow \mathbb{R}_+$  function. The asymptotical UCB algorithm chooses the step which maximises the UCB formula:

$$A_{t+1} = \arg \max_a \hat{\mu}_a(t) + \sqrt{\frac{2 \log(f(t))}{T_a(t)}}$$

There are several reasonable choices for  $f$ . For instance, in the book [6] the  $f(t) = 1+t \log^2(t)$  option was thoroughly examined. More precisely calculation shows that  $f(t) = t$  or an even slower-growing function can perform better.

## 3.2 Monte Carlo Tree Search

Essentially Monte Carlo Tree Search (MCTS) is a heuristic search algorithm for Markov Decision Processes that proved to be efficient in various board games. For instance, AlphaZero, the algorithm that achieved state-of-the-art results in Go, used MCTS and other deep learning techniques. Another interesting fact is that Tesla's Autopilot software also uses MCTS. Nevertheless, there are many variants of MCTS, but the most widely used [7] utilises the UCB formula.

Let  $\hat{Q}_n(s, a)$  denote the average value gained for playing  $a$  from state  $s$  during the first  $n$  steps. Furthermore, let  $N_n(s)$  denote the number of times  $s$  has been visited until the  $n$ th step and let  $N_n(s, a)$  be the number of times that  $a$  was chosen from state  $s$  in the first  $n$  steps.

**Definition 3.15.** Let  $\mathcal{M}$  be an episodic MDP. The Monte Carlo Tree Search interacts with  $\mathcal{M}$  by following the asymptotical UCB formula:

$$A_{t+1} = \arg \max_{a \in \mathcal{A}} \left\{ \hat{Q}_t(s, a) + \alpha \sqrt{\frac{\log(N_t(s))}{N_t(s, a)}} \right\}$$

where  $\alpha > 0$  is some control parameter. (*The episodic assumption on  $\mathcal{M}$  ensures that multiple trajectories are enrolled.*)

It is easy to see that every state-action pair is played infinitely many times; therefore, the whole environment will be searched. In this chapter, we only considered gaussian bandits because our goal was to introduce the Monte Carlo Tree Search, where even with some prior knowledge about the reward distribution of each state, the accumulated discounted reward, i.e. the  $Q$ -function, is a sum of several of these distributions. Hence it approximates a normal distribution according to the central limit theorem. That explains why we use the UCB formula in MCTS, even though it was introduced for the one-armed bandit problem (with normal reward distributions).

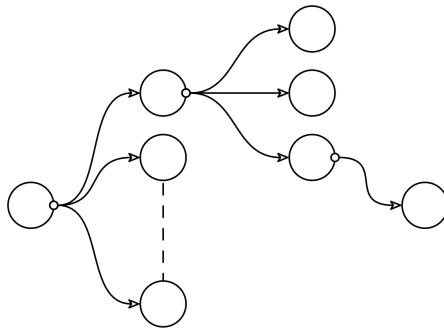


Figure 1: Illustration of the Monte Carlo Tree Search

## 4 Deep learning

Machine learning is the science of learning and building knowledge from experience. In practice, this means that it is capable of identifying patterns in the structure of the input by observing a large number of examples. These tasks are often split into the following categories.

**Supervised learning** is the simplest form of learning. In this scenario, the model is provided by a dataset consisting of inputs with labels. The task is to predict the corresponding label for each input and generalise for unseen data points.

**Unsupervised learning** algorithms discover patterns and previously undetected information in a given dataset. The main difference from supervised learning is that the data points are not labelled.

**Reinforcement learning**'s goal is to build an intelligent agent that takes actions in an environment to maximise the cumulative reward. Contrary to supervised learning, the agent has to figure out how actions contributed to the accumulated rewards and balance between exploration and exploitation.

Deep learning is an area of machine learning that utilises large architectures such as neural networks to solve optimisation problems. Due to the size of the architectures, an optimal solution is hard to find explicitly, so deep learning relies on other optimisation methods like gradient descent. Gradient descent is a first-order iterative optimisation method that finds local optimum in a family of differentiable functions. Let  $L : X \times \Omega \rightarrow \mathbb{R}$  be a loss function, where  $\Omega \subset \mathbb{R}^d$  is the parameter and  $X \subset \mathbb{R}^n$  is the input space. The goal is to find an  $\omega$  parameter that minimise  $L$  over an  $\mathcal{D} \in \mathcal{P}(X)$  input distribution, formulated as the following optimisation problem:

$$\arg \min_{\omega \in \Omega} \mathbb{E}_{x \sim \mathcal{D}} L(x, \omega)$$

Let  $\omega_0$  be the parameter at initialisation, then gradient descent defined as

$$\omega_{t+1} = \omega_t - \alpha \nabla_{\omega} \mathbb{E}_{x \sim \mathcal{D}} L(x, \omega) = \omega_t - \alpha \mathbb{E}_{x \sim \mathcal{D}} \nabla_{\omega} L(x, \omega)$$

where  $\alpha$  is a parameter controlling the size of update steps. We will see later that the idea of gradient descent is used with incredible results for a variety of optimisation problems.

### 4.1 Neural architectures

This chapter introduces some of the most widely used neural architectures as function approximators in various problems.

#### Neural network

**Definition 4.1.** A dense layer is an  $l : \mathbb{R}^{d_i} \rightarrow \mathbb{R}^{d_o}$  function of the following form

$$l(x) = \sigma(Ax + b)$$

where  $A \in \mathbb{R}^{d_j \times d_i}$ ,  $b \in \mathbb{R}^{d_o}$  are the parameter matrix and vector. The  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  is a non-linear function applied to each entry.

**Definition 4.2.** An  $n$ -layered neural network is an  $f_\omega : \mathbb{R}^{d_0} \rightarrow \mathbb{R}^{d_n}$  function defined as

$$f_\omega(x) = l_n \circ l_{n-1} \cdots \circ l_1(x)$$

where  $\omega$  denotes the parameters of each  $l_i$  as an element of the euclidean space.

**Definition 4.3.** The softmax :  $\mathbb{R}^d \rightarrow \mathbb{R}^d$  defined as

$$\text{softmax}(x) = \left( \frac{e^{x_i}}{\sum_{j=1}^d e^{x_j}} \right)_i$$

## Transformer

In recent years, a rather successful neural architecture has been the transformer. It was proposed in [8], and quickly became a popular choice for a variety of learning tasks, especially for natural language processing.

**Definition 4.4.** Let  $Q \in \mathbb{R}^{l_Q \times d_{QK}}$ ,  $K \in \mathbb{R}^{l_{KV} \times d_{QK}}$ ,  $V \in \mathbb{R}^{l_{KV} \times d_V}$  be the query, key and value matrix respectively. The  $l$  parameter is sequence length, while  $d$  is the dimension of the corresponding input. Then the Attention defined as

$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_{QK}}} \right) V$$

Finally, the transformer consist of two parts, an encoder and a decoder. An encoder layer takes an input sequence  $X$  and will use it as the query, key and value matrix. Since each  $Q, K, V$  is essentially the same matrix, it is often referred to as self-Attention.

**Definition 4.5.** Let  $X \in \mathbb{R}^{l \times d}$  be the input sequence and  $h$  the number of 'heads', then the  $L_\omega : \mathbb{R}^{l \times d} \rightarrow \mathbb{R}^{l \times d}$  **Multi-Head Attention** defined as

$$L_\omega(X) = l^{\text{out}} \circ \text{CONCAT}_{i=1}^h \left( \text{Attention}(l_i^Q(X), l_i^K(X), l_i^V) \right)$$

where  $l_i$  are different dense layers and CONCAT concatenates the vectors resulting an output from  $\mathbb{R}^{l \times dh}$  which is projected back into  $\mathbb{R}^{l \times d}$  by the last  $l^{\text{out}}$  dense layer.

Other popular architectures are convolution networks, graph networks, recurrent neural networks, etc. A detailed catalog of architectures can be found in [9].

## 4.2 Deep reinforcement learning

### Fitted Q-learning

A huge disadvantage of vanilla Q-learning is that we need to keep track of each Q-value. To address this issue, we define a set of Q-functions  $\{Q_\omega : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R} \mid \omega \in \Omega\}$ ,

where  $\omega$  denotes the parameter and  $\Omega$  the parameter space. Following the idea of  $Q$ -learning, we want to update the  $\omega$  parameter such that the new  $Q$ -function is the closest one to

$$Y_t(s, a) = \mathbb{E}_{s' \sim T(\cdot|s, a)} \left( R(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q_t(s', a') \right)$$

where the distance between the target and the  $Q$ -function often given by the 2-norm. So, the  $Q$ -function picked as

$$Q_{t+1} = \arg \max_{Q_\omega} \|Y_t - Q\|_2^2$$

## Deep temporal difference learning

In fitted  $Q$ -learning, we have a family of  $Q_\omega$  function approximators parameterised by  $\omega \in \Omega$ . If the  $\Omega$  parameter is large, then it is almost infeasible to select the closest candidate to the target. Therefore, we leverage the idea of deep learning and instead we consider a family of function approximators that are differentiable in the parameter. Let  $\pi$  be a fixed policy,  $\Omega \subset \mathbb{R}^d$  be the parameter space and  $V_\omega : \mathcal{S} \rightarrow \mathbb{R}$  be value-functions such that  $V_\omega(s)$  differentiable in  $\omega$  for every  $s$ . Recall, that in temporal difference learning our goal was to minimise the difference between  $\mathcal{R}_t^\lambda$  and  $\hat{V}_t(s)$ . The  $\omega$  parameter updated by the gradient descent step. The gradient is

$$\nabla_\omega \left( \mathcal{R}_t^\lambda - \hat{V}_t(s) \right)^2 = - \left( \mathcal{R}_t^\lambda - \hat{V}_t(s) \right) \nabla_\omega V_{\omega_t}$$

Therefore, the update parameter is

$$\omega_{t+1} = \omega_t + \alpha \left( \mathcal{R}_t^\lambda - V_{\omega_t}(s) \right) \nabla_\omega V_{\omega_t}(s)$$

The Lemma 2.19 allows us to decompose  $\mathcal{R}_t^\lambda - V_{\omega_t}(s)$  which gives us the slightly modified version of TD( $\lambda$ ) algorithm. The pseudo-code provided below.

---

### Algorithm 4 Lambda-temporal difference learning

---

- 1: **while**  $V$  converges **do**
  - 2:    $S, R, S' \leftarrow \text{SampleStep}()$
  - 3:    $\delta \leftarrow R + \gamma \cdot V_\omega[S'] - V_\omega[S]$
  - 4:    $z \leftarrow \gamma \cdot \lambda \cdot z + \nabla_\omega V_\omega(X_t) \mathbb{1}_{S=s}$
  - 5:    $V \leftarrow V + \alpha \cdot \delta \cdot z$
- 

## Policy gradient methods

The previous section introduced several algorithms that estimate the value function. Then the policy is calculated from  $V$  or  $Q$  by taking the most promising action. Another natural approach is to introduce a policy  $\pi_\omega$  with an  $\omega$  parameter and updating it to maximise the expected return. Therefore the precise formulation of the policy gradient

method is the following: Let  $\pi_\omega(\cdot|s) \rightarrow \mathcal{P}$  a policy function with an  $\omega \in \Omega$  parameter such that it satisfies certain technical assumption that we specify later.

Recall that  $V^\pi(s) = \mathbb{E}_{\pi \sim \pi(\cdot|s)} Q^\pi(s, a)$  and our goal is to maximize  $V^\pi(s_0)$ . As the title of this subsection suggests, we want to use gradient descent to solve this optimisation problem, so we need to approximate  $\nabla_\omega V^{\pi_\omega}(s_0)$ . The Policy gradient theorem [10] gives us a formula to estimate the gradient. As we will see, it is especially useful, since the derivative of  $Q^{\pi_\omega}$  does not occur in the formula.

**Definition 4.6.** Let  $\Omega \subset \mathbb{R}^d$  be a parameter space and  $\mathcal{P} = \{\pi_\omega | \omega \in \Omega\}$  a family of policies parametrised by  $\omega \in \Omega$ . The  $\mathcal{P}$  family is suitable if

1. There is a  $\lambda$  measure that dominates  $\mathcal{P}$ .
2. The  $\rho_\omega(\cdot|s) = \frac{d\pi_\omega(\cdot|s)}{d\lambda}$  Radon-Nikodym derivative of  $\pi$  with respect to  $\lambda$  is differentiable in  $\omega$ .
3. There is a  $C : \mathcal{S} \rightarrow \mathbb{R}_+$  integrable function such that  $\|\nabla_\omega \rho_\omega\|_1 \leq C$ .

If the action space is finite, then the counting measure is the most common choice for  $\lambda$ .

**Theorem 4.7** (Policy gradient theorem). *Let  $\mathcal{M}$  be an MDP,  $\mathcal{P}$  be a suitable policy family with  $\Omega$  parameter space. Then, we can write*

$$\nabla_\omega V^\pi(s_0) = \mathbb{E}_{s,a} (\gamma^t \nabla_\omega \log \rho_\omega(a|s) Q^\pi(s, a))$$

*Proof.* First, we use the formula for  $V^\pi$ . Then use Leibniz rule (Theorem 1.3) to swap the derivative and the integrals, where the assumptions are fulfilled due to Definition 4.6.

$$\begin{aligned} \nabla_\omega V^\pi(s) &= \nabla_\omega \mathbb{E}_{a \sim \pi(\cdot|s)} Q^\pi(s, a) \\ &= \nabla_\omega \int Q^\pi(s, a) \frac{d\pi(\cdot|s)}{d\lambda}(a) d\lambda(a) \\ &= \int (\nabla_\omega \rho_\omega(a) Q^\pi(s, a) + \rho_\omega(a) \nabla_\omega Q^\pi(s, a)) d\lambda(a) \\ &= \int (\nabla_\omega \rho_\omega(a) Q^\pi(s, a) + \rho_\omega(a) \nabla_\omega \mathbb{E}_{s' \sim T(\cdot|s,a)} (R(s, a, s') + \gamma V^\pi(s'))) d\lambda(a) \\ &= \int (\nabla_\omega \rho_\omega(a) Q^\pi(s, a) + \rho_\omega(a) \gamma \mathbb{E}_{s' \sim T(\cdot|s,a)} \nabla_\omega V^\pi(s')) d\lambda(a) \end{aligned}$$

By repeatedly substituting  $V^\pi(s')$  back into the above equation and using the fact that  $\lambda(a) = \pi_\omega(da|s)/\rho_\omega(a)$ , we get that:

$$\begin{aligned} \nabla_\omega V^\pi(s_0) &= \int \left( \frac{\nabla_\omega \rho_\omega(a_0) Q^\pi(s_0, a_0)}{\rho_\omega(a_0)} + \gamma \mathbb{E}_{s' \sim T(\cdot|s_0, a_0)} \nabla_\omega V^\pi(s') \right) \pi_\omega(da_0|s) \\ &= \int (\nabla_\omega \log(\rho_\omega(a_0)) Q^\pi(s_0, a_0) + \gamma \nabla_\omega V^\pi(s')) d\tau \\ &= \int \left( \sum_{i=0}^{\infty} \gamma^i \nabla_\omega \log(\rho_\omega(a_i)) Q^\pi(s_i, a_i) \right) d\tau \end{aligned}$$

Finally, we can rewrite our equation

$$\nabla_{\omega} V^{\pi}(s_0) = \mathbb{E}_{s,a} (\gamma^t \nabla_{\omega} \log \rho_{\omega}(a|s) Q^{\pi}(s, a))$$

where  $s, a$  is the state-action distribution from the interaction  $pi_{\omega}$  with  $\mathcal{M}$  and  $t$  is the corresponding time step.  $\square$

Now, that we obtained an equation to approximate the gradient, we are almost ready to put together the Actor-critic algorithm. Before that, we should consider how efficiently can we approximate the gradient with Monte-Carlo method. Unfortunately, it exhibits a rather high variance that is because even when  $\mathbb{E}(\nabla_{\omega} \rho_{\omega} Q^{\pi}) = 0$  the  $\nabla_{\omega} \rho_{\omega}$  gradients are not necessarily zero. The following lemma, provides a solution to significantly reduce the variance.

**Lemma 4.8.** *Let  $s \in \mathcal{S}$  be a fixed state and  $C \in \mathbb{R}$  an arbitrary constant, then*

$$\mathbb{E}_{a \sim \pi(\cdot|s)} (\nabla_{\omega} \log \rho_{\omega} C) = 0$$

*Proof.* This immediately follows from the following simple observation

$$\begin{aligned} \mathbb{E}_{a \sim \pi(\cdot|s)} (\nabla_{\omega} \rho_{\omega}(a|s) C) &= C \int \nabla_{\omega} \rho_{\omega}(a|s) da \\ &= C \nabla_{\omega} \int \rho_{\omega}(a|s) da \\ &= C \nabla_{\omega} 1 = 0 \end{aligned}$$

$\square$

This allows us to substitute any value to the place of  $Q^{\pi}(s, a)$  that have  $Q^{\pi}(s, a)$  expected value up to some constant depending on  $s$ . For different substitutes, we get different methods, each listed below.

$$\begin{aligned} \nabla_{\omega} V^{\pi}(s_0) &= \mathbb{E}_{s,a} (\gamma^t \nabla_{\omega} \log \rho_{\omega}(a|s) \mathcal{R}_t) && \text{REINFORCE} \\ &= \mathbb{E}_{s,a} (\gamma^t \nabla_{\omega} \log \rho_{\omega}(a|s) Q^{\pi}(s, a)) && Q \text{ Actor-Critic} \\ &= \mathbb{E}_{s,a} (\gamma^t \nabla_{\omega} \log \rho_{\omega}(a|s) A^{\pi}(s, a)) && \text{Advantage Actor-Critic} \\ &= \mathbb{E}_{s,a} (\gamma^t \nabla_{\omega} \log \rho_{\omega}(a|s) \delta) && \text{TD Actor-Critic} \end{aligned}$$

where  $\mathcal{R}_t = \sum_{k=t}^{\infty} \gamma^{k-t} R_k$  is the discounted accumulated reward from the  $t$ th time step and  $\delta$  is the temporal difference  $\delta_{t+1} = R_{t+1} + \gamma \hat{V}(X_{t+1}) - \hat{V}(X_t)$ .

## Actor-critic method

The variance of the gradient estimation in the policy gradient theorem can be reduced by introducing baselines. Therefore, actor-critic algorithms [11] consist of two-part an actor, which is a parametrised policy function and a critic, which is the approximation of the value function and used as a baseline function. Vanilla actor-critic (A2C) can be further improved by technical ideas giving rise to the asynchronous actor-critic (A3C) [12], the most widely used reinforcement learning algorithm.

---

**Algorithm 5** Actor-critic

---

```
1: while  $\theta, \omega$  converges do
2:    $S, R, S' \leftarrow \text{SampleStep}()$ 
3:    $\delta \leftarrow R + \gamma V_\omega[Y] - V_\omega[X]$ 
4:    $\theta \leftarrow \theta + \alpha \delta \nabla_\theta \log \pi_\theta(a|s)$ 
5:    $\omega \leftarrow \omega + \alpha \delta \nabla_\omega V[X]$ 
```

---

### 4.3 Autoencoders

Autoencoders were introduced to allow neural networks to learn in an unsupervised fashion. Their primary purpose is to extract information from the dataset by creating a more informative representation of the data points. The formal definition of the problem is the following:

**Definition 4.9.** Let  $\mathcal{A}$  and  $\mathcal{B}$  be sets of functions from  $\mathbb{R}^n \rightarrow \mathbb{R}^d$  and  $\mathbb{R}^d \rightarrow \mathbb{R}^n$  respectively. We want find  $A, B$  such that an  $L$  reconstruction loss is minimalized.

$$\arg \min_{A \in \mathcal{A}, B \in \mathcal{B}} \mathbb{E}_{x \sim \mathcal{D}} (L(x, B \circ A(x)))$$

In statistics, one of the most available tools to estimate quantities is the Monte Carlo method. Deep learning is just a branch of statistics, where we use gradient descent to tackle complex optimisation problems, which otherwise would be infeasible to solve. Therefore, the following theorem is particularly useful.

**Theorem 4.10.** Let  $q_\omega(z)$  be a parameterized distribution and  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  a differentiable function, then we have

$$\nabla_\omega \mathbb{E}_{z \sim q_\omega} [f(z)] = \mathbb{E}_{z \sim q_\omega} [f(z) \nabla_\omega \log q_\omega(z)]$$

*Proof.* Let  $\lambda$  be a dominating measure of the  $\{q_\omega\}$  family. By changing the base of integrand to  $\lambda$ , we have the following calculation

$$\begin{aligned} \nabla_\omega \mathbb{E}_{z \sim q_\omega} [f(z)] &= \nabla_\omega \int f(z) \frac{dq_\omega}{d\lambda} d\lambda \\ &= \int f(z) \nabla_\omega \frac{dq_\omega}{d\lambda} d\lambda \\ &= \int f(z) \nabla_\omega \log(q_\omega(z)) dq_\omega \end{aligned}$$

And this is what we wanted to prove. □

### Variational Autoencoders

As presented in the previous section, the autoencoder suffers from many problems. Variational Autoencoders (VAE), first introduced in [13], made a significant improvement in the area by trying to build an autoencoder and simultaneously learn the data distribution.



For the precise formulation of the problem, first, consider a  $\mathcal{D}$  dataset distribution. Furthermore, we assume that the data generation involves an unobserved  $Z$  random variable that is often addressed as the latent variable. So, the generation process starts with the generation of  $z$  from some prior  $p_{\omega^*}$  distribution and continues with the generation of  $x$  from the  $p_{\omega}(x|z)$  conditional distribution. We assume a parametric family of distributions can model the two distributions and that their Radon-Nykodim derivatives are differentiable almost everywhere w.r.t. both  $\omega$  and  $z$ .

From the perspective of autoencoders, the  $q_{\omega}(z|x)$  can be interpreted as a probabilistic encoder since for a given  $x$  datapoint, it produces a distribution of  $z$  from which the  $x$  sample could have been generated. Similarly, the  $p_{\omega}(x|z)$  will be referred to as a decoder because it returns a distribution over the possible values of  $x$ .

The curicial observation is the following:

**Lemma 4.11.** *The log-likelihood can be rewritten as*

$$\log p_{\omega}(x) = D_{KL}(q_{\omega}(z|x) \parallel p_{\omega}(z|x)) + \mathcal{L}(\omega, \phi; x)$$

where  $\mathcal{L}$  is the variational lower-bound and can be expressed as

$$\mathcal{L} = \mathbb{E}_{z \sim q_{\omega}(z|x)} [-\log q_{\omega}(z|x) + \log p_{\omega}(x, z)]$$

*Proof.*

$$\begin{aligned} D_{KL}(q_{\omega}(z|x) \parallel p_{\omega}(z|x)) &= \int \log \left( \frac{q_{\omega}(z|x)}{p_{\omega}(z|x)} \right) q_{\omega}(z|x) dz \\ &= \int \log \left( \frac{q_{\omega}(z|x)}{p_{\omega}(z, x)} p_{\omega}(x) \right) q_{\omega}(z|x) dz \\ &= \int p_{\omega}(x) q_{\omega}(z|x) dz + \int \log \left( \frac{q_{\omega}(z|x)}{p_{\omega}(x, z)} \right) q_{\omega}(z|x) dz \\ &= p_{\omega}(x) - \mathcal{L} \end{aligned}$$

□

The variational autoencoder tries to minimise the  $-\log(p_{\omega}(x))$  negative log-likelihood and the  $KL$ -divergence between  $q_{\omega}(z|x)$  and  $p_{\omega}(z|x)$ . The lemma states that it is equivalent of maximising the  $\mathcal{L}$  variational lower-bound.

$$\mathbb{E}_{x \sim \mathcal{D}} (-\log p_{\omega}(x) + D_{KL}(q_{\omega}(z|x) \parallel p_{\omega}(z|x))) = \mathbb{E}_{x \sim \mathcal{D}} (-\mathcal{L})$$

So, VAE is defined as

**Definition 4.12.** Let  $p_{\omega}, q_{\omega}$  be parametrised distribution, then the VAE algorithms updates the parameter as

$$\omega = \omega + \nabla_{\omega} \mathbb{E}_{x \sim \mathcal{D}} \mathbb{E}_{z \sim q_{\omega}(\cdot|x)} [\log p_{\omega}(x|z) + \log p(z) - \log q_{\omega}(z|x)]$$

Note that  $p_{\omega}(x|z)$  and  $q_{\omega}(z|x)$  are functions introduced earlier, so the above estimation can be easily calculated, unlike  $p_{\omega}(x)$ .

## 5 Dreaming to Prove

In the last chapter of the thesis, we finally can discuss, how we use all the reinforcement learning theories to build automated theorem prover algorithms. Automated theorem proving is a long-standing problem in mathematics and computer science. There were many advancements in this field in the past. However, the most reliable approaches focus on constructing tactics such as applying induction or a theorem. These tactics were often combined with powerful planning algorithms such as Monte-Carlo Tree Search.

Despite the rapid development of Artificial Intelligence, deep learning-based approaches could not overtake previous technics. One direction is building large Transformer networks to synthesise grammatically correct calls of tactics. Another approach, and our project’s goal, is to go back to the roots and convert theorem-proving assistants into environments that can be interpreted as Markov Decision Processes, which allows us to utilise reinforcement learning algorithms.

### 5.1 Dreamer algorithm

The reinforcement learning algorithms are often categorized as **model-free** or **model-based**. A model-free algorithm primarily relies on learning, while a model-based one depends on planning as its main component. For instance, the value-learning algorithms we introduced in Chapter 2 are model-free, and an algorithm which leverages Monte-Carlo-Tree search is considered model-based.

The first breakthrough in deep reinforcement learning was achieving human-level in Atari games with a variant of  $Q$ -learning, a model-free algorithm. Even though model-based learning comes with many benefits, such as more robust learning and the ability to plan. They are also more complicated; hence it took researchers a long time to create the first model-based deep reinforcement algorithm, which was better than the previous solutions. That algorithm is the so-called DreamerV2 algorithm [14].

The algorithm consists of the following parameterized components:

$$\begin{aligned} \text{Recurrent model:} & \quad h_t = f_\omega(h_{t-1}, z_{t-1}, a_{t-1}) \\ \text{Representation model:} & \quad z_t \sim q_\omega(z_t|h_t, x_t) \\ \text{Transition predictor:} & \quad \hat{z}_t \sim p_\omega(h_t) \\ \text{Image predictor:} & \quad \hat{x}_t \sim p_\omega(\hat{x}_t|h_t, z_t) \\ \text{Reward predictor:} & \quad \hat{r}_t \sim p_\omega(\hat{r}_t|h_t, z_t) \\ \text{Discount predictor:} & \quad \hat{\gamma}_t \sim p_\omega(\hat{\gamma}_t|h_t, z_t) \end{aligned}$$

where  $t$  is the time step,  $h_t$  is the hidden state,  $z_t$  is the latent state,  $x_t$  is the state,  $r_t$  and  $\gamma_t$  are the reward and discount. The Image predictor and the Representation model together form an autoencoder. The Representation model calculates the latent state distribution conditioned on the input  $x_t$  and some hidden state  $h_t$ , and then the  $x_t$  is reconstructed by the image predictor from  $z_t$  and  $h_t$ . It also imitates the transition of the underlying MDP within the latent space with the Transition model, so the algorithm can

predict the next latent state without knowing the actual input. The dreamer algorithm’s main idea is that it first extracts the crucial information from the state by an autoencoder and then performs reinforcement learning on the latent space.

The following figures are borrowed from [14]. Figure 5.1 shows the process of embedding the input into the latent space and recreating the image, reward and the next latent state from the current state (and from the action). The authors of the paper used  $32 \times 32$  discrete latent space, and they claim that one of the crucial improvements was switching to discrete latent space because instead of minor accumulated errors, only falsely predicted subsequent states could cause problems. Overall, this made the inference more stable.

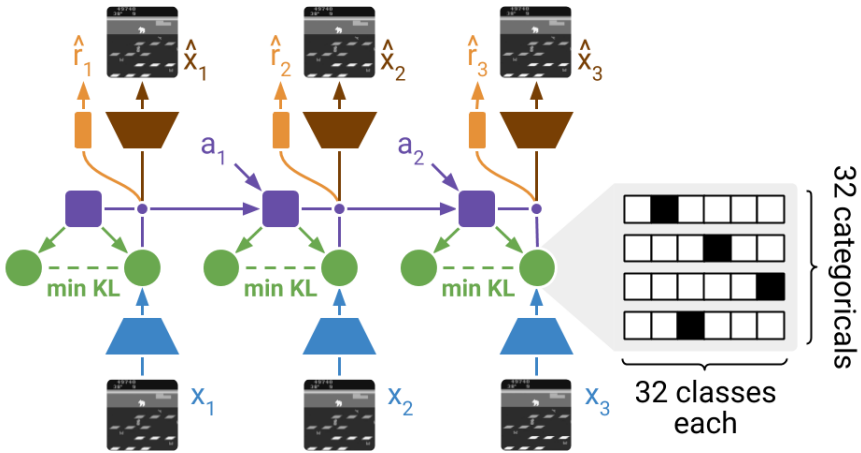


Figure 2: Dreamer world model

Figure 5.1 illustrates that with a learned world model, i.e with an image autoencoder and a dynamic module, the agent can use reinforcement learning algorithms on the learned latent space, and the agent can learn the best policy without interacting with the environment, hence the name Dreamer.

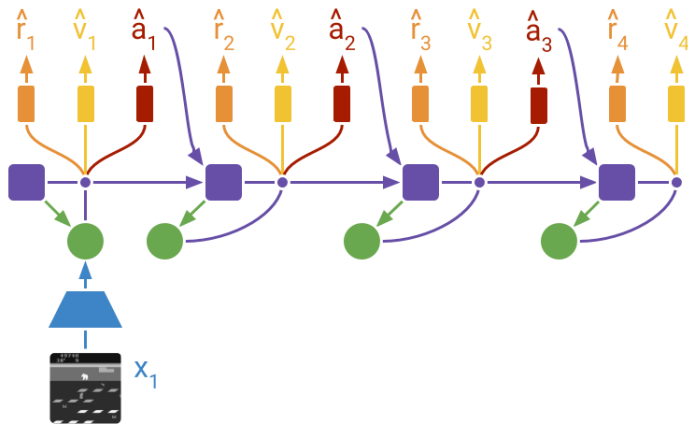


Figure 3: Dreamer dynamics

## 5.2 Dreaming to Prove

### Introduction

Finally, we introduce the **Dreaming to Prove** (D2P) project<sup>1</sup> that builds upon all the theories introduced in previous chapters. The project’s name reflects that it is based on the Dreamer algorithm, which is applied to the problem of proving theorems. A proving problem is given as a set of assumptions (axioms and lemmas) and a conjecture, and the task is to find proof of the conjecture from the assumptions. A proof is a finite sequence of atomic inference steps, and each step has to be selected from a finite set of possible moves, determined by the particular problem and the current state of the proof attempt. Automated theorem proving can be naturally rephrased as an MDP where maximising rewards corresponds to finding proofs. We use the leanCoP [15] theorem prover written in Prolog, encapsulated into a reinforcement learning environment [16] in Python. A formal introduction of automated theorem proving and the leanCoP system is beyond the scope of this thesis. An important feature of the leanCoP system is that it decomposes a mathematical statement into goals. Although these goals are not independent, they can be processed separately. (*A goal is a literal and an action is an ordered formula.*)

Before we dive into the details of our architecture, let us discuss the intuitions and motivations behind the project. There are two reasons for adopting this algorithm to theorem proving systems. The first concerns the time because, with trained dynamics, we do not have to query the environment, which can accelerate the training repeatedly. Unfortunately, this is not important because evaluating proving attempts with the lean-cop system is rapid, especially when compared to a neural network. However, there is another part where we can accelerate the training and inference. The most computationally expensive part of the model is the encoder. However, since the agent also learns the dynamics, it does not rely on calling the encoder at each step. To better understand this particular idea, consider the following real-world example. The state of the art theorem provers are humans, and when we solve a theorem, we usually take more time to understand the problem at the very beginning of it and spend less time on understanding how our problem changes if we manipulate it; for instance, substitutes the value of a variable into our equation.

The second motivation we had when we started this project is that the model is given more objectives to learn by learning the dynamics. Moreover, autoencoders significantly reduce state space size (because it projects the input state into a latent space), so the policy and value function family defined on the latent dynamics is smaller. Therefore, it avoids overfitting more efficiently. Essentially, it will better generalise to unseen cases, which is especially important in theorem proving because positive rewards are infrequent;

---

<sup>1</sup>The code of D2P is available at my github repository <https://github.com/Ayers1013/DreamingToProve>.  
(The latest commit hash: `b0ae7fa734d199d056ca1ffc5fa40689d270bfc7`.)

therefore, it is hard to obtain a large enough dataset where the agent not only memorises but also generalises.

### The architecture: World Model

Similarly to the Dreamer algorithm, we have a world model that learns to simulate the environment and manipulate mathematical statements. The world model consists of the following components:

1. Goal and action embedder
2. State embedder
3. Heads
4. Dynamics module

The following figure provides a brief visual representation of the world model.

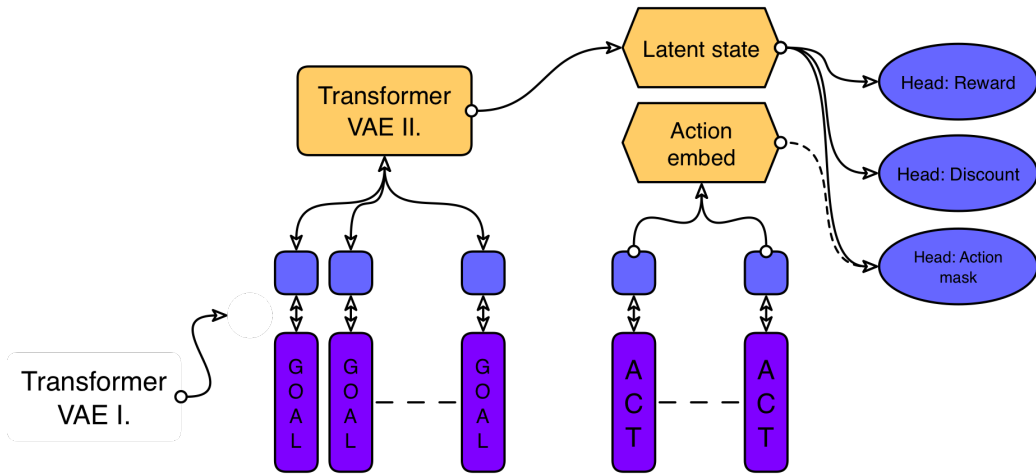


Figure 4: World Model (1-3)

**1. Goal and action embedder.** The goals and actions are flattened into a sequence of tokens. In natural language processing, a token usually represents a word, but in our case, it is a notion of mathematical logic. Then each token is represented by a  $d$ -dimensional learnable vector. In Chapter 4, we introduced the transformer architecture and the autoencoders. A transformer model can process the input once the input is turned into a sequence of  $d$ -dimensional vectors. Note that a transformer model only learns the distributions of the sequences by trying to predict the subsequent tokens, but following the idea of the Dreamer algorithm, we need a model that projects the input into the latent space. Fortunately, with a slight modification [17], we can get a suitable architecture. So, the goal and action embedder is a transformer-based variational

autoencoder introduced in Section 4.3. The Dreamer algorithm used a discrete autoencoder, but the variational autoencoders exhibit a more stable and robust learning curve in the Dreaming to Prove architecture. Figure 4 illustrates how the goals and actions are embedded into a latent space (blue squares).

**2. State embedder.** The first component learns to represent goals separately, but in order to fully comprehend the mathematical statement, the embedded goals should be aggregated so that the model can identify patterns and relations between the goals. That explains the module’s name because the entire input, i.e. the state, is incorporated in a latent space. The D2P architecture considers the embedded goals as a sequence that another VAE transformer can further process. In the figure, this processing step is illustrated by the yellow hexagon. By leveraging autoencoders, the model can obtain a representation of the mathematical problem without depending on any reinforcement learning algorithm.

**3. Heads.** The purpose of this component is to learn other objectives that are important but not part of the input because they can be calculated by knowing that input. The D2P architecture uses three heads: reward, discount and action mask. The model learns the reward associated with the current state and the discount that tells whether a terminal state is reached. (*In the environment provided by leanCop, a terminal state can occur when a valid proof is found or when no more actions can be applied.*) The third head is the action mask. It learns to predict the action mask, which tells which actions are valid at that state. That is rather important because in theorem proving, the action space is large, but usually, only a few are valid. One could argue that valid actions could be learned by a reinforcement learning algorithm where the environment returns a negative reward for taking invalid action. But given the size of the action space, that would result in a large number of unnecessary queries of the environment. Therefore, it is better to provide an action mask to the model implicitly. We also considered a fourth head that learns input meta information, such as the number of specific pattern occurrences. In the early version of D2P, this additional head played a crucial role, but as D2P developed, it became redundant.

**4. Dynamics module.** Note that the above components’ purpose is to help detach the model from the environment by learning the state distribution and the reward function. The last component of an MDP is the transition function that is learnt by the dynamics module. The transition function gives a distribution over the subsequent states given the current state and the action, so the dynamics module does the same, but in the latent space, i.e., it takes a latent state, and an embedded action (the first component encodes the action), then calculates a distribution over the state latent space. This module is trained such that the predicted distribution is close to the predicted latent space distribution by the state embedder calculated from the next state, where ‘close’ means that the KL-divergence is minimised. (*The dynamics module is illustrated on Figure 5.*)

Theoretically, a trained world model (*that perfectly learned all objective*) can fully

simulate the environment so that we can derive a latent environment: Let us fix a problem that we want to prove; this turned into an MDP environment with a starting state  $s$  and set of possible actions  $\mathcal{A}$ . Then  $s$  is projected into the latent state  $s'$  by components 1-2, and every element of  $\mathcal{A}$  is embedded by component 1 to get  $\mathcal{A}'$ . (*Note that actions only embedded once for every problem.*) The trained dynamics module then can predict the next latent state and associated reward for any action. We call the projected environment the **latent environment**.

### The architecture: Behaviour Model

Once the world model is trained on initial data (*Data that is sampled by a random policy.*), we can apply the algorithms of the 2 Reinforcement Learning chapter and the 4.2 Deep reinforcement learning section. The following figure illustrates the D2P architecture.

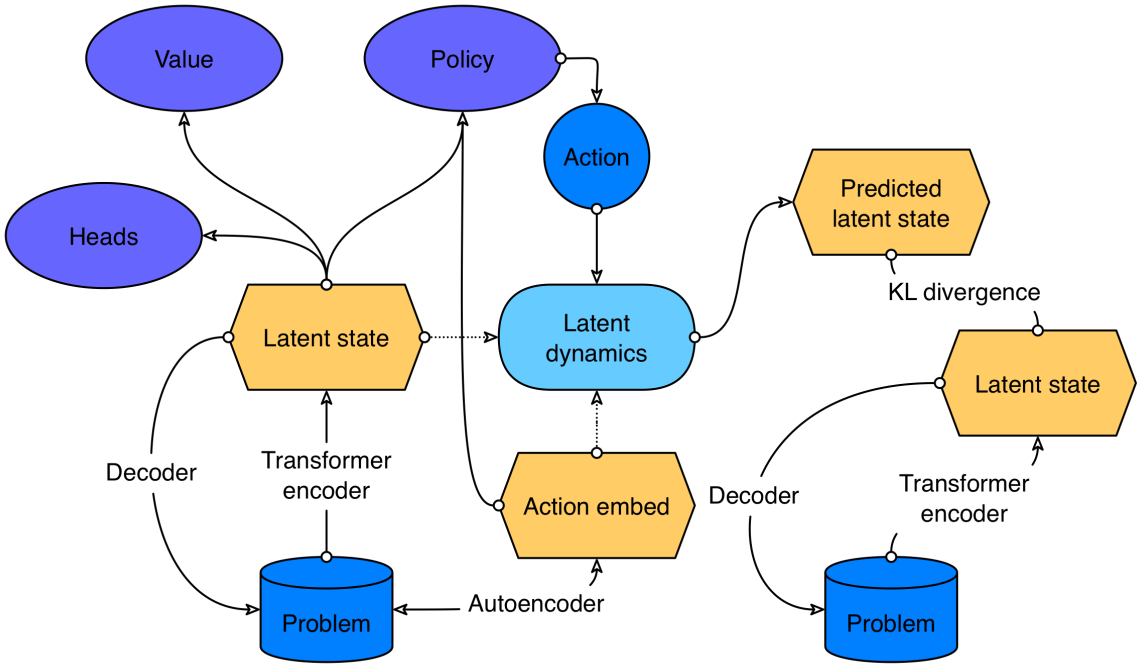


Figure 5: D2P architecture

We discussed how a transformer projects the state into the latent state and how the latent dynamics work. The missing pieces are the value function and the policy. Both take an  $s$  latent state as an input along with the embedded actions, and then they are trained by the Actor-Critic algorithm (Section 4.2). Since this process takes place in the latent environment, it is completely detached from the real environment; hence it is called *dreaming*.

## The algorithm

We have introduced how the different components of the D2P architecture work together, and finally, we put the last pieces into the puzzle to see how the algorithm learns. First, we initialise a dataset consisting of carries from different math problems. Then the training loop starts that continues until the model converges. We can further divide the loop into two processes, the first updates the dataset by deleting less relevant samples and replacing them with new queries, and the second trains the modules.

**1. Dataset update.** First, the algorithm samples a problem, and then new proof attempts are obtained by using the policy and value functions of the model. That is when planning becomes essential because we want to gather good quality data. In the current version, we use epsilon-greedy algorithm (that explore with epsilon probability and takes the best action otherwise) to ensure balanced exploration-exploitation, but more sophisticated algorithms (discussed in Chapter 3) such as MCTS (Section 3.2) might be used to improve performance. Finally, the new samples are added to the new dataset. The D2P has VAE components that rely on consistent data distribution, so the dataset size must be significantly larger than the number of new samples to ensure a slow enough shift in the data distribution that the VAE components can handle. As the training progresses, the old samples become less relevant, so we frequently clean the dataset. That way, we can maintain a fixed size dataset.

**2. Model training.** The world model parameterised by  $\omega$  is trained on samples from the collected dataset. It is important to note that the sampling is not uniform over the dataset. The reason behind it is that correct proof attempts make up only a tiny portion of the data even though they are more critical for the training; therefore, the model gets to see correct proof attempts with higher frequency. The world model consists of several components, and each has its loss function. By taking a weighted sum of these loss functions, we get a loss function that we use to train the world model. Then policy and value functions parametrised by  $\theta$  are trained using the latent environment and the A3C (Section 4.2) algorithm.

---

**Algorithm 6** Dreaming to Prove

---

```
1:  $\mathcal{D} \leftarrow \text{InitDataset}()$ 
2: while  $\omega, \theta$  converges do
3:    $\mathcal{M} \leftarrow \text{ChooseProblem}()$ 
4:   for  $t = 1$  to queryNums do
5:      $\mathcal{D}' \leftarrow \text{PlannedQuery}_{\omega, \theta}(\mathcal{M})$ 
6:      $\mathcal{D} \leftarrow \text{UpdateDataset}(\mathcal{D}, \mathcal{D}')$ 
7:   for  $t = 1$  to trainSteps do
8:      $\omega \leftarrow \text{TrainWorldModel}_{\omega}(\mathcal{D})$ 
9:      $\theta \leftarrow \text{Dream}_{\theta}()$ 
```

---



## Status of the project

Dreaming to Prove is still an ongoing project at the writing of this thesis. Nevertheless, the D2P project has made a long trip. Initially, the D2P architecture used Graph Neural Network (GNN) to embed mathematical statements into a latent space, but after numerous experiments, GNNs could not meet the expectations. So, the embedder architecture was changed to a transformer-based autoencoder. However, transformers are used for solving autoregressive problems (*The task is predicting the next token from previous ones.*) and are not specifically meant to serve as an autoencoder architecture. Therefore, following further experiments, a modified transformer architecture was created that is reliably capable of learning to embed goals and actions. As a result, at this point of the research, the D2P model can embed goals and reconstruct them from the latent space with high accuracy. Unfortunately, the state embedder currently underperforms because despite it can aggregate a few goals and reliably embed them into latent space, it fails for states with many goals. That might be because it is immensely hard to concentrate that much information into a vector space. Another explanation is that the current transformer architecture does not behave invariantly under scaled input size, which might prevent it from proper generalisation. Nevertheless, the current version of D2P can successfully operate on simple problems (with less than a dozen goals), but it does not scale for more complicated problems (with hundreds or even more goals).

The main advantage of the Dreamer architecture is that the model does not solely rely on the learning signal from the reinforcement learning algorithm. That is especially useful in automated theorem proving, where positive rewards are rare, so reinforcement learning is less effective. Nevertheless, the way the dreamer algorithm exploits this advantage relocates the stress to the world model because the policy and value functions rely on a well-trained world model. Promising future work is to redesign the system such that the dependence of different components on each other is alleviated. On the other hand, D2P could benefit from Geometric Deep Learning (*A branch of deep learning that specialises in constructing models on domains that invariant under certain transformations.*). That way relation between mathematical statements could be extracted more efficiently.

## Conclusion

The Dreaming to Prove project builds upon reinforcement learning and deep learning theories and utilises ideas of the Dreamer algorithm and the variational autoencoders. We briefly introduced reinforcement learning in Chapter 2 and deep learning in Chapter 4 and showed how the two could be combined to create new learning algorithms. Then, we discussed one-armed bandits between the two chapters that showed how the explore-exploit trade affects performance. Finally, we gave an insight into a novel neural architecture and algorithm for proving theorems that is part of an exciting research area within the field of automated theorem provers that uniquely utilises deep learning.

## References

- [1] Y. K. Chan. Notes on Constructive Probability Theory. *The Annals of Probability*, 2(1):51–75, 1974. Publisher: Institute of Mathematical Statistics.
- [2] Csaba Szepesvári. Algorithms for Reinforcement Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, January 2010.
- [3] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.
- [4] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*, 11(3-4):219–354, 2018. arXiv:1811.12560 [cs, stat].
- [5] Dmitry B. Rokhlin. Robbins-Monro conditions for persistent exploration learning strategies, October 2018. Number: arXiv:1808.00245 arXiv:1808.00245 [cs, stat].
- [6] Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms*. Cambridge University Press, 1 edition, July 2020.
- [7] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212, pages 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.
- [8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, December 2017. Number: arXiv:1706.03762 arXiv:1706.03762 [cs].
- [9] Smys S., Joy Iong Zong Chen, and Subarna Shakya. Survey on Neural Network Architectures with Deep Learning. *Journal of Soft Computing Paradigm*, 2(3):186–194, July 2020.
- [10] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

- [11] Vijay Konda and John Tsitsiklis. Actor-Critic Algorithms. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.
- [12] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning, June 2016. Number: arXiv:1602.01783 arXiv:1602.01783 [cs].
- [13] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes, May 2014. Number: arXiv:1312.6114 arXiv:1312.6114 [cs, stat].
- [14] Danijar Hafner, Timothy Lillicrap, Mohammad Norouzi, and Jimmy Ba. Mastering Atari with Discrete World Models. *arXiv:2010.02193 [cs, stat]*, February 2022. arXiv: 2010.02193.
- [15] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36(1-2):139–161, July 2003.
- [16] Zsolt Zombori, Josef Urban, and Chad E. Brown. Prolog Technology Reinforcement Learning Prover - (System Description). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 2020.
- [17] Le Fang, Tao Zeng, Chaochun Liu, Liefeng Bo, Wen Dong, and Changyou Chen. Transformer-based Conditional Variational Autoencoder for Controllable Story Generation. *arXiv:2101.00828 [cs]*, July 2021. arXiv: 2101.00828.