MULTI-AGENT REINFORCEMENT LEARNING IN TWO-PLAYER ZERO-SUM GAMES

MARC PAULO MOLINA

Thesis supervisor: JOSEP VIDAL MANZANO (Department of Signal Theory and Communications)

Thesis co-supervisor: MARGA CABRERA BEAN (Department of Signal Theory and Communications)

Degree: Bachelor's Degree in Data Science and Engineering

Thesis report

Facultat d'informàtica de Barcelona (FIB)
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona (ETSETB)
Facultat de Matemàtiques i Estadística (FME)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Agraïments

En primer lloc, voldria agrair als meus pares i a la meva germana, pel seu ajut al llarg de la meva trajectòria com a estudiant i, en especial, al llarg del desenvolupament d'aquest projecte. Sense la seva paciència i suport, recórrer aquest camí no hauria estat possible.

També voldria donar les gràcies al professor Josep Vidal Manzano (director del projecte) i a la professora Margarita Cabrera Bean (codirectora del projecte) per la seva supervisió i seguiment. El seu consell i orientació han estat peces clau en la creació del projecte.

Finalment, voldria agrair a tots els professors del Grau en Ciència i Enginyeria de Dades per la bona qualitat de la formació rebuda. Així mateix, també agrair a totes aquelles persones que han dedicat el seu temps a desenvolupar els camps de l'Aprenentatge Profund i l'Aprenentatge per Reforç. Projectes com aquest són possibles gràcies a l'esforç de tots aquests investigadors i investigadores.

Resum

El principal objectiu d'aquest projecte és comparar com diferents algorismes d'Aprenentatge per Reforç aprenen a jugar a jocs de suma-zero. En concret, ens centrem en el joc del Connecta 4 (o Quatre en Ratlla). En primer lloc, comencem introduint els conceptes teòrics bàsics per a entendre el projecte. Seguidament, proposem un procés d'entrenament que combina Aprenentatge Supervisat i Aprenentatge per Reforç. Inicialment, els agents aprenen a imitar els moviments d'un jugador de nivell mitjà. Sobre aquest coneixement après, s'apliquen diferents algorismes d'Aprenentatge per Reforç amb l'objectiu de millorar el nivell de joc de cada agent. Per a avaluar els agents entrenats, els fem competir entre ells i els comparem per a acabar concloent quin algorisme ha permès adquirir un millor nivell de joc. Finalment, presentem una senzilla Interfície d'Usuari perquè el/la lector/a pugui jugar al joc del Quatre en Ratlla contra tots els agents que s'han entrenat en aquest projecte.

Resumen

El objetivo principal de este proyecto es comparar como distintos algoritmos de Aprendizaje por Refuerzo aprenden a jugar a juegos de suma-cero. En concreto, nos centramos en el juego del Conecta 4 (o Cuatro en Raya). En primer lugar, empezamos comentando los conceptos teóricos básicos para entender el proyecto. Seguidamente, proponemos un proceso de entrenamiento que combina Aprendizaje Supervisado y Aprendizaje por Refuerzo. Inicialmente, los agentes aprenden a imitar los movimientos de un jugador de nivel medio. Sobre este conocimiento aprendido, se aplican distintos algoritmos de Aprendizaje por Refuerzo con el objetivo de mejorar la calidad de juego de cada agente. Para evaluar los agentes entrenados, los hacemos competir entre ellos y los comparamos para acabar concluyendo qué algoritmo ha permitido adquirir un mayor nivel de juego. Finalmente, presentamos una sencilla Interfaz de Usuario para que el/la lector/a pueda jugar al juego del Cuatro en Raya contra todos los agentes que han sido entrenados en este proyecto.

Abstract

The main objective of this project is to compare how different Reinforcement Learning algorithms learn to play zero-sum games. Specifically, we focus on Connect 4 (or Four in a Row). Firstly, we start by introducing the basic theoretical concepts to understand the project. Afterward, we propose a training process that combines Supervised Learning and Reinforcement Learning. Initially, the agents learn to mimic the actions of a mid-level player. On this learned knowledge, we apply different Reinforcement Learning algorithms to improve the performance of each agent. To evaluate the trained agents, they compete against each other, so we can compare them and conclude which algorithm has achieved the highest level of play. Finally, we present a simple User Interface to let the reader play Connect 4 against all the agents that have been trained in this project.

Contents

1. Introduction

In this first chapter, we present the purpose of this project. It starts by providing some context on the problem we tackle and explaining why it is of our interest [1.1]. Afterward, we define the scope and goals of the project, and we also briefly introduce the strategy we set out to follow in order to accomplish these goals [1.2]. Finally, we explain the structure of the document, what is found in each chapter, and how they are related to each other [1.3].

1.1. Problem statement

Artificial Intelligence (AI) is already changing our world. Modern AI systems can process their environment in real-time while making optimal decisions toward specific objectives. In recent years, AI has achieved astonishing results in several fields such as Computer Vision and Natural Language Processing. Despite its recent success, there are some areas in which AI has always been of great interest even in the times when it was not so powerful. Board games are one of those areas in which AI has always been present (even as an abstract idea at the beginning) and has recently achieved superhuman performance in many games.

Board games are played in any part of the world by people of all ages. There are many varieties of board games, and their rules and complexity can range from the very simple (e.g. *Snakes and Ladders*) to deeply complex (e.g. *Shogi*). In the late 1990s, board games became increasingly popular on the Internet because it was easy for players to find opponents of different levels. Back then, the idea of an Artificial Intelligence playing better than humans aroused great interest, and the properties of board games made them a perfect environment to test AI algorithms. Moreover, what is learned to play games may also be useful to solve other real-world problems.

In this project, we train some Artificial Intelligence algorithms to learn to play zero-sum board games. In particular, we focus on the game of Connect4, and the algorithms we use are based on Supervised Learning and Deep Reinforcement Learning. In the following chapters, we explain these concepts and how we approach this problem in more detail.

1.2. Objectives

Our objective is to compare how different Deep Reinforcement Learning algorithms learn to play the classic Connect4 game. Our planned training strategy consists of two stages. Firstly, we set out to train a policy network to predict the moves of a mid-level player (a heuristic that we designed). Secondly, starting from this pre-trained network with basic knowledge of the game, we intend to improve it by applying some Deep Reinforcement Learning algorithms with self-play in order to create high-level Connect4 players. The Deep Reinforcement Learning algorithms that we test in this project are Vanilla Deep Q-Network, Dueling Deep Q-Network, REINFORCE with baseline, and Proximal Policy Optimization.

Before training the agents, our first objective is to create a Reinforcement Learning environment to simulate Connect4 games. This process involves creating a meaningful reward function and implementing the game rules to allow interaction between the agents and the environment.

After the training phase, our goal is to compare all the trained agents to conclude which algorithm works better in this particular setting. The trained agents will compete against each other and against some baselines, and we will measure their win rates and other metrics in order to elaborate a comprehensive analysis.

Finally, we also set out to create a simple User Interface to play Connect4 and let the reader interact with the trained agents and play against them. At each turn, the policy followed by the agent will be displayed, so it also serves as a visual tool for evaluation and debugging.

1.3. Document structure

In Chapter 2, we introduce the technical concepts required to understand the project. It starts with a general description of Reinforcement Learning and some of the algorithms we used to train different agents to play Connect4. Then, we explain how Deep Neural Networks are applied to Reinforcement Learning problems, settings with multiple agents, and two-player zero-sum games. This chapter ends by explaining the case of *AlphaGo*, the first computer program to defeat a human professional player in the full-sized game of Go. The training pipeline used in our project is partially inspired by the one used to train *AlphaGo.*

Chapter 3 describes the classic Connect4 game as a Reinforcement Learning environment. First, we analyze the nature and rules of the game. Afterward, we present the game as an environment and explain how the agents interact with. Finally, we show the reward function that we designed for this environment.

Chapter 4 is a comprehensive explanation of how we trained the agents. First of all, we present the Baseline Agents for evaluation purposes. Then, we explain how we trained a policy network to predict the actions of a mid-level player. This pre-trained network was improved with different Deep Reinforcement Learning algorithms adapted to two-player zero-sum games. The algorithms implemented here are Vanilla Deep Q-Network, Dueling Deep Q-Network, REINFORCE with baseline, and Proximal Policy Optimization. This chapter covers the entire training process and the proposed solutions for each algorithm.

In Chapter 5, we evaluate the agents trained in the previous chapter. We describe how we organize a fair competition between two agents and how we analyze the results. Finally, we present a simple User Interface to let the reader play against our agents and evaluate them. Further details on how to run the code are provided in that section.

Lastly, Chapter 6 presents the overall conclusions of the project, the contribution we have made, and some suggestions for future work based on the work we started.

2. Theoretical background

In this chapter, we introduce the technical concepts required to understand the project. It starts with a general description of the key concepts in Reinforcement Learning [2.1] and some of the algorithms we used to train different agents to play Connect4. These algorithms are Q-learning [2.1.1] REINFORCE with baseline [2.1.2], and Proximal Policy Optimization [2.1.3]. Afterward, we analyze how Deep Neural Networks are applied to solve Reinforcement Learning tasks [2.2]. In particular, we introduce two Deep Learning variants of Q-learning that were also used in this project: Vanilla Deep Q-Network [2.2.1], and Dueling Deep Q-Network [2.2.2]. Then, we present different settings in Multi-Agent Reinforcement Learning [2.3], and we focus on the specific case of two-player zero-sum games [2.4]. This chapter ends by explaining the case of *AlphaGo* [2.5], the first computer program to defeat a human professional player in the full-sized game of Go. The training pipeline used in our project is partially inspired by the one used to train *AlphaGo.*

2.1. Reinforcement Learning

Reinforcement Learning (RL) [1] is the area of Machine Learning that is concerned with training intelligent agents to solve complex decision problems to maximize a notion of cumulative reward. At each time step, the agent observes part of the environment state, takes an action, and receives a reward (or punishment) that tells the quality of that action in that environment state. The action taken by the agent affects the environment and may alter its state.

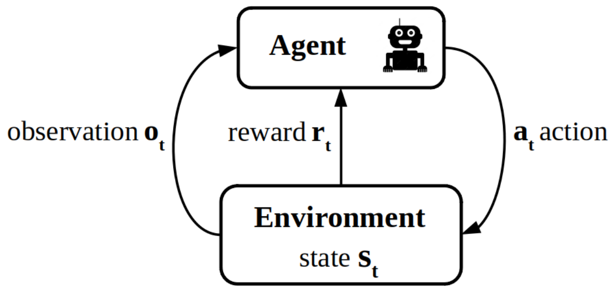


Figure 2.1. *Scheme of the typical agent-environment interaction in Reinforcement Learning*

The main difference between RL and Supervised Learning (SL) is that in RL no reference answers are available. Due to the lack of a labeled training dataset, the way to find the optimal strategy is by interacting with the environment using trial-and-error and learning from the gathered experiences. In SL, the data comes from an external source that might certify some data properties such as consistency, completeness, or timeliness. But in RL things are different, the data depends on a variable agent that interacts with a potentially unknown environment.

One of the main challenges in RL is finding a good trade-off between exploration and exploitation. When the agent exploits the environment, it takes the most effective actions according to the information it has gathered so far from experience. However, the agent's experience is usually limited, so their estimation of actions' effectiveness may be inaccurate. To learn better strategies, the agent has to explore the environment. In other words, the agent takes sub-optimal actions to discover better policies. One of the most simple and well-known strategies to address this dilemma is *decaying epsilon-greedy*. The agent chooses exploration with probability ϵ and exploitation with probability $1 - \epsilon$. As the agent gathers more experience, the exploration rate decreases and the focus is on exploiting the environment.

In the following lines, some key components in the RL terminology will be introduced.

Reward signal: it is the feedback that the agent receives from interacting with the environment. It is usually a scalar value that tells the agent how well it has performed. The reward signal defines the task to solve. At a time-step t , the reward is represented as r_t.

Return: at time *t,* the return is the cumulated reward obtained from the entire duration of an episode. It is the sum of the subsequent rewards weighted by the discount factor $\gamma \in [0, 1]$. This factor indicates the importance of the future rewards at the present moment. The equation of the return is as follows (on the right, a simpler expression using a recursion):

$$G_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \cdots = \sum_{k \geq 0} \gamma^k \cdot r_{t+k+1} \qquad or \qquad G_t = r_{t+1} + \gamma \cdot G_{t+1}$$

Policy: it is a map $\pi(a|s)$ that gives the probability of taking an action a when in a given state s, so it defines the agent's behaviour. It can be either stochastic or deterministic, and it changes during the learning process. The agent tries to optimize the policy to get the highest return.

Value function: it represents the expected cumulative reward starting from a given state. In other words, it expresses the goodness of a state in the long run when following a policy π. A state might have a low reward but still have a high value, and vice versa. It is important to know that values are more relevant than instant rewards when making decisions to maximize long-term gains. The expression for the value function is as follows:

$$v_\pi(s) = E_\pi[G_t | S_t = s]$$

This expression can be decomposed into two parts: the immediate reward, and the future discounted value. The resulting expression is known as the *Bellman Equation* for the value function. It relates the value function for a given state to the value function of its subsequent states in a recursive way. The Bellman Equation for the value function is as follows (s' stands for the subsequent states):

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r'} p(s', r'|s, a)[r' + \gamma v_\pi(s')]$$

Action-value function: the value function for each state-action pair can be defined as well. It refers to the expected return when the agent is in a particular state and takes a particular action. This function is also called the *Q function.* The expression is as follows:

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

In an RL setting, the environment is typically modeled as a *Markov Decision Process* (MDP). The standard approach for solving RL problems stated as an MDP is by using *Dynamic Programming* (DP) to solve the Bellman equation, but RL offers some key advantages. Especially when the state space is relatively large, RL is considered a much more efficient solution. Moreover, DP assumes full knowledge of the environment dynamics (*model-based*), and it is usually unreasonable for most real-world applications. On the other hand, RL proposes several trial-and-error algorithms (*model-free*) to deal with unknown environments.

RL can be used to solve a wide range of real-world problems, in particular those that involve decision-making, resource allocation, and optimization. In general, it can be said that every problem that can be translated into a useful reward signal can be potentially solved using an RL algorithm. In the following section, we will dive into some of the most well-known model-free algorithms that will be used throughout this project.

2.1.1. Q-learning

Q-learning [1] [2] is a popular *off-policy* RL algorithm to learn the value of an action in a particular state. It does not assume that environment dynamics are known, so it is a *model-free* algorithm. An agent tries an action in a particular state and evaluates that transition based on the reward it receives and its estimate of the value of the next state. By trying a wide range of state-action pairs, it learns which action yields the highest return in each state. For any RL problem stated as a finite *Markov decision process,* the Q-learning algorithm has been proved to converge asymptotically to the optimal Q-function provided that all the state-action pairs are infinitely visited.

To estimate the Q-values, the agent runs many episodes and follows the update rule shown in *Figure 2.2*. At each training episode, the agent takes actions following a policy derived from the estimation of the Q-values. For each state-action pair (s, a), the agent receives a reward r

and observes the next state s'. The value $Q(s,a)$ is updated following the *update* expression that depends on the transition (s, a, r, s'). The next action a' is chosen greedily on $Q(s',:)$.

```
function Q-LEARNING
    Initialize $Q(s,a)$ arbitrarily
    for each episode do:
        Initialize state $s$
        for each step of the episode do:
            Choose an action $a$ from $s$ using a policy derived from $Q$ (e.g. $\epsilon$-greedy)
            Take action $a$ and observe $r$, $s'$
            $Q(s,a) = (1 - \alpha) \cdot Q(s,a) + \alpha \cdot \left[r + \gamma \cdot max_{a'} Q(s', a')\right]$    (update)
            $s \leftarrow s'$
        until state $s$ is terminal
        end for
    end for
    return $Q$
end function
```

Figure 2.2. *Pseudocode of Q-learning algorithm*

The tabular Q-learning algorithm stores the action-state values in a $|S| \times |A|$ look-up table (where S and A are the state and action spaces, respectively). However, this approach is limited to discrete action and state spaces, and even if they are discrete there might be memory limitations if the tables are too big. The solution is to use a parametric function and optimize the weights to find the optimal action-state values. In general, Neural Networks are a popular choice due to their great performance in complex domains. However, linear functions or decision trees may also work well in several contexts.

2.1.2. REINFORCE with baseline

One of the main problems with value-based methods is that even if the value estimations are accurate, the policy derived from them might not be as good as expected. Policy Gradient methods try to address this issue by learning directly a parametrized optimal policy by performing gradient ascent to maximize the expected return. The REINFORCE algorithm [3] is one of the most basic Policy Gradient algorithms. The *Policy Gradient Theorem* [4] states that the derivative of the expected reward is the expectation of the product of the reward and the natural logarithm of the policy. It means that Policy Gradient techniques are *model-free* and don't assume previous knowledge about the environment.

Within the REINFORCE training loop, the agent runs an episode using its current policy and collects the rewards and the log probabilities of the chosen actions. At the end of the episode, it computes the discounted return for each intermediate time step. The weights of the network are updated to maximize the expected return. That means that the probability of actions that yield a high return will be increased, and the ones that yield a low return will be decreased.

REINFORCE is also known for its high variance and slow convergence. One way to palliate this unstable behaviour is to add an arbitrary baseline function [3]. One of the most common baseline functions to reduce the variance is an approximation of the state values. There will be another parameterized function $b_\omega(s)$ that learns the state value and it is trained with the discounted rewards of each episode (a regression task). In this case, the difference between the discounted reward and the value estimation is called the *advantage function.* The definition of the advantages is one of the key differences between policy gradient algorithms. The following *Figure 2.3* illustrates the training loop of the REINFORCE with baseline.

```
function REINFORCE with baseline
    Define a differential policy parametrization $\pi_\theta$
    Define a differential state-value function parametrization $b_\omega$
    Initialize $\theta$, $\omega$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do:
        for $t = 1$ to $T - 1$ do:
            $G_t$ = return from step $t$
            $\hat{A} = G_t - b_\omega(s_t)$
            $\theta \leftarrow \theta + \alpha_\theta \cdot \hat{A} \cdot \nabla_\theta log(\pi_\theta(s_t, a_t))$      (update)
            $\omega \leftarrow \omega + \alpha_\omega \cdot \hat{A} \cdot \nabla_\omega b_\omega(s_t)$      (update)
        end for
    end for
    return $\theta$
end function
```

Figure 2.3. *Pseudocode of REINFORCE with baseline algorithm*

2.1.3. Proximal Policy Optimization

Proximal Policy Optimization (PPO) [5] is a policy gradient algorithm concerned with mitigating the high variance of other policy gradient methods such as the REINFORCE algorithm. PPO tries to improve the Trust Region Policy Optimization (TRPO) algorithm [6], which also addresses the high variance problem. PPO has proven to improve the results of

TRPO in several contexts, it also reduces the complexity of the computations within the algorithm, and it is easy to implement and fine-tune. PPO was first introduced by OpenAI, and on their web page they claim that "*PPO has become the default reinforcement learning algorithm at OpenAI because of its ease of use and good performance".*

As a Policy Gradient algorithm, the goal of PPO is to optimize the policy to maximize the expected reward. The REINFORCE algorithm also has this goal, but its learning is slow and unstable. TRPO implements a limit on the *Kullback–Leibler divergence* between the old policy and the new one, so it ensures that the deviation from the previous policy is relatively small. This way, the variance is reduced. With the same goal in mind, the improvement that PPO implements consists of clipping the probability ratios $\varphi(\theta)$ between the old policy and the new one. Then, this concept is used to create a new *clipped surrogate objective* $L^{CLIP}(\theta)$, that serves as a lower bound for the performance of the policy. PPO has also a Kullback-Leibler variant, but the *clip* function is the preferred option by the OpenAI team. The expression of the *clipped surrogate objective* is as follows.

$$L^{CLIP}(\theta) = E_t\left[min\left(\varphi_t(\theta) \cdot \hat{A}_t, \, clip(\varphi_t(\theta), \, 1 - \epsilon, \, 1 + \epsilon) \cdot \hat{A}_t\right)\right]$$

$$\text{where} \quad \varphi_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)}$$

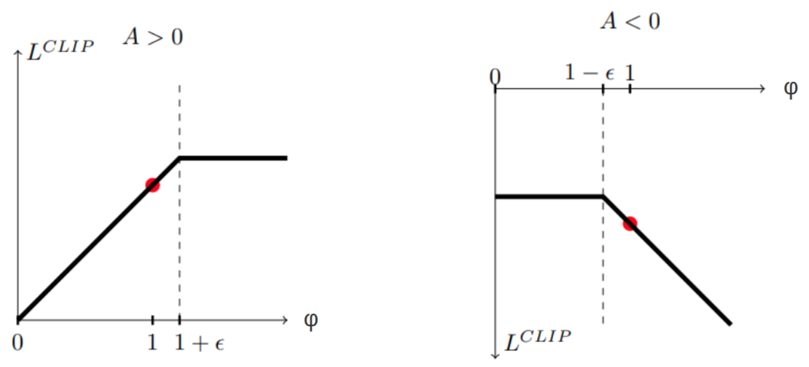


Figure 2.4. *Plots showing one term of the surrogate function L^{CLIP} as a function of the probability ratio ϕ, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., ϕ = 1* [5].

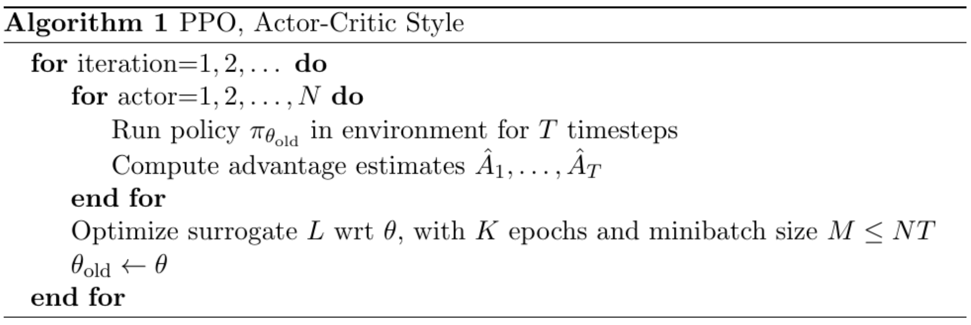
The *clip* function in L^{CLIP} prevents the ratio $\phi(\theta)$ from moving outside the interval $[1 - \epsilon, 1 + \epsilon]$. Taking the minimum between the clipped and unclipped objective is a lower bound on the unclipped objective. In Figure 2.4, it can be seen that for positive advantages the new policy is not allowed to diverge more than $1 + \epsilon$ from the old one, whereas for negative advantages the ratio is not allowed to be less than $1 - \epsilon$.

In the original paper, the creators of PPO address the case of a backbone neural network followed by two prediction heads (the actor and the critic). In this case, the final objective is a weighted combination of the (actor) policy objective L^{CLIP}, the (critic) value objective L^{VF}

(any regression loss function, e.g. *MSE*), and a policy entropy bonus $S[\pi]$ to ensure sufficient exploration. The final objective function that is *maximized* (gradient ascent) is as follows (c_1 and c_2 are coefficients to control the relevance of L^{VF} and $S[\pi]$, respectively):

$$L_t^{CLIP+VF+S}(\theta) = E_t\left[L_t^{CLIP}(\theta) - c_1 \cdot L_t^{VF}(\theta) + c_2 \cdot S[\pi_\theta(s_t)]\right]$$

REINFORCE is very inefficient in terms of data reusability because each transition is used only once to update the network weights. The ability to reuse training data several times is crucial in reinforcement learning since the interaction with the environment might be expensive in some scenarios. In the case of PPO, there is a buffer that stores different timesteps from different episodes generated using the same policy weights. Then, it runs many epochs on this data to update the network weights. The following Figure 2.5 presents the pseudocode of the general PPO algorithm.



```
**Algorithm 1** PPO, Actor-Critic Style
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \le NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

Figure 2.5. Pseudocode of the *PPO algorithm. From the original paper* [5].

2.2. Deep Reinforcement Learning

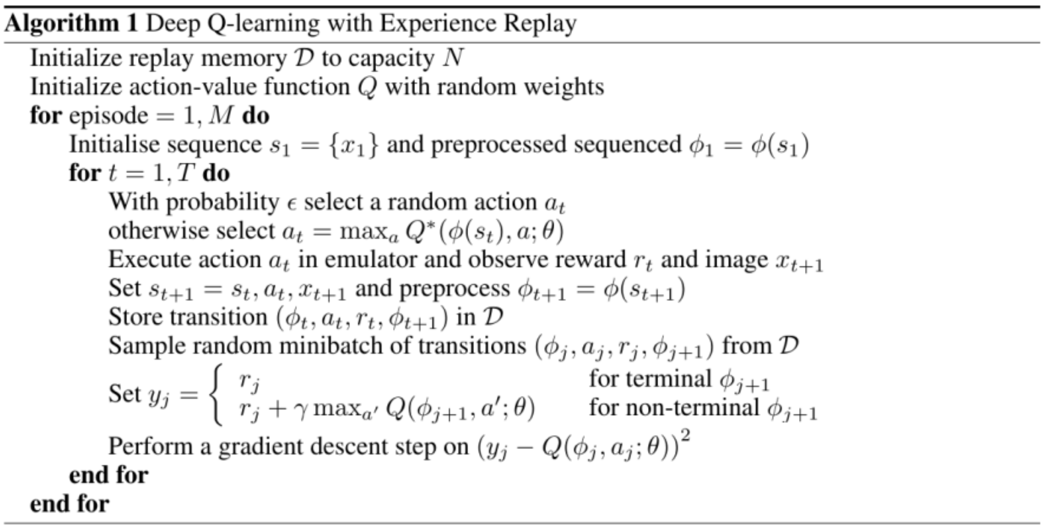
Deep Reinforcement Learning (Deep RL) [1] is the area of Machine Learning that combines Reinforcement Learning (RL) and Deep Learning (DL). In many problems that RL deals with, the state space might be too large to be solved using traditional RL algorithms (e.g. the pixels of an image or a computer screen). Here is where DL comes into play by using Deep Neural Networks to learn the policy, the value functions, and/or the action-value functions. Deep RL has been achieving increasingly impressive results in several fields such as self-driving cars, computer games, robotics, and healthcare.

One of the main advantages of using Deep RL is that neural networks are more capable of generalizing to unseen data than standard RL algorithms. When neural networks are applied to an RL problem, they learn to extract meaningful features from the input data that may be useful for unseen data. Moreover, Deep RL can benefit from *transfer learning*: a model pre-trained on a similar task can be reused in the problem at hand. However, if a problem is simple enough to be solved using basic RL techniques, Deep RL might not be the best option due to its inherent instability.

2.2.1. Vanilla Deep Q-Network

With the success of Deep Learning in several domains, one area of exploration was to apply Neural Networks to implement the concepts of basic Reinforcement learning algorithms, such as Q-learning [2.1.1]. A Deep Learning version of the Q-learning algorithm was presented by DeepMind, and they showed that the network learned to play many Atari 2600 games directly from the screen's raw pixels using a convolutional neural network. The network was trained to estimate the Q-values so it was called *Vanilla Deep Q-Network (Vanilla DQN)* [7], and it surpassed human-level performance and other state-of-the-art approaches in some of the games.

In order to use Neural Networks to learn the Q-values, the correlation within sequences of consecutive transitions from the same episode poses a problem. In the original paper, DeepMind proposed the use of an *Experience Replay Memory*. This memory keeps the most recent transitions and randomly samples batches of previous transitions to feed the network with uncorrelated data. *Figure 2.6* shows the training process of the DQN with Experience Replay. Note that the definition of the target y_j is based on the Bellman Equation that we described in [2.1].



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
      end for
  end for
```

Figure 2.6. *Pseudocode of Deep Q-Network with Experience Replay (from the original paper [7])*

In this approach, the target is computed using the network that is also being trained. This dependence between the target and the prediction may lead to unstable learning. For the prediction and the target to be independent, a different network must be used for each one.

There are many ways to address this issue, but one of the most common solutions is to use a *target network* with a different set of weights θ' that are used to compute the target: $max_{a'} Q(\phi_{j+1}, a' ; \theta')$. These weights θ' are always an old version of θ, and are updated (θ' ← θ) every C training steps.

2.2.2. Dueling Deep Q-Network

The success of *Deep Q-Networks* opened the door to exploit the power of Neural Networks to estimate the Q-values. *Dueling Deep Q-Network (Dueling DQN)*[8] is one of the variants that have been proposed to improve the performance of *Vanilla DQN*. The *Dueling* architecture uses two prediction heads to estimate the advantage of each action and the value of the state as well. Then, these values are combined to estimate the Q-values. The advantage subtracts the value of the state from the Q-value to obtain the relative importance of each action, and separating these two elements works particularly well in the presence of many similar-valued actions. Since both *Vanilla* and *Dueling* architectures aim at estimating the Q-values, almost every improvement designed for the *Vanilla* case is likely to also work for the *Dueling* case (e.g. experience replay, target network).

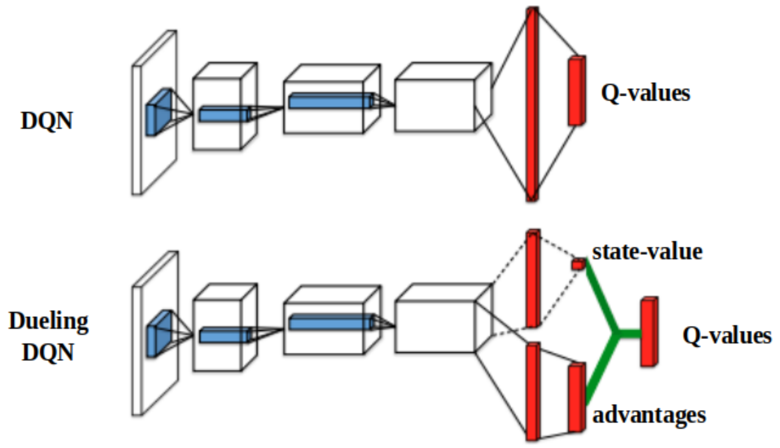


Figure 2.7. *Comparison between the popular DQN architecture (top) and the Dueling DQN architecture (bottom) (image from [8]).*

We mentioned that the advantages and the value of the states are predicted separately and then combined to estimate the Q-values. Intuitively, we would use the expression $Q(s, a) = A(s, a) + v(s)$, but this would lead to poor performance. Following that expression, given a value $Q(s, a)$ it is not possible to recover $A(s, a)$ and $v(s)$ uniquely. One possible solution is to force the advantage of the chosen action a to be zero, so $Q(s, a) = v(s)$. The following expression could be used to solve this problem of identifiability:

$$Q(s,a) = v(s) + [A(s,a) - max_{a'} A(s,a')]$$

There is also another approach that adds more stability to the learning process. It consists of replacing the *max* operator with the average over the advantages. Subtracting the mean also helps with identifiability, and it adds stability because the changes in the mean are smoother than the potential changes in the optimal action's advantage when the *max* operator is used. The following expression replaces the *max* operator with the average.

$$Q(s,a) = v(s) + [A(s,a) - mean\,(A(s,\ :))]$$

2.3. Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) [9] is the area of Reinforcement Learning that studies how multiple agents interact in a common environment. The goal of MARL is to learn a policy for each agent such that all agents together achieve the goal of the system. In general, the problems that MARL deals with are categorized as *NP-Hard* due to their complexity. With the motivation of recent success in deep reinforcement learning, several researchers have been focusing on MARL.

Essentially, the goals of the agents can be classified into three categories. When the agents work towards a common goal, it is said to be a cooperative setting (e.g. image classification with a swarm of robots). Contrarily, if the agents compete with one another to accomplish a goal, it is said to be a competitive setting (e.g. chess). The last category is some mix of the two (e.g. two football teams playing a match).

Collaborative MARL is arousing great interest in large-scale cooperation problems. These are arguably the biggest conflicts in our society and our species, and they can be approached from a MARL perspective. In a collaborative setting, the agents have to learn to communicate effectively with each other, because the communication bandwidth and the memory capacity are usually limited. Moreover, the agents must seek an optimal solution under the constraint that this solution is in consensus with its neighbours, and that is not always easy.

In a competitive setting, agents have conflicting goals and they need to take into account their opponents when updating their policies. The *minimax* principle can be applied here: maximize one's benefits under the worst-case assumption that the opponent will always aim to minimize it. In general, adapting your policy to your opponents requires you to anticipate your opponents' next actions to counterattack them effectively. The competitive setting is widely studied in game theory and economics.

2.4. Two-player zero-sum games

Two-player zero-sum games [10] are a mathematical framework for representing scenarios where two players interact and the advantage for one player is the equivalent loss for the other. That is, all the resources are available at the beginning of the game, and the game only allows a redistribution of the initial amount of resources. Contrarily, in nonzero-sum games, both players can win or lose simultaneously.

Zero-sum games are a case of competitive Multi-Agent Reinforcement Learning since the players have opposite goals and interact in the same environment. For this reason, zero-sum games are usually solved using the *minimax* theorem: maximize one's benefits under the worst-case assumption that the opponent will always aim to minimize it.

In the particular case of two-player zero-sum games with deterministic state transitions (i.e. the current state and action unequivocally determine the next state) and zero rewards except at a terminal time-step T, there is a unique optimal value function that determines the outcome from the state s following perfect play by both players. The outcome of the game is the terminal reward at the end of the game from the perspective of the active player: $z_T =+ r(s_T)$ for a certain win, and $z_T =- r(s_T)$ for a certain loss. The expression for the optimal value function takes into account that players alternate turns to beat each other and win the game (i.e. states s_t and s_{t+1} are player by different players). The expression is as follows [11].

$$v * \left(s_t\right) = \begin{cases} z_T & if\ s_t\ is\ terminal \\ max_a - v^*\left(s_{t+1}\right) & otherwise \end{cases}$$

2.5. AlphaGo

In 2015, DeepMind introduced a new search algorithm that mastered the game of Go using Monte-Carlo simulations with two deep neural networks to estimate the policy and the state value. These networks were trained by a combination of supervised learning from human expert games, and reinforcement learning from games of self-play. The program *AlphaGo* [11] became the first computer program to defeat a human professional player in the full-sized 19 × 19 game of Go. It achieved a 99.8% winning rate against other sophisticated Go programs and defeated the world champion Lee Sedol 4 to 1.

Monte-Carlo Tree Search (MCTS) is a heuristic search algorithm widely used to play board games by solving the game tree. Traversing the tree using exhaustive search is unfeasible in big games such as chess and Go. But *AlphaGo* implements two networks to reduce the effective search space. First, the effective breadth may be reduced by sampling actions from the policy network. Second, the effective depth may be reduced by using the value network to predict the outcome (i.e. value function) from a given state.

The training pipeline consists of three stages. Firstly, a policy network is trained to predict expert moves in the game of Go. Then, the policy network is improved by policy gradient reinforcement learning using the REINFORCE algorithm with a baseline for variance reduction. Finally, the value network is trained to predict the game outcome from states. Once these three stages are completed and both networks are trained, *AlphaGo* combines them with an MCTS to create a high-performance tree search engine.

The data used in the supervised learning stage came from the KGS Go server, which contains nearly 30 million positions from 160k games played by KGS professional human players. The games used in the second and third training stages were generated using self-play, in other words, letting the policy network play against itself. The data processing steps consisted of computing all the features relative to the current color to play; for example, the stone color at each intersection is represented as either player or opponent rather than black or white. Then, each feature is one-hot encoded in 19×19 binary planes. In total, there are 48 stacked binary features after the one-hot encoding, so the input to the networks is a $19 \times 19 \times 48$ stack of feature planes.

A few months later, Deep Mind went a step further and published *AlphaGo Zero* [11], an algorithm based solely on self-play reinforcement learning, without human data, guidance, or domain knowledge beyond game rules. Starting *tabula rasa, AlphaGo Zero* achieved superhuman performance, winning 100-0 against the previously published *AlphaGo*. One of the improvements is that *AlphaGo Zero* uses a backbone network with two prediction heads (the actor and the critic), rather than separate policy and value networks. Moreover, the input features were only the black and white stones from the board. Since the algorithm only uses self-play, *AlphaGo Zero* was able to discover new non-standard strategies beyond the scope of traditional Go knowledge.

3. Environment

In this chapter, we analyze the nature (complexity, properties, solutions) and rules (game structure, how to play, how to win) of Connect4 [3.1]. Afterward, we present the game as a Reinforcement Learning environment. We will explain in detail how agents alternate turns to interact with the environment [3.2]. Finally, we define the reward function that is used by the environment to tell the quality of a given action in a given state [3.3].

3.1. Connect Four

Connect Four (or Connect4) [12] is a two-player connection game. Each player has 21 colored tokens and they alternate turns to place them in a 6 × 7 grid. At each turn, the active player chooses one column to drop a token that will occupy the lowest available position within that column. The first player who forms a horizontal, vertical, or diagonal line of four of one's tokens becomes the winner. If the board is filled before either player achieves four in a row, then the game is a draw.

Connect Four is a *perfect information game*, in which there is no hidden information. It means that what differentiates a good player from a bad player is how effectively they handle the information of the game. It is also a well-known *zero-sum game*, and this property entails that if one player wins, the other player loses. When a player tries to maximize their probability of winning, it is also minimizing the opponent's probability of winning (i.e. maximizing the opponent's probability of losing).

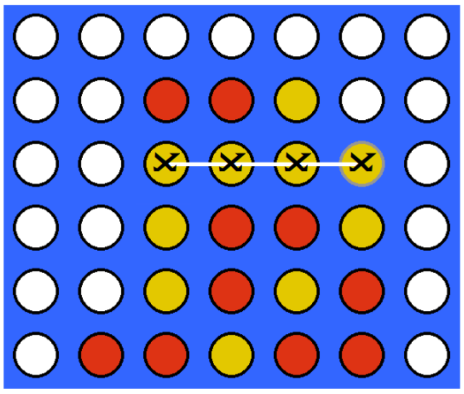


Figure 3.1. *An example of a Connect4 game won by the yellow player (horizontal line)*

The fact that the game rules and configuration are simple does not mean that the game complexity is low. In the case of the classic Connect 4, there are roughly 4,5 trillion valid board states. The game was created in 1974, and the initial attempts to solve it didn't rely on brute-force approaches because they seemed infeasible due to the game's complexity and the computational power available at the time.

In 1988, the game was solved for the first time. The proposed solution described a *first-player-wins* strategy, in which the first player can force a victory by starting in the central column, even if the second player plays optimally. If the first player starts in the columns adjacent to the center, the second player can force a draw. If the first move is played in one of the outer columns, the second player can force a win. In 1995, the game was solved using brute-force methods that became feasible thanks to the technology at that time.

In general, board games have been broadly studied from a mathematical point of view, especially in the fields of game theory and artificial intelligence. In these games, the rules tend to be simple, they usually allow the creation of variants of different complexities, and they open the door to research work on finding new strategies and solutions. For these reasons, we decided to explore the possibilities that Deep Reinforcement Learning algorithms can offer to play Connect4.

3.2. Environment structure

In Reinforcement Learning, the environment is the world in which an agent learns to solve a given task. Agents are allowed to interact with the environment, but not to change its rules. In this project, our agents learn to play Connect4, and the environment is the game itself. In the following paragraphs, we explain how we turned Connect4 into a Reinforcement Learning environment, how the reward function was defined, and how to deal with two agents interacting in the same environment.

In the previous section [3.1] we explained the properties and the rules of Connect4. We mentioned that the classic game board has 6 rows and 7 columns, and players alternate turns to select columns to drop their pieces. If we use some Reinforcement Learning terminology, Connect4 is considered a *discrete* environment. The *action space* A is the finite set of valid columns (i.e. not filled columns) at each turn, so $|A| \leq 7$. The *state space* S is large but finite, and its size is the number of possible board configurations, which is estimated to be $|S| \approx 4,5 \cdot 10^{12}$. Connect4 is a *perfect information game,* so agents can observe the full state of the environment (no information is hidden).

It is also considered a *deterministic* environment because the next state is unequivocally determined by the current state and the agent's action. Since there are *terminal states* in which no more actions can be played (e.g. a win or a draw), it is said to be an *episodic* environment. The main difference with respect to the common Reinforcement Learning settings is that in Connect4 two agents with conflicting goals take turns to change the state of the same environment.

At each turn (or time step), the state of the environment is the board at that turn, and it is always shown from the perspective of the active player. It means that the board is represented as a two-dimensional matrix with three different values: ‘0’ for empty positions, ‘1’ for the active player’s pieces, and ‘-1’ for the opponent’s pieces. This way, all agents can assume that they always play the ‘1’ pieces. It is the environment that is in charge of alternating the roles (‘-1’ and ‘1’ pieces) of the active player and the inactive player to ensure the correct functioning of the game. In other words, from the agents' point of view, there is no such thing as a *red player* and a *yellow player*, it is *me* and *my opponent*. Since the agent is not attached to any color, in Chapter 4 we can use *self-play* (a single agent playing for both colors in the same game) to generate training data.

Below there is *Figure 3.2* which shows an example of two consecutive turns (one per player). Note that none of the agents know which color they are playing because the board they observe is encoded using the aforementioned {-1, 1} or {*my opponent, me}* notation. Also note that when *Agent1* takes action, the environment state is updated before moving to the next turn.

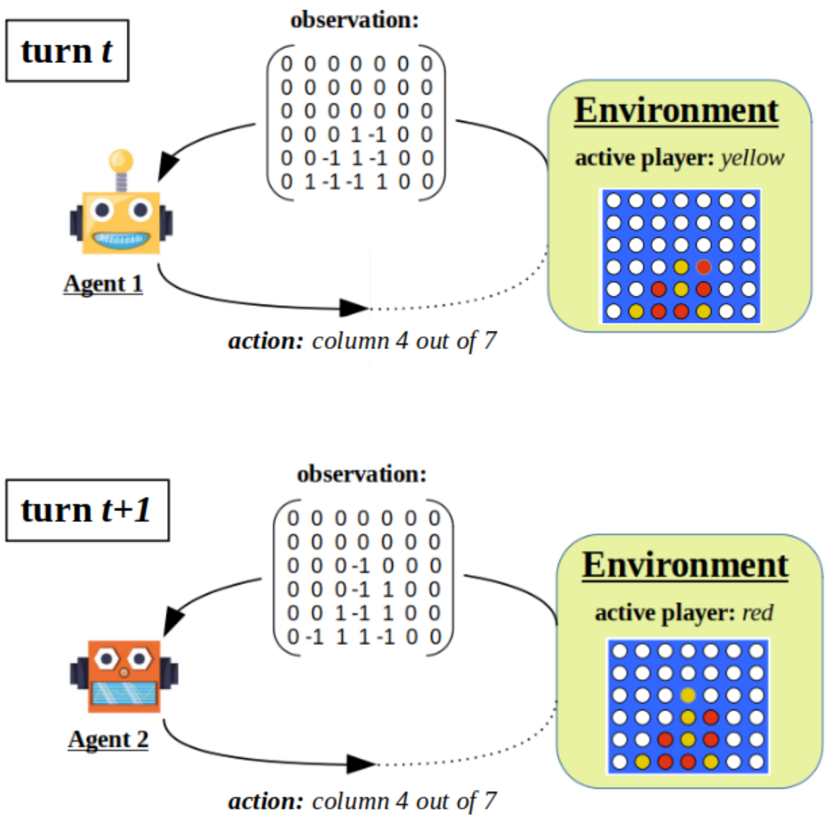


Figure 3.2. *An example of how two agents interact with the environment to play the game*

3.3. Reward function

When a Reinforcement Learning agent is trained to solve a task, what the agent actually learns is to maximize the expected return. For this reason, the definition of the reward function is crucial in every RL problem since it is the feedback signal that agents receive to update their strategy. Unfortunately, in most real-world problems the reward signal is not as good as one would like. The rewards are often sparse and delayed, as happens in Connect4.

In the particular setting of two-player zero-sum games, there is rarely a simple way to tell the exact quality of an action in the middle of the game. It could arguably be said that the reward at an intermediate turn depends on the game outcome in some sense. For instance, in the game of chess, the queen is one of the most powerful pieces, so when the queen is sacrificed to win the game it is considered a good move. But if done in vain, it is considered one of the worst moves. Discovering how each intermediate action affects the outcome of the game is known as the *credit assignment problem.*

Before dealing with the reward of intermediate turns, let us first focus our attention on the last actions of the game (which are played by the winner), and also on the penultimate turns (which are played by the loser). In other words, we first address the case of terminal states, certain wins, and certain losses. For the aforementioned scenarios, we designed a non-zero reward function to help the agent learn how to act in these critical situations. Regardless of the intermediate turns, when the outcome of the game is at stake it is easy to tell if an action is good or bad. We took into account the zero-sum property of Connect4: note that when an agent receives a positive reward the opponent's situation becomes worse. The following list explains in more detail these non-zero rewards and the situations where they are given.

- **+1** if the agent wins the game or forces a certain win (i.e. when it has more than one option to win so the opponent can't do anything to prevent it).

- **-1** if the agent loses the game or does not block an opponent's clear chance to win in the next turn. The agent is penalized even if the opponent does not seize the opportunity to win, as it is a bad action anyway.

- **+0.5** if the agent blocks an opponent's clear chance to win in the next turn.

- **-0.5** if the agent does not seize a clear opportunity to win in the current turn.

- **0** otherwise.

Despite addressing the reward function for terminal and close-to-terminal states, there are still a lot of intermediate actions that receive a zero reward. In theory, it would not be a problem because agents learn to maximize future rewards, not the immediate ones. They should be able to estimate the expected future reward of an action even if the current reward is zero. However, in practice, the learning process benefits more from non-zero rewards.

We know that intermediate actions somehow have an impact on the outcome of the game for each player, but we do not know how to quantify that impact accurately. Our approach here is an estimation based on two important assumptions. First, we assume that the latest actions of a game have more impact on the outcome than the earlier actions. Second, we assume that all actions leading to a game win are good to some degree and that all actions leading to a loss are bad to some degree. Based on these two assumptions, we defined an expression to estimate the impact (reward) that an intermediate turn *t* has on the outcome of a game that lasted *T* turns. In other words, this expression is used to *backpropagate* the terminal rewards to the intermediate turns for both the winner and the loser. It includes an exponent *n* that can be fine-tuned and is used to control the slope of the backpropagation.

$$reward\left(t,\, T,\, r_T\, ;n\right) \;=\; r_T \cdot \left(\frac{t}{T}\right)^n$$

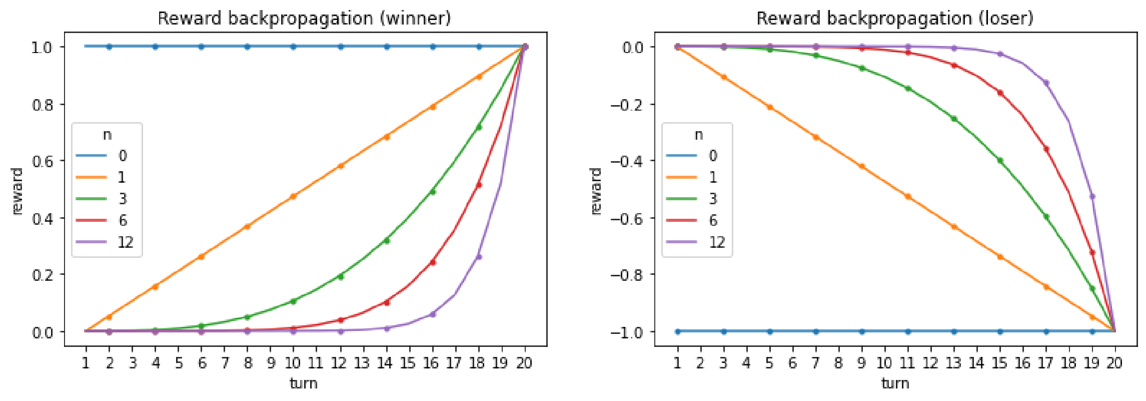


Figure 3.3. *Reward backpropagation for different 'n' values in a game that lasted 20 turns*

Figure 3.3 shows how the exponent *n* affects the backpropagation of the terminal rewards. In the last turn of the game (the twentieth turn in this example), the winner receives a +1 reward and the loser receives a -1. The winner backpropagates the positive reward to their intermediate actions to take them more often in future games. On the other hand, the loser backpropagates its negative reward to discourage those actions that led to losing the game.

When $n = 0$, the backpropagation is constant so all turns are considered to be equally important, and when $n \to \infty$ there is no backpropagation at all. When $n = 1$, the slope decreases linearly, and for $n \geq 2$ earlier turns receive a reward that is close to zero. After testing some values, we finally chose $n = 3$ (green line in Figure 3.3) to train our agents in Chapter 4.

From our empirical experience, regardless of the n value, backpropagating the last reward achieved better results (i.e. higher win rate against the baselines) than not backpropagating at all. We finally chose $n = 3$ because it was a good trade-off between giving more credit to the latest actions and also taking into account the earlier ones. It is important to mention that backpropagation is applied only to those intermediate states with a zero reward. It means that if a close-to-terminal intermediate state receives a non-zero reward from the list we described above, we use the reward from that list because it is more reliable than the backpropagated one.

4. Agents

In this chapter, we present how we trained different Deep Reinforcement Learning agents to play the game of Connect4. In general, Deep RL algorithms face several challenges. For instance, they tend to be unstable and their performance heavily depends on the chosen values for the hyperparameters. Apart from that, plenty of data and computational power are often required to train an agent, even for simple tasks. And when the agent leaves the simulated training environment to enter the real world, new problems may appear. These problems may become bigger when agents have to learn from scratch, without any human supervision. One of the most intuitive ways to give more stability to the algorithms is by introducing some previous knowledge in the form of human-generated data. This is what we did in this project.

When DeepMind trained *AlphaGo* [11] to master the game of Go, the first stage of the training pipeline consisted of teaching a policy network to predict the next move of professional players. This supervised learning task provided immediate feedback and high-quality gradients. In that stage, the goal of the policy network was to imitate human experts. Then, the network was refined using policy-gradient reinforcement learning with self-play to maximize the expected return and learn the value of each state. In other words, it was trained to win games and improve their performance, rather than maximizing the classification accuracy.

Our training pipeline has two main stages and was partially inspired by how *AlphaGo* was trained (for a comprehensive explanation of *AlphaGo*, refer to section [2.5]). Our first training stage is a supervised learning one-class classification task. We created a synthetic dataset with games played by a hand-crafted heuristic (a mid-level player), and we used the data to train a convolutional network to predict the actions of the heuristic. Then, we applied *transfer learning* to reuse the model trained for this first task (Supervised Learning) as the starting point for the models in our second task (Reinforcement Learning). The pre-trained weights of the network were used as the starting point of all the RL algorithms that we tested. The convolutional block was regarded as a *feature extractor,* so the convolutional pre-trained weights were frozen. Then, for each Deep RL algorithm, the fully connected block of the network (which may have either one or two prediction heads depending on the algorithm) was trained to solve the RL task in each case. *Figure 4.1* shows a visual representation of this *transfer learning* approach.

Beyond the type of game that is studied, there are important differences between our approach and *AlphaGo*. The first one is the objective of the project. *AlphaGo* aimed at being the first computer program to defeat a human professional player in the full-sized game of Go, whereas our main goal is to compare how different Deep RL algorithms learn to play Connect4. The second difference is the use of *Monte-Carlo Tree Search*. *AlphaGo* uses a high-performance lookahead search to simulate several games and choose its actions, whereas our RL agents are not allowed to simulate future moves. The third difference is the available computer resources. *AlphaGo* ran several simulations in parallel and was carefully designed to be as efficient as possible. Our training pipeline is intended to run on a single GPU and was designed to be general and simple so we could make a fair comparison between the different Reinforcement Learning algorithms we tested.

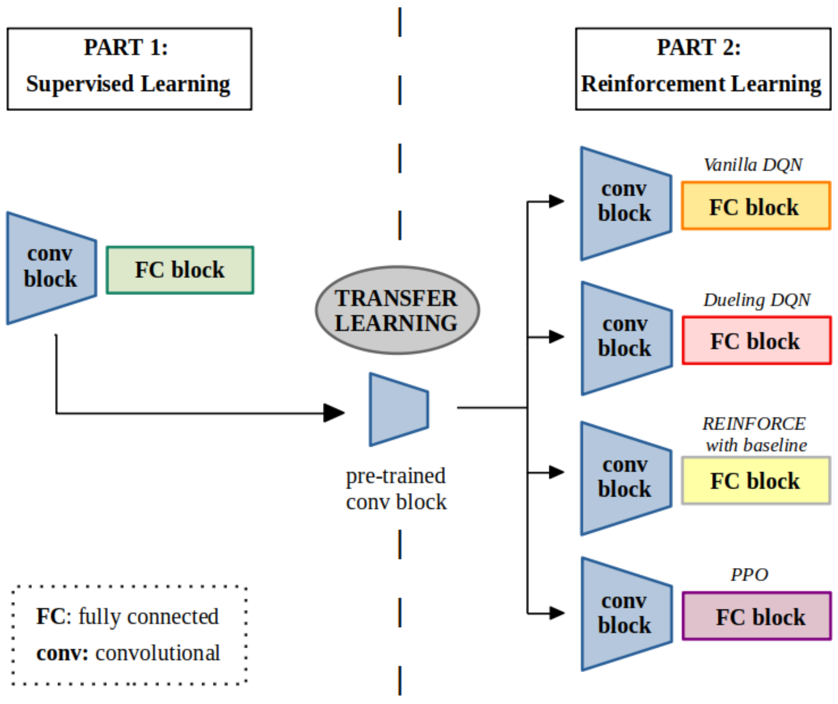


Figure 4.1. *Visual representation of how 'transfer learning' is applied in our training pipeline.*

DeepMind went a step further and created *AlphaGo Zero* [12], a better version of *AlphaGo* that was trained solely by self-play reinforcement learning and outperformed *AlphaGo* and other sophisticated computer programs. Without human supervision, *AlphaGo Zero* managed to learn non-standard strategies that went beyond the scope of traditional Go knowledge. However, the goal of our project is not to discover new strategies to play Connect4. Learning uniquely from self-play requires millions of timesteps before finding good solutions. For this reason, we first trained a network on a supervised task and then we moved on to self-play reinforcement learning with a pre-trained network that already had some knowledge of the game.

In this chapter, we start by introducing the two baselines we have implemented in this project [4.1], and we explain their main advantages and disadvantages. These baselines are used to measure the performance of the trainable agents. Afterward, we explain the supervised learning task in more detail [4.2]: how we generated the dataset [4.2.1], how we trained the policy network [4.2.2], and what results we obtained in this first stage [4.2.3]. Then, we move on to the Reinforcement Learning algorithms. Here we tried to refine the policy network to improve its performance. For each algorithm, we explain how it was adapted to be used in two-player zero-sum games with alternating turns (a competitive Multi-Agent Reinforcement

Learning setting) and the training process. We also present our solutions and the obtained results. The Deep RL algorithms we tested in this project are Vanilla and Dueling Deep Q-Networks [4.3], REINFORCE with baseline [4.4], and Proximal Policy Optimization [4.5].

4.1. Baseline agents

We developed two non-trainable agents based on different heuristics to play the game. These Baseline Agents are used to measure the performance of the Reinforcement Learning agents. Since the logic applied by the baseline agents is known, if a trainable agent beats a Baseline Agent, it might mean that they have learned some knowledge beyond the baseline's scope. In this section, we introduce our Baseline Agents: the *Random Agent* and the *N-Step Lookahead Agent.*

4.1.1. Random agent

In every scenario, the most basic agent is always the *Random Agent*. In Connect 4, the random agent selects one of the available columns to drop their pieces. It does not have a strategy because it does not know the goal of the game, so the columns are selected at random regardless of the state of the game. The only rule the *Random Agent* knows is that if a column is full, it can't be chosen. The Random Agent is a very weak player, and a minimum knowledge of the game will be enough to start winning against it. A reasonably good agent should consistently beat the Random Agent roughly 90-95% of the time.

4.1.2. N-step lookahead agent

In games where players alternate turns, a human player is concerned about their current turn but also about how the opponent can counterattack. In other words, a human player can look ahead and estimate the probabilities of winning the game for each of their available moves. Depending on the person's capacity, it might also take into account how the opponent will respond, and imagine even more turns ahead. In the game Connect4, the game tree structure is shown in *Figure 4.2* below:

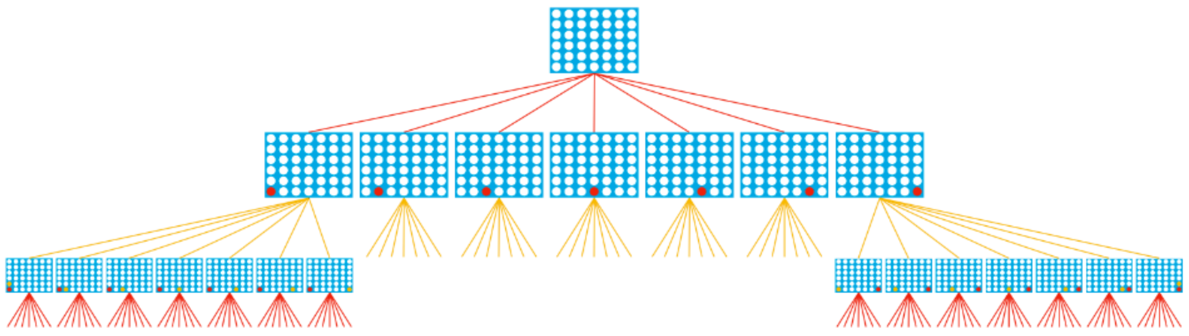


Figure 4.2. *Connect 4 game tree for a lookahead search (from Kaggle tutorials* [14])

Based on this idea of simulating future games, we implemented an agent that we call *N-Step Lookahead Agent* (*NstepLA* from now on). When the *NstepLA* takes a turn, it builds the game tree of depth *N,* i.e. it explores all the possible combinations for the next *N* turns. Then, it uses a hand-crafted evaluation function to assign a score to each board in a leaf node. Finally, it goes up the game tree collecting the partial results using a *minimax search.* In a minimax search, the active agent chooses the action that yields the highest score, and it assumes that the opponent will counteract this by choosing actions to force the score to be as low as possible. That is why it is called minimax, because the active player aims at maximizing the score and the opponent aims at minimizing it. It is considered a pessimistic heuristic because it assumes that the opponent plays optimally and this is not always true.

The leaf nodes (boards) receive a score that is defined by a hand-crafted evaluation function that is specifically designed for this game. The idea is to define a set of patterns that are relevant to the game and assign a score to each of them. Positive scores indicate patterns to seek, whereas negative scores indicate patterns to avoid. The goal of Connect4 is to connect four of your pieces in the same line, so positive patterns check the connectivity of your pieces, and negative ones check the connectivity of the opponent's pieces. *Figure 4.3* is shown below and presents the patterns and scores that have been used in this project. Many score values may work, but the ones we used came from the Kaggle Reinforcement Learning tutorial [14]. With these scores, the *1stepLA* consistently beats the Random Agent in almost every game (≈99% of the time).

The patterns presented in *Figure 4.3* are sought vertically, horizontally, and diagonally. Not just these exact patterns, but also the equivalent ones. For instance, the set of equivalent patterns for *[red, red, red, empty]* is all the possible combinations of four connected spaces in which just one of them is blank and three of them are red. The score assigned to a leaf node is the sum of the pattern scores weighted by the number of occurrences of each pattern in any of the allowed directions

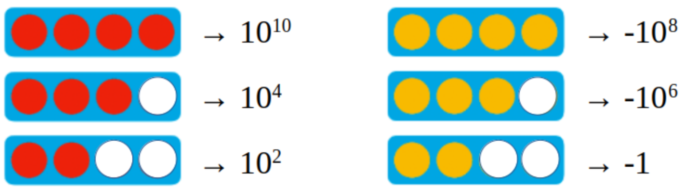


Figure 4.3. *Patterns and scores to evaluate a leaf node (board) in the game tree. These scores are for the red player to maximize the overall score.*

When the *minimax search* is done, the *NstepLA* has assigned a score to the actions they can take in the current turn. When there is more than one action sharing the highest score, there are two ways to break the tie. The first option is to select at random one of the best-scored actions. The other option is to select the action that is closer to the central column because central columns are more valuable in Connect4. If the first option is chosen the policy is stochastic, whereas the second option leads to a deterministic policy and also a higher win rate in some cases.

Without any evidence, we could speculate that the level of an average human being is somewhere between a one-step and a two-step lookahead search. In the common 6 × 7 Conect4 board, it may be easy for a human player to imagine the impact of the 7 actions in their current turn. However, it may be too difficult to imagine the impact of the $7^2 = 49$ possible combinations of their move and the opponent's next move, let alone the $7^3 = 343$ possible combinations of three steps ahead. We guess that an average human player can efficiently perform a full one-step lookahead search and a partial two steps lookahead search focusing on the moves that lead to crucial game states. This estimation is just a basic and unofficial human baseline to evaluate the agents.

4.2. Supervised Learning Task

In this section, we trained a policy Neural Network to predict the actions of a mid-level player. Since we did not have access to a dataset of games played by professional Connect4 players, we were forced to create a synthetic dataset of moves played by one of our baselines. We considered the *1-Step Lookahead Agent* (*1StepLA*) to be a mid-level player, and we created a dataset of 200k unique *(state, action)* pairs played by this agent. Then, we tried different network architectures to solve this one-class classification task: to predict the actions played by *1StepLA*. At the end of the training, the best network achieved ≈85% test accuracy with minimum overfitting. We designed a convolutional architecture with 186k trainable parameters that we called *CNET128*.

Here we describe the creation of a synthetic dataset of actions played by *1StepLA* [4.2.1] and every detail of the training process: data preprocessing, hyperparameters, and assumptions [4.2.2]. Finally, we present the results and introduce the next chapter that will improve the model with different Reinforcement Learning algorithms and self-play [4.2.3].

4.2.1. Dataset creation

The lack of a large dataset to train our supervised policy network forced us to create a synthetic one. We considered one of our baseline agents to be a mid-level Connect4 player. In particular, we chose the *1-Step Lookahead Agent* (*1StepLA* from now on) to generate the data. Of course, other agents that look more steps ahead will play better. But the *1StepLA* offers a trade-off between performance and speed. Looking more steps ahead would increase the quality of the actions but at the expense of an exponential increase in the computational cost to simulate and evaluate more boards.

The *1StepLA* played against itself and generated 200k unique (*state, action)* pairs. When evaluating nodes in the *minimax search, 1StepLA* breaks the ties by choosing the most central action. This way, the policy is deterministic and the *state* completely determines the *action.* In the common Connect4 game, a *state* is a 6×7 board from the active player's perspective. It means that the board has values: '0' for empty positions, '1' for positions with an active player's piece, or '-1' for those with an opponent's piece. The *actions* represent the columns and are values from 0 (leftmost column) to 6 (rightmost column). For more information on the dynamics of the environment and how agents interact with it, refer to section [3.2]. Before saving the data, the states were flattened (i.e. collapsed into one dimension) and turned into string format. The data pairs were saved in a text file. When the data is loaded, the states are reshaped into a 6×7 grid of integers.

4.2.2. Training

Once the dataset was ready, we trained the convolutional network to estimate the policy of our *1StepLA*. We chose a convolutional architecture to take advantage of the board structure of the game. There is a convolutional block that is followed by a sequence of fully connected layers. All the layers are followed by rectified linear units (ReLU) and there are neither dropout nor pooling layers. The input of the layer is the state of the environment after some preprocessing steps that are described in this section. A *Softmax* activation is applied to the output in order to get the policy. We designed an architecture with 186k trainable parameters which is shown in Figure 4.4 and we called it C*NET128*.

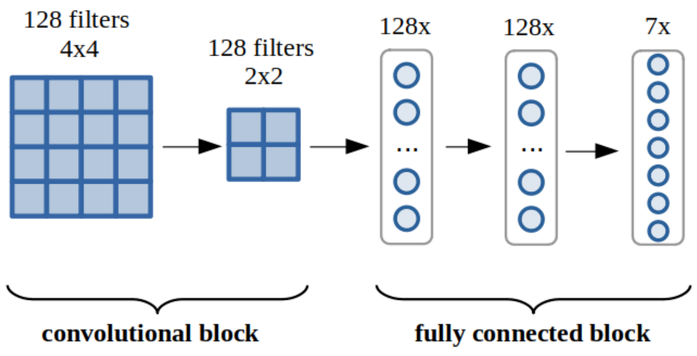


Figure 4.4. *Our CNET128 architecture (186k trainable parameters)*

Before feeding the network, the *states* have to be preprocessed. In the common Connect4 game, a raw *state* is a 6 × 7 board from the active player's perspective. As we explained in section [3.2], it means that the *state* of the environment is a board with values: ‘0’ for empty positions, ‘1’ for the active player’s pieces, and ‘-1’ for the opponent’s pieces. The preprocessing of a *raw state* has two steps. The first step is a one-hot encoder of the pieces of both players that results in two binary 6 × 7 planes. The first plane has values ‘1’ for the active player’s pieces and ‘0’ for empty positions and opponent’s pieces. The second plane has values ‘1’ for the opponent’s pieces and ‘0’ for empty positions and the active player’s pieces. When stacked, the result is a 6 × 7 × 2 three-dimensional board. The second step is to highlight the positions that can be filled by the active player in the current turn. These positions are set to *-1* in both channels so the network can easily see the position that can be filled at that turn. We also tried to feed the network with the raw states (i.e. without the one-hot encoding and the available positions highlighted), but the test accuracy was ≈5% lower. We decided to implement these two data preprocessing steps because they could help the Reinforcement Learning algorithms ([4.3], [4.4], [4.5]) to solve their respective tasks. *Figure 4.5* offers a visual example of how the environment states are preprocessed.

We split the 200k data samples into 160k training samples (80%), 20k validation samples (10%), and 20k test samples. We trained for 20 epochs, using a batch size of 64, a learning rate of 5e-4, and a weight decay (L2 regularization) of 2e-3. We also tested other pairs of batch sizes and learning rates: (64, 1e-4), (128, 1e-4), (128, 5-4). We finally used the combination (64, 5e-4) but all of them led to very similar results in terms of training speed and test accuracy when using *CNET128*.

We tested different networks with the same structure as *CNET128* shown in *Figure 4.4*, but instead of using 128 convolutional filters or 128 units in each layer, we used 96, 160, and 192. The convolutional kernel sizes were the same for all of them (4 × 4 and 2 × 2). We finally chose *CNET128* because it was the best combination of low complexity and high performance. The results were worse using smaller networks and did not improve significantly when more complexity was added.

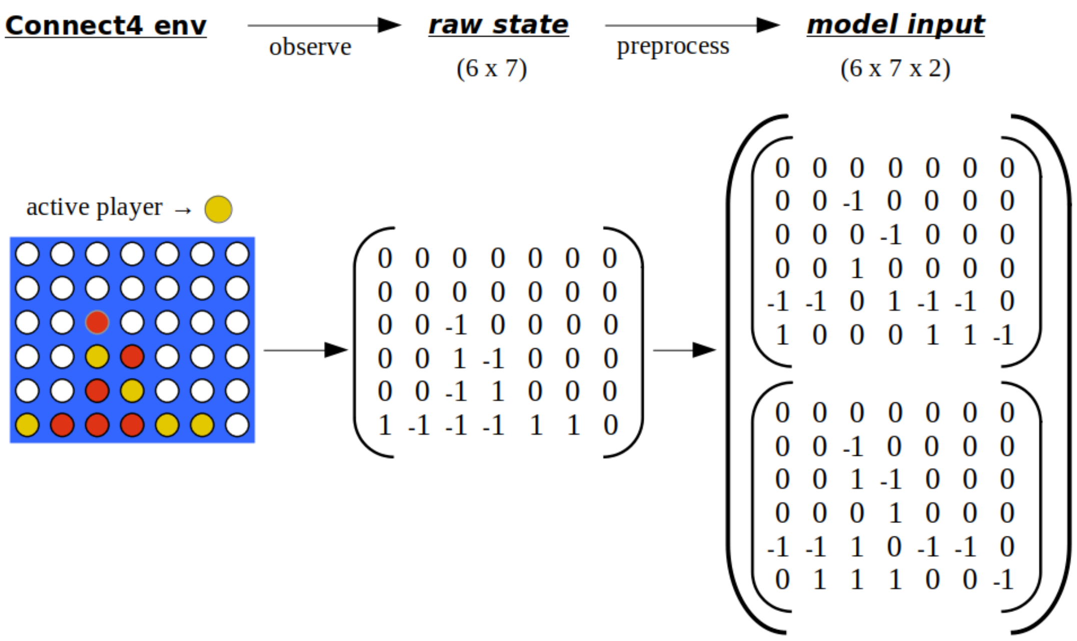


Figure 4.5 *An example of data preprocessing before feeding the neural network.*

4.2.3. Results

After training for 20 epochs, *CNET128* achieved ≈87% training accuracy and ≈85% test accuracy with minimum overfitting. We selected the best network weights to continue the learning pipeline so they serve as a starting point for the Reinforcement Learning algorithms. An agent following the trained *CNET128* would beat the *Random Agent* ≈95% of the time and would beat the *1StepLA* ≈50% of the time. Figure 4.6 shows the training and validation losses (cross-entropy loss) and accuracies. In the plots, it can be seen that the overfitting was minimal and that after 50k updates the network converged.

After achieving 56% test accuracy, *AlphaGo* improved its game by playing against itself and optimizing its policy following a Reinforcement Learning algorithm (in particular, REINFORCE with baseline). After achieving 85% test accuracy, our network will try to improve its game using different Reinforcement Learning algorithms. The knowledge of this part is *transferred* to the RL agents as shown in Figure 4.1. The convolutional block will be a non-trainable *feature extractor* and the fully connected block will be trained to solve the Reinforcement Learning task in each algorithm. In the following sections, the network will learn to win games, rather than maximize the classification accuracy.

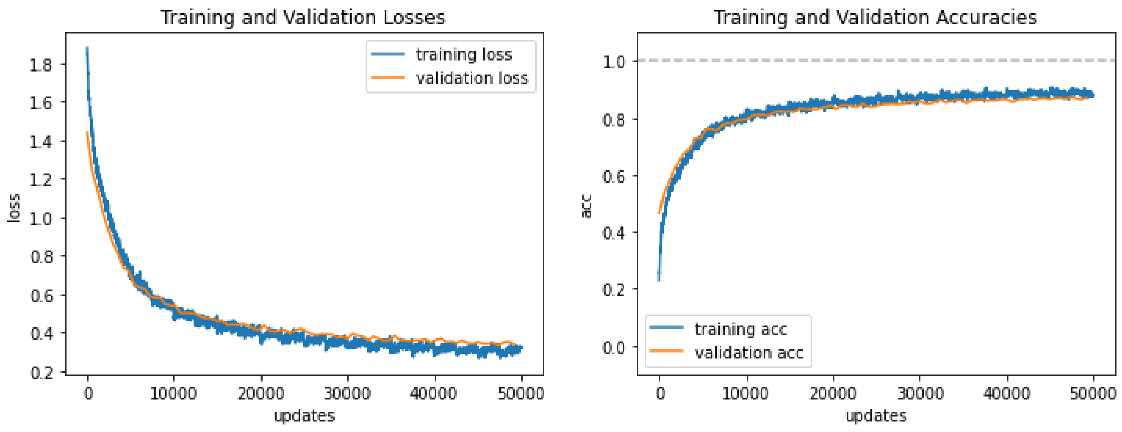


Figure 4.6. *CNET128 training and validation results: losses and accuracies.*

Some of the Deep RL algorithms in the next section require two different predictions. This can be accomplished by either using two different networks or by using a single backbone network with two prediction heads. In this project, we implemented the latter: a two-headed network. The convolutional block and the first fully connected layers are our *backbone network.* Then, two independent heads are appended to learn two different tasks. The first head outputs seven units (one for each column of the board) to predict either the policy, the Q-values, or the advantage of each action. The second head (when it is required) outputs the value of the input state and has 16k trainable parameters. Below there is *Figure 4.7* which shows how the two-headed version of *CNET128* is structured and the role that each part of the network plays in the next training process.

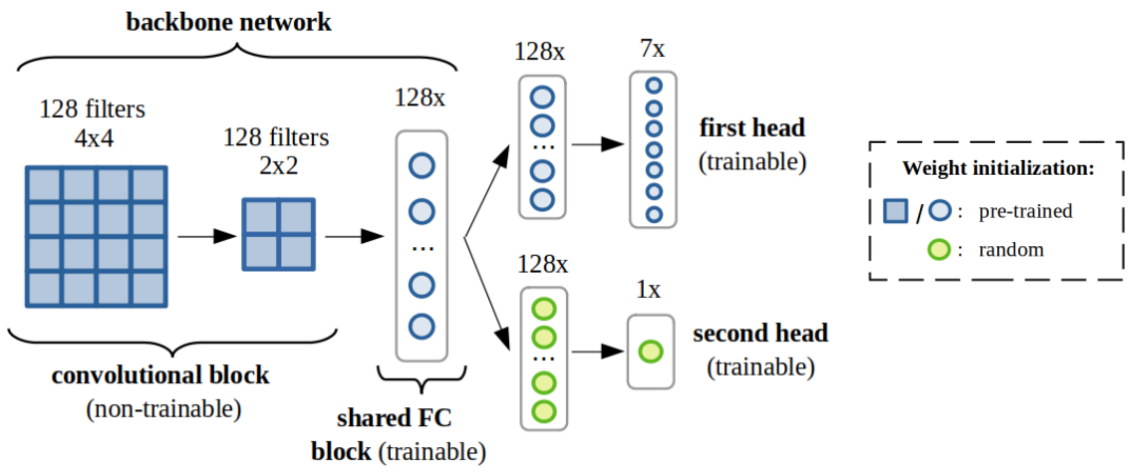


Figure 4.7. *Two-headed version of CNET128 architecture. It is used as the starting point for some of the Deep RL algorithms in the following sections.*

We mentioned that the one-headed version of *CNET128* has 186k parameters (*Figure 4.4*). In this supervised learning task, all of them were trained. However, in the following sections, the convolutional block (70k parameters) is frozen, as shown in *Figure 4.7*. It means that in the next sections (where we train Deep RL algorithms), the one-headed version of *CNET128* will have 116k trainable parameters (186k in total), and the two-headed version will have 132k trainable parameters (16k for the second head, 202k in total).

Among the RL algorithms used in this project, Vanilla Deep Q-Network [4.3] is the only one that uses a one-headed network to estimate Q-values, whereas the rest of them require a two-headed architecture (i.e. two different predictions). Dueling Deep Q-Network [4.3] estimates the advantage for each action and the state value. REINFORCE with baseline [4.4] estimates the policy and the state value, and so does PPO [4.5]. In the following sections, we describe in detail how we used Deep RL algorithms to improve the pre-trained *CNET128* model to create stronger Connect4 players.

4.3. Vanilla and Dueling Deep Q-Network

In Chapter 2 we introduced the Q-learning algorithm [2.1.1] to learn the value of an action in a particular state. We also introduced *Vanilla Deep Q-Network (Vanilla DQN)* [2.2.1] and *Dueling Deep Q-Network* [2.2.2], two Deep Learning variants of Q-learning which use Neural Networks to estimate the Q-values. However, the original version of these algorithms was created to deal with the typical single-agent setting, and Connect4 is a two-player zero-sum game. To train our agents, we first have to adapt the original algorithms to our particular environment. More importantly, Connect4 is a turn-based game, so when an agent is in a given state s, it must take into account that the next state s' (the next turn) is played by an opponent who is also trying to win the game (and beat the first agent).

In order to use these value-based techniques in a two-player zero-sum game (a competitive multi-agent setting), some changes in the target have to be made. (Jianqing Fan *et al., 2020)* [15] published a paper in which they introduced the *Minimax-DQN algorithm* for zero-sum Markov games with two players playing simultaneously. Fortunately, there is also a variant for turn-based games like Connect4. In this variant, the Q-value for a given pair (s, a) is updated taking into account that the next state s' is played by the opponent.

The only difference with respect to the *DQN* training loop shown in *Figure 2.6* is the definition of the non-terminal target (y_j). In turn-based zero-sum games, the next Q-value is subtracted (instead of added) from the current reward because the expected return for the active player is the opposite of the expected return for the opponent (one of them will win and the other one will lose). In the expression below, we use $Q(s, a; \theta, player)$ to refer to the expected return from the state s if $player$ takes turn using a Q-network with weights θ.

$$y_j = Q(s_j, a_j; \theta, player) = r_j - \gamma \cdot max_{a'} Q(s_{j+1}, a'; \theta', opponent)$$

In order to add more stability to the predictions, we took advantage of the vertical symmetry of the board by taking the average of symmetric Q-values. For a given state (board) s, we compute the symmetric board s^{sym} by flipping horizontally (*fliplr*) the state s. Based on the rules of Connect4, these two boards are exactly the same, so their actual Q-values are also the same (but flipped horizontally as well). By taking the average of these symmetric values, we reduce the variance and avoid the overestimation bias that comes from using just one estimation. The expression below shows the computation. We use *fliplr* to refer to the horizontal flip operation (left/right direction). This technique is only used when playing games in competitions, not when training the network.

$$Q^{avg}(s, :) = \frac{1}{2}\left[Q(s, :) + fliplr\left(Q(s^{sym}, :)\right)\right] \qquad where\ s^{sym} = fliplr(s)$$

In the next two sections, we present our solutions for these two value-based Deep RL algorithms. The only difference between *Vanilla DQN* and *Dueling DQN* is in how they compute the Q-values (i.e. the network architecture), but the training steps are exactly the same and we also used the same hyperparameters in both cases. For these reasons, we explain their training process in just one section [4.3.1] instead of in two different ones. After training the agents, we present the best solutions and results in section [4.3.2].

4.3.1. Training

The training process that is explained here is used to train both *Vanilla DQN* and *Dueling DQN.* We use the term *Deep Q-Network (DQN)* to refer to the general parts of the training that are common in both algorithms. In those parts where the implementation is specific for each algorithm, we discuss each one separately using the terms *Vanilla DQN* and *Dueling DQN* to differentiate them.

Vanilla DQN uses the one-headed version of our *CNET128* (defined in [4.2.2]) which has 186k parameters (as shown in *Figure 4.4*), whereas *Dueling DQN* uses the two-headed version which has 202k parameters (as shown in *Figure 4.7*). Before the training starts, the *DQN* inherits the weights of the pre-trained *CNET128* model [4.2.3] from the previous supervised learning task. So, in this section, the convolutional block (*feature extractor)* has 70k pre-trained parameters that are frozen and remain unchanged. Appended to the convolutional block is a sequence of fully connected layers, which are optimized to estimate the Q-values. In the case of *Vanilla DQN,* there are 116k trainable parameters in the fully connected layers, whereas this number is 132k for *Dueling DQN* (the second head has 16k trainable parameters).

Apart from the *online network* that is trained, two more networks are also created (but not trained). The first one is the *target network* that is used to compute the target and avoid overestimating the Q-values. The *target network* θ' starts with the same weights θ as the *online network,* and it is updated every 400 iterations ($\theta' \leftarrow \theta$). The second network is the *old network* that keeps the best weights found so far. When the performance of the *online network* decreases, the last updates are undone and it goes back to the *old weights*. We considered that the performance decreased when the win rate against the *1StepLA* was 8% lower than the best one achieved so far. Every time that the *online network* achieves a new best win rate against the *1StepLA,* the *old weights* are replaced with the *online weights* ($\theta_{old} \leftarrow \theta$).

Regarding the training data, the training episodes are generated using self-play, i.e. the *online network* plays against itself. We used an *Experience Replay Memory* to store transitions in the format $(s, a, r, s', done)$. It is a *first-in-first-out* structure in which the older transitions are removed to add new ones. At each training step, the network samples a random batch of 48 transitions to update the weights. The memory we used had a capacity for 60k transitions, and the training did not start until it had 30k of them (sufficient to ensure that they are not correlated). New self-play games are constantly generated to update the data in the memory.

As explained in [3.3], the terminal rewards are backpropagated to give some credit to the intermediate actions, and this process is done within the memory. When a new self-play game (episode) is played, the memory preprocesses the transitions of the episode and stores them. It is assumed that what the memory receives is an ordered sequence of transitions that belong to the same finished episode, so the reward backpropagation can be done properly.

For each new episode that is added to the memory, a symmetric version is created and added as well. In Connect4, the board has a vertical symmetry that can be exploited to double the number of data samples and also to help the network learn to take advantage of this symmetry. To create a symmetric version of a given transition $(s, a, r, s', done)$, the states s and s' are flipped horizontally, and the new action becomes $\#columns - a$. The values r and $done$ remain unchanged.

In order to address the *exploration-exploitation dilemma*, we implemented a *decaying epsilon-greedy* strategy. When the training starts, the focus is on exploring the environment, but eventually, the focus is on exploiting it. The expression below defines the decaying exploration rate we used for a training episode at timestep *t*.

$$exploration\_rate(t) = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) \cdot exp\left(\frac{-1 \cdot t}{\epsilon_{decay}}\right)$$

$$where \quad \epsilon_{start} = 0.8; \quad \epsilon_{decay} = 500; \quad \epsilon_{end} = 0.05$$

In our initial attempts we realized that, when the exploration rate was minimum, the games generated with self-play were too repetitive. In the end, the Replay Memory had little variety of game boards. In order to keep exploiting the environment and still visit a wide range of new game boards, we decided that the initial board would be random. The common initial empty board was no longer the first environment state. Instead, we let the *Random Agent* use self-play to generate a sequence of non-terminal transitions so the *online network* had to finish the game. The random moves were not added to the Replay Memory, so in the end, the memory had a richer variety of high-quality transitions that were chosen by the *online network*.

Every 1000 training updates, the *online network* competes against the *Random Agent*, against the *old network*, and against the *1StepLA*. In each competition [5.1], 100 games are played and the win rates and the average game length are tracked. The win rate against the *Random Agent* is expected to be greater than 95% throughout the training. It serves as a sanity check to make sure that the performance is not decreasing significantly. The win rate against the *old network* is not really informative because the *old network* is constantly updated throughout the training, so this value is usually around 50%. In this self-competition, we were more interested in the average game length. Lastly, the win rate against the *1StepLA* is 50% before training the network, so the main objective is to exceed it.

The training loop lasted for 100k updates, which are 10k episodes taking into account the symmetric versions that we generated. The hyperparameters used were the same in both *Vanilla* and *Dueling DQN*: we used the *Adam* algorithm to perform *stochastic gradient descent*, the loss function was the *Mean Squared Error (MSE),* the batch size was 48, the learning rate was 1e-4, the weight decay (L2 regularization) was 5e-4, and the discount factor was 0.95. For each new episode that was added to the memory, 20 training steps were performed.

4.3.2. Results

Our starting point was a pre-trained network that learned to predict the actions of *1StepLA* (a mid-level player) with ≈85% of accuracy. Unsurprisingly, the win rate of this pre-trained network against the *1StepLA* was ≈50%. Our will in this section was to further improve this network using two variants of the Q-learning algorithm that are based on Deep Learning. After learning the Q-values, the *Vanilla DQN* beat the *1StepLA* 87% of the time (+37% increase in win rate). The training of *Dueling DQN* was also successful and was able to beat the *1StepLA* 94% of the time (+44% increase in win rate).

Figure 4.8 shows the plots of the evolution of the losses throughout the training loop. For visualization purposes, the first 1000 losses were discarded because they were extremely high. The reason is that the network needed some updates to transition from the supervised classification task to the Q-values regression task. The loss values that are shown here were smoothed using a moving average of 200 points. It is important to bear in mind that the training data varies throughout the training (new self-play games are generated and stored in the Replay Memory, and the oldest ones are replaced), so the loss is not our key indicator of the quality of the learning.

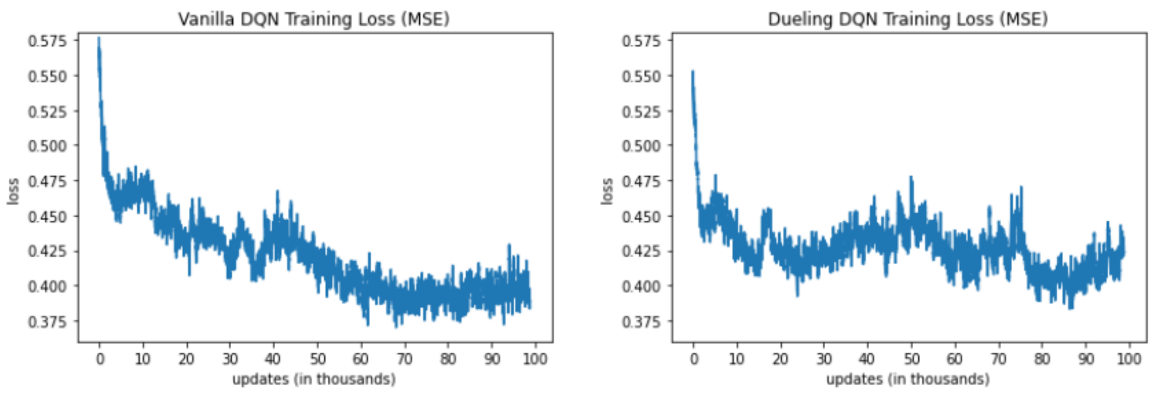


Figure 4.8. Training loss for both Vanilla and Dueling DQN

Every 1000 updates, the online network competes against its older version to see if the last updates make the performance increase or decrease. In this self-competition, the most important indicator is the average length of the games. The fact that the games last longer means that the network has more and more problems when it tries to win its older version: the network is learning to win games but also to prevent the opponent from winning, and this is very important to become a strong player. Both *Vanilla* and *Dueling DQN* started averaging less than 15 turns per game. In both cases, the maximum average game length was around 30 turns. *Dueling DQN,* unlike *Vanilla DQN*, manages to maintain this good value.

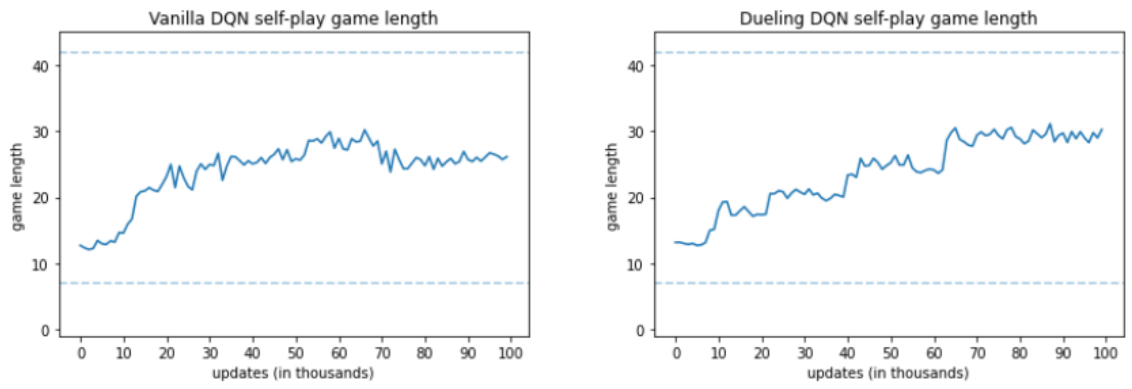
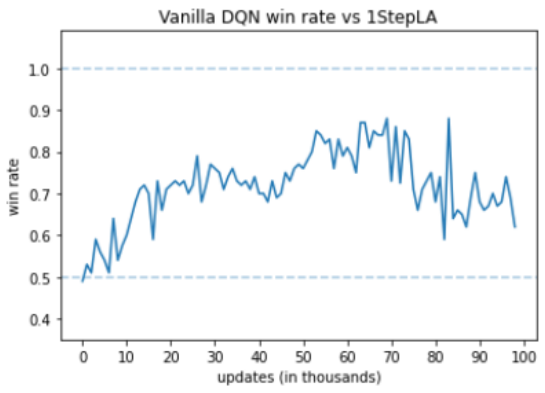


Figure 4.9. *Average game length in self-play games for both Vanilla and Dueling DQN.*

Finally, we show the evolution of the win rate against the *1StepLA*. The results for the *Dueling DQN* look more stable, but both of them are quite good. At the end of the training, we keep the model that achieved the highest win rate. These best models will join the global competition in Chapter 5.



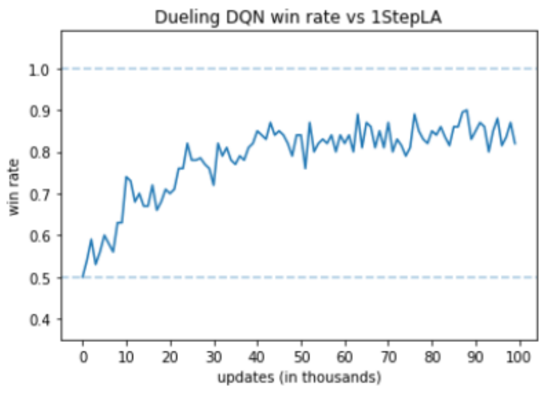


Figure 4.10. *Evolution of the win rate against 1StepLA for both Vanilla and Dueling DQN*

4.4. REINFORCE with baseline

In this section, we present our approach using REINFORCE with baseline [2.1.2] to learn to play Connect4. Honestly, the results we obtained were not as good as we expected. We tried to reduce the variance by using a baseline function and by fine-tuning the hyperparameters to take smaller update steps toward more stable directions. And yet, despite our efforts, the learning process was quite unstable and the maximum win rate against the *1StepLA* was 59% (+9% increase in win rate).

The network did not seem to learn new meaningful strategies of the game throughout the training, and the average win rate against the *1StepLA* was around 50% both at the beginning and at the end. However, we wanted to include these results to compare them with the ones obtained using *Proximal Policy Optimization* (in the next section [4.5]) so we could analyze the benefits that PPO offers to reduce the variance.

As happened in [4.3] with *Deep Q-Networks*, to train our agents using the REINFORCE algorithm we also have to adapt the original version to our particular environment. As we mentioned, Connect4 is a turn-based game, so when an agent is in a given state s, it must take into account that the next state s' (the next turn) is played by an opponent who is also trying to win the game (and beat the first agent). As shown in the expression below, we changed how the return is computed. Note that, since it is a turn-based game, the turn t is played by the active player but the opponent plays the turn at time $t+1$.

$$G_t = r_t - \gamma \cdot G_{t+1}$$

In order to add more stability to the predictions, we took advantage of the vertical symmetry of the board by taking the average of symmetric probabilities (as we did in [4.3] with the Q-values). For a given state (board) s, we compute the symmetric board s^{sym} by flipping horizontally (*fliplr*) the state s. Based on the rules of Connect4, these two boards are exactly the same, so their optimal policies are also the same (but flipped horizontally as well). By taking the average of these symmetric probabilities, we reduce the variance and avoid the overestimation bias that comes from using just one estimation. The following expression shows the computation. We use *fliplr* to refer to the horizontal flip operation (left/right direction). This technique is only used when playing games in competitions, not when training the network.

$$\pi^{avg}(s, :) = \frac{1}{2}\left[\pi(s, :) + fliplr\left(\pi(s^{sym}, :)\right)\right] \qquad where\ s^{sym} = fliplr(s)$$

In the next two sections, we present our solution using REINFORCE with baseline. The training steps and the choice of hyperparameters are explained in section [4.4.1]. As we mentioned, the results obtained were not as good as we expected. However, we decided to put them in section [4.4.2] in order to compare them with the Proximal Policy Optimization approach [4.5.2], which is the other policy-gradient algorithm tested in this project.

4.4.1. Training

REINFORCE with baseline uses the two-headed version of our *CNET128* (defined in [4.2.3]) which has 202k parameters (as shown in *Figure 4.7*). Before the training starts, the network inherits the weights of the pre-trained *CNET128* model [4.2.3] from the previous supervised learning task. So, in this section, the convolutional block (*feature extractor)* has 70k pre-trained parameters that are frozen and remain unchanged. Appended to the convolutional block is a sequence of fully connected layers that split into two prediction heads. In total, there are 132k trainable parameters in the fully connected layers: 17k for the policy head, 16k for the value head, and 99k that precede the prediction heads and are shared by both of them.

Apart from the *current network* that is trained, there is a second network called the old *network* that keeps the best weights found so far. When the performance of the *current network* decreases, the last updates are undone and it goes back to the *old weights*. We considered that the performance decreased when the win rate against the *1StepLA* was 8% lower than the best one achieved so far. Every time that the *current network* achieves a new best win rate against the *1StepLA,* the *old weights* are replaced with the *current weights* ($\theta_{old} \leftarrow \theta$).

Regarding the training data, the training episodes are generated using self-play, i.e. the *current network* plays against itself. We used a *buffer* to store transitions in the format $(s, a, r, done)$. At each iteration, the buffer is filled with a new self-play episode generated using the most recent weights of the network. Then, as explained in [3.3], the terminal rewards are backpropagated to give some credit to the intermediate actions, and this process is done within the buffer. Each transition is only used once to update the weights.

For each new episode that is added to the buffer, a symmetric version is created and used as well. In Connect4, the board has a vertical symmetry that can be exploited to double the number of data samples and also to help the network learn to take advantage of this symmetry. To create a symmetric version of a given transition $(s, a, r, done)$, the state s is flipped horizontally, and the new action becomes $\#columns - a$. The values r and $done$ remain unchanged.

The *exploration-exploitation* dilemma is addressed by learning a stochastic policy and sampling actions from it. However, in order to let the network visit a wider range of game boards, we decided that the initial board would be random. The common initial empty board was no longer the first environment state. Instead, we let the *Random Agent* use self-play to generate a sequence of non-terminal transitions so the *network* had to finish the game. The random moves were not added to the buffer, so in the end, the network was able to visit a richer variety of high-quality transitions.

Every 100 training episodes, the *current network* competes against the *Random Agent*, against the *old network*, and against the *1StepLA*. In each competition [5.1], 100 games are played and the win rates and the average game length are tracked. The win rate against the *Random Agent* is expected to be greater than 95% throughout the training. It serves as a sanity check to make sure that the performance is not decreasing significantly. The win rate against the *old network* is not really informative because the *old network* is constantly updated throughout the training, so this value is usually around 50%. In this self-competition, we were more interested in the average game length. Lastly, the win rate against the *1StepLA* is 50% before training the network, so the main objective is to exceed it.

To reduce the variance, we tried different small learning rates such as 5e-5, 1e-5, and 5e-6. We finally chose 5e-6, but our experiments showed that smaller learning rates did not prove to reduce the variance and achieve better results. We trained for 100k updates using the *Adam* algorithm to perform *stochastic gradient ascent*. The weight decay (L2 regularization) was 1e-3 and the discount factor was 0.95. The loss function for the value head was the *Smooth L1 Loss* (L^{value}), and the loss for the policy head (L^{policy}) was the one described in [2.1.2]. The objective function to *maximize* is a weighted combination of these two partial losses (with their proper signs). We tested different c_1 values but they made no significant difference in the results. In a general scenario, the recommended values for c_1 are in the interval [0.5, 1], so we chose 0.75.

$$L_t(\theta) = L_t^{policy}(\theta) - c1 \cdot L_t^{value}(\theta) \qquad where\ c1 = 0.75$$

4.4.2. Results

Our starting point was a pre-trained network that learned to predict the actions of *1StepLA* (a mid-level player) with ≈85% of accuracy. Unsurprisingly, the win rate of this pre-trained network against the *1StepLA* was ≈50%. Our will in this section was to further improve this network using REINFORCE with baseline. Our best solution got a 59% win rate against *1StepLA*. However, despite all our effort to reduce the variance and improve the learning process, the win rates fluctuated at each iteration and never seemed to start improving consistently.

In *Figure 4.11*, it can be seen that the training loss did not improve during the training. The network seemed to be always trying new strategies without favouring either of them. We guess that this indecisive behaviour comes from using self-play to generate training episodes. At each self-play game, the network applies its best strategy to play the role of both players. But the game always ends with a winner and a loser, and in self-play, the network is both of them. In other words, it is extremely difficult for the network to learn a winning strategy (i.e. a policy that guarantees a higher expected reward) because the self-play data shows that all policies led to the exact same number of wins as the number of losses. The same happens with the estimation of the value function (i.e. the baseline), the network is unable to learn which states are more valuable (i.e. more likely to lead to winning the game). Due to the high variance, the loss values that are shown here were smoothed using a moving average of 1000 points.

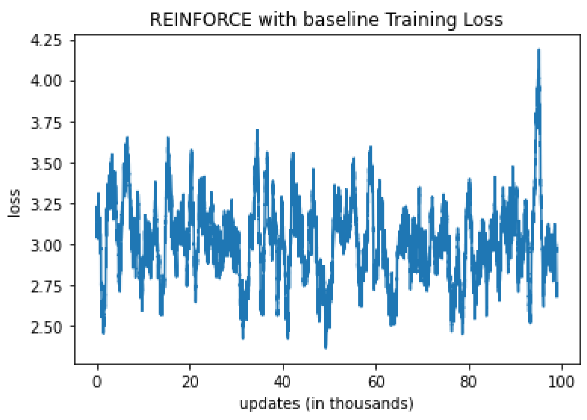


Figure 4.11. Training loss for REINFORCE with baseline

Every 100 training episodes, the network competes against its older version to see if the last updates make the performance increase or decrease. In this self-competition, the most important indicator is the average length of the games. The fact that the games last longer means that the network has more and more problems when it tries to win its older version: the network is learning to win games but also to prevent the opponent from winning, and this is very important to become a strong player. In the case of REINFORCE with baseline, the average game length did not increase, it remained below 15 turns. This means that the network was not acquiring more knowledge of the game since beating its older version was not getting more difficult.

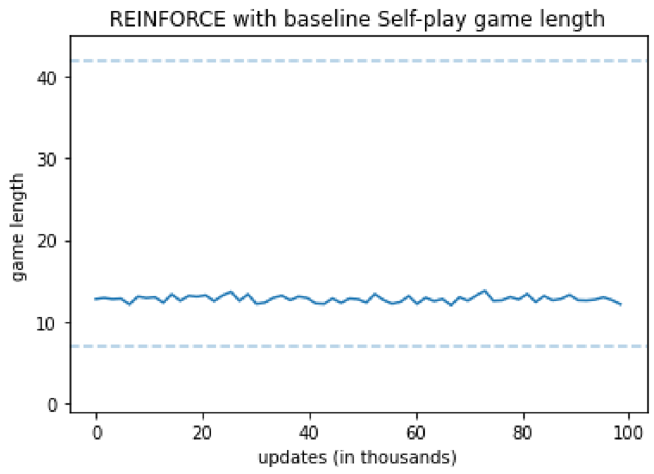


Figure 4.12. *Average game length in self-play games for REINFORCE with baseline.*

Finally, we show the evolution of the win rate against the *1StepLA*. The values range from 40% to 60%, and there is no clear evidence that the network is improving its performance as the learning progresses. In fact, the highest win rate is achieved within the first 12k training steps. At the end of the training, we keep the model that achieved the highest win rate of 59%. This model will join the global competition in Chapter 5.

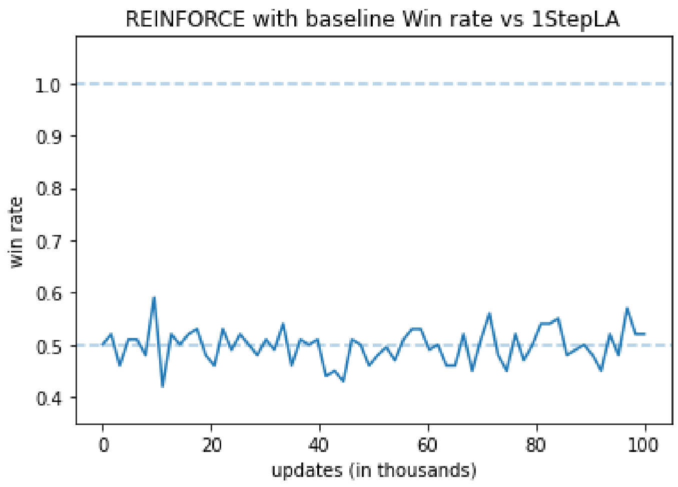


Figure 4.13. *Evolution of the win rate against 1StepLA for REINFORCE with baseline*

4.5. Proximal Policy Optimization

In the previous section [4.4], we presented the results we obtained using the REINFORCE algorithm with a baseline for variance reduction. Despite our efforts, the results showed no evidence of the network increasingly improving its knowledge of the game throughout the training. We achieved a maximum win rate of 59% against the *1StepLA*, but the average win rate was around 50% both at the beginning and at the end of the training. Proximal Policy Optimization [2.1.3] was created to reduce the high variance of some policy-gradient algorithms (e.g. REINFORCE) and to improve the data reusability by allowing the network to use each transition many times to update the weights. In this section, we will see if the novel ideas that PPO implements are useful to learn to play Connect 4 and hopefully get better results than the ones using REINFORCE.

As happened with other RL algorithms we tested, the original version of PPO was created to deal with the typical single-agent setting, and Connect4 is a two-player zero-sum game. To train our PPO agent, we first have to adapt the original algorithm to our particular environment. As we mentioned, Connect4 is a turn-based game, so when an agent is in a given state s, it must take into account that the next state s' (the next turn) is played by an opponent who is also trying to win the game (and beat the first agent). As shown in the expression below, we changed how the return is computed. Note that, since it is a turn-based game, the turn *t* is played by the active player but the opponent plays the turn at time t+1.

$$G_t = r_t - \gamma \cdot G_{t+1}$$

As we did in the previous section with the REINFORCE algorithm, in order to add more stability to the predictions, we took advantage of the vertical symmetry of the board by taking the average of symmetric probabilities. For a given state (board) s, we compute the symmetric board s^{sym} by flipping horizontally (*fliplr*) the state *s*. Based on the rules of Connect4, these two boards are exactly the same, so their optimal policies are also the same (but flipped horizontally as well). By taking the average of these symmetric probabilities, we reduce the variance and avoid the overestimation bias that comes from using just one estimation. The following expression shows the computation. We use *fliplr* to refer to the horizontal flip operation (left/right direction). This technique is only used when playing games in competitions, not when training the network.

$$\pi^{avg}(s, :) \;=\; \frac{1}{2}\Big[\pi(s, :) + fliplr\big(\pi(s^{sym}, :)\big)\Big] \qquad where\ s^{sym} = fliplr(s)$$

In the next two sections, we present our solution using the PPO algorithm. The training steps and the choice of hyperparameters are explained in section [4.5.1]. We also present the obtained results and our final solution in section [4.5.2]. Fortunately, the improvement that PPO implements helped the network to follow a more stable learning path than the REINFORCE algorithm with baseline.

4.5.1. Training

Proximal Policy Optimization uses the two-headed version of our *CNET128* (defined in [4.2.3]) which has 202k parameters (as shown in *Figure 4.7*). Before the training starts, the network inherits the weights of the pre-trained *CNET128* model [4.2.3] from the previous supervised learning task. So, in this section, the convolutional block (*feature extractor)* has 70k pre-trained parameters that are frozen and remain unchanged. Appended to the convolutional block is a sequence of fully connected layers that split into two prediction heads. In total, there are 132k trainable parameters in the fully connected layers: 17k for the policy head, 16k for the value head, and 99k that precede the prediction heads and are shared by both of them.

Apart from the *current network* that is trained, there is a second network called the *old network* that keeps the best weights found so far. When the performance of the *current network* decreases, the last updates are undone and it goes back to the *old weights*. We considered that the performance decreased when the win rate against the *1StepLA* was 8% lower than the best one achieved so far. Every time that the *current network* achieves a new best win rate against the *1StepLA,* the *old weights* are replaced with the *current weights* ($\theta_{old} \leftarrow \theta$).

Regarding the training data, the training episodes are generated using self-play, i.e. the *current network* plays against itself. As shown in *Figure 2.5* (pseudocode for the PPO algorithm), we used a *buffer* to store a set of 2000 transitions in the format $(s, a, r, s', log\_prob, done)$ from different episodes using the *current network*. In each transition, log_prob is the natural logarithm of the probability of choosing action a in the state s following the policy defined by the *current network*. When the buffer is full, we run 5 epochs over the entire buffer using a batch size of 32 samples. The update rule for the PPO algorithm is explained in [2.1.3]. After all the epochs are done, the buffer is discarded. As explained in [3.3], for each episode, the terminal rewards are backpropagated to give some credit to the intermediate actions, and this process is done within the buffer.

For each new episode that is added to the buffer, a symmetric version is created and added as well. In Connect4, the board has a vertical symmetry that can be exploited to double the number of data samples and also to help the network learn to take advantage of this symmetry. To create a symmetric version of a given transition $(s, a, r, s', log\_prob, done)$, the states s and s' are flipped horizontally, and the new action becomes $\#columns - a$. The log_prob is computed for the symmetric action in the symmetric state, and the values r and $done$ remain unchanged.

The *exploration-exploitation* dilemma is addressed by learning a stochastic policy and sampling actions from it. However, in order to let the network visit a wider range of game boards, we decided that the initial board would be random. The common initial empty board was no longer the first environment state. Instead, we let the *Random Agent* use self-play to generate a sequence of non-terminal transitions so the *current network* had to finish the game. The random moves were not added to the buffer, so in the end, the network was able to visit a richer variety of high-quality transitions.

Every time an old buffer is discarded and a new one is created, the *current network* competes against the *Random Agent*, against the *old network*, and against the *1StepLA*. In each competition [5.1], 100 games are played and the win rates and the average game length are tracked. The win rate against the *Random Agent* is expected to be greater than 95% throughout the training. It serves as a sanity check to make sure that the performance is not decreasing significantly. The win rate against the *old network* is not really informative because the *old network* is constantly updated throughout the training, so this value is usually around 50%. In this self-competition, we were more interested in the average game length. Lastly, the win rate against the *1StepLA* is 50% before training the network, so the main objective is to exceed it.

We trained for 100k updates using the *Adam* algorithm to perform *stochastic gradient ascent*. The weight decay (L2 regularization) was 5e-5, the learning rate was 1e-4, the clip parameter (ϵ) was 0.2, and the discount factor was 0.95. The loss function for the value head was the *Smooth L1 Loss* (L^{VF}), and the *clipped surrogate objective* for the policy head (L^{CLIP}) was the one described in [2.1.2]. The objective function to *maximize* is a weighted combination of these two partial losses (with their proper signs) and an entropy bonus to ensure sufficient exploration $S[\pi]$. We used c_1=0.75 because it was the value that we used with REINFORCE [4.4.2], and c_2=0.04. The final expression to maximize is as follows.

$$L_t^{CLIP+VF+S}(\theta) = E_t\left[L_t^{CLIP}(\theta) - 0.75 \cdot L_t^{VF}(\theta) + 0.04 \cdot S[\pi_\theta(s_t)]\right]$$

4.5.2. Results

Our starting point was a pre-trained network that learned to predict the actions of *1StepLA* (a mid-level player) with ≈85% of accuracy. Unsurprisingly, the win rate of this pre-trained network against the *1StepLA* was ≈50%. Our will in this section was to further improve this network using Proximal Policy Optimization. Our best solution achieved an 84% win rate against *1StepLA*, surpassing the 59% that we achieved using REINFORCE with baseline [4.4.2]. Despite being outperformed by both Deep Q-Networks [4.3], PPO managed to follow the smoothest learning path compared to all the Deep RL algorithms that we tested before.

In *Figure 4.14*, we observe a behaviour similar to that observed using REINFORCE (Figure 4.11): the training loss did not improve during the training (explained in more detail in [4.4.2]). The network seems to be always trying new strategies without favoring either of them. Because we use self-play to generate new data, all policies lead to the exact same number of wins as the number of losses, so the network seems to be unable to tell which policy guarantees a higher expected return. However, unlike the case of REINFORCE, here we observed that the win rate against the *1StepLA* constantly increased throughout the training. For visualization purposes, the loss values were smoothed using a moving average of 1000 points.

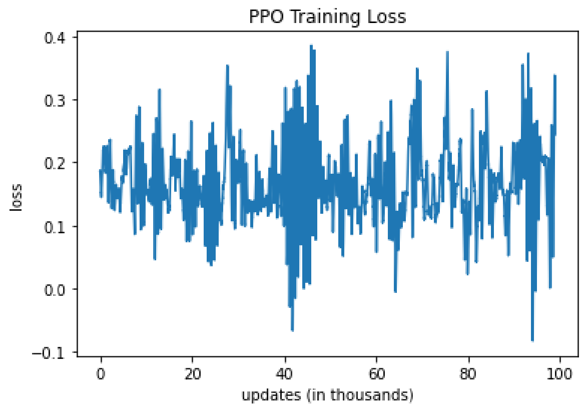


Figure 4.14. *Training loss for Proximal Policy Optimization Algorithm*

Every time an old buffer is discarded and a new one is created, the network competes against its older version to see if the latest updates make the performance increase or decrease. In this self-competition, the most important indicator is the average length of the games. The fact that the games last longer means that the network has more and more problems when it tries to win its older version: the network is learning to win games but also to prevent the opponent from winning, and this is very important to become a strong player. In the case of PPO, the average game length increased consistently up to 25 turns per game.

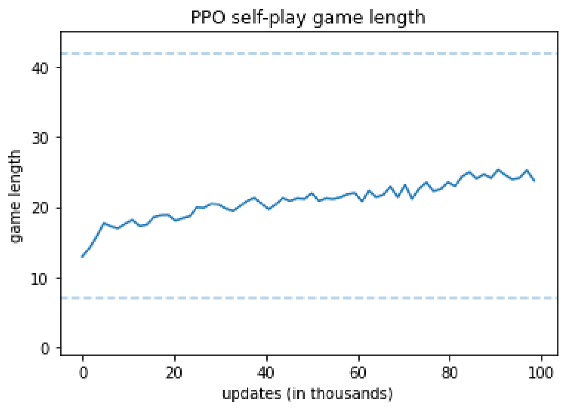


Figure 4.15. *Average game length in self-play games for Proximal Policy Optimization*

Finally, we show the evolution of the win rate against the *1StepLA*. Unlike REINFORCE (Figure 4.13), PPO managed to gradually increase its performance. At the end of the training, we keep the model that achieved the highest win rate of 84%. This model will join the global competition in Chapter 5.

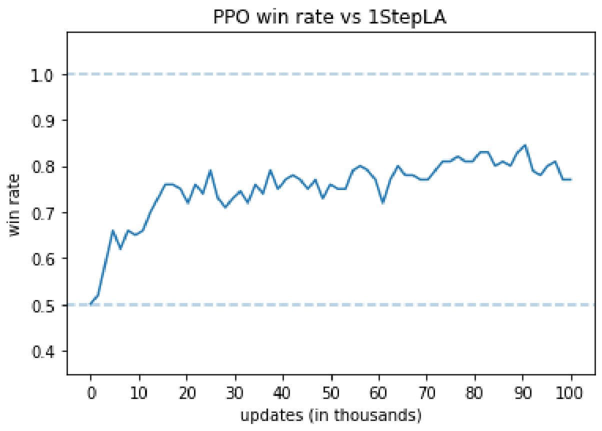


Figure 4.16. *Evolution of the win rate against 1StepLA for Proximal Policy Optimization*

5. Evaluation

In this chapter, we describe the evaluation process to measure the performance of the best agents trained in the previous chapter. We start by describing the competition system that we designed to evaluate the agents. This type of competition involves two agents and consists of 100 "*special*" games to test how the agents act in a wide range of game boards [5.1]. Then, we display the results (win rates and average game lengths) in different tables, and we compare the overall performance and skills of all the agents [5.2]. Finally, we introduce the User Interface that was designed to let the reader interact with all the agents created in this project and try to beat them [5.3].

5.1. Evaluation setup

One of the most effective ways to measure the performance of an RL agent is to run a set of episodes and average the received cumulative reward. To give meaning to this value, it must be compared with some baselines or previous values from the same agent to check whether it is better. However, in a competitive multi-agent setting we can't evaluate the performance of an agent alone. Instead, we have to measure their performance against other agents. To play Connect4, two players are needed.

We did a pairwise evaluation of the agents by making them compete against each other. For the competition between two players to be fair, each player must take the first turn half of the time. In Connect4, the first player can force a victory if it plays optimally. Even if it does not play optimally, the first player is always one piece ahead of the second player. For this reason, we track the win rates separately depending on which agent plays first.

In some cases, an agent may follow a deterministic (or close to deterministic) policy. For instance, those agents that learn the Q-values have a deterministic policy that consists of taking the action with the highest expected return. Apart from that, policy-based agents can also act greedily on their stochastic policies and derive a deterministic policy. If two deterministic agents (*Agent1* and *Agent2*) play many games starting on an empty board, they will always play the same turns, resulting in just two different games (when *Agent1* plays first, and when *Agent2* plays first). In other words, the results of the competition will be the same whether they play two games or ten thousand games.

One way to add more diversity to the competition is by forcing the agents to take random actions with some probability. However, this random element may have a huge impact in turns where the game outcome is at stake (i.e. one of the players could win the game in the current or the next turn), or when the competition is tight. The solution we propose is to add two random turns (one per agent) at the beginning of the game. Then, the agents will finish the games following their own policies. It means that they will have to adapt to the initial boards. In the next paragraph, we elaborate more on this idea and how to control these first two random turns to make the competition fair.

Since the classic Connect4 board has 7 columns, there are 7^2=49 combinations of the first two turns (one per agent) of the game. If we also count the empty board, there are 50 different combinations of zero or two turns. We did not consider one-turn initializations because they only involve the first player and we want both players to have the same initial conditions (in terms of randomness). Of course, even if the first two turns are different for each game, some of them might eventually lead to the same final board. But at least, the sequence of actions taken in each game will be unique. So, a competition between *Agent1* and *Agent2* consists of 100 games: 50 games in which *Agent1* plays first, and 50 games in which *Agent2* plays first. In each case, these 50 games will be initialized with one of the combinations of the first zero or two random turns, and *Agent1* and *Agent2* will play to finish each of the games.

In a competition, the win rate is the most important metric. It tells the percentage of wins in the competition. It is important to bear in mind that the win rate is not a general quality of an agent, since it heavily depends on the opponent. The formula we use to compute the win rates is as follows.

$$win\,rate \;=\; \frac{wins\; + 0.5 \cdot ties}{total\; games}$$

5.2. Competition results

Using the competition system described in [5.1], we evaluated our agents by making them compete against each other. We also included our baselines *1StepLA* and *2StepLA* in the competition. Since all our agents beat the *Random Agent* in almost every game, we did not include it. Versions of *NStepLA* with $N \geq 2$ were not included because the *2StepLA* is already an upper bound for the agents' performance.

First of all, we measured the average time that each type of agent needs to choose an action. The general *NStepLA* has to evaluate 7^N boards at each turn, whereas the trained agents have to preprocess the state and run a forward pass through the neural network (*CNET128* architecture*)*. In *Table 5.1*, we present our results. The exact times may differ depending on the hardware in which the experiment is run (in our case, we used a CPU AMD Ryzen 7 4800h). We measured the total time to play 400 turns and we took the average for each agent. In our experiment, the results show that the trained agents are *20x* faster than *1StepLA* and *70x* faster than *2StepLA*.

	Trained Agents	1StepLA	2StepLA
time [milliseconds]	1.683	34.012	117.437

Table 5.1. *Table of the average time that each type of agent needs to choose an action*

In turn-based games like Chess or Connect4, being the first player gives you an important advantage and increases your probability of winning the game. In the case of Connect4, the first player is always one piece ahead of the second player, so the probability of *Agent1* beating *Agent2* may heavily depend on who plays first. For this reason, we decided to present the win rates separately for each agent playing first. In other words, in a competition *Agent1-vs-Agent2*, we differentiate the games in which *Agent1* plays first and those in which *Agent2* plays first. From these partial win rates, we can compute the general win rates of the competition.

The table presented below (*Table 5.2*) contains the win rates of all the possible competitions between the agents introduced in this project. This table is intended to be read row by row because the win rates are from the perspective of the *row player*. The value of each cell is the win rate of the *row player* versus the *column player,* only considering those games where the *row player* plays first. Since the stochastic policies of *PPO* and *REINFORCE* ('*REI*') can be turned into deterministic policies (i.e. acting greedily instead of sampling), we added both types of policies as different agents. We use the abbreviation '*sto*' to refer to *stochastic* policies, and '*det*' to refer to *deterministic* policies. We use a green cell background to indicate the values that are greater than 0.5, and a red background otherwise.

row player win rates	**vs. 1StepLA**	**vs. 2StepLA**	**vs. DQN**	**vs. DuDQN**	**vs. PPOsto**	**vs. PPOdet**	**vs. REIsto**	**vs. REIdet**
1StepLA	- - -	0.23	0.20	0.12	0.26	0.26	0.56	0.58
2StepLA	0.79	- - -	0.75	0.49	0.70	0.79	0.94	0.82
DQN	0.96	0.44	- - -	0.18	0.52	0.54	0.86	0.86
DuDQN	1.00	0.48	0.62	- - -	0.64	0.68	0.88	0.96
PPOsto	0.82	0.36	0.48	0.44	- - -	0.86	0.82	0.86
PPOdet	0.86	0.40	0.44	0.40	0.82	- - -	0.94	0.68
REIsto	0.60	0.17	0.30	0.20	0.24	0.22	- - -	0.62
REIdet	0.66	0.16	0.24	0.20	0.26	0.34	0.62	- - -

Table 5.2. *Table of competition win rates.*

For instance, the first non-empty cell (first row, second column) is the win rate of *1StepLA* (*row player*) versus *2StepLA* (*column player)* in those games in which *1StepLA* plays first.

Since this value is 0.23, it means that the win rate of *2StepLA* when *1StepLA* plays first is *1-0.23=0.77*. Following the same example, if we look at the cell in the second row of the first column, we can see the same information but from the perspective of *2StepLA* and for those games in which *2StepLA* plays first (here, *2StepLA* is the *row player*). In this case, the win rate of *2StepLA* is 0.79 and the win rate of *1StepLA* is *1-0.79=0.21*. In this example, we would say that the expected result in a 100-game fair competition between *1StepLA* and *2StepLA* is 22 to 78 (the average of the partial win rates mentioned above).

The first two rows and columns are the win rates against the baselines. When our trained agents play first, all of them beat *1StepLA* (first column) but none of them beat *2StepLA* (second column). REINFORCE is the only trained agent that does not beat *1StepLA* when *1StepLA* plays first (first row). The best results against the baselines are achieved by *Dueling DQN*: a consistent 0.94 win rate against *1StepLA* and a 0.495 win rate against *2StepLA*. We could argue that *Dueling DQN* has the same level of the game as *2StepLA*.

In the case of REINFORCE and PPO, the results using *stochastic* policies do not look too different from the ones using *deterministic* policies. It might mean that the average entropy is low in both cases. Interestingly, the win rates of *PPOsto-vs-PPOdet* are 48-52 (in percentage), but when *PPOsto* plays first these values are 86-14 (*sto-det*), and when *PPOdet* plays first are 18-82 (*sto-det*). This is a perfect example of how the probability of winning the game heavily depends on who plays first. Something similar happens in the competition *REIsto-vs-REIdet.*

In *Table 5.3*, we gathered the results from *Table 5.2* to compute the general win rate for each agent and create a ranking with all of them. For each agent, its *average win rate* takes into account all the competitions of that agent.

ranking	**Agent**	**average win rate**
1	Dueling DQN	0.7307
2	2StepLA	0.7171
3	Vanilla DQN	0.5950
4	PPO (sto)	0.5857
5	PPO (det)	0.5607
6	REINFORCE (det)	0.2929
7	REINFORCE (sto)	0.2664
8	1stepLA	0.2514

Table 5.3. *Final ranking of the agents, based on their general win rate.*

On average, all our trained agents are better than *1StepLA*, and our *Dueling DQN* achieves a higher win rate than the *2StepLA*. It is interesting to see that *PPO* benefits more from the *stochastic* policy than the *deterministic* policy, but in the case of REINFORCE, the opposite happens. Among the different Reinforcement Learning algorithms tested in this project, *value-based* (*off-policy*) algorithms have proven to achieve better results than *policy-based* (*on-policy*) *algorithms*.

game length	**vs. 1StepLA**	**vs. 2StepLA**	**vs. DQN**	**vs. DuDQN**	**vs. PPOsto**	**vs. PPOdet**	**vs. REIsto**	**vs. REIdet**
1StepLA	12.5	26.1	17.6	17.6	15.9	16.0	12.2	12.7
2StepLA	23.1	39.0	31.2	31.2	31.2	30.3	19.8	23.1
DQN	17.2	34.2	27.4	29.0	27.7	25.3	17.0	17.0
DuDQN	15.2	32.1	26.0	29.8	23.0	23.8	15.8	16.2
PPOsto	15.9	28.2	27.1	21.3	18.3	25.7	14.8	15.1
PPOdet	15.9	28.2	25.6	21.2	24.1	19.3	16.0	15.5
REIsto	12.2	23.7	17.1	16.8	16.6	16.1	13.1	13.1
REIdet	12.5	23.3	19.0	18.0	16.6	16.5	12.6	12.7

Table 5.4. *Table of the average game length of each competition*

Finally, in *Table 5.4* we show the average game length of each competition. Here, we also included the self-competitions (values in the main descending diagonal). The table is read the same way as the table of win rates (*Table 5.2*). There are several values greater than 30, especially in those competitions that involve high-level players like Dueling DQN, Vanilla DQN, or 2StepLA. On the other hand, agents with a lower level of the game (e.g. REINFORCE and *1StepLA*) rarely exceed 20 turns per game.

5.3. User Interface

In the previous section, we measured the performance of the trained agents in an *agent-versus-agent* mode. Since the objective of this project was to compare how different Reinforcement Learning algorithms learn to play Connect4, sections [5.1] and [5.2] were enough to accomplish our goals. Even so, we wanted to go a step further and let the readers play against the agents. With this idea in mind, we implemented a simple User Interface (UI) to play Connect4 against the best agents we have seen within this project. In this section, we explain in detail the features of the UI, how to use it, and how to run the code.

The application was programmed using Python. In particular, we used a Python module called Pygame which allows us to create simple video games such as Connect4. The application runs on your local machine, so you must have the code of the entire project (agents, environment, models, etc) on your local computer. Apart from that, you have to run the application every time you want to play the game. The code created in this project can be found in a public GitHub repository following this link. The README file explains the steps to follow in order to set up the Python environment and run all the code.

Everything related to the User Interface is found in the *'src/game'* directory. Inside the *'src/game/game_config'*, there are the configuration files to customize the game as you want (every color, the screen size, the board size, etc). To play the game, do: (1) clone the repository (if not cloned yet) and activate the Python environment, (2) open a terminal, (3) navigate to the directory where the *'src'* directory of the project is located, (4) enter *python3 src/game/connect_game_main.py* or *python src/game/connect_game_main.py* in your terminal.

When you run the program, it shows an initial menu to choose your opponent (Figure 5.5). The list of opponents is fully customizable and can be easily modified in the file *'src/game/game_logic/opponents_list.py'*. Use your mouse to right-click on the colored box with the name of your preferred opponent.

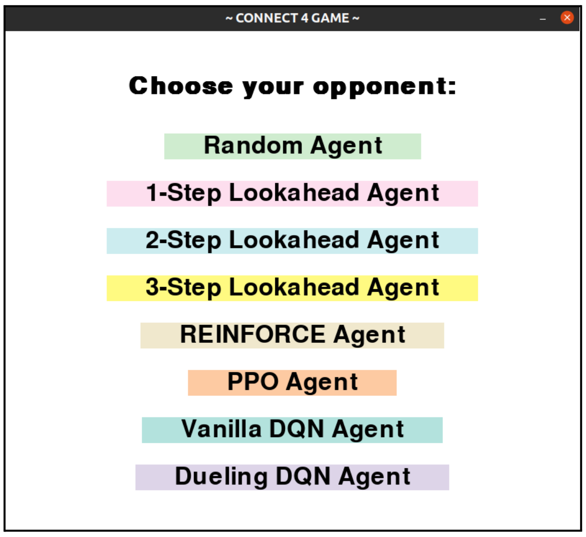


Figure 5.5. *User Interface. Game menu to choose your opponent.*

After choosing your opponents, the menu is closed and a new game board is created. The first player is chosen at random, so if your opponent does not drop one of their pieces, it means that you play first. Use your mouse to click on the columns where you want to drop your pieces. The information about the current state of the game (who is taking the current turn, who has won the game, etc) is found in the top-left corner of the screen. Click 'RESTART' (in the top-right corner) to start a new game at any moment.

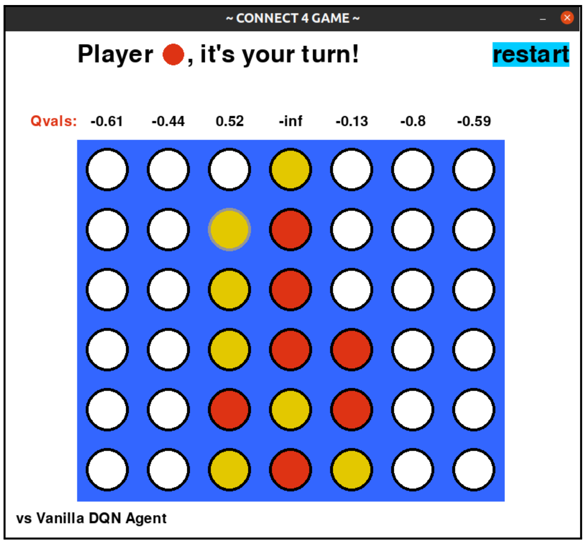

~ CONNECT 4 GAME ~
Player ●, it's your turn!
restart
Qvals: -0.61 -0.44 0.52 -inf -0.13 -0.8 -0.59
vs Vanilla DQN Agent

Figure 5.6. *User Interface. An example of an ongoing game against Vanilla DQN*

Above each column, there is a value that indicates the '*score*' that your opponent assigns to that column in the given board. These '*scores*' depend on the type of agent and define the logic that is followed to choose actions. For scored-based agents (N-Step Lookahead Agent), the '*scores*' are the scores computed using the minimax search and the evaluation function described in [4.1.2]. For value-based agents (Deep Q-Networks), the '*scores*' are the Q-values. For policy-based agents (REINFORCE, PPO) the 'scores' are the probabilities. These values are displayed at every turn, even for those turns that are not played by the agent. Of course, when it is your turn, you don't have to follow these *'scores'* since they are displayed only for evaluation purposes. You are free to follow your own strategy.

At any turn, it is possible to review past actions and game states, but you are not allowed to change them. Use your left arrow key to move one turn backward in time. You can visit all the past boards with their respective scores. Use your right arrow key to come back to the current turn and finish the game.

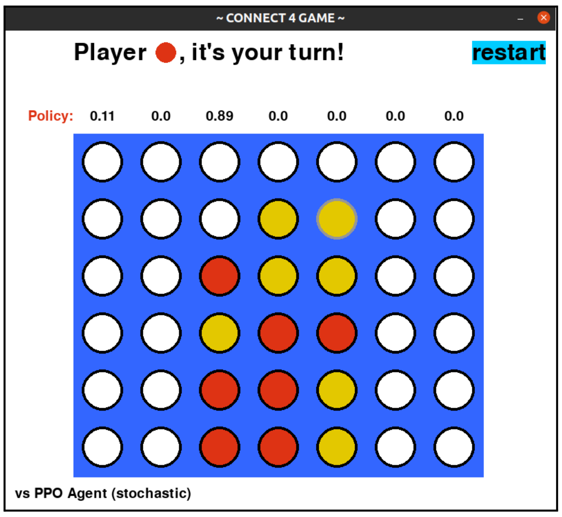

~ CONNECT 4 GAME ~
Player ●, it's your turn!
restart
Policy: 0.11 0.0 0.89 0.0 0.0 0.0 0.0
vs PPO Agent (stochastic)

Figure 5.7. *User Interface. An example of an ongoing game against PPO.*

6. Conclusions

In this last chapter, we present the overall conclusions of the project [6.1], the contribution we have made [6.2], and some suggestions for future work based on the work we started [6.3].

6.1. Conclusions

In this project, we tested two *off-policy* RL algorithms to learn the optimal Q-values (*Vanilla and Dueling DQNs* [4.3]*)*, and two *on-policy* algorithms to learn the optimal policy (*REINFORCE with baseline* [4.4] and *PPO* [4.5]). After training them, they competed against each other to check their performance and their knowledge of the game.

Our experimental results show that the two *off-policy* algorithms achieved better results than the two *on-policy* algorithms in the particular case of zero-sum games when the training data is generated using self-play. In sections [4.4.2] and [4.5.2] we presented the training results for *REINFORCE* and *PPO,* respectively. We plotted the evolution of the loss function throughout the training (Figure 4.11 and Figure 4.14) and there was no evidence that the network was learning anything. Then, we realized that the data was generated using self-play. Since Connect4 is a zero-sum game, there is always a winner and a loser, and the network is both of them in self-play games. It was impossible for the network to optimize the policy so the training data had more wins than losses.

On one hand, we could argue that using self-play in zero-sum games provides the network with a healthy balance of positive and negative transitions, so it does not get stuck in a local minimum. On the other hand, the learning becomes more difficult, and if the algorithm has high variance (e.g. REINFORCE), it might be unable to complete the training successfully. Despite the noisy losses, *PPO* managed to take more stable update steps and become a stronger player.

The value-based *off-policy* methods that we tested seem to be more robust to this kind of training data (i.e. zero-sum games using self-play). It might be because they learn the Q-values and derive a policy from them, rather than learning a policy directly to maximize the expected return.

In section [4.1.2], we defined the *N-Step Lookahead* baseline and we speculated that the level of an average human being is somewhere between a one-step and a two-step lookahead search. We guess that the level of an average human player would be the equivalent of performing a full one-step lookahead search and a partial two-step lookahead search focusing on the moves that lead to crucial game states. This estimation is not backed by any evidence, it is just an unofficial human-level approximation. Based on this assumption, we could arguably say that our *Dueling DQN* achieves average human-level performance in Connect4. The results of the competitions [5.2] show that our *Dueling DQN* achieves better results (i.e. higher win rates) than the *2-Step Lookahead* agent (which simulates and evaluates 49 future boards at each turn).

6.2. Contribution

The first part of this document provides a theoretical overview of some of the basic concepts in Single-Agent Reinforcement Learning, Multi-Agent Reinforcement Learning, and Deep Reinforcement Learning applied to two-player zero-sum games.

We also presented an implementation of Connect4 as a Reinforcement Learning environment, including the reward function that we designed for this game. Both the environment structure and the reward function could inspire other approaches to solve similar zero-sum board games.

We used different Deep Reinforcement Learning algorithms to train a set of agents to play Connect4. We made them compete against each other and analyzed their performance and their knowledge of the game. We finally drew some conclusions about how each algorithm performed in this particular zero-sum game [6.1].

Finally, we provide free access to all the Python code implemented in this project: scripts, algorithms, documentation, notebooks, and trained models. Moreover, we also provide the code to run the application that we designed to play against the trained agents (more information about the application is found in section [5.3]). To access the public repository refer to https://github.com/marcpaulo15/RL-connect4.

6.3. Future Work

The work we started can be extended and improved in several ways. There are other more advanced techniques to create stronger Connect4 players than the agents we trained here. In this last chapter, we explore some ways to improve the methods we used and the results we got.

The most advanced techniques to solve zero-sum board games involve a look-ahead search to simulate future games. This type of search is usually known as *Monte-Carlo Tree Search*.

However, for most games, exploring the entire game tree is unfeasible due to their complexity. To reduce the search space, the breadth and depth of the tree must be pruned. For the purpose of our project, the agents we implemented were not allowed to perform any look-ahead search. However, some of them could be used to guide a *Monte-Carlo Tree Search* (e.g. using a Deep Q-Network to evaluate the states, or a policy network to select actions). Even with a shallow search, our results could be remarkably improved. Of course, simulating future moves increases the computation required to take each turn.

Our training approach consists of two stages: a Supervised Learning task, and self-play Reinforcement Learning. It would be better to design an *end-to-end* training pipeline based solely on Reinforcement Learning and self-play. This approach would not depend on the particular game so it could be easily used for other zero-sum games. Moreover, the agents would not be limited to any human knowledge and would be able to potentially learn new strategies of the game at hand. However, it would probably require more computational resources because the training process might be slower and more unstable.

Bibliography

[1] Sebastian Raschka, Vahid Mirjalili. "*Python Machine Learning. Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*". Chapter 18, "*Reinforcement Learning for Decision Making in Complex Environments*". Third edition. December 2019. Published by Packt Publishing Ltd.

[2] C. J. C. H. Watkins. *"Learning from delayed rewards"*. Ph.D. dissertation. King's College. Cambridge. 1989.

[3] R. J. Williams, "*Simple statistical gradient-following algorithms for connectionist reinforcement learning*". Machine Learning, vol. 8, no. 3, pp. 229–256, May 1992.

[4] Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour. "*Policy Gradient Methods for Reinforcement Learning with Function Approximation*". AT&T Labs - Research, 180 Park Avenue, Florham Park, NJ 07932.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "*Proximal Policy Optimization Algorithms*". arXiv e-prints. p. arXiv:1707.06347. Jul. 2017.

[6] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel. "*Trust region policy optimization*". in Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ser. ICML'15. JMLR.org, 2015, pp. 1889–1897

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller. "*Playing Atari with Deep Reinforcement Learning"*. DeepMind Technologies. arXiv:1312.5602v1 [cs.LG] 19 Dec 2013

[8] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. "*Dueling Network Architectures for Deep Reinforcement Learning*". Google DeepMind, London, UK. arXiv:1511.06581v3 [cs.LG] 5 Apr 2016

[9] Kaiqing Zhang, Zhuoran Yang, Tamer Basar. "*Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms". arXiv:1911.10635v2 [cs.LG] 28 Apr 2021*

[10] Zero-sum game. (2023, May 28). In Wikipedia. https://en.wikipedia.org/wiki/Zero-sum_game

[11] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "*Mastering the game of go with deep neural networks and tree search*". Nature, vol. 529. no. 7587. pp. 484–489. Jan 2016.

[12] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. "*Mastering the game of go without human knowledge*". Nature. vol. 550. no. 7676. pp.354–359. Oct 2017.

[13] Connect Four. (2023, June 2). In *Wikipedia*. https://en.wikipedia.org/wiki/Connect_Four

[14] Kaggle. Intro to Game AI and Reinforcement Learning. accessed June 1st, 2023. https://www.kaggle.com/learn/intro-to-game-ai-and-reinforcement-learning

[15] Jianqing Fan, Zhaoran Wang, Yuchen Xie, Zhuoran Yang. "*A Theoretical Analysis of Deep Q-Learning*". arXiv:1901.00137v3 [cs.LG]. 24 Feb 2020.