Reinforcement Learning in Simple Games

Student: Ryazanov Stanislaw
Scientific Supervisor: Bruno Frederik Bauwens

Higher School of Economics
Faculty of Computer Science
Moscow, Russia
March 13, 2025

Contents

Annotation

This thesis focuses on discovering and implementing Reinforcement Learning algorithms. We will touch upon the essence of Reinforcement Learning, its applications in various fields of our lives, types of RL algorithms, their mathematical foundation, intricacies of their implementation and, of course, how they perform in a few simple games.
The project files can be accessed through this GitHub page.

> ***Remark***
> I did not invent any of the methods mentioned in this thesis. All algorithm descriptions are based on Richard S. Sutton and Andrew G. Barto "Reinforcement Learning: An Introduction"[10].

Keywords

Reinforcement Learning, Deep Learning, Data Science

1 Introduction

1.1 What is Reinforcement Learning?

Reinforcement Learning is a paradigm of machine learning, just as supervised and unsupervised learning are. In case of supervised learning, an algorithm learns patterns in the feature data to predict the value of a target(s) variable(s); in case of unsupervised learning, the algorithm searches for patterns in feature data to extract insights from unlabeled data; whereas in reinforcement learning, an algorithm (*agent*) is placed in a dynamic *environment* and is asked to take such *actions* that would maximize a *reward* signal that comes from the environment.

1.2 Applications of Reinforcement Learning

Reinforcement Learning is a great Swiss-knife tool for solving optimal control problems: RL algorithms learn to make sequential decisions to achieve long-term goals from experience gained by interacting with the environment. It has been used in a variety of ways, here are some of them:

- **Automobiles.** Among the first things that come to mind when talking about Artificial Intelligence are, of course, self-driving cars. Research teams at companies such as Yandex and Tesla use custom software to simulate the behavior of a car in its natural habitat - the road. The car is rewarded for long and safe trips and is punished for violating traffic rules.
- **Robotics.** Any industrial factory needs complex heavy machinery to perform its work; Reinforcement Learning helps robotized arms and production platforms adjust the positions of their components to better complete target actions and find flaws in produced goods.
- **Finance.** Firms such as IBM use Reinforcement Learning algorithms to better time High-Frequency Trading deals. On-line learning plays a huge role in making trading decisions, ever-changing financial and political playing fields require venture funds to be at the edge of the current trends.
- **Retail.** Marketplaces and online stores like Amazon exploit Reinforcement Learning algorithms' ability to predict next actions to create recommendation algorithms for their services: using large amounts of user data, such algorithms can suggest users products and services they might want to buy.
- **Logistics.** For businesses like USPS logistic optimization is the heart of proper operation. Reinforcement Learning algorithms are great at generalizing data and finding complex patterns, which is effectively used in optimization of delivery routes for logistic companies.

In Section 4 we will also take a look at how Reinforcement Learning evolved throughout the years.

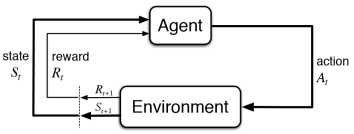
2 Reinforcement Learning Theory

2.1 The Agent-Environment Interface

To look deeper into the world of Reinforcement Learning, we first need to get the hang of its mathematical background. Let's take one more look at the definition we gave at the beginning of this thesis:

> In reinforcement learning, an algorithm (*agent*) is placed in a dynamic *environment* and is asked to take such *actions* that would maximize a *reward* signal that comes from the environment.

I propose we break that definition down. In context of Reinforcement Learning we are mainly talking about episodic tasks, that is each task consists of a number of states that the algorithm has to go through: from the initial start state and to the finishing state, each observed subsequently. Each state is associated with some information about it (the agent's surroundings), be that the card we get in a game of Blackjack or the charge level of a robot. Observing all available information, the algorithm must take an action, for example - hold our cards or send the robot back to its charging station. Based on the action taken, the environment will respond with the next state and, possibly, a reward. The reward may be positive in case the algorithm takes the right action (wins a game) and may be negative in case the algorithm makes a mistake (ignores low charge level and lets the robot perform an action which drains the battery completely).



2.2 Markov Decision Processes

2.2.1 State-action-reward Sequence

To convert the problem statement into something a machine can understand, *Markov Decision Processes* are used. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations, or states, and through those future rewards. They allow the program to break the problem down to a sequence of states, actions and rewards: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$ At each time step t the agent receives some representation of the environment's *state*, $S_t \in \mathcal{S}$, and selects an *action*, $A_t \in \mathcal{A}$, which leads it to the next time step $t+1$, where the agent receives a numerical *reward*, $R_{t+1} \in \mathcal{R}$, and the new *state* of the environment, $S_{t+1} \in \mathcal{S}$. This process repeats up to the terminal state of the process.

We will only be considering finite MDPs, that is MDPs that have a final state, where execution halts. In a finite MDP sets of possible states $\mathcal{S}$, actions $\mathcal{A}$ and rewards $\mathcal{R}$ have a finite number of elements, which allows random variables R_t and S_t to have well-defined discrete probability distributions dependent only on the preceding state and action:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-t} = a\}.$$

The function p defines the *dynamics* of the MDP. In a *Markov* Decision Process, the probabilities of S_t and R_t only depend on the immediately preceding state S_{t-1} and action A_{t-1}. We will be assuming the Markov property, that is each state includes all available information about the past agent-environment interaction. We can simplify the equation above to the following *state-transition probability* form:

$$p(s'|s, a) \doteq Pr\{S_t = s \mid S_{t-1} = s, A_{t-t} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a).$$

2.2.2 Goals, Rewards, Returns

As we have already mentioned, the goal of reinforcement learning algorithms is maximizing a *reward signal*. The rewards come from the environment, most often machine learning engineers must design the conditions on which an algorithm receives rewards. Let's take a look at a few examples:

- **Tic-Tac-Toe.** This is a pretty simple game and we may punish the algorithm with a negative reward when it loses, a smaller negative reward at a draw and a positive reward when the algorithm wins.
- **Walking robot.** Assume you are teaching a humanoid robot to walk. The robot starts at an upright position and is asked to move. Naturally, it is reasonable to positively reward the robot for the distance it was able to move itself.
- **Logistic management.** Assume an algorithm is asked to manage logistic routes. The algorithm shall receive negative reward at any congestion or delivery delays it caused.

The problem of reinforcement learning lies in the process of obtaining the highest *cumulative* reward, not the highest *immediate* reward. Yet not every problem could be split into episodes, like the *continuing tasks*: on-going process-control tasks go on continually, without limits. To unify the representation of episodic and continuing tasks, we introduce the absorbing state for the episodic tasks: this state only redirects the agent to itself with zero for the reward. To overcome the problem of infinite sums of rewards over time, we come up with a notion of *discounted return*:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ is the *discount rate*, $\gamma \in [0, 1]$. With $\gamma = 0$, the agent only maximizes the immediate reward, with greater γ values the agent becomes more and more farsighted. One convenient property of the discounted return formula is the following:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ... \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

2.3 Policies and Value Functions

2.3.1 Definitions

Basically, a *policy* π is a function that maps states to the probabilities of selecting each possible action. Reinforcement Learning algorithms define how the agent's policy must be changed as a result of its experience. For example: at time step t the probability of the agent selecting an action a following the policy π goes as follows:

$$\pi(a \mid s) = Pr\{A_t = a \mid S_t = s\}.$$

Value functions are mappings of *states* or *state-action pairs* to estimates of utility the agent gains by being in a given state or performing a given action in a given state. Different types of algorithms approach estimating the value functions differently, but generally the estimates are computed from expected returns G_t under policy π. Value function (*state-value function*) can be formally defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \text{ for all } s \in \mathcal{S},$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable given that the agent follows the policy π. The action-value function goes as follows:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right].$$

2.3.2 Optimality of Policies and Value Functions

How do we decide which policy is the best? To do that, we first need to define criteria that will help us compare policies. We call policy π' better than or equal to π only if $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$. For each fixed environment, there exists a policy that is better than any other, which is called the

optimal policy π_*. Although there may be multiple optimal policies, they share the same *optimal value functions*:

$$v_* (s) \doteq \max_\pi v_\pi (s), \text{ for all } s \in \mathcal{S}$$

$$q_* (s) \doteq \max_\pi q_\pi (s, a), \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}$$

2.4 The Bellman Equation

2.4.1 Definition

We are getting into the nitty-gritty of reinforcement learning. Let's combine formulas for state-value function and expected return:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}\left[G_t | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \sum_a \pi\left(a|s\right) \sum_{s'} \sum_r p\left(s', r|s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s'\right]\right] \\ &= \sum_a \pi\left(a|s\right) \sum_{s'} \sum_r p\left(s', r|s, a\right) \left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathcal{S} \end{aligned}$$

The last equation is known as the *Bellman Equation* for v_π, what it does is represent the relationship between the value of the current state and the values of its successor states. It weights all possible rewards by their probability of occurring. Value function v_π is the *unique solution* to its Bellman equation. The Bellman Equation for q_* takes the following form:

$$q_*(s, a) = \sum_{s'} \sum_r p\left(s', r|s, a\right) \left[r + \gamma \sum_{a \in \mathcal{A}} \pi\left(a'|s'\right) q_\pi\left(s', a'\right)\right]$$

2.4.2 Optimality

In solving reinforcement learning tasks, we are interested in value functions that yields the best results. Since there may be a number of optimal policies, we must define the equation for v_* (the optimal value function) that does not depend on any particular policy π:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}} q_{\pi_*}\left(s, a\right) \\ &= \max_a \mathbb{E}_{\pi_*}\left[G_t | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \\ &= \max_a \sum_{s'} \sum_r p\left(s', r|s, a\right) \left[r + \gamma v_*(s')\right]. \end{aligned}$$

Last two equations are known as the *Bellman optimality* equation for v_*. For q_* the Bellman optimality equation goes as follows:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*\left(S_{t+1}, a'\right) \middle| S_t = s, A_t = a\right] \\ &= \sum_{s'} \sum_r p\left(s', r|s, a\right) \left[r + \gamma \max_{a'} q_*\left(s', a'\right)\right]. \end{aligned}$$

For finite MDPs these equations have unique solutions.

2.5 Exploration-Exploitation Trade-off

One of the greatest problems reinforcement learning algorithms face is the exploration-exploitation trade-off.

Exploration refers to the ability of the algorithm to discover new features of the environment. It usually involves ditching greedy actions, which yield the highest expected reward, and trying other less explored actions and states to acquire more information. This way, an agent may uncover more promising strategies and avoid getting stuck in a suboptimal policy.

Exploitation on the other hand is defined by the agent's ability to make decisions aimed at maximizing the expected reward signal. Using previously encountered data the agent chooses the action that grants it the greatest future (long-term or short-term) reward.

The problems comes from the conflicting nature of those algorithms: exploration drives the agent to try new strategies, while exploitation demands the agent to stick to known high-reward strategies. To avoid that conflict, some randomness is added to the learning process, like the $\epsilon-$greedy strategies.

> ***Remark***
> ϵ-greedy policy π is a policy that makes the agent take the greedy action with probability $1-\epsilon$ and take a random action with probability ϵ

3 Principles of Reinforcement Learning

Okay, we have now covered required mathematical background for understanding how the practical part works theoretically. Before we get coding, let's go through key mechanisms by which reinforcement learning algorithms work.

3.1 Generalized Policy Iteration

The cornerstone idea of reinforcement learning is *Generalized Policy Iteration* (GPI), which consists of two parts: *Policy Evaluation* and *Policy Improvement*.

3.1.1 Policy Evaluation

Assume an arbitrary policy π. We can compare it to any other policy using value functions, which quantify the expected return an agent can expect when starting from a state s following the policy π.

Baseline method for estimating value functions is through *Iterative Policy Evaluation*. To better understand that algorithm, lets recall the *Bellman Equation*:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s') \right], \text{ for all } s \in \mathcal{S}.$$

What it does is, given a state, sums expected returns weighted by the probability of receiving those returns by following a given policy. This procedure can easily be represented as an iterative process over the states:

Algorithm 1: Iterative Policy Evaluation

Input: π, a policy to be evaluated
θ, a threshold parameter
Result: $V \approx v_\pi$

Initialize arbitrary $V(s)$ for $s \in \mathcal{S}$ and $V(terminal) = 0$;
while $\Delta < \theta$ **do**
  $\Delta \leftarrow 0$;
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V_\pi(s')]$;
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
  end
end

We need a threshold parameter θ which will define algorithm's tolerance to error. This detail is required because the optimal policy might take too much compute to be obtained.

3.1.2 Policy Improvement

Policy Improvement, on the other hand, is the process of changing a policy to suit the updated value function V_π. Let's look at the pseudo-code to better understand how it works:

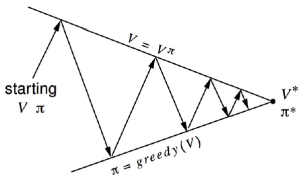
Algorithm 2: Policy Improvement

Input: $V_\pi(\cdot)$, value function of an evaluated policy
π, a policy to improve
Result: $\pi' > \pi$

policy_stable $\leftarrow$ *true*;
for each $s \in \mathcal{S}$ **do**
  old_action $\leftarrow \pi(s)$;
  $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V_\pi(s')]$;
  if *old_action* $\neq \pi(s)$ **then** *policy_stable* $\leftarrow$ *false*;
end
if *policy_stable* = *true* **then return** $V \approx v_*,\ \pi \approx \pi_*$;
else goto *policy_evaluation*();

This procedure yields us a policy that is compliant with the updated value function.

3.1.3 Generalized Policy Iteration



The *Generalized Policy Iteration* is the heart and soul of all reinforcement learning. It lets processes of Policy Evaluation and Policy Iteration interact, regardless of their granularity and any other details. The resulting algorithm is able to achieve a near-optimal policy for a wide range of problems. In the picture you can see how this process is commonly depicted: the algorithm starts with arbitrary policy and value function, then iteratively performs policy evaluation and policy improvement until the two processes converge. Policy evaluation makes the value function consistent with the current policy, while policy improvement makes the policy greedy with respect to the current value function. These two processes work in tandem, each driving the other towards optimality.

3.2 Specifications of Algorithms

Of course, there are many types of reinforcement learning algorithms, they all differ by a number of criteria. Before we start implementing algorithms in code, let's run through these criteria.

3.2.1 Discrete VS Continuous state space

First of all, as any other computer algorithm, reinforcement learning models work with numerical data. There are two types of numerical variables an algorithm can encounter: discrete and continuous. Depending on the type of variable, we might want to use different types of models. *Temporal-Difference* family of methods, for example, uses *Q-Tables* to find the optimal policy, which are basically $n-$dimensional tables, which store value function values for all possible states. Now assume the *Mountain Car*[5] task, in which the car exists in a continuous state-space 2D plane. It is possible to quantize continuous values into discrete via a step-parameter, but for more complex task it would be time-consuming and nearly impossible to set efficient step-parameters for each of the variables. Thus, for such tasks DQNs are more suitable, thanks to neural networks, which take the role of the *Q-functions*. On the other hand, tasks with discrete state spaces (like the blackjack game) are suitable for models of Temporal-Difference family, because the state space is discrete and reasonable sized.

3.2.2 Model-based VS Model-free algorithms

The other criteria which is important during choosing the model for solving a given task is choosing either a model-based or a model-free algorithm. Model-based algorithms try to learn the environment they operate in, that is they try to estimate the transition probabilities between states and the reward function. Model-free algorithms, on the other hand, do not estimate the environment model, but try to estimate the optimal policy or value function by trial and error.

Model-based algorithms are known for their *planning* ability: the strategies they use are based on looking several steps ahead. Examples of such models are Monte Carlo Tree Search, Dyna-Q, AlphaZero, etc. They are also more sample efficient than model-based algorithms, because they use simulations to predict environment's behavior, requiring less data to train on. Though, models of this type require much more compute power, due to the need of learning the policy *and* the environment model, and are prone to bias, which arises from model estimation.

Model-free algorithms show great performance in complex and dynamic environments, which model-based methods would fail to estimate. Model-free algorithms are less biased, as they build their policies and value-functions on real interactions with the environment. For the same reason, they require more computational power.

3.2.3 On-policy vs Off-policy algorithms

The last characteristic I would like to mention is whether an algorithm is *on-policy* or *off-policy*. For now, assume that any algorithm has two policies: a *behavior policy*, which dictates the algorithm how it should act, and the *evaluation policy*, which is actually evaluated and learned.

In **on-policy** methods the behavior and evaluation policies are the same thing, meaning the algorithm is directly improving the policy it is acting upon. To abide the *exploration-exploitation trade-off*, some degree of randomness is incorporated into the algorithms (e.g., $\epsilon-$greedy approach). This type of methods is more stable, yet also less sample efficient, requiring to sample new data under the new policy after each update.

In **off-policy** methods, the behavior and the evaluation policies are separate. That means, that the algorithm acts based on one policy and learns another one. This allows the agent to explore using an $\epsilon-$greedy policy, while evaluating the greedy policy. This type of algorithms is more sample-efficient, since it can learn on data generated by virtually any policy, and is better at avoiding local optima. However, off-policy methods can face convergence issues if the behavior and evaluation policies differ too much.

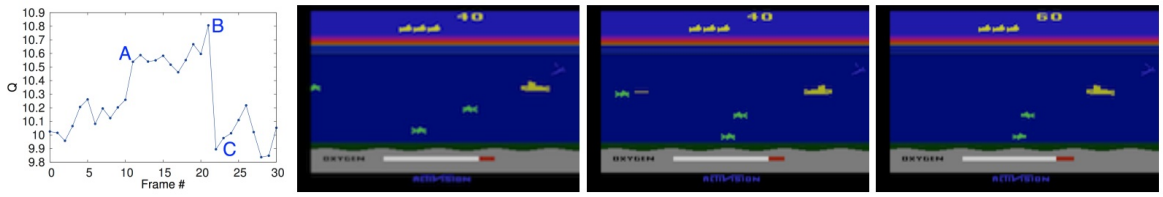
4 Origins of Reinforcement Learning

In 1957 Richard Bellman published his work "A Markovian Decision Process"[1]. MDPs provided a mathematical framework for modeling sequential decision-making in stochastic environments, representing a discrete and probabilistic counterpart to the continuous problems often considered in optimal control. Later that year, Bellman published his other cornerstone work "Dynamic Programming"[2], which provided a comprehensive theoretical foundation for optimization and control theory. Bellman's dynamic programming provides a crucial set of theoretical tools for reinforcement learning, particularly in the context of evaluating and improving policies through iterative algorithms grounded in the *Bellman equation*:

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

Dynamic programming has contributed to creation of the Temporal-Difference family of *model-free* algorithms, such as TD(0) and TD(λ) (created by Richard S. Sutton[9]), which bootstrap from the current estimate of *the value function*. Unlike existing Monte Carlo methods, which only adjust their estimates once the final outcome of an episode is known, TD methods have the capability to adjust predictions to match later, more accurate predictions about the future, even before the final outcome is known. TD learning was a huge leap forward in the field of Reinforcement Learning and paved its way for Q-learning and SARSA. Q-learning[11] is an *off-policy* TD control algorithm that aims to learn an optimal policy by estimating the optimal action-value function (the *Q-function*). The core of the Q-learning algorithm lies in its update rule, which is based on the *Bellman equation* and iteratively refines the Q-value for a state-action pair based on the immediate reward received and the maximum Q-value achievable in the next state. SARSA[6] is an *on-policy* TD control algorithm, which means it learns the Q-values associated with the policy that the agent is currently following. The update rule in SARSA considers the next action that will be taken by the agent according to its current policy, making it a more conservative learning approach compared to Q-learning.

The watershed moment for Reinforcement Learning was the "Playing Atari with Deep Reinforcement Learning"[4] paper by the Google DeepMind team in 2013. Researchers used a Convolutional Neural Network to extract features from the game screen (emulated Atari console games) and a Deep Q-Learning Network (a Q-learning algorithm with a neural network as a Q-function). Researchers found that DQNs were able to spot patterns in games and were able to successfully build winning strategies.



DeepMind's trained DQN's Q-function valued the frame with presence of a *vulnerable* target (the green shark-shaped figure left to the submarine on the frames A and B) greater than frames without such a target.

DQN have marked a pivotal moment in history of Artificial Intelligence, laying ground to new models and architectures like Double DQN, Policy Gradient methods and Actor-Critic approaches. In 2016 DeepMind team made headlines with their Alpha Go model, which has beat two of the most professional Go players on earth, Fan Hui and Lee Se dol (matches were held in October 2015, but the news were delayed until January 2016 to coincide with the Nature publication[12]). Since then, Alpha models were trained to successfully beat pro-players in Chess, StarCraft 2 and Dota 2.

During the same time *Policy Gradient* methods were making an appearance. They work by using gradient ascent to increase the probabilities of actions that lead to higher rewards and decrease of the actions that lead to lower rewards. Unlike all previously discussed methods, Policy Gradient methods directly optimize their policies instead of learning value functions to derive policies.

The breakthrough Policy Gradient algorithm, discovered in 2015, was *Trust Region Policy Optimization* (TRPO)[8]. It improved on the Vanilla Policy Gradient method, which was subject to large policy updates leading it to regions of parameters with significantly lower performance. TRPO is an on-policy policy gradient method relying on *trust regions* defined by the Kullback-Leiber divergence between old and updated policies. This characteristic allows the algorithm to perform stable updates on its policy. Additionally, the algorithm guarantees monotonic improvement in model performance.

Until this point in time Reinforcement Learning algorithms performed well in tasks involving making decisions in discrete action spaces, but no algorithm could make proper decisions in continuous control tasks. In late 2015 this changed when *Deep Deterministic Policy Gradient* (DDPG) method was introduced[3]. DDPG is a model-free off-policy algorithm following *actor-critic* architecture, where the actor learns the deterministic policy that maps states to actions and the critic learns to evaluate the utility of actions taken by the actor. DDPG is built on principles of Deterministic Policy Gradient (DPG) and Deep Q-Network methods.

Now the baseline Reinforcement Learning algorithm of choice is OpenAI's 2017 Proximal Policy Optimization (PPO)[7] algorithm. It is a model-free on-policy actor-critic algorithm, which addresses the complexity of implementation of the TRPO algorithm while retaining its stability and monotonic policy improvement. Till this day PPO algorithm is a state-of-the-art solution to optimal control tasks.

5 Reinforcement Learning Algorithms

Now, knowing the theory and basic principles behind reinforcement learning, its high time we put our knowledge to practice. We will start exploring the world of reinforcement learning algorithms starting with simpler Monte Carlo methods, then tackle TD methods, and, finally, try to recreate what DeepMind's team has done in their "Playing Atari with Deep Reinforcement Learning"[4] paper.

5.1 Learning Environments

Before we start discussion our models, we shall first get to know the environments the models will be learning in.

5.1.1 Blackjack

This is one of the most popular learning environments for testing simpler algorithms. This environment represents the card game of Blackjack, where the goal is to obtain such cards, so that their value sums up closer to 21 than dealer's cards. The value of the cards are the following:

- Cards 2 - 10 $\rightarrow$ value equal to their number
- Jack, Queen, King $\rightarrow$ 10 points
- Ace $\rightarrow$ either 11 ('usable ace') or 1

The game starts with the player and the dealer having two cards: player's cards are face up and the dealer's cards are face up and face down. All cards are drawn with replacement. The player can either hit (request additional cards), stick (stop hitting) or go bust (exceed 21). As the player sticks, dealer's face-down card is flipped and the dealer starts hitting cards until he reaches the score of 17 or goes bust. The winner is the player that did not go bust or whose score is closer to 21 than opponent's.

- **Observation space.** Is represented via players current sum (4-21), Dealer's face-up card (1-10) and whether the ace is usable or no (0/1).
- **Action space.** Two possible actions: sticking (0) and hitting (1).

- **Reward.** The player receives a reward of -1 on a lost game, a reward of $+1$ on a won game and a reward of 0 on a draw.

5.1.2 Frozen Lake

Frozen Lake is a 2D game where the player controls a character who has to come from the top left corner of the map to the bottom right corner while avoiding "frozen lakes". The environment is dynamic: there is a $\frac{2}{3}$ chance the character will slip and move perpendicular to the intended direction.

- **Observation space.** Current position on the map.
- **Action space.** Four possible actions: go up, go down, go left, go right.
- **Reward.** The player receives 1 point for each time the character gets to their destination.

5.1.3 Checkers

Checkers is a well-known board game, played on a chess board with pawns which move diagonally. The goal of the game is to capture all of the opponent's pawns. This environment was developed by me from scratch since there was no other environment like that to test my Monte Carlo Tree Search algorithms.

- **Observation space.** Whole game board, including all of the pawns positions.
- **Action space.** The set of all legal (possible and non-contradictory) moves the player can perform.
- **Reward.** The player receives 1 point for each won game, -1 for each lost game and 0 for each draw.

5.1.4 Atari Pong

> ***Remark***
> Atari 2600 is home video game console released in 1977. It featured the ability to play games off cartridges, allowing for third-party games being developed for the console. Games for this console kicked things off for DQNs in DeepMind's 2013 paper[4].

The goal of the game is to control the right paddle in a way to hit the approaching ball and outplay the opponent. The game is played until one of the players reaches 21 points

- **Observation space.** Greyscale images of the screen, with resolution of 210 by 160 pixels.
- **Action space.** Three possible actions: do nothing, move up and move down.
- **Reward.** The player obtains 1 point for each won round and loses 1 point for a lost round.

5.1.5 Atari Breakout

The goal of the game is to bounce the ball off the bottom of the screen for it to hit blocks at the top of the screen. The game ends once player misses four balls or all the blocks are destroyed.

- **Observation space.** Greyscale images of the screen, with resolution of 210 by 160 pixels.
- **Action space.** Three possible actions: do nothing, move left and move right.
- **Reward.** The player gets 1 point for each hit block.

5.1.6 Atari Space Invaders

The goal of the game is to control the spaceship at the bottom and fight back aliens at the top of the screen. The player has destructible shields that defend the player from the attacks of aliens. The game end when the player loses three spaceships or destroys all the aliens.

- **Observation space.** Greyscale images of the screen, with resolution of 210 by 160 pixels.
- **Action space.** Four possible actions: do nothing, move left, move right and fire.
- **Reward.** The player gets points for destroying aliens, each row of aliens has a constant score assigned to them (the higher the alien - the more it is worth).

5.2 Monte Carlo Methods

5.2.1 Mechanics

Monte Carlo methods work on the principle estimating the value function by averaging returns from complete episodes of agent-environment interactions, only updating the value function after the episode has been complete. This type of algorithm is model-free, meaning it learns from experience and does not require prior knowledge on the environment's dynamics. To maintain the balance between exploration and exploitation Monte Carlo algorithms use various strategies, most used being the $\epsilon-$soft policies.

5.2.2 Monte Carlo Exploring Starts algorithm

Let's see how a Monte Carlo Exploring Starts algorithms works to estimate the optimal policy:

Algorithm 3: Monte Carlo First Visit Exploring Starts algorithm

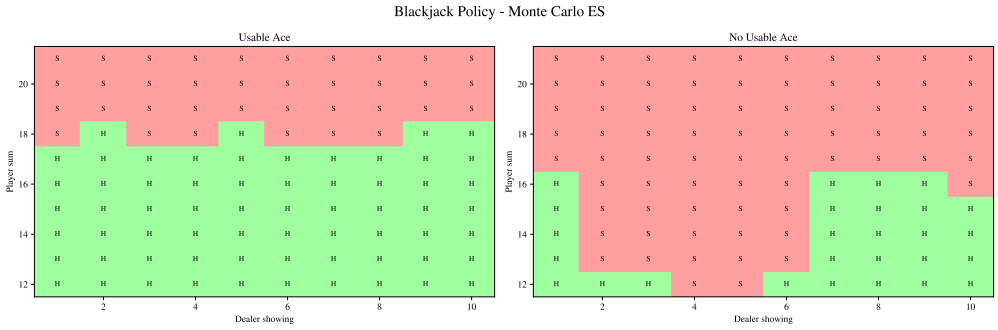
Input: arbitrary $\pi(s) \in \mathcal{A}(s)$, for all $s \in \mathcal{S}$
arbitrary $Q(s,a) \in \mathbb{R}$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Result: $\pi \approx \pi_*$

$returns(s,a) \leftarrow$ empty list;
while true **do**
  $(S_0, A_0) \leftarrow$ random (S, A), such that $S \in \mathcal{S}, A \in \mathcal{A}(S)$;
  Following π, generate an episode: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$;
  Return variable $G \leftarrow 0$;
  for $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$;
    if $(S_t, A_t) \notin ((S_0, A_0), (S_1, A_1), ..., (S_{t-1}, A_{t-1}))$ **then**
      Append G to $returns(S_t, A_t)$;
      $Q(S_t, A_t) \leftarrow mean(returns(S_t, A_t))$;
      $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$;
    end
  end
end

So, this is the *model-free on-policy* version of the Monte Carlo algorithm, meaning it is not given any prior knowledge on the behavior of the environment and acts with the same policy that it updates.

I propose we try to implement that algorithm in Python from scratch. You can see it on the GitHub page in the *"monte_carlo"* folder. It is exactly the algorithm we have discussed above, turned into a class representation of an agent.

To put the algorithm to test, I have tried to apply it on the Blackjack example from the Holy Bible of the field, the "Reinforcement Learning: An Introduction"[10] book by Richard Sutton and Andrew Barto [pp. 99-101]. Below you can see the result we get after letting the agent learn the game for 750.000 episodes with $\gamma = 1.0$ (meaning the reward is exactly the return). It is super close to the policy shown in the book, differing only by a few states.



Optimal policies learned by our Monte Carlo Exploring Starts agent for a game of Blackjack.

5.2.3 Constant-α Monte Carlo algorithm

Now, switching to the Constant-α Monte Carlo agent, let's see how it works:

Algorithm 4: Constant-α Monte Carlo algorithm

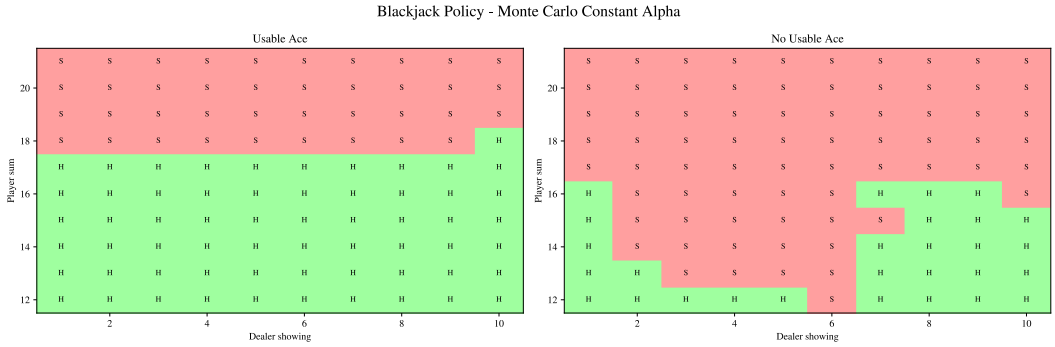
Input: α, step-size parameter
γ, discount factor
arbitrary ϵ-greedy $\pi(s) \in \mathcal{A}(s)$, for all $s \in \mathcal{S}$
arbitrary $Q(s,a) \in \mathbb{R}$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Result: $\pi \approx \pi_*$

$returns(s,a) \leftarrow$ empty list;
while true **do**
  Following π, generate an episode: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$;
  Return variable $G \leftarrow 0$;
  for $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$;
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G - Q(S_t, A_t))$;
  end
end

This is an incremental version of the Monte Carlo algorithm. The difference with the previous (First Visit Exploring Starts) algorithm lies in a few key parameters:

- Instead of using Exploring Starts, which limits adaptability to dynamic environments, Constant-α algorithm relies on ϵ-greedy policies to both explore new strategies and exploit well-rewarded strategies.
- To better suit stochastic environments, Constant-α algorithm uses a constant step-size α while updating its Q-values, which allows to prioritize more recent rewards.



Optimal policies learned by our Constant-α Monte Carlo Exploring Starts agent for a game of Blackjack.

5.2.4 Off-Policy Monte Carlo algorithm

Now we are going to look at the off-policy version of the Monte Carlo Algorithm. Remember, off-policy methods have two policies: a *behavior* policy, which tells the agent how to act, and a *target* policy, which is evaluated and improved. Also, this algorithm uses importance sampling - a method used for more precise expected value approximation, which considers the frequency of the agent's state visits:

Algorithm 5: Off-Policy Monte Carlo algorithm

Initialize: $Q(s,a) \in \mathbb{R}$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
$C(s,a) \leftarrow 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}$
$\pi(s) \leftarrow \text{argmax}_a Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$

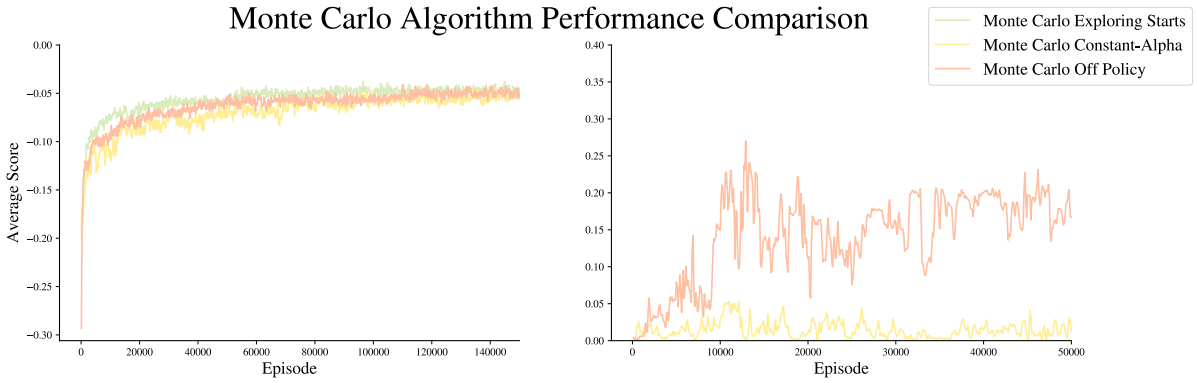
Result: $\pi \approx \pi_*$

while true **do**
  $\pi' \leftarrow \epsilon$-greedy policy Following π', generate an episode: $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$;
  Return variable $G \leftarrow 0$;
  Weight variable $W \leftarrow 1$;
  for $t = T-1, T-2, ..., 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$;
    $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$ $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} (G - Q(S_t, A_t))$;
    $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$;
    if $A_t \neq \pi'(S_t)$ **then**
      continue to next episode
    else
      $W \leftarrow W \frac{1}{b(A_t|S_t)}$
    end
  end
end

Due to the complexity of implementation, it is impossible to display the policy that the algorithm has came up with, but we will be comparing performance of algorithms in the following subsection.

5.2.5 Monte Carlo algorithm comparison

Okay, we have taken a look at how three different Monte Carlo algorithms function, now it's time we see how their performance compares. In the following test the algorithms play games of Blackjack (left) and Frozen Lake (right). The algorithms are evaluated after each 100 episodes of learning:



As we can see, all three models were able to converge to similar results for the game of Blackjack but they all had problems with the Frozen Lake game. Blackjack is a simpler static game with well-defined rules while in Frozen Lake agents face a dynamic environment which is hard to adapt to, so more complex models are better at the Frozen Lake game: Exploring Starts algorithm could not even get once to the end goal, while Off-Policy algorithm showed the best performance.

5.2.6 Monte Carlo Tree Search algorithm

Monte Carlo Tree Search (MCTS) algorithm is one of the main components behind AlphaGo[12], a Reinforcement Learning agent that has beat a 18-time world champion at a game of Go. This is a model-based off-policy *rollout* model, which means it is a decision-time planning algorithm.

5.2.6.1 Mechanics

What it does is simulate all possible trajectories of a set length from the current state in a Monte Carlo fashion to find the action that would lead to the state with the highest reward.
Each state of the simulated trajectory is represented as a node, which has:

- **Children.** A child node represents the state that comes after the current state in a trajectory. A node can have many children nodes, all corresponding to trajectory states 'accessible' from the current trajectory states by taking a legal (possible) action.
- **UCB1 Score.** To compare children nodes a UCB1 (also known as UCT) formula is used: $UCB1\left(v_i\right) = \frac{Q_i}{N_i} + C\sqrt{\frac{ln(N_{parent})}{N_i}}$, where Q_i is reward accumulated from node i, N_i is the number of visits to i, N_{parent} is the number of visits to the parent node and C is the exploration constant (usually set to $\sqrt{2}$). This formula allows the algorithm to favor both exploiting nodes with high rewards ($\frac{Q_i}{N_i}$) and exploring less visited nodes ($\frac{ln(N_{parent})}{N_i}$).

The whole algorithm can be broken up into four main subtasks:

1. **Selection.** At this step the algorithm traverses the nodes using a *tree-policy*, balancing exploration and exploitation. To do so, at each node the algorithm chooses the child node that has the highest UCB1 value. This step ends at a leaf node: a terminal node or a node with an *unexpanded* child node.
2. **Expansion.** During expansion the game tree is expanded at the node which has children that have not been visited yet. An unexpanded action is taken, taken randomly if several actions have not been taken yet.
3. **Simulation.** The value of the unexpanded node is estimated by simulating a random trajectory to a terminal state.
4. **Backup.** Once simulation is over (a terminal state has been reached), the attributes of the nodes along the traversal path are updated (number of visits, accumulated reward Q_i, etc).

This four steps are repeated over and over until a preset number of iterations has been achieved. Finally, the action that leads to the node (state) with the highest UCB1 value is taken.

5.2.6.2 Implementation

My implementation of the Monte Carlo Tree Search algorithm is located in the *"monte_carlo_tree_seacrh"* folder on the GitHub page. It contains implementations of Nodes and the MCTS model itself, the checkers environment (described above), an algorithm test function and a main loop. To simplify things just a little, the algorithm plays against an opponent who takes random actions.

The results are the following: of 1000 games **the MCTS algorithm won 981 games, played 19 games in a draw and lost 0 games**. The Agent was the first player to make a move for 500 games and the second player to make a move in the other 500 games. Of course, the games were played against a random opponent, but it is still a surprising result.

5.3 Deep Q-Network

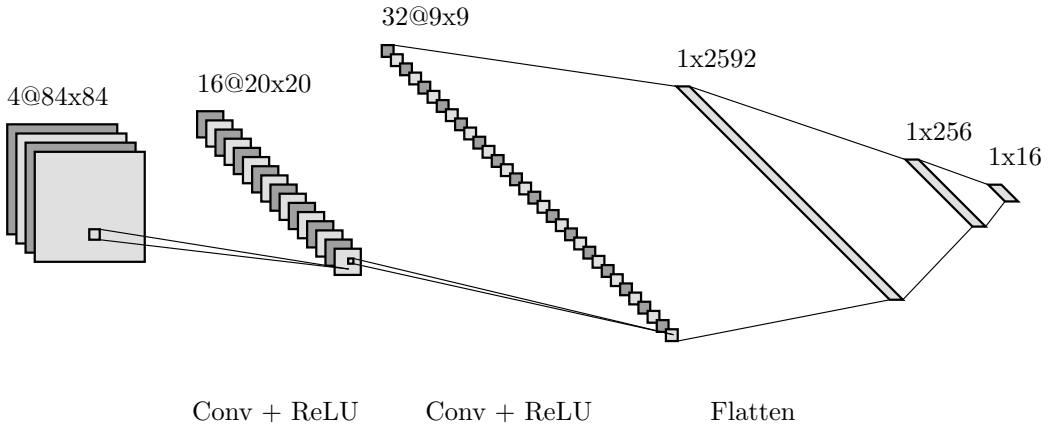
5.3.1 Mechanics

DQNs are one of the more advanced methods we are touching upon in this thesis. Created by Google DeepMind, DQN's goal was to overcome limitations of Q-Learning, mainly handling high-dimension state-spaces. In their 2013 paper "Playing Atari with Deep Reinforcement Learning"[4] DeepMind has taught the agent to play Atari games only through experience. They have introduced a few key concepts:

- **Action Replay.** This serves as a buffer of (s_t, a_t, r_t, s_{t+1}) tuples. Such buffer allows to break correlations between consecutive frames, allowing for better estimation of the optimal policy. Additionally it allows for bootstrapping samples, making the learning more efficient (requires less episodes to be played).
- **Q-Functions.** Q-Tables restrict RL models applications to small and discrete environments, but DeepMind has standardized using neural networks for learning Q-Functions, which allowed for better function approximation and generalization to states that have not been encountered by the agent yet. In addition to that, DQNs use two networks to learn: a policy network and a target network. Policy network is constantly updated, whilst the Target network is copied from the policy network once every N steps and is used for loss computation. This trick is used for avoiding divergence during the learning process and thus improving stability.

DeepMind has used the following architecture for their paper [4]:

1. A 210×160 128-color frame is taken from an Atari emulator, greyscaled and downsized to 110×84 pixels, which is then cropped to a 84×84 image. The processed frame is fed into a Action Replay (replay buffer), which has the capacity of 1 million frames.
2. When the replay buffer is filled, a series of four images is taken and passed through a CNN and a FCNN:
   - Convolutional Neural Network is comprised of two layers: one with a 8×8 kernel, 16 output channels and 4-pixel stride and the second one with a 4×4 kernel, 32 output channels and 2-pixel stride. Each layer has a ReLU activation function after it.
   - A Fully Connected Neural Network has three layers: a layer of 2592 neurons, which is a flattened output from the CNN, a layer of 256 neurons, a ReLU activation function, and a final layer of N neurons, where N is the number of valid actions for a given game.
   - The optimizer of choice was RMSprop with learning rate lr equal to 0.00025, smoothing constant ρ equal to 0.95 and the ϵ-parameter (added to the denominator for numerical stability) is set to 0.01.
3. The network has learned for 10 million frames. For the first million of frames the $\epsilon-$parameter of the DQN (the chance to make a random move) is linearly decreasing from 1.0 to 0.1, staying constant after that.



5.3.2 Implementation

My framework of choice was PyTorch - a well-documented and widely used deep learning framework, it has all necessary modules for creating neural networks, including regular layers, convolutional layers, optimizers, etc.

The Deep Q-Network implementation fully mimics such of DeepMind: two down-sampling convolutional layers with ReLUs followed by two fully-connected layers with ReLU in-between and outputs to all valid game actions. The *DQN* class also has functionality to save and load models.

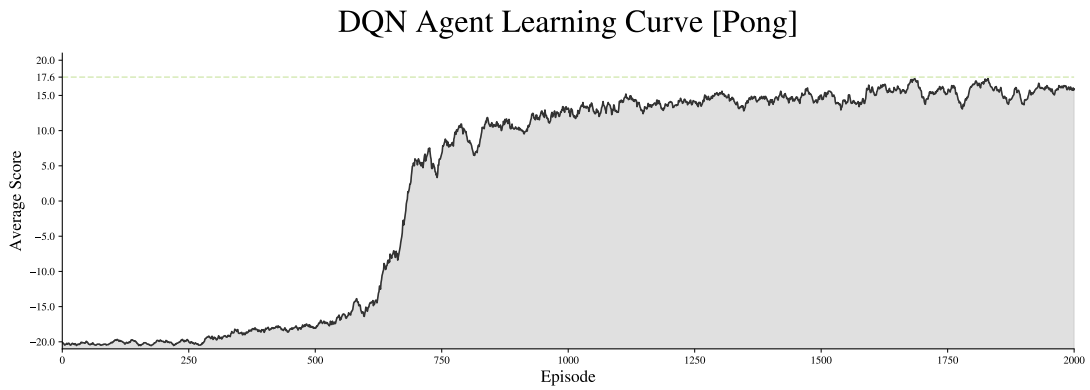
The Action Replay (named *ReplayBuffer* in code) is basically a deque data structure, with added functionality of sampling batches of data.

I have tested the model on three Arcade Learning Environment (ALE) environments: 'Pong-v4', 'Breakout-v4' and 'SpaceInvaders-v4'. Each environment comes through a preprocessing pipeline: the frames are greyscaled and scaled to 84×84 images, additionally the agent skips every four (three for Space Invaders due to rocket's flickering) frames for easier learning.

During the learning process the model uses the ϵ-greedy policy: for the first 1.000.000 frames of the game the ϵ is linearly descends from 1.0 to 0.1. At each decision-making time the model either performs a random action or follows its estimated policy.

In my implementation I have tried to follow the DeepMind's architecture as close as possible, the only differences in hyperparameters being a shorter size of Replay Buffer (100.000 last frames) and a shorter learning period (around 9 million frames or 4400 games).

Learning Pong took around 24 hours on my hardware (RTX 2070 Super, Ryzen 3900X and 32GB of 2400MHz RAM). Heres the learning curve of the first 2000 games of Pong:



Green line marks average performance of the best performing model saved in the *dqn_pong_best_model.pth* file.

The game of Pong is played until one of the players hits the score of 21, that's why the graph starts at around -21 and goes up to 21. Due to the learning process involving a ϵ-greedy policy, the average score during learning is actually lower than during inference without random actions. Overall average score the model could achieve in Pong is 20.42, indicating its great performance.

I wanted to replicate DeepMind's success at training *the same exact model* different games but, unfortunately, my model got stuck at suboptimal policies for games of Breakout and Space Invaders: it only got to average scores of 14.23 for Breakout and 555 for Space Invaders.

Snippets of Agent's gameplay can be seen on the .gif image in the readme part of the GitHub page.

6 Conclusion & Further Work

In this thesis we have explored basic and more advanced methods of Reinforcement Learning, including Monte Carlo algorithm and the Deep Q-Network method, by implementing them in Python. These models have shown positive results in a number of game environments: Blackjack, Checkers and Atari games such as Pong, Breakout and Space Invaders. Great results prove the ability of Reinforcement Learning algorithms to learn proper decision-making in dynamic environments.

Even though this thesis limits application of Reinforcement Learning algorithm to simple games, in real world this type of algorithms is widely used in the IT sphere for solving optimal control problems ranging from routing parcels and properly timing trading deals to adjusting parameters of industrial equipment and driving cars.

Happy with the results, I am interested in exploring other types of Reinforcement Learning algorithms such as Policy Gradient methods. A special interest field for me would be applications of Reinforcement Learning and Computer Vision in modern AI solutions like autonomous agents.

References

[1] Richard Bellman. "A Markovian Decision Process". In: Journal of Mathematics and Mechanics 6.5 (1957), pp. 679–684. URL: http://www.jstor.org/stable/24900506.

[2] Richard Bellman and Stuart Dreyfus. Dynamic Programming. Vol. 33. Princeton University Press, 2010. ISBN: 9780691146683. URL: http://www.jstor.org/stable/j.ctv1nxcw0f.

[3] Timothy P. Lillicrap et al. Continuous control with deep reinforcement learning. 2019. arXiv: 1509.02971 [cs.LG]. URL: https://arxiv.org/abs/1509.02971.

[4] Volodymyr Mnih et al. Playing Atari with Deep Reinforcement Learning. 2013. arXiv: 1312.5602 [cs.LG]. URL: https://arxiv.org/abs/1312.5602.

[5] Andrew W. Moore. "Efficient memory-based learning for robot control". In: 1990. URL: https://api.semanticscholar.org/CorpusID:60851166.

[6] G. Rummery and Mahesan Niranjan. "On-Line Q-Learning Using Connectionist Systems". In: Technical Report CUED/F-INFENG/TR 166 (1994). URL: https://www.researchgate.net/publication/2500611_On-Line_Q-Learning_Using_Connectionist_Systems.

[7] John Schulman et al. Proximal Policy Optimization Algorithms. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[8] John Schulman et al. Trust Region Policy Optimization. 2017. arXiv: 1502.05477 [cs.LG]. URL: https://arxiv.org/abs/1502.05477.

[9] Richard Sutton. "Learning to Predict by the Methods of Temporal Differences". In: Machine Learning 3 (1988), pp. 9–44. DOI: https://doi.org/10.1007/BF00115009.

[10] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[11] Christopher Watkins and Peter Dayan. "Technical Note: Q-Learning". In: Machine Learning 8 (1992), pp. 279–292. DOI: https://doi.org/10.1007/BF00992698.

[12] Wikipedia. AlphaGo. URL: https://en.wikipedia.org/wiki/AlphaGo.