INTERNATIONAL HELLENIC UNIVERSITY

DEPARTMENT OF INFORMATION AND ELECTRONIC ENGINEERING

Training an agent on playing a video game using Reinforcement Learning

Author:
Panagiotis Topalidis
Student id: 164756

Supervisor:
Dr. Konstantinos Diamantaras

Date 29/09/2020

Thesis Title: Training an agent on playing a video game using Reinforcement Learning

Thesis id: 20134

Student: Panagiotis Topalidis

Supervisor: Konstantinos Diamantaras

Assignment Date: 21/04/2020

Submission Date: 26/08/2020

I hereby declare that I am the author of this dissertation and that any help I received in order to prepare and complete it is properly acknowledged and cited. All the sources from which I used data, ideas, images and text are included, either the material was used as it is or paraphrased. In addition, I declare that the present assignment was entirely written by me as a dissertation for the Department of Information and Electronic Engineering of IHU.

The approval of the dissertation by the the Department of Information and Electronic Engineering of the International Hellenic University of Greece does not necessarily imply acceptance of the views of the author on behalf of the Department.

Τίτλος Π.Ε: Εκπαίδευση ευφυούς πράκτορα σε βιντεοπαιχνίδι με χρήση μάθησης με ενίσχυση

Κωδικός Π.Ε: 20134

Ονοματεπώνυμο φοιτητή: Παναγιώτης Τοπαλίδης

Ονοματεπώνυμο εισηγητή: Κωνσταντίνος Διαμαντάρας

Ημερομηνία ανάληψης Π.Ε: 21/04/2020

Ημερομηνία περάτωσης Π.Ε: 26/08/2020

Βεβαιώνω ότι είμαι ο συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω καταγράψει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, εικόνων και κειμένου, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επιπλέον, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά, ειδικά ως διπλωματική εργασία, στο Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του ΔΙ.ΠΑ.Ε.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Μηχανικών Πληροφορικής και Ηλεκτρονικών Συστημάτων του Διεθνούς Πανεπιστημίου της Ελλάδος, δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων του συγγραφέα, εκ μέρους του Τμήματος.

Abstract

Reinforcement Learning (RL) is the area of Machine Learning (ML) which mainly focuses on maximizing the total reward an agent can receive, based on a sequence of actions taken by the agent. Reinforcement Learning is considered as a standalone Machine Learning formula, differentiating itself from the two basic formulas of Supervised and Unsupervised Learning. This formula is widely used to train video game agents but also to solve real world problems, such as robotics or text mining. Even though Reinforcement Learning algorithms may seem very powerful and efficient there are also drawbacks that prevent them from being optimal for every problem's solution. For instance, a reinforcement learning agent is model free, meaning that it doesn't have any prior exposure to the environment it is trained on. Unlike Supervised Learning, an RL agent has to train itself through a trial-and-error process and figure out on its own the optimal next action that must be taken. By taking into consideration the aforementioned drawback, the required training hours are extremely long, with an extensively slow learning rate, in order to achieve an effective learning process.

Neural network architectures such as the Convolutional Neural Network, initially proposed by Dr.Fukushima in 1980, are highly beneficial for the learning process of the agent. In 2015, DeepMind introduced the Deep Q-Network through a paper on Deep Reinforcement Learning. By utilizing replaying, stacking the maximum frames,replacing the target network and normalizing the rewards, the agent's learning process can be achieved by "observing" raw pixels. This sort of technique tackles the instability problems encountered when using a non-linear approximator function, and allows the agent to adapt in any environment, ignoring any parameters that should have been taken into consideration if we were using another Reinforcement Learning algorithm.

In relation to the present thesis, I am implementing a Deep Q-Network using a virtual Reinforcement Learning environment named GYM, developed by OpenAI. GYM contains Reinforcement Learning tools, suitable for training an agent to play Atari games. Atari games are composed of a high dimensional state space, even though allowing me to train agents way faster than i would in a 3D game. By preprocessing the frames before stacking them, learning becomes a lot more lightweight and quick. Furthermore, I am using the GPU of Google Colab for faster training, and the Ubuntu(Linux) distro for demonstrating the output networks from Colab. The previously mentioned process is being done more than once, thus i am retrieving more than one different model networks. Those networks are being compared in efficiency, training time and the output agent's performance.

Περίληψη

Η Μάθηση με Ενίσχυση (Reinforcement Learning) είναι ένας κλάδος της Μηχανικής Μάθησης όπου στοχεύει στην μεγιστοποίηση της συνολικής ανταμοιβής που μπορεί ένας πράκτορας να λάβει, μετά από ένα σύνολο από ενέργειες του πράκτορα. Η Μάθηση με Ενίσχυση θεωρείται ως μία μεμονωμένη προσέγγιση της Μηχανικής Μάθησης, διαφορετική από αυτές των Μάθησης με Επίβλεψη και Μάθησης χωρίς Επίβλεψη. Η συγκεκριμένη προσέγγιση χρησιμοποιείται ευρέως για εκπαίδευση πρακτόρων βιντεοπαιχνιδιών αλλά και για την επίλυση προβλημάτων πραγματικού κόσμου, όπως την ρομποτική ή εξόρυξη γραφής. Παρ' όλο που οι αλγόριθμοι Μάθησης με Ενίσχυση δείχνουν να είναι πολύ ισχυροί και αποτελεσματικοί, υπάρχουν μειονεκτήματα που δεν τους καθιστούν ικανούς για να λύσουν οποιοδήποτε πρόβλημα. Για παράδειγμα, ένας πράκτορας Μάθησης με Ενίσχυση δεν βασίζεται πάνω σε κάποιο μοντέλο, το οποίο σημαίνει πως ο πράκτορας εκτίθεται για πρώτη φορά στο περιβάλλον που εκπαιδεύεται. Σε αντίθεση με την Μάθηση με Επίβλεψη, ένας πράκτορας Μάθησης με Ενίσχυση πρέπει να εκπαιδευτεί μέσω μίας διαδικασίας δοκιμής και σφάλματος ώστε να ανακαλύψει μόνος του την επόμενη βέλτιστη κίνηση που πρέπει να διαλέξει. Λαμβάνοντας υπ' όψιν το προαναφερθέν μειονέκτημα, ο χρόνος εκπαίδευσης είναι μεγάλος, με αργή ταχύτητα μάθησης, ώστε να υπάρχει μία αποτελεσματική διαδικασία μάθησης.

Αρχιτεκτονικές νευρωνικών δικτύων όπως αυτή του Συνελικτικού Νευρωνικού Δικτύου, όπως αρχικά προτάθηκε από τον καθηγητή Φουκουσίμα το 1980, είναι εξαιρετηκά ευεργετικά για την διαδικασία μάθησης του πράκτορα. Το 2015, η DeepMind εισήγαγε το Deep Q-Network μέσα από μία δημοσίευση σχετικά με την Μάθηση με Ενίσχυση. Χρησιμοποιώντας επανάληψη, σωρεύοντας τα μέγιστα στιγμιότυπα, αντικαθιστώντας το στοχευμένο δίκτυο και κανονικοποιώντας της ανταμοιβές, η διαδικασία μάθησης του πράκτορα μπορεί να επιτευχθεί ΄παρατηρώντας΄ ακατέργαστα εικονοκύτταρα. Αυτή η τεχνική αποφεύγει προβλήματα αστάθειας τα οποία μπορεί να εμφανιστούν εάν χρησιμοποιούνταν μία συνάρτηση μη γραμμικού προσεγγιστή, και επιτρέπει στον πράκτορα να προσαρμοστεί σε οποιοδήποτε περιβάλλον, αγνοώντας παραμέτρους που θα έπρεπε να ληφθούν υπ' όψιν στην περίπτωση χρήσης ενός άλλου αλγορίθμου Μάθησης με Ενίσχυση.

Σχετικά με την πτυχιακή εργασία, υλοποιώ ένα Deep Q-Network σε ένα εικονικό περιβάλλον Μάθησης με Ενίσχυση που ονομάζεται GYM, ανεπτυγμένο από την OpenAI. Το GYM αποτελείται από εργαλεία Μάθησης με Ενίσχυση κατάλληλα για την εκπαίδευση ενός πράκτορα ώστε να παίξει παιχνίδια του Atari. Τα παιχνιίδια του Atari αποτελούνται από ένα μεγάλο εύρος καταστάσεων, αλλά παρ' όλ' αυτά μου επιτρέπεται να εκαπαιδεύσω έναν πράκτορα γρηγορότερα απ' ότι σε ένα τρισδιάστατο παιχνίδι. Προεπεξεργάζοντας τα στιγμιότυπα του παιχνιδιού πριν μπουν σε μία στοίβα, η μάθηση γίνεται αρκετά ελαφριά και γρήγορη. Επιπλέον, χρησιμοποιώ την κάρτα γραφικών του Google Colab για γρηγορότερη εκπαίδευση, και το λειτουργικό σύστημα Ubutnu (Linux) για την επίδειξη των δικτύων που παρήχθησαν από το Colab. Η προαναφερθείσα διαδικασία γίνεται περισσότερες από μία φορά, οπότε δημιουργούνται περισσότερα από ένα δίκτυα. Αυτά τα δίκτυα εν τέλει , συγκρίνονται ως προς την αποτελεσματικότητα, τον χρόνο εκπαίδευσης και την αποδοτικότητα του πράκτορα.

Acknowledgements

I would like to thank my teacher Konstantinos Diamantaras and my fellow students for introducing me to the field of deep learning and sequently to Reinforcement Learning and of course, my family and relatives for their continuous support and patience all over my studies journey up until now, as well as in the near future.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

The following, is the introduction to my thesis. The introduction includes the motivation for writing the thesis, and well as its objective, the process and the results and of course the structure.

1.1 Motivation

If we look back in the history of humanity, it is clear to state that every creature is striving to survive. Even though such an example is very generic, surviving requires actions that must be taken. Therefore, those actions are based on what we, as humans, or the animals know. Now, knowledge comes from learning, which is the main objective of the thesis. Learning could be achieved through observation of the current state, an action that could be taken and last but not least, by a trial-and-error process. Reinforcement learning is the area of Machine Learning which mainly focuses on how a machine can learn by observing the current state and taking the proper actions, like a living creature does.

Reinforcement Learning is about training agents that could learn through a trial-and-error process and maximize their performance over the training time. By utilizing an algorithm, an agent learns by the outcome of its actions so far. Therefore, there is no need for any specific commands to be given to the agents, since they will go through that trial-and-error process. Despite the fact that agents can learn solely on their own, a policy representation or a value function could be utilized. Those options would help the agent in taking the right actions at a specific state. In the recent years, the revolutionary discovery of the deep neural networks, or in other words Deep Learning, became a huge advantage to that time's state of the art of Reinforcement Learning. Deep Learning also proved to be extremely beneficial for many areas, such as Computer Vision, Natural Language Processing or Speech Recognition. Deep Learning brought Reinforcement Learning new challenges that were out of scope in the past.

Like every technique, Reinforcement Learning is not absolutely stable. Q-learning is one of the main Reinforcement Learning techniques, which emphasizes on instructing the agent to take the best action based on his previous actions and the rewards the agent received. Q-learning's target variable constantly changes through the process, but Deep Learning's target variable remains the same during the whole training process. In other words, assuming we are playing a video game, our goal is constantly changing due to our previous

actions and the current state we are observing, allowing us to safely assume that the initial policy must also change, in order to achieve better results in the future.

A solution that was proposed in 2015 by DeepMind, proved to be highly efficient. The paper named Human-level control through deep reinforcement learning, demonstrates a Deep Q-Network that is applied in Atari 2600 games. The outcome proved that a Deep Q-Network performs better, compared to previous algorithms, since the agent could perform as good as a human player would. Several more researches were done, while some of them are still in progress. A notable mention is OpenAI's Five, an agent that learnt how to play the video game Dota 2, by playing against itself. Five could defeat professional human players, after a specific time of training. In the current thesis, I am applying a Deep Q-Network on the Atari game named Pong, and comparing its performance between different approaches of the network's model. The agent is learning to play against the game's default AI.

Figure 1.1 Reinforcement Learning observation depiction

The figure above depicts a state of the game Dark souls 3. Very generally, this state includes the Environment, which is the environment the agent is in, the Agent, which is the player the human usually controls and the Reward, which is the goal of the game, in this case defeating the opponent. The Agent is free to take any permitted action against the opponent. Even though for every good action the agent takes there is a reward given to him, the biggest reward will be given by defeating the opponent.

1.2 Thesis Objective

Applying the Deep Q-Network to an Atari game named Pong. Training and comparing different approaches of the network model considering the training time and the performance of the agent.

The proper way of training for such an agent is the Deep Q-Network. Due to the high dimensional state space, the Deep Q-Network is a good option to handle such a situation. Raw Q-Learning would lead to an extremely huge Q-table that would not be not possible to utilize for the training of the agent. Assuming we have a sum of 10,000 states and a sum of 1,000 actions per state, makes up a 10 million-cell table, therefore relying on raw Q-Learning is catastrophically wrong. The Markov Decision Process could prove to be useful at some point. In the Markov Decision process each state within an environment is a consequence or its previous state, in other words a result of the previous state. Albeit, keeping a storage of all that information, would become readily infeasible. Therefore, Deep-Q-Learning becomes my proper solution, by solving the following problems of Q-Learning:

- The amount of memory required to update the table would increase as the states increase, and that's where the neural networks come in useful,
- The amount of time required to explore states would be tremendously huge

1.3 Thesis Process and Results

I am building the Deep Q-Network using Python as the scripting language and PyTorch as a machine learning library for the convolutional neural network. The source code was written and tested on Ubuntu(Linux) distro and the agent is trained on Google Colab without rendering the environment. The demonstration of the model used for the performance of the agent is done on Ubuntu as well. The main network consists of 3 hidden convolutional layers and 2 fully connected with the last one outputting the actions the agent must take. Even though the DeepMind's paper instructs to use 4 hidden layers, I am training the agents on networks of 3 or 2 hidden layers alternatively too, in order to make a training and performance comparison. Furthermore, I am using GYM, a Reinforcement Learning environment for Atari games, which provides functions to return the state, observation, reward and done. By utilizing the Gym Wrappers I am adjusting and adding functionalities to the environment such as handling a certain amount of frames, resetting the stack or preprocessing those frames to tackle memory overflow. During the training, the DQN has to choose the max frame between the last 4 frames. I am using a prediction and a target network to estimate the target. As I mentioned before, this solves the target problem that diverges Q-Learning and Deep Learning.

After a certain training time, on a certain number of episodes the agent is trained and ready to perform. The agent learns how to play against the default AI of pong. The performance of the model is done on Ubuntu where the agent plays against the default AI and wins against it. Furthermore, I am comparing different structures of the model considering the training and the performance of the agent.

1.4 Thesis Structure

In the current Chapter(first) of the thesis I am introducing the objective of the thesis and the brief description of the implementation I did. The Chapter 2 includes the background of the thesis, explaining any tools and architectures used for the implementation. The Chapter 3 analyzes the problem. The Chapter 4 analyzes the structure of the Deep Q-Network and how it actually works. In Chapter 5, I am extensively explaining the architecture of the implementation of Deep Q-Network, how the training was done and the results. Chapter 6 contains the appendix, some different frameworks and libraries, that could implement a different approach to the problem of the thesis. Finally, Chapter 7 explains the conclusions deduced upon reaching the end of the thesis, where I am briefly explaining the architecture, the Reinforcement Learning's State of the Art and the Future Potential of it.

CHAPTER 2

Background

First and foremost, the following chapter analyzes the basic concepts of Reinforcement Learning and Deep Convolutional Neural Networks, as well as the Deep Q-Network. Deep Q-Network consists of those two concepts combined. Therefore, I believe it is highly essential to dive deeper into the aforementioned terms as they are going to make the reader feel more familiar to the objective of this thesis. By providing their structure, but also the advantages and drawbacks of the basic concepts of Reinforcement Learning, the need that led DeepMind to create the Deep Q-Network becomes all the more clearer to the reader.

2.1 Q-Learning

Beginning with Q-Learning[1], it is a model-free algorithm which is used for the agent to learn what action must be taken during the current state the agent is in. Q-Learning is characterized as model-free because the prior exposure to the environment the agent is being trained in is needless. Consequently, it handles the problems with stochastic transitions and rewards. Q-Learning initializes a Q-Table which consists of Q-Values. Q-Learning estimates the best Q-values of an MDP(Markov Decision Process). Every action taken is heavily affected by the Q-Values the agent has learnt so far. As for the very first actions, the agent is required to take random actions and estimate the starting Q-Values by rewarding it for every successful action taken. An action is chosen by using the greedy policy by using **ε** (epsilon), which is a parameter between 0 and 1. In the next paragraph, I will be explaining further the ε hyperparameter With respect to the policy, the agent randomly selects one of the possible actions that can be taken. Accordingly, the Q-Value for this action is estimated with a Bellman equation as a values iteration update, based on the weighted average of the old value and the reward received. The Bellman Equation of Q-Learning used is the following:

$$Q^{new}(s_t, a_t) := Q(s_t, a_t) + a \cdot [r_t + \gamma \cdot \underset{a}{max} Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where r_t is the reward obtained by the current state and α is the learning rate stating to what extent our new value should override the previous one, which varies between 0 and 1,

as ($0<\alpha<=1$) and gamma which is set to be lower than 1 in order to make the infinite sum finite. Simply put, gamma divulges how much our next rewards should be affected based on the previous highest reward. Each time t the agent selects an action, receives a reward r_t and gets into a new state $s_t + 1$, leading Q to be updated.

Q-Learning Algorithm

```
1: Initializing Q arbitrarily
2: for each episode:
3:     initialize s
4:     for each step of episode:
5:         choose a from s using policy from Q
6:         take action a and receive r and observe s'
7:         calculate Q using the Q function
8:         replace s with s'
9:     end for loop when s is terminal
10: end for when final episode is reached
```

By taking random actions on a certain state, the agent will reach the point of knowing all possible incoming states, and which action must be taken to reach them. Here comes the question of how to enable the agent to learn faster. Even though the process is pretty straightforward, by tweaking the hyperparameters and adjusting them correctly could make the learning quicker. The ε is responsible for the greedy action that will be taken. The higher the ε value is set, it is more likely that a greedy (random action) will be taken. By reducing the ε constantly, the agent commences taking actions biased by the weights of his previous actions progressively. It is highly recommended to reduce the epsilon by a very low value of 0.00001 for example. Up until the epsilon reaches the value of 0.01 (or any other value set by the user), it guarantees enough time for the agent to learn by it's random actions so far. Once ε is finally reduced to 0.01, the agent approaches the matter differently and takes actions based on the shaped Q-Value. Further values that could be consciously tweaked are the learning rate and the gamma. Learning rate is defining up to what extent our previous value should be overridden by our new Q-Value. Gamma determines how heavily our agent will be affected by the future rewards. Learning rate is preferred to be instantiated as a low value e.g. 0.0001, and it is optional whether it should be adjusted as the training is in process. In my current situation, setting gamma as high as 0.99 proved to be vastly efficient. Last but not least, notable mentioning is that Q-Learning could be truly efficient by using a Deep Learning model along with it.

A striking limitation of Q-Learning is that the algorithm is not capable of handling large scale problems with high dimensional state space. A Q-Table is useful for solving problems with a low state space. Simply put, this limitation led the researchers to look for a different approach of handling the storage of what the agent has learnt so far.

The previously stated approach is to be expounded extensively as we are moving on in the thesis, by reaching the point of explaining the Deep Q-Network.

2.2 Markov Decision Process

Markov Decision Process (MDP)[2] is a discrete-time stochastic control process which partially randomizes the action that should be taken and partially relies on decision making. MDP is useful for optimization problems based on Reinforcement Learning. MDP is based on the Markov Chains[3] which derives from the mathematician Andrey Markov. The MDP is utilized as the following process described. Assuming the agent is currently in a state ***s***, the decision maker can choose any action of ***α***, which is available at the current state it is in. As a result, the agent is moving into a state ***s'*** based on the action ***α*** taken, and the agent gets a reward of ***R(s,s')***. The state transition function ***π*** handles the output of the influence that the action will have on the new state ***s'***. Worth mentioning is that state ***s*** and action ***α*** are conditionally independent from the previous states and actions taken. Accordingly, the state transition function is based on the Markov Property. Markov Property is a memoryless property of a stochastic process. Considering the components of MDP, they are defined as ***(S,A,Pa,Ra)***, where ***S*** is the state space which contains all the states, ***A*** is the action space which contains all the actions, ***Pa(s,s')*** is the probability that in the current state ***s***, the action ***α*** at the certain time ***τ*** will lead to state ***s'***, in the following time of ***τ*** + 1. As for the ***π*** function, its purpose is to choose a policy that will maximize some cumulative function of the random rewards received so far and it is optically written as follows in Mathematics:

$$\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})$$

where $\alpha_t = \pi(st)$, which is the actions taken from the policy applied, ***γ*** is the the gamma, which similarly ranges between 0 and 1, like 0<=***γ***<=1, st is the state at the current time ***t*** and s_t+1 indicates the state in the next time ***t***. A policy that would maximize the ***π*** function is characterized as an optimal policy and it is symbolized as π^*.

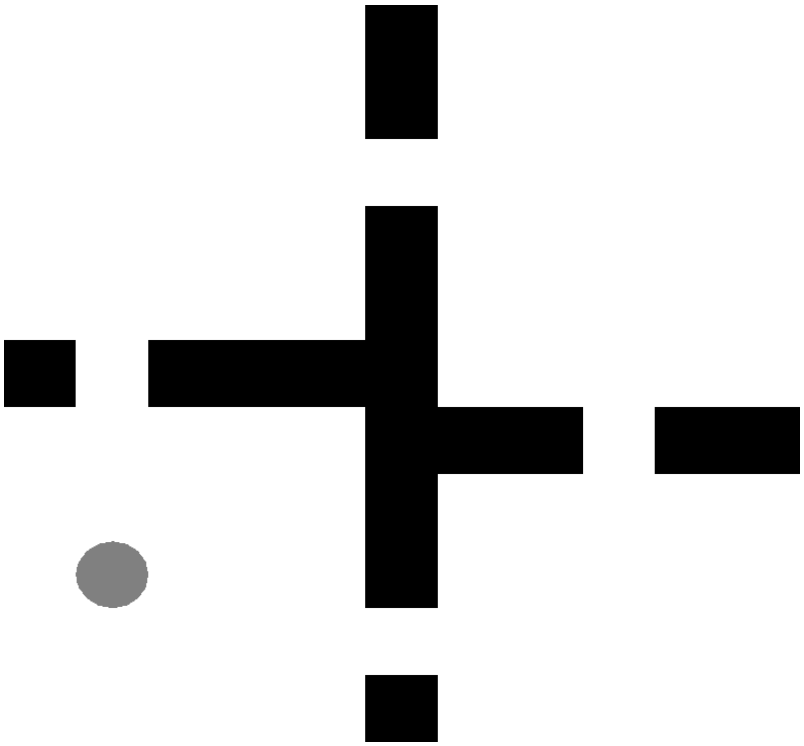

Figure 2.1 Grid World problem

A way to represent a MDP problem is the grid world problem. A grid world is a 2D environment in which an agent can move north, south, east or west by one unit each time step, provided there are no walls in the way. The above image shows a simple grid world with the agent's position represented by a gray circle and walls of the environment painted black. Typically, the goal in a grid world is for the agent to navigate to some location, but there are a number of variants.

Coming to the solutions of the MDP, there is a range of them that should be mentioned. Such methods are composed of finite state and action spaces, with specific reward functions and transition probabilities given. In order to approach the optimal policy, there are a few things that will be required. Firstly, an array that consists of the real values, which could be named values ***V*** and also an array named policy ***π*** which contains the actions. Both of those arrays are indexed by the state. Briefly, those two arrays contain values for each state so far, like a log of what was done by which action. More specifically, as the algorithm draws to an end, the ***π*** array will contain the solution and ***V*** the average rewards that will be received, at their specific states respectively, by following that solution(the sequence of actions so far). The structure of the algorithm is comprised by two steps, indicated as the following two mathematical equations:

$$V(s) := \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi(s)}(s, s') + \gamma V(s'))$$

$$\pi(s) := argmax_a \left\{ \sum_{s'} P(s' \mid s, a)(R(s' \mid s, a) + \gamma V(s')) \right\}$$

Both of them are recursively estimating and updating the optimal policy and state value using a previous estimation of those values. The steps are repeated for every state. The

algorithm is terminated once there are no more changes to be done. The order of those steps can be done in any way it fits the problem best. Furthermore, the steps can be applied for all states at once, state by state or by ordering the states the way it's more efficient.

Value Iteration: A variation of the solution presented is the Value Iteration[4]. In this algorithm the π function is calculated situationally in the V function. The mathematical formula for Value Iteration is a Bellman equation, written as:

$$V_{i+1}(s) := \max_a \left\{ \sum_{s'} P_a(s, s')(R_a(s, s') + \gamma V_i)(s')) \right\}$$

The equation is a value update equation as well. V_i is the expected sum of rewards, accumulated when starting from state s and acting optimally for a predetermined number of steps. Therefore, Value iteration is utilized in order to get the best out of every policy, which means the policy that will extract the maximum value.

Value Iteration Algorithm

```
1: Initializing value function V for all states
2: Repeat until convergence
3: for each state:
       calculate the Vi+1(s)
4: end for
```

2.3 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks(CNN)[5] are a form of deep neural networks, commonly used for image and video recognition. In 1980 Dr. Fukushima introduced "Neocognitron"[6], which contained two basic layers of a CNN, the Convolutional Layer and a Downsampling Layer. The word "deep" indicates that the network consists of more than one layer, therefore it is called deep. More specifically, CNNs are a variation of multilayered perceptrons. A multilayer perceptron is a fully connected network, where every neuron of one layer is connected to all the neurons of the next layer. Furthermore, aside the neurons, each layer contains learnable weights and biases. Each neuron obtains inputs, processes those inputs, then produces a product and finally forwards it to the next layer. Worth mentioning is that CNNs handle a small pre-processing, compared to other image classification algorithms. Given that the network is learning the filters, the amount of hardcoded hyperparameters is being reduced. Accordingly, the forward function is more efficient to be implemented. That is the basic idea of how a CNN works, and now we can move on to the components that comprise a CNN.

A generic CNN structure consists of an Input Layer, a Convolutional Layer, a Rectified Linear Units Layer, a Pooling Layer and finally a Fully-connected Layer.

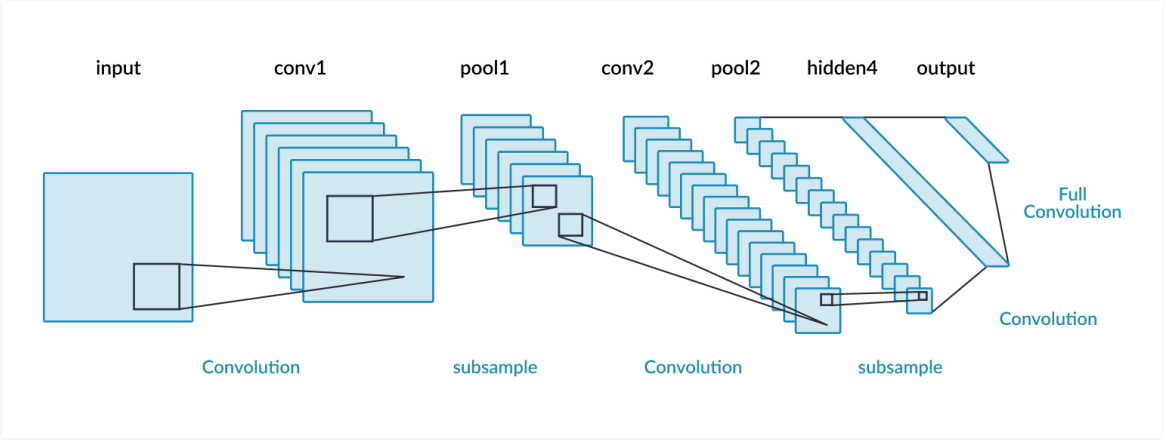

Figure 2.2 Convolutional Neural Network structure. The figure is extracted from missinglink.ai.

The Figure above is depicting the structure of a Convolutional Neural Network, which is composed of an input layer, 2 convolutional layers, 2 pooling layers placed after each of the 2 convolutional layers, a third hidden layer and finally the output.

2.3.1 Input Layer

The Input Layer holds the values of every raw pixel of the image. The layer considers the image's structure as width x height x number of channels. By symbolizing the structure it could be described as W x H x C, where W is the width, H the height and C the number of channels. Assuming we have the RGB scheme, the number of channels are 3, those of red, green and blue. Of course, the less channels we need to handle the faster the network will be processing our inputs, therefore here comes in useful the preprocessing. Preprocessing will be explained later in the thesis.

2.3.2 Convolutional Layer

The convolutional layer is highly essential for the CNNs, accordingly that's where the network's name derives from. The convolutional layer is the core of the network and does most of the computational heavy lifting. The convolutional layer's parameters are composed of learnable filters. Every filter is built of a small dimension and extends through the full depth of the input volume. Assuming a filter of the first convolutional layer has the shape of 10x10x3, where 10x10 is the 10 pixels width and height and 3 the number of channels i mentioned earlier, e.g. RGB. During the forward pass, the filters convolve through the width and height of the input volume, compute dot products between the entries of the filter and the input at any position. Accordingly, each filter produces a 2-dimensional activation map. Therefore, the network learns filters that are activated once they detect a specific visual feature located at spatial position in the input. Every activation of the filters being put together forms the output of the convolution layer. Every entry in the output can be considered as the output of a neuron of the network, which observes a part of the spatial input. The parameters of every neuron are known to every other neuron in the same activation map. This sums up the procedure a CNN follows to handle input images.

2.3.2.1 Spatial Interrelation

Considering the spatial interrelation of the CNN, there are three hyperparameters that define the output volume's size. Starting with the Depth of the output volume, which corresponds to the number of filters used. Every one of them looks for something different in the input. Neurons observing the same region of the input are characterized as a depth column. Coming back to the neuron looking for something different, as mentioned earlier, that could be the edge of the image or an intense spot of color. Next hyperparameter is the Stride. The Stride defines how many pixels we will slide through at a time. Assuming our Stride equals to 1, we are sliding through the image pixel by pixel and 2 would be 2 pixels at a time etc. It is a good practice to set the Stride of the first layers higher than the last ones, but a starting stride of higher than 4 could prove to be less efficient. Last but not least, is the Zero-Padding. Zero-padding pads the border of the input with zeros. This way we can control the dimensional size of the output volumes, which guarantees the same width and height of the input and output volumes.

2.3.2.2 Local Connectivity

The spatial extent of Local Connectivity is the hyperparameter Receptive Field of the neuron, which is also known as filter size. The spatial extent of the connectivity along the axis is always equal to the depth of the input volume. This is very important when dealing with high-dimensional input images. It is difficult to deal with them, upon connecting all neurons to all neurons of the previous volume. By connecting each neuron to a specific local region of the input volume proves to be more efficient. It is very important to state that the connections are local spatially but always full along the entire depth of the input volume. Suppose an input volume had size [16x16x20]. Then using an example receptive field size of 3x3, every neuron in the Conv Layer would now have a total of 3*3*20 = 180 connections to the input volume. Notice that, again, the connectivity is local in space (e.g. 3x3), but full along the input depth (20).

2.3.2.3 Parameter Sharing

In order to handle the parameters in the CNN one should follow the parameter sharing pattern. Let's assume we are convolving an image with a receptive field size, a stride and a pad, outputting a volume size of 30x30x80, where 80 is the depth of the convolutional layer. This makes us a total of 30x30x80=72,000 neurons. Reminding that the stride, filter and padding shrink the output volume's size, but even shrunk the neurons are quite plenty. All neurons in each depth column are connected to the same region or the input, let's say 10x10x2. Every neuron has a different weight though. This makes a total sum of 10x10x2=200 weights and 1 bias. All together this brings us a 72,000x200 multiplication, equal to 14,400,000 parameters solely on the first convolutional layer. It turns out that there has to be a way to greatly reduce that amount of parameters. And here's where the parameter sharing steps in. The parameter sharing assumes that if a patch feature proves useful to compute at some spatial position, it could also come in useful for another spatial position. Therefore, by defining a depth slice which points a 2-dimensional slice of depth, the neurons in that depth slice are forced to have the same weights and bias. In our example, the number of weights will be limited to 80 unique sets of weights, and therefore, our total weights become 80x10x10x2, producing the number of 16,000 unique weights. Since our biases are the product of unique weights plus the number of depth slices, we have a total of 16,080(16,000 + 80) parameters. During the backpropagation, each neuron in the volume will compute the gradient for its weights. Although, these gradients will be added up across each depth slice and update just a single set of weights per slice. Those sets of weights are referred to as a filter(or kernel size), convolved with the input, and accordingly derives the name Convolutional Layer.

2.3.3 Rectified Linear Units Layer

The RELU Layer is not a separate component of the CNN process, but the reason for utilizing the RELU function is to increase the non-linearity in our images, since our images are naturally non-linear. RELU layer applies an elementwise activation function, such as the max(0,x). This leaves the currently processed volume untouched, as the dimensions received. Even though RELU is not the only activation function existing, it is the most commonly used activation function in the neural networks, and also the one I am going along with in the Deep Q-Network.

2.3.4 Pooling Layer

The pooling layer is of the same importance as the convolutional layer. Pooling performs a down-sampling operation over the input volume. By putting it simply, in order to reduce the amount of parameters used and the computational load in the neural network, pooling is a function that progressively reduces the spatial size of the representation. Furthermore, the pooling layer reduces the memory footprint. Hence, it is able to control overfitting. The most frequently used pooling function is the max pooling. The max pooling function handles the partitioning of the input image, reshaping it into a set of non-overlapping rectangles. Subsequently, for each sub-region created, it outputs the maximum. It is common to periodically insert a pooling layer between the convolutional layers in the CNN structure. A usual structure of a pooling layer is with filters of a 2x2 size with a stride of 2. Accordingly, it handles a max operation over 4 neurons which leads to a 75% reduction of the size for the higher layers. The following mathematical formula describes the pooling's max operation, which maintains the depth dimension:

$$f_{X,Y}(S) = \max_{a,b=0}^{1} S_{2X+a,2Y+b}$$

Letting aside the max pooling, there are more variations of pooling, such as average pooling or $\ell 2$-norm pooling. Despite the fact that the variation of average pooling works well, it is lacking in performance compared to max pooling, which respectively outperforms average pooling.

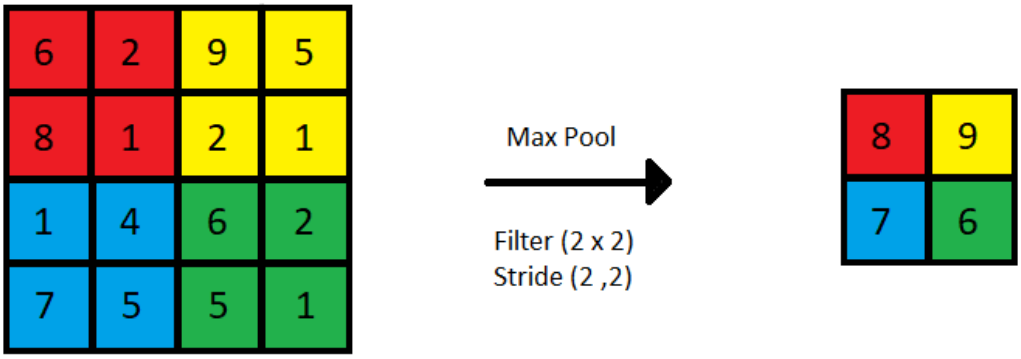

Figure 2.3 Pooling Layer process

2.3.5 Fully-connected Layer

As for the Fully-connected Layer, it can be the final layer of the convolutional neural network. A neural network can contain more than one fully-connected layer. Each neuron of this layer is fully-connected to all the neurons of the last layer, hence that's where this layer's name derives from. The fully-connected layer works like that for any neural network, and not just for the CNN. In my case, the final fully-connected layer outputs the optimal next action the agent must take.

2.3.6 Loss Layer

The Loss Layer defines the deviation between the predicted output labels and the true labels our network is trained on. The loss layer is usually the final layer in a neural network. The loss layer is implemented in a function called loss function. There are more than one loss functions, each used for the appropriate task. Some of them are the softmax loss function, the sigmoid cross entropy loss function, the Euclidean loss function or the mean squared error(MSE) loss function. The last is the one I am going along with in the Deep Q-Network. The MSE loss function is the most commonly used function for regression. The loss is the mean overseen data of the squared differences between true and predicted values. The following formula satisfies the MSE loss function mathematically:

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} (y - \hat{y}_i)^2$$

where $\hat{y}$ is the predicted value.

2.4 Python

The programming language I am using is Python[7], a high level general-purpose programming language. The reason i stick with Python is that it supports the most well-known Deep Learning and Reinforcement Learning frameworks, such as PyTorch, Tensorflow, Chainer and toolkits like Gym. Python is flexible in the aspect of it's programming structure. With that being said, Python supports the procedural structure, the object oriented structure and the functional structure. Furthermore, Python is supported by many Operating Systems, of course amongst them the Ubuntu(Linux OS) and Windows OS. Python is considered to be amongst the most popular languages of the last years up until now. According to 2019 Stack Overflow's developer survey results, Python is the fastest-growing major programming language. Part of Python's philosophy state:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

Therefore, one can imply that the simple and clean code structure is the main reason that makes Python outshine compared to other languages and be widely used to build libraries for many computer science fields. As for Deep Learning, the field itself is about recognizing patterns in data. Since Python is an easy to read language it makes the data easier to handle in Deep Learning.

2.5 PyTorch

PyTorch[8] is an open source deep learning framework, based on the Torch library developed by Facebook. It is one of the most commonly used frameworks amongst Chainer and Tensorflow. PyTorch is supported by Python and Java/C++. Considering the operating systems, PyTorch is compatible with Linux, Windows and Mac and supports CUDA as well. PyTorch has a couple of features and capabilities. Production Ready, which provides transition between eager and graph models with Torchscript and acceleration to production using TorchServe. Robust Ecosystem, which is a rich ecosystem of tools and libraries that extend PyTorch and supports development in computer vision,

natural language processing and more fields. PyTorch also supports Distributed Training, is a scalable distributed training and performance optimization, either in research or production, done by torch distributed backend. And finally, as stated by the PyTorch team, the Cloud Support, Pytorch is well supported on major cloud platforms providing frictionless development and easy scaling. Moreover, PyTorch provides tensor computing with strong acceleration via Graphic Processing units and Deep neural networks are built on a tape-based automatic differentiation system. It is important to mention that PyTorch defines a class named torch.Tensor, which stores and operates on homogeneous multidimensional rectangular arrays of numbers. PyTorch supports various sub-types of Tensors. Those tensors are similar to NumPy arrays but can also be operated on a CUDA Nvidia GPU.

2.6 Gym

Gym[9] is a toolkit developed by OpenAI, for developing and comparing reinforcement learning algorithms. As stated by OpenAI, it supports teaching agents anything, from walking to playing a game itself. Gym supports from teaching a graphically depicted robot agent to lift objects, to Atari agents on how to play and win against the computer or a human. The toolkit's library provides an open source easy-to-use interface to reinforcement learning tasks and the environment on which one can implement their algorithm, using any library of Deep Learning they want, such as Theano, Pytorch, Tensorflow etc. Gym has a great range of Atari games, which one can train on, either with RAM or with pixels(images) as input. The later technique can be approached by using the Deep Q-Network or other algorithms that take as input raw pixels. Furthermore, every game has a set of input actions. Also, every game's documentation cites that each action taken is performed repeatedly for a duration of k frames, and k is uniformly sampled in a certain range of integers. For example, pong has an observation image shaped like (210, 160, 3) as an array, the number of actions are 2, up or down, while 0 means that the agent stands still, and k ranges between 2 and 4. Another useful feature is the Gym's wrapper classes. The wrapper classes allow one to modify the environment as in a modular way. Even if the files are restructured later on, the wrappers are kept in the top level folder. For instance, one can modify the ObservationWrapper and preprocess the frames. Instead of getting the default image array shape, one can exclude the unneeded information from the frame received or even change the colors, if they don't really affect the training. The new environment will be formed based on the modified classes.

2.7 Google Colab

Google Colab[10], also known as Colaboratory is a Jupyter notebook that executes Python code on the cloud. Google Colab depicts the following features:

- Zero configuration required
- Free access to GPUs
- Easy sharing

Just like in Jupyter, the code is separated in cells, where each cell can execute as a standalone code. Colab notebooks can combine executable code and rich text in a single document, along with images, HTML, LaTeX and more. The Colab notebooks that one creates are stored in their Google Drive account. Using Colab, one can visualize and analyze data using the most popular Python libraries for Data Science. As for Data Science, data can be imported in Colab notebooks from the Google Drive account, including from spreadsheets, as well as from Github and other sources. Considering Machine Learning, one can import an image dataset and train a classifier on it in order to produce a model. The training is done on the Google cloud servers, by utilizing Google's GPUs(Graphic Processing Units) or TPUs(Tensor Processing Units) and not the local machine's CPU,GPU/TPU. The machine learning community uses Colab for applications such as Developing Neural Networks, Experimenting with TPUs, Disseminating AI Research.

CHAPTER 3

Problem Analysis

The purpose of the thesis is to create an agent that can maximize the rewards it receives by taking the right actions. Assuming we have an agent A in an environment E, the agent chooses and takes an action from the action space AS, and receives rewards R based on the actions taken. Therefore, the process is the following. The agent chooses an action and takes it, depending on whether the action was positive or not, the agent receives a positive reward and negative reward respectively. The agent wants to maximize the reward, accordingly, it is the positive reward the one the agent pursues, rather than the negative. The negative reward could also be referred to as punishment for the agent. The agent does not have any prior exposure to the current environment, consequently we can safely assume that the agent can only learn from the interactions it has with the environment. Furthermore, worth mentioning is that the updated reward received is returned in raw pixels by the environment, since i am taking screenshots to make the agent learn.

3.1 Pong

Pong is the Atari game where 2 players are controlling a vertical rectangle, and try to hit the ball in order to change its direction to the opponent's side. The one who fails to hit the ball and let’s it pass loses the round. A match is a total sum of rounds. Therefore, the one who reaches a certain score of won rounds first, wins the whole match. In my case, the agent is trained against the computer and finally manages to beat the computer after thousands of rounds played. In Gym’s Reinforcement Learning environment, the match is referred to as an Episode, and the player who reaches the score of 21 won rounds first, wins the Episode(match). Furthermore, each time the agent wins a round, its score is increasing by 1 and gets a reward of +1 too. Upon losing a round, the score stays as it is, and receives a reward of -1. This way the agent starts playing without knowing how the reward will be received and has to find that out on its own.

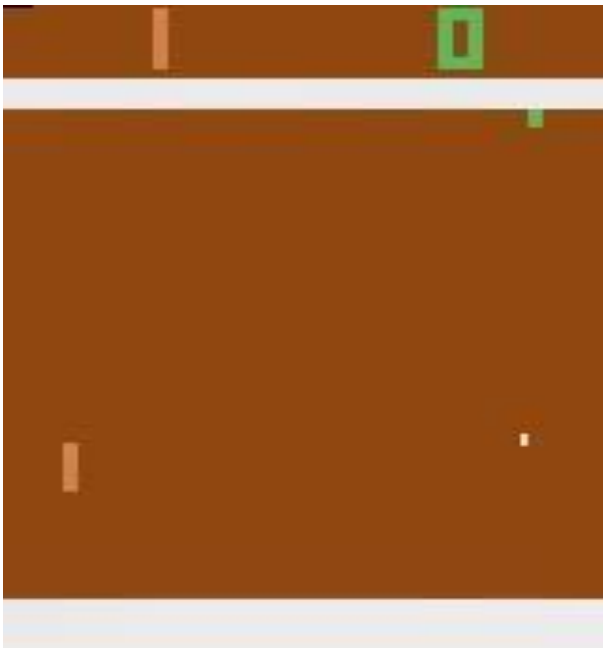

Figure 3.1 GYM Pong Environment

CHAPTER 4

Deep Q-Network

As mentioned earlier, the Q-Learning is a good method to enable the agent to start learning through a trial-and-error process. Also, as stated, the Q-table containing the optimal Q-Values of a Markov Decision Process, can store values up to a limit. Solving low-dimensional state space Reinforcement Learning problems might evince a raw Q-Learning algorithm useful, and therefore solve the problem. As for the problems that are structured to produce a high-dimensional state space, a Q-table is not capable of handling the problem. And that's where the Neural Networks step in to solve the size of the table problem and the amount of time required to explore and create the Q-Table. Therefore, the Deep Q-Network[11] glues the Q-Learning method and the Neural Networks to solve the problem. In this case, the neural network used is a Convolutional Neural Network that can process RGB images. The CNN can receive as input an image and output the optimal action that can be taken. The similarities to the simple Q-Learning method are a lot, even though there are many optimizations that have to be done. This pattern was firstly introduced by DeepMind in 2015, which works along with their so-called technique Experience Replay and the Target Network.

4.1 Q-Learning

Since Q-Learning has already been explained in Chapter 2, its functioning process will not be repeated here and I will move straight into explaining the Deep Q-Learning.

4.2 Deep Q-Learning

As stated before, most problems are difficult to be solved by using a Q-Learning algorithm with just a Q-Table. Instead, a neural network can be trained that contains parameters θ, in order to estimate the Q-values, for example Q(s,α;θ) = Q*(s,α). By using a loss function, the previous equation's loss can be minimized at every step ι.

$$L_i(\theta_i) = E_{s,a,r,s' \sim \rho(.)}[(y_i - Q(s,a;\theta_i))^2]$$

where $y_i = r + \gamma max_{\alpha'} Q(s',a';\theta_{i-1})$

yi is the Temporal Difference target, and yi - Q is the Temporal Difference Error. ρ represents the behaviour distribution. The distribution refers to the transitions s,a,r,s' where s is state, a is action, r is reward and s' the new state, and those transitions are retrieved from the environment the agent interacts with. The Q, is the Q-Learning's Bellman equation.

The hyperparameter of θ_i - 1 is constant and does not receive any updates. In practice, snapshots of the network parameters from previous iterations are used, instead of the last iteration. This is the target network.

Deep Q-Network follows the same process considering the greedy policy, similar to that of the Q-Learning. To remind this to the reader, the policy is called ε-greedy policy which selects a greedy action with the probability of 1 - ε and a random action with the probability of ε.

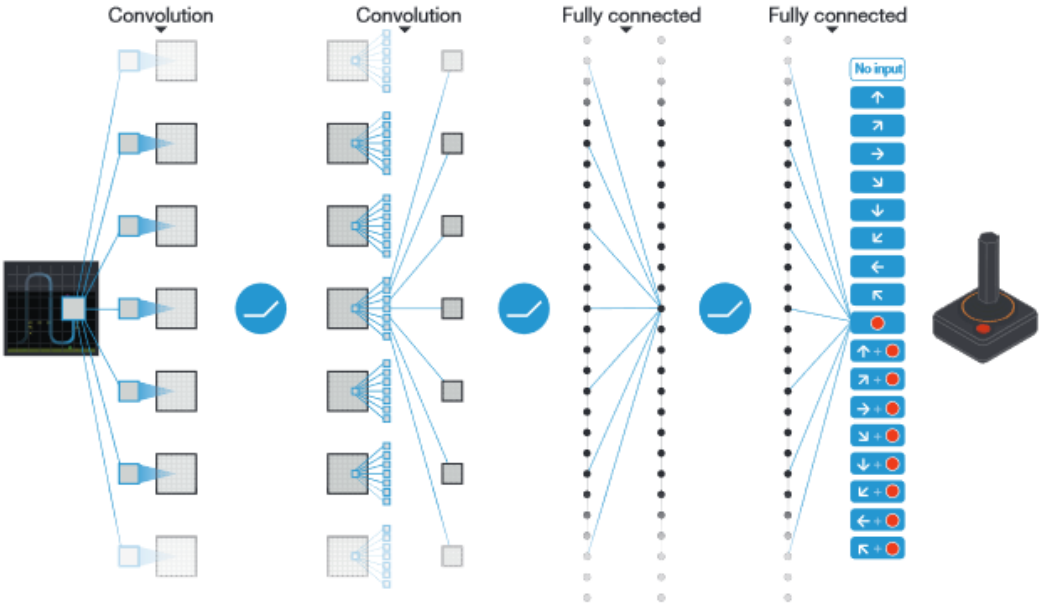

Figure 4.1 Deep Q-Network architecture

The figure above shows the Deep Q-Network architecture. The figure is extracted from the "Human-level control through deep reinforcement learning" paper.

4.2.1 Target Network and Prediction Network

As stated earlier, the Deep Q-Learning utilizes the target network. The target network is formed by the snapshots taken from the parameters of the previous iterations. Therefore except the target network. There is also a prediction network utilized during the training time. The reason Deep Q-Learning is composed of 2 networks(target and prediction one) is done in order to avoid the changing target. In deep neural networks, the target, which is the

network the model is trained on, is stable and never changes. But as mentioned before, in Q-Learning, the model(agent) is constantly exploring and changing the target network. To simplify the previous statement, imagine playing a video game where your goal is constantly changing, for example leveling up because you are not powerful enough, reaching an opponent which must be beaten to progress through the game or looking for a certain item that might be useful in the story of the game. Our goal could be whichever one of the previous three mentioned and sometimes even more. To stabilize the network, we are using the target and the prediction networks. More specifically, the target network has the same architecture as the function approximator but with frozen parameters. The prediction network is the network that every C iterations copies the parameters to the target network, where C is a hyperparameter we are declaring before the training begins.

4.2.2 Experience Replay

To avoid computing the whole expectation of the network's outcome in the loss function of the Deep Q-Network (DQN), it can be minimized by using the stochastic gradient descent. But if the loss is calculated based on the last state, action ,reward and new state, the DQN becomes a normal Q-Learning algorithm. Therefore that's where the Experience Replay is utilized.

With Experience Replay, the network's updates are more stable. The transitions (s,a,r,s') are added to a circular buffer named Replay Buffer. According to the previous statement, through the time of the agent's training, the DQN is utilizing mini-batches of the Replay Buffer to compute the loss and the gradient. This way the DQN avoids using the latest data and proves to produce better data efficiency by reusing each transition in many updates, and as stated before, a more stable network updated because of the uncorrelated transitions in a batch.

For instance, let's suppose a video game agent, where each frame is a different state from the agent's observation. In order to have unbiased and uncorrelated frames, we could randomly choose 4 frames from the last 1000 frames, (assuming we have that many in the Buffer) to train the agent on. This way we are providing sampling efficiency too.

CHAPTER 5

Case Study

In this Chapter, i am analyzing the method i followed to build the Deep Q-Network, the set up and how the Training was done and finally the results of what the agent learnt. In order to analyze these matters, I will be referring to parts of the previously inspected Chapters. Even though the structure of the Deep Q-Network is generally the same, every different case of using it, demands a different approach to it. Therefore, i believe it is essential to dive deeper into the implementation of the problem, in order to have a clearer view of the coding structure of it. After the network's analysis, i am expounding the Training part of the problem, where i am referring to the grounds of training and the process of it. As for the results, contain a breakdown of how the agent came up to this point and it's journey up until reaching those results. That is all about the prologue, let me begin with the implementation of the Deep Q-Network.

5.1 The implementation of the Deep Q-Network

Before the training begins, there are a few things that can be tweaked, which can save computational speed and accordingly the training time required to produce the model. Before initializing the gym environment, some of the classes that are used for the initialization can be overridden, considering their functions and their parameters. Their classes are known as Wrappers. As mentioned earlier Wrappers are used to transform an environment in a modular way, that fits better to our preferences on how the environment will be initialized before the training begins.

5.1.1 Frame handling and Preprocessing

First I am inserting the number of frames that will be checked and amongst them the maximum frame will be returned. The maximum frame is being selected and returned by the step function. Basically, what the step function returns is the observation of the maximum frame, as well as the reward, the done flag and the info that the gym returns. After selecting the maximum frame's observation, the arrays that contained the 4 last frames are re-setted and re-initialized using the reset function. This can be achieved by customizing the Gym wrapper classes.

As for the preprocessing, initially I am preprocessing the frame to change the observation space. The shape of the frames I will be using is 84x84x1 where 84x84 is the resolution of the frame and 1 is the number of channel(s). The number of channel(s) is 1 because I'm converting the color of the frame to gray. In my case, the color does not affect the training quality, so by converting the frame to a grayscale image works as expected. To conclude, I will be feeding the network cropped frames of 84x84 size in gray color. Then in the FrameStack class, I am redefining the observation space. In the reset function I am clearing the Stack that contains the preprocessed frames and resetting the observation as well. Then stack the cleared stack with the re-setted observation and return the stack into the shape of the observation space shape. As for the observation function, I am simply appending any new observation and of course shaping the stack like the observation space shape and then returning it.

Finally, before the agent begins its training in the Gym environment, I am gluing every modification I have done so far, this can be done by initializing the environment and passing it through every modified Wrapper class. This way the environment is setted up in the modified way and ready for the agent to be trained on.

5.1.2 Convolutional Neural Network

The next part of the Deep Q-Network is the Convolutional Neural Network(CNN). The structure and its function were extensively explained in Chapter 2. Therefore, I will bypass explaining it again and get straight into the structure I chose. Even though I will be training the agents using different CNN model approaches, the main model will be as follows. The CNN consists of 3 hidden layers and 2 fully connected at the end. The first convolutional layer has 4 channels, as much as the frames that will be inserted, and 32 are the output channels that will be forwarded to the next layer. The kernel size is 8 and a stride of 4. Briefly reminding that the kernel is a filter that extracts the features of the image fed to the network and the stride is the number of pixels that the kernel will be simultaneously processing. The second convolutional layer contains an input of 32 in total channels and an output of 64 channels for the next layer, with a kernel size of 4 and a stride of 2. The last convolutional layer contains 64 input channels and 64 output channels, a kernel size of 3 and a stride of 1. Now, the first fully connected layer will take as input the product of the third convolutional layer dimension size which is 7 x 7 x 64, and the output of the fully connected layer is a total of 512 neurons. The second and final fully connected layer takes as input the output of our previous layer, 512, and outputs the number of actions. Obviously, the layer input and output channels are increasing as we get deeper into the network, and in contrast that kernel size and the stride are decreasing. Simply put, the deeper the input is getting into the network, the more specific features are the layers corresponding to. And that is the reason the kernel and the stride are decreased over time too. As for the optimizer, the network uses a RMSProp optimizer and an MSE Loss function. Briefly, the optimizer is used to optimize the weights and the learning rate of a

neural network in order to reduce the loss, where loss is being calculated by the loss function. Since the network is built up, what follows is the forward function that will be forwarding the output of every layer to the next. First I am passing the current state of the input into the first layer and activate it using a relu function. The same process is being done again where the output of the first layer after the activation is being passed to the second layer and is activated as well using the relu function and the same happens in the third layer. Then, comes the output of the third layer, after the activation is reshaped and flattened, in order to be fed into the first fully connected layer. The first fully connected layer takes as input the output of the previous layer and is activated as well. Finally, the second fully connected layer and the last one from the network retrieves the output of the previous fully connected layer and outputs the estimation of every possible action that can be taken and of course returns the values.

Hyperparameter	value
ε initial	1.0
lowest ε	0.1
learning rate	0.00015
gamma	0.99
replay memory size	40000
update target network	1000
batch size	32
number of frames to iterate over	4

Table 5.1 Hyperparameters of the Deep Q-Network

5.1.3 Replay

The Replay is used to store and then sample the information received from the environment to the network. I am using the numpy library of python to handle the memory of the agent. The function that stored the memory of the agent contains numpy arrays of the current state, the action taken, the reward received, the new state and the done, returned from the environment. That is where we are randomly choosing states, actions, rewards, new states and dones out of the memory instead of the last 5 or 10 of them. As explained before, sampling the last memory replays could bias our network, therefore we have to

choose random previous replays for better efficiency. Each time a random replay is chosen and sampled, it is flagged as False so it can not be sampled again in the future. The replay memory is filled while the agent is training and sampled during the learning process of the agent.

5.1.4 Agent

As for the Agent class structure, it holds up the functions of choosing action, based on the epsilon's value then sampling the states, actions, rewards, new states and dones into PyTorch tensors and of course most importantly the learning function. The action chosen depends on the epsilon's value. I am taking a random value and comparing it to the epsilon's current value. If the random value is less than epsilon, a random action will be taken from the agent. Else, if the random value is higher, the agent will take a greedy action based on what it learnt so far. Therefore, I am reducing the epsilon during every episode(a total number of games), by a very small value. This value is different for every kind of training and can be tweaked until it seems to be optimal for the training. It is a good choice to reduce it by a very small value, so the agent has enough time to learn by his actions so far. Until the epsilon is reduced to a very low value, the agent has enough time to form a memory in order to take greedy actions from that time on. As for the tensor sampling function, it handles the numpy arrays, sampling them into tensors and forwarding them to the device used, the CPU or the GPU, where in my case it is Colab's GPU, and then returns them to the learning function in order to be fed to the network. Finally, is the learning function. The learning function is the most important part of the agent class, because it utilizes most of the functions that have been built up thus far. The first thing to do before start learning is to reset the optimizer's gradients. After the optimizer reset I am replacing the target network. This is done every 1000 times the learning function is being called. Replacement is done in order to avoid calculating the older values of the target network. The learning function then samples the data into tensors by using the sampling function I mentioned before. After sampling the data, the form of the data can be fed to the networks. Even though i explained before how the networks work i think it is essential to explain it more practically here. The Deep Q-Network is using 2 networks, the target and the prediction. The reason is that, in contrast to the typical supervised deep learning models, the DQN does not have a stable model to be trained on. Therefore, I am preparing the prediction network in which I am feeding the states and then with the target network I am calculating the maximum values of the new states extracted from the states fed into the prediction network. Basically, this is comparing the values and moving closest to the maximum action of the next state. And of course the target as a value is the rewards, as long as the next state is not a terminal state, added to the gamma multiplied by the output of the target network(the maximum value of the next state). If the next state is terminal, the target value equals the rewards. The reason this is done, is because we can not really calculate a prediction for the next state if there is not a next state, so any rewards are added

and we are moving to the next set of games. Closing up with evaluating the loss, and of course back propagating the loss so it takes effect on the network and minimizes the error in the next episodes. This wraps up the learning function and the next thing to do is to move to the Training of the Agent.

5.2 The Training

After explaining the Agent's structure it is time to explain the part of the Training. The training is done on the Google Colab platform, where i am using a GPU to train the agent. It is highly recommended to use a GPU or a TPU for deep neural networks' training. The reason is that a GPU consists of thousands of CPUs. Therefore, a GPU handles the specific situation of training, differently and more effectively. Practically, for a CPU it would take days to train the problem's agent but a GPU can train the agent on the same amount of episodes in hours. After initializing the customized environment, I am setting a total of 500 episodes through which the agent will go through, the best score the agent has achieved, the scores and a log for the epsilon. To clarify, while the agent is training, the epsilon is slowly reduced until it reaches the minimum epsilon set, which is 0.1 in my case. In order to test the networks that are produced after the training is over, the epsilon has to be setted to 0, since we want every action taken to be a greedy action based on our networks. Furthermore I am setting the agent's memory to 40000, since that's almost the maximum memory Colab's ram can handle, which proves to be enough for the training in the end. And so it begins, for every episode, I am resetting the observation by resetting the environment, the scores to 0 and a done flag set to false. Obviously, the environment's observation and scores are resetted in order to have a clear view of the progress of the network so far. The done flag will be set to True, once the agent reaches the end of the episode's training and accordingly terminates the loop. While in the loop, the agent must choose an action by utilizing the function built within the agent's class and then retrieve everything the step function returns. The reward returned is added to the current episode's score. The agent saves the parameters retrieved so far to its memory and then the learning function is called to do all the processing. After the learning function is done, the observation is replaced with the new observation and the loop continues until the done flag is set to True. Once the loop is over, the score is appended to the scores log. Each time a network proves to have a score higher than the average of the last 100 episode's score, it replaces the best network up until now and of course, the best score becomes the best network's average score.

5.3 The Results

Finally the results are the networks outputted after the training is over and a plot of the agent's progress thus far. In order to explain the rewards I will be using the following plot as a base for explaining.

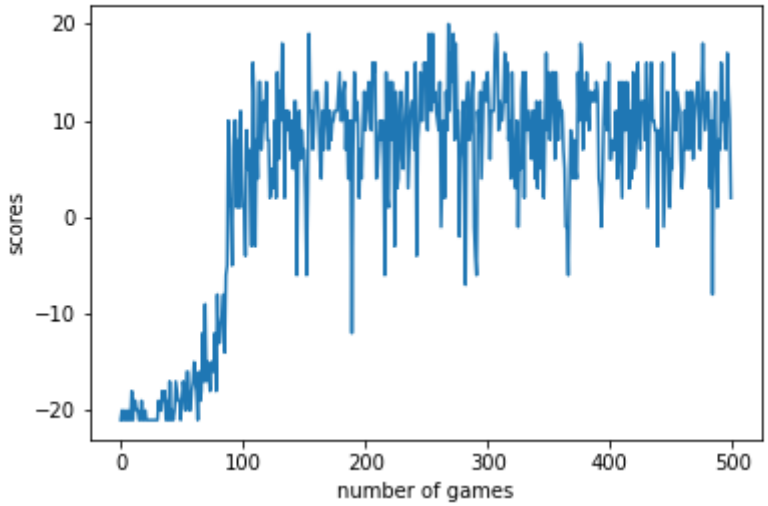

Figure 5.1 Agent's training process plot

The y axis indicates the scores of the agent during the training and the x axis the number of games the agent played. Therefore, this plot describes the agent's learning process(based on its score), over the training time. The number of games the agent was trained on are 500. The scores are in the range of -20 to 20. The agent does a huge step of learning nearly at the 100th episode, where its score is significantly increasing all the more. After that, the learning is stabilized and mainly ranges between 0 and 20, but the average score is near 10. Those results might also be achieved in a shorter time of training, but in order to be clearer in the plot, I decided to let the agent train for 500 hundred episodes, instead of 300 or 400. Furthermore, even though there seem to be some huge divergences, e.g. near the 200th game where the score drops below -10, the agent's performance during the demonstration is stable and performs as expected.

CHAPTER 6

Appendix

In this final chapter of the thesis, I will be analyzing alternative frameworks and libraries to approach the Reinforcement Learning implementation. To clarify, the analysis will contain frameworks similar to Torch, responsible for deep learning algorithms and of course Reinforcement Learning frameworks which contain Reinforcement Learning algorithms that do that whole process of the data retrieved from environments such that of GYM. Of course, it is not mandatory to use a presetted reinforcement learning framework for the training, since the functions can all be built from the scratch and customized specifically for a certain environment, similarly to the process i followed.

6.1 Deep Learning

The following are some alternatives to PyTorch Deep Learning frameworks.

6.1.1 Chainer

Chainer[12] is an open source Deep Learning framework developed by Preferred Networks (PFN). It is characterized as a powerful, flexible and intuitive deep learning framework. Chainer framework is notable for the utilization of the define-by-run scheme in deep learning. To explain further, in the define-by-run scheme, the connection with the network is not determined when the training is started but the network is determined during the training as part of the calculation performed. The flexible and intuitive characteristics refer to the lack of complexity of the define-by-run mechanism. The absence of complex conditionals and loops make the model flexible. And therefore, due to the define-by-run scheme, python’s constructs such as if statements and for loops can be used to describe such flow. Another advantage is that one can just suspend the calculation with the language's built-in debugger and inspect the data that flows on the code of the network. Later, frameworks such as PyTorch and TensorFlow adopted the define-by-run approach, due to its efficiency. Notable mentioning is also that chainer supports parallel execution(multi-node).

Chainer contains a set of extension libraries, such as ChairnerMN, ChainerRL, ChainerCV and ChainerUI. The ChainerRL is a deep reinforcement learning library which I will be explaining further later.

6.1.2 Tensorflow

Tensorflow[13] is one of the most well-known deep learning frameworks, developed by the Google Brain Team and written in Python, C++ and CUDA. APIs are provided in Python and C. Initially released for internal use and later in 2015 released to the world. It is an open source end-to-end deep learning platform. Tensorflow supports easy building models, utilizing high-level the Keras API. Tensorflow 2.0 was introduced in 2019, with many changes such as removal of old libraries, cross-compatibility between trained models on different versions of TensorFlow, and significant improvements to the performance on GPU. A very significant change was to also replace the static computational graph automatic differentiation scheme, with the Define-By-Run scheme, which was initially used by Chainer.

Keras: Keras API[14] is written in Python and runs on top of the Tensorflow framework. As the website mentions “It was developed with a focus on enabling fast experimentation.”. A deep learning model can be designed in very few lines of code using Keras. It provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity.

6.1.3 Theano

Theano[15] is another deep learning library for Python, written in Python and CUDA, and developed by the Montreal Institute for Learning Algorithms(MILA). It is also an optimizing compiler for manipulating and evaluating mathematical expressions, especially matrix-valued ones. Any computations are done using the NumPy syntax. Theano is compatible running on GPU and CPU architectures. As mentioned, heano features tight integration with NumPy and transparent use of GPU, which computes up to 140x faster than a CPU would. Furthermore, Theano can compute derivatives for functions of one or many inputs and features speed and stability optimizations, avoiding computational expressions bugs. Dynamic C code generation to evaluate expressions faster and extensive unit-testing and self-verification are also some of its features. Theano has been powering large-scale computationally intensive scientific research since 2007, but it is also approachable enough to be used in the classroom.

The following, is a comparison table, which compares the 3 previously mentioned deep learning frameworks, as well as PyTorch which i chose to go along with and explained in Chapter 2.

Name	Release Date	Open Source	Platform	Written in	Interface	OpenMP Support	OpenCL Support	CUDA Support
PyTorch	2016	Yes	Linux, macOS, Windows	Python, C, C++, CUDA	Python, C++, Julia	Yes	Yes*	Yes
Chainer	2015	Yes	Linux, maxOS	Python,	Python	No	No	Yes
Tensorflow	2015	Yes	Linux, macOS, Windows, Android	Python, C++, CUDA	Python (Keras), C,C++, Java, Go, JavaScript, R, Julia, Swift	No	Yes*	Yes
Theano	2007	Yes	Cross-platform	Python	Python	Yes	No*	Yes

Figure 6.1 The deep learning framework

6.2 Reinforcement Learning

The following, are some Reinforcement Learning libraries:

6.2.1 ChainerRL

As mentioned earlier, the Chainer[16] framework comes with libraries built for more specific utilizations. ChainerRL is a Deep Reinforcement Learning library that implements a range of the state-of-the-art deep reinforcement learning algorithms, such as Deep Q-Network, A3C, Rainbow and more. Deep reinforcement learning algorithms can be implemented to Atari games or any other problem as long as it is modeled as an environment. For instance, GYM, which provides such environments and an interface to fully retrieve any information required for the training. For ChainerRL to come in useful, there has to be an observation space and an action space defined as well as a reset and a step function (similar to those explained earlier in my implementation). Briefly reminding that the reset function resets the environment and returns the default observation and step

function executes an action, moves to the next state and returns 4 values, new observation, reward, done(whether the current state is the final or not) and any additional information. After setting up and environment, there has to be a Q-Function that receives an observation and outputs the expected future return for each action the agent can take. ChainreRL also owns predefined Q-Functions. In my case, to create a Deep Q-Network agent, there have to be some more parameters predefined such as the discount factor and the experience replay in order to begin the training. After the training is over, the agent can be tested by using the agent.act and agent.episode_stop functions for a specified number of iterations, of course, with epsilon(greedy action taking) reduced to 0. Finally, the agent can be saved using the agent.save function.

The following is a full list of the current Reinforcement Learning algorithms supported by ChainerRL:

Algorithm	Discrete Action	Continous Action	Recurrent Model	Batch Training	CPU Async Training
DQN (including DoubleDQN etc.)	✓	✓ (NAF)	✓	✓	x
Categorical DQN	✓	x	✓	✓	x
Rainbow	✓	x	✓	✓	x
IQN	✓	x	✓	✓	x
DDPG	x	✓	✓	✓	x
A3C	✓	✓	✓	✓ (A2C)	✓
ACER	✓	✓	✓	x	✓
NSQ (N-step Q-learning)	✓	✓ (NAF)	✓	x	✓
PCL (Path Consistency Learning)	✓	✓	✓	x	✓
PPO	✓	✓	✓	✓	x
TRPO	✓	✓	✓	✓	x
TD3	x	✓	x	✓	x
SAC	x	✓	x	✓	x

Figure 6.2 ChainerRL supported algorithms

6.2.2 Keras-RL

Keras-RL[17] similarly implements some of the state-of-the-art deep reinforcement learning algorithms in Python and seamlessly integrates with the deep learning library

Keras. Of course, Keras-RL works with GYM as well. Aside from GYM, Kreas-RL can also be used in a custom environment. Built-in Keras callbacks and metrics can be used or define custom ones as well. The algorithms can also be tweaked by extending some simple abstract classes. Keras-RL contains algorithms such as Deep Q-Network, Double Deep Q-Network, Deep SARSA etc.

The following, is a list of what Reinforcement Learning algorithms KerasRL supports:

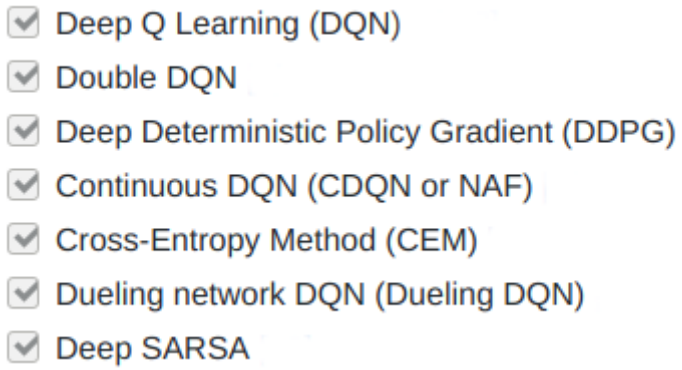

Figure 6.3 Keras-RL supported algorithms

6.2.3 Tensorforce

Tensorforce[18] is an open source Reinforcement Learning library built on top of Google's Tensorflow Deep Learning framework. It is relatively straightforward and easy to use and understand. Tensorforce is compatible with Python 3. Tensorforce follows a set of high-level design choices which differentiate it from other similar libraries, as the introduction to the library mentions. Some key features are Modular component-based design, Separation of RL algorithm and application, Full-on TensorFlow models. Feature implementations, above all, strive to be as generally applicable and configurable as possible, potentially at some cost of faithfully resembling details of the introducing paper. Algorithms are agnostic to the type and structure of inputs (states/observations) and outputs (actions/decisions), as well as the interaction with the application environment.The entire reinforcement learning logic, including control flow, is implemented in TensorFlow, to enable portable computation graphs independent of application programming language, and to facilitate the deployment of models.

To conclude, such Reinforcement Learning libraries bypass the process of building a reinforcement learning algorithm from scratch, by providing ready-to-use Reinforcement Learning algorithms. As stated, some of the libraries' functions can be tweaked.

CHAPTER 7

Conclusions

In this thesis, I am implementing and analyzing the process of the implementation of the Deep Q-Network algorithm as well its partial components, the Q-Learning algorithm and the Deep Neural Networks. The agent learns how to play from raw pixels, by forming a Neural Network which's final form outputs the actions to retrieve the maximum reward. The demonstration depicts the agent performing and winning against the computer.

7.1 Architecture

The thesis is about the Reinforcement Learning Deep Q-Network architecture. The traditional reinforcement learning implementations have limits considering the dimensional state spaces. The Deep Q-Network handles the high dimensional state space problems by utilizing a Deep Neural Network. The paper introduced by DeepMind uses a Convolutional Neural Network, in order to produce the networks that calculate the best action for the AI, based on the frames that are fed to that network. The Reinforcement Learning method could also be approached by extracting the states of the game from the RAM. For instance, OpenAI's GYM environment provides the data to train the agent through RAM instead of pixels too. Nevertheless, my approach is that of DeepMind's with the Convolutional Neural Network.

The Agent was implemented using the Python language, the GYM reinforcement learning environment, the PyTorch deep learning framework. The implementation was done on Ubuntu(Linux) and trained on Google Colab. The demonstration of the agent playing against the computer bot is done on Ubuntu as well, since the Colab can not render the game while playing (or training). The DQN algorithm handles the stability issues which normally occur when a non-linear approximate is being utilized in the reinforcement learning algorithms.

The main process is the agent playing games against the computer. Every time the agent does the right action, it is receiving a reward. The hyperparameter ε is set to 1. The agent computes a random float. As long as that float is lower than the ε, the agent takes a random action. If it is higher the agent takes a greedy action based on what it learnt so far. The ε is constantly reduced with a hard coded variable. Before the ε is reduced to its minimum value, there is enough time for the agent to form a memory from which it can learn in the future. This process continues until the agent learns how to play and maximizes its score by the reward it receives.

7.2 Comparison

As mentioned earlier I am also training the agent with a different Deep Q-Network architecture and comparing the results. In my main approach, I am using a 3-Convolutional-layer network. The second approach contains a 2-Convolutional-Layer network, where the first layer's input is 4 and the output 32, with a kernel size of 8 and a stride of 4. The second layer's input is 32 and an output of 64 with a kernel size of 3 and a stride of 1. The fully connected layers remain the same, as well as the optimizer and the loss function I previously used.

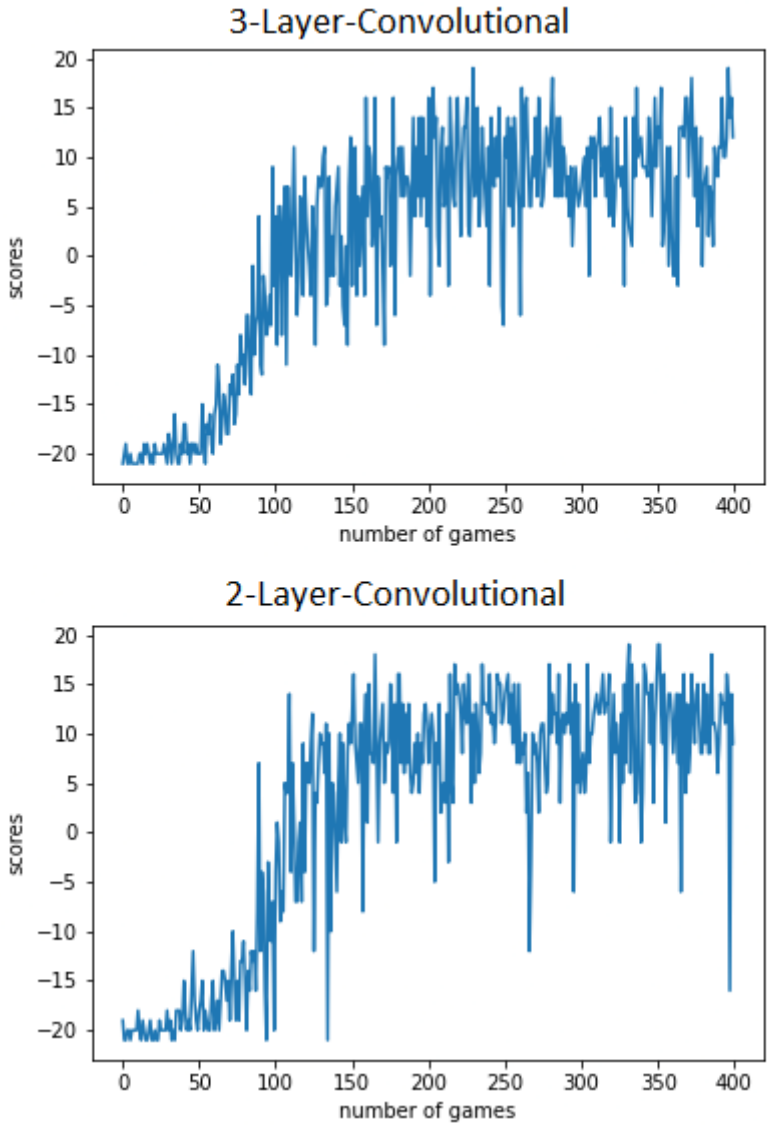

Figure7.1 Models comparison

As depicted in the figure above, the 2-Layer model performed as well as the first. Albeit, the 3-Layer model seems a little bit more stable. The 3-Layer model was trained in 14041 seconds and the 2-Layer model in 13441 seconds.

Therefore, it is safe to state that in my case, both of the models could get the job done. However, in larger scale problems, the 3-Layer or higher model might be the only way to go along with.

7.3 State of the Art

Reinforcement Learning is a powerful learning model that can train an Agent from zero. To explain further, there is no need for an already existing dataset in order to train the agent. The agent explores the environment it is in and for every positive action it receives a reward, on the other hand, for every negative action gets punished by reducing the score it formed so far. Therefore, through this trial and error process the agent learns how to behave in that certain environment. Currently, one of the biggest projects that utilized the Reinforcement Learning architecture is that of OpenAI's Five. Five is an AI that learnt how to play the video game Dota 2. A Dota 2 game is composed of 2 teams of 5 members, competing against each other until one of them wins. The 5 members can choose amongst a great range of characters. Five initially defeated amateur human teams, and in the future defeated the best Dota 2 professional team of that time. Five trains itself by playing against itself. Other implementations of Reinforcement Learning were done in the past on games like Go, where the agent could defeat the best Go player in the world.

Some of the main advantages of Reinforcement Learning are the following. Through trial and error, the agent can fully explore the environment it is trained in, and reach the point of discovering exploits of that environment that a human could not. Another advantage is that an agent can be trained in less time than a human needs. Let's assume a human and an artificial agent are being trained for a competitive video game. A day of training for the human might be a total of 100 matches, but for the agent, a day of training could be a total of 10000 matches. The way the human perceives time differs from that of the agent, because in the parallel universe of the agent, a human day could be 100 days for the agent.

With that being said, Reinforcement Learning is a powerful model that can learn and adapt in any environment. The main challenge is to form an environment that can extract organized data, based on which the agent will form a good performance model in the future.

7.4 Future Potential

The future potential of the deep learning field in general is huge. As the computational speed increases all the more, deep learning is scaling alongside it. More specifically, the increased computational speed could be highly beneficial for Reinforcement Learning

since it could reduce the training time and handle problems with higher dimensional state spaces.

Reinforcement Learning could be benefited by the future of quantum mechanics. A quantum computer developed by Google, which contains a processor of 54-qubit, could solve a certain calculation roughly in 200 seconds, that required 2.5 days by the best electronic computer to be calculated. The quantum computers are a case which is still in development through research. Albeit, the quantum computers are not performing better than the electronic computers in every case. Assuming the quantum computers' performance is stabilized and widely used in Deep Learning, the training speed would be shortened by a huge amount of time. Therefore, the quantum speedup could have a huge impact on Reinforcement Learning.

Another field that could be benefited in the future by Reinforcement Learning is autonomous vehicles. As stated by George Hotz, a co-founder of comma.ai, the problems in autonomous vehicles are 3. The static, which assumes a self-car driving on a certain route without any cars around it, using maps. The dynamic which assumes that a self-car is encountering an obstacle(e.g. another car) and has to react based on that. The third problem is the counterfactual, which involves the behavior of the other cars, which are being driven by humans. Accordingly, in the third problem the algorithm has to predict what the other human will do. And that's where Reinforcement Learning could step in. By training an agent in the environment(the road) with other agents(cars driven by humans) it could learn perfectly about the others' behavior and react correspondingly in any case. Even though this is just a proposition, it could prove to be a decent option to deal with the problem.

The next term I want to explain is Meta Learning. Meta Learning is about using metadata in order to achieve automatic learning for problems' solutions. This could describe an algorithm that is "learning to learn". The algorithm could improve its performance on its own, by using the metadata or even learn the learning algorithm itself. Coming back to Reinforcement Learning where the Meta Learning scheme could be implemented in[19]. The implementation would indicate how prior tasks can inform an agent about how to explore effectively in new situations.

With such topics being explained, it is safe to assume that Reinforcement Learning could be a major step forward for the Machine Learning field and Artificial Intelligence in the near future.

Bibliography

[1].http://burlap.cs.brown.edu/tutorials/cpl/p3.html

[2].http://burlap.cs.brown.edu/tutorials/bd/p1.html#mdp

[3].https://brilliant.org/wiki/markov-chains/

[4].https://arxiv.org/abs/1106.0251

[5].https://cs231n.github.io/convolutional-networks/

[6].https://search.ieice.org/bin/summary.php?id=j62-a_10_658

[7].https://www.python.org/about

[8].https://pytorch.org

[9].https://gym.openai.com/

[10].https://colab.research.google.com/

[11].https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf

[12].https://docs.chainer.org/en/stable/

[13].https://www.tensorflow.org/

[14].https://keras.io/about/

[15].http://deeplearning.net/software/theano/

[16].https://chainerrl.readthedocs.io/en/latest/

[17].https://keras-rl.readthedocs.io/en/latest/agents/overview/

[18].https://tensorforce.readthedocs.io/en/latest/

[19].http://papers.nips.cc/paper/7776-meta-reinforcement-learning-of-structured-exploration-strategies