Reinforcement Learning in Video Games

BSc (Hons) Artificial Intelligence

Sulaiman Alhmoudi

Department of Computer Science

The University of Manchester

Supervised by Dr. Wei Pan

April 21, 2023

Abstract

Reinforcement learning
in video games

Author: Sulaiman Alhmoudi

Reinforcement learning is a type of machine learning that involves an agent learning through trial-and-error to maximize rewards in a given environment. In recent years, deep reinforcement learning, which combines reinforcement learning with deep neural networks, has shown significant progress in solving complex problems such as playing video games.

As a starter, the project develops a Q-Network to solve the CartPole game. The second obective involves developing three deep reinforcement learning algorithms: Deep Q-Network, Double Deep Q-Network, and Dueling Deep Q-Network, in two different games: Flappy Bird and Super Mario. The games were chosen as they represent different levels of complexity, where the algorithms will be evaluated and compared to each other.

The success of the project in building powerful agents in these games is a testament to the effectiveness of deep reinforcement learning algorithms and their ability to solve a wide range of problems. Moreover, the ability to replicate existing algorithms and apply them to new problems is an important skill in the field of artificial intelligence and can lead to new breakthroughs and advancements.

As a result of developing the agents, the first agent managed to play the CartPole game and is able to solve the game and achieve the maximum possible reward for 300 consecutive episodes. As for the other two experiments, the results showed that each agent had its strengths and weaknesses and performed differently depending on the game. For example, Double Deep Q-Network performed the best in FlappyBird, while Dueling Deep Q-Network was outperformed by the other algorithms in both FlappyBird and Super Mario. However, all agents managed to perform at a superhuman level.

The project's contribution to the field lies in replicating and applying deep reinforcement learning algorithms to different games, which can inspire further research and advancements in the area as well as solve real-world problems in areas such as robotics and autonomous vehicles.

Supervisor: Dr. Wei Pan

Acknowledgements

During the early stages of this project, I encountered difficulties in devising a plan for the project. In fact, the project's main objective was initially different. It was a rough start, and I would like to thank Dr. Wei Pan for guiding me more than necessary in the initial phases of the project. I would like to also thank my family and friends for helping me, especially Ahmed for his invaluable support throughout the project, as well as proof-reading and commenting on this report.

Contents

List of Figures

List of Tables

List of Algorithms

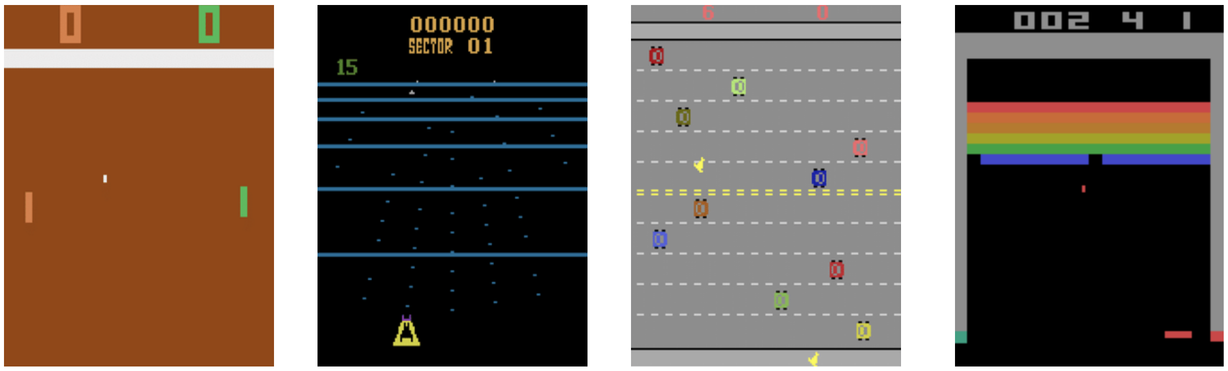
1 Introduction

1.1 Motivation

In the 1970s and 1980s, researchers began to develop more advanced reinforcement learning algorithms that could learn to solve more complex problems. One of which was the Q-learning algorithm, developed by Chris Watkins in 1989 [21]. The Q-learning algorithm was a type of reinforcement learning algorithm that could learn to make decisions based on a value function. The value function assigned a value to each possible action in a given state, based on the expected future rewards that would result from that action.

In the 1990s and 2000s, researchers continued to develop more advanced reinforcement learning algorithms, including algorithms that could learn to play games like chess and Go. However, these algorithms were still limited by the complexity of the problems they could solve, until the development of deep learning in the 2010s that reinforcement learning really took off.

In 2013, DeepMind published a paper titled "Playing Atari with Deep Reinforcement Learning" [12]. The paper developed an algorithm that could learn to play Atari games at a superhuman level using only raw pixel data from the game environment. The algorithm used in this research was a type of deep neural network called Deep Q-Network (DQN).



(a) Pong (b) BeamRider (c) FreeWay (d) Breakout

Figure 1.1: Some of the environments used in the DeepMind paper

The DQN algorithm worked by learning to predict the value of each possible action in a given game state. This allowed the algorithm to predict the best possible action in any given state. The algorithm was trained using a technique called reinforcement learning, which involves giving the algorithm a reward based on the action taken. This research opened up many possibilities for the use of artificial intelligence in gaming, and more importantly in other areas such as robotics and autonomous vehicles.

1.2 Objectives

Reinforcement learning has been very successful in a range of applications, including playing classic Atari 2600 games. The ability to take raw pixel data as input and output the best possible action in each state allows agents to learn the rules of the game and gain the most possible rewards. The development of 'generic' agents that can learn several different games with very minor changes in the algorithm is an exciting area of research. This approach is inspired by human learning, where we can quickly learn to play new games just by looking at the screen, observing the game rules, and maximizing the rewards. This project aims to implement and compare some of the reinforcement learning algorithms. There are several different reinforcement learning algorithms that can be used, and each has its strengths and weaknesses. Comparing the results of different versions of the agents developed is an essential step to evaluate their performance and identify areas for improvement. One significant advantage of using reinforcement learning to play games is that it can lead to a better understanding of the game mechanics and strategies. The agents can learn to identify patterns and make decisions based on the current state of the game, allowing them to perform better than traditional rule-based approaches.

The main objective of the project is to develop and compare several different agents to play video games. The most significant feature of the agents is taking raw pixel data as input and outputting the best possible action in each state, allowing the agent[1] and model[2] to learn the rules of the game and gain the most possible score.

1.3 Achievements

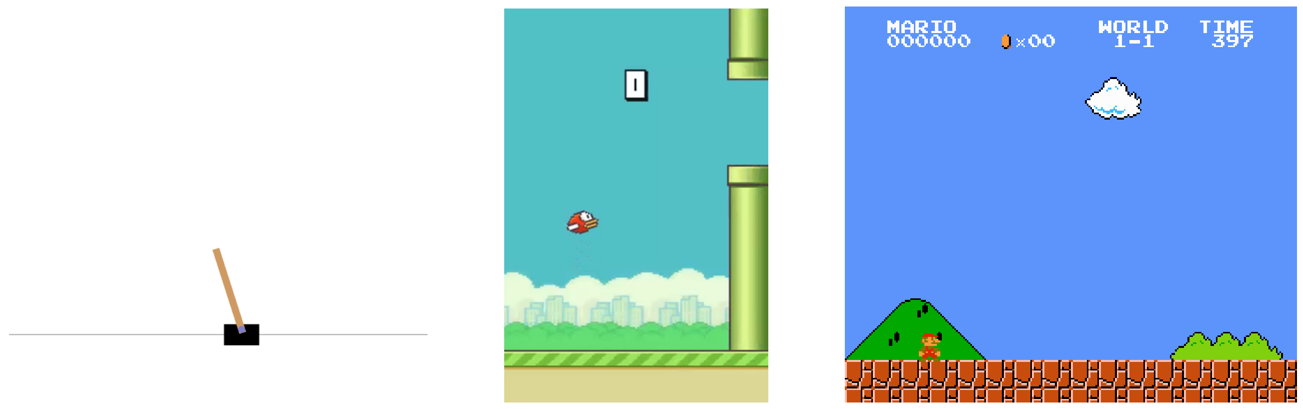
As a start, the project has successfully developed a Q-Network agent and a model that manages to solve the game of Cart Pole perfectly in a short period of time. Furthermore, this project

[1] An agent refers to an entity that interacts with an environment in order to achieve a certain goal.

[2] A model is a computational representation of a system or data that can be trained to make predictions. In this project, neural networks will be used as the model.

has successfully replicated three deep reinforcement learning agents: Deep Q-Network, Double Deep Q-Network and Dueling Deep Q-Network - in two different kinds of games: Flappy Bird and Super Mario - which are depicted in Figure 1.2.

The games chosen for the project are interesting choices as they represent different levels of complexity and require different types of strategies to solve. Cart Pole is a simple game that can be solved with a basic Q-learning algorithm, whereas Flappy Bird and Super Mario are much more challenging and require more sophisticated algorithms and techniques, such as experience replay and deep Convolutional Neural Networks (CNN). It is interesting to observe how the algorithm used in the DeepMind paper will perform on more complex game environments than simple Atari 2600 games.



(a) Cart Pole (b) Flappy Bird (c) Super Mario

Figure 1.2: Screenshot of environments used in project

The success of the project in building these powerful agents is a testament to the effectiveness of deep reinforcement learning algorithms and their ability to solve a wide range of problems. Moreover, the ability to replicate existing algorithms and apply them to new problems is an important skill in the field of artificial intelligence and can lead to new breakthroughs and advancements.

2 Background

2.1 Reinforcement Learning

Reinforcement learning is a type of machine learning algorithm that learns by understanding patterns in an environment by performing certain actions and receiving feedback in the form of a reward or a punishment. The goal of reinforcement learning is to maximize the cumulative reward over a period of time by learning an optimal policy that maps states to actions. The basic components of reinforcement learning are the agent and the environment. The agent is the learning system that interacts with the environment by taking actions. On the other hand, the environment is the world in which the agent operates, and it provides feedback in the form of rewards or punishments. The rewards indicate how well the agent is performing, and the agent uses them to update its policy. The process of reinforcement learning involves the following steps:

1. The agent observes the state of the environment.
2. The agent selects an action to perform based on the state.
3. The agent performs the action and transitions to a new state.
4. The agent receives a reward based on the new state.
5. The agent updates its policy based on the reward and state.

Reinforcement learning has been successfully applied to various real-world problems such as game playing, robotics, and autonomous driving. One of the major advantages of reinforcement learning is its ability to learn from experience without the need for supervision. Normally, the 'experience' is stored in a table containing each state and action. However, this brings up a major limitation: the limited memory available to store its experience with environments that have a large number of states. One way to address this issue is by using deep reinforcement learning, which involves merging reinforcement learning with deep learning.

2.2 Deep Reinforcement Learning

Deep reinforcement learning is a subset of reinforcement learning that uses neural networks to approximate the optimal policy. Deep reinforcement learning combines the power of deep learning with reinforcement learning to learn complex behaviors in a high-dimensional state space. The basic idea behind deep reinforcement learning is to use a neural network as a function approximator for the policy. The neural network takes the current state as input and outputs a value for each possible action. The agent selects an action based on the output of the neural network, and the environment transitions to a new state. The neural network is then updated based on the reward and the new state.

One of the key advantages of deep reinforcement learning is its ability to learn from raw sensory inputs such as images or audio. This makes it possible to apply deep reinforcement learning to a wide range of real-world problems such as image recognition, speech recognition, and many more. However, deep reinforcement learning also has some challenges. One of the major challenges is the issue of over-fitting. The neural network may become too specialized to the specific training data and fail to generalize to new situations. Another challenge is the issue of instability. The learning process can be unstable, and the neural network may oscillate between different policies. To address these challenges, several techniques have been developed such as experience replay, target networks, and exploration strategies. Experience replay involves storing experiences and sampling from it during training. Target networks involve using a separate network to generate target values for the neural network. Exploration strategies involves encouraging the agent to explore different actions (usually by selecting random actions) instead of always selecting the action with the highest probability. These topics will be explained in detail in chapter 3.

2.3 Deep Q Learning on Atari games

As mentioned in previous sections, V.Mnih et al., a team working in DeepMind, released a paper that uses DQN which combines deep reinforcement learning, CNN, and experience replay [12]. As was noted with the introduction of Deep Q-Learning the algorithm suffers from overestimating the value of some actions. This can lead to poor performance on more complex Atari games such as Space Invaders. Experience replay was used to solve this problem. Experience replay is an important technique used in reinforcement learning, where an agent learns from the experiences it has encountered in the past. The idea behind experience replay is to store the agent's experiences in a memory buffer and then randomly sample a batch of experiences from the buffer to train the agent. In other words, the agent does not learn from a chronological series of frames, but rather we store all the experience and learn using random samples of this memory. It also allows the agent to learn from infrequent experiences, which might be forgotten or underrepresented in the agent's training data. Further detailed descriptions of the Q-Learning algorithm is provided in section 3.3.

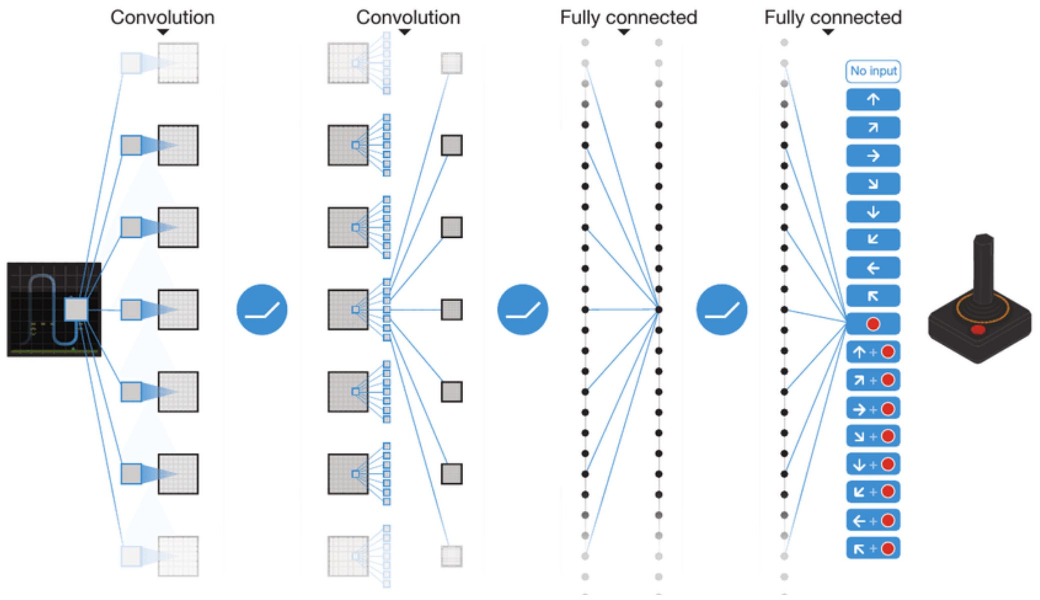


Figure 2.1: Deep Q-Network used in the DeepMind paper [11]

The above figure is a DQN architecture which shows how an image is being fed into the network and processed. The image passes through two CNNs and then flattened and fed into a MLP before outputting the final values. This report will briefly explain the theory behind these layers in chapter 3.

3 Theory

3.1 Reinforcement learning

3.1.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used to model decision-making problems in situations where the outcomes of decisions are uncertain and depend on the current state of the system. MDPs are widely used in reinforcement learning, operations research, and control theory. Below are definitions quoted from the Reinforcement Learning course by David Silver [18].

Definition 3.1. *A Markov Decision Process is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$*, where*

- $\mathcal{S}$ *is a finite set of states*
- $\mathcal{A}$ *is a finite set of actions*
- $\mathcal{P}$ *is a state transition probability matrix,*
- $\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$
- $\gamma \in (0, 1)$*, a discount factor*
- $\mathcal{R}$ *is a reward function,* $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$

As defined above, MDP is a great model for reinforcement learning problems. In order to express the strategies in MDP, we have to define what a policy is.

Definition 3.2. *A policy* π *is a distribution over actions given states,*

$$\pi(a|\, s) = \mathbb{P}(A_t = a | S_t = s)$$

The idea here is to find a policy π, to maximise the cumulative reward. To define an optimal policy, we need a series of definitions, as shown below:

Definition 3.3. *The return G_t is the total discounted reward from time-step t.*

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Definition 3.4. *The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and then following policy π*

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

Definition 3.5. *The optimal action-value function $q_*(s, a)$ is the maximum action-value function over all policies.*

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

As mentioned earlier, the goal of MDP is to find a policy π such that q_*(s, a) $\approx$ q_π(s, a). Given some state, the optimal action-value function used in the MDP will help identify the action that will lead to the highest possible future reward.

3.1.2 Q-Learning

Q-learning was first introduced by Watkins in 1989 and has since become a cornerstone of reinforcement learning [21]. It is a popular reinforcement learning algorithm used to solve MDP. Q-learning is a model-free, off-policy algorithm that can learn an optimal policy from experience. It learns by updating estimates of the expected rewards of different actions in different states.

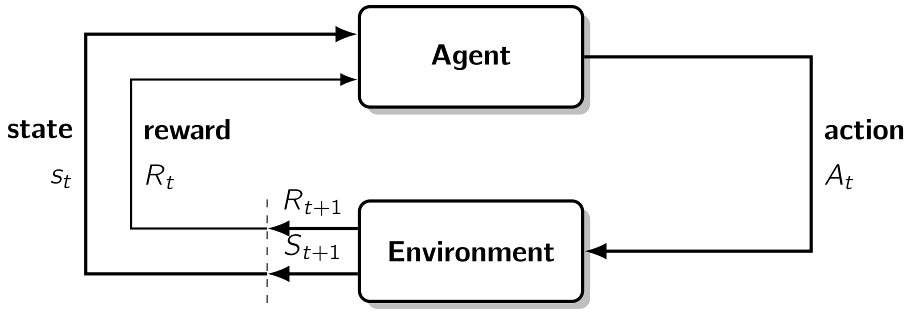


Figure 3.1: The Agent-Environment interaction in Reinforcement Learning

In Q-learning, the agent learns a Q-function that maps state-action pairs to an expected reward. The Q-function is learned through an iterative process of exploring the environment, choosing actions based on the current estimate of the Q-function, and updating the Q-function based on the reward and the estimated values of the next state and action. The algorithm does this by using the Bellman equation, which is defined below.

Definition 3.6. *The Bellman equation used to update Q-values is defined by*

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r + \gamma \; max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

- *Q(s, a), q-value for state-action pair*
- *α, learning rate*
- *r, immediate reward from enviroment*
- *γ discount factor, $\gamma \in [0, 1]$*
- *$max_a Q(s_{t+1}, a)$, estimate of optimal future reward*

The Q-learning algorithm iteratively updates the Q-value using the Bellman equation as seen above, until it converges to the optimal Q-value. One of the key advantages of Q-learning is that it does not require knowledge of the transition probabilities or the reward function, making it a useful algorithm for problems where these parameters are unknown.

	$Action_1$	$Action_2$	$Action_3$	...	$Action_n$
$State_1$	2.23431	-0.29423	1.59381	...	-1.11134
$State_2$	3.53441	5.19443	1.85341	...	6.48183
$State_3$	0.73646	-0.59322	-2.55325	...	1.48713
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$State_n$	-2.31231	-3.24213	0.19342	...	-4.19384

Table 3.1: Example of tabular Q-Learning

Since the Q-Learning algorithm is iterative, at each time-step, we need to update the Q-value based on the current state and the action taken. In section 3.3 this idea will be expanded upon, by using neural networks as a function approximator, replacing the need to store the entire table of Q-values. Algorithm 1 below shows how the Q-learning is implemented to both, predict the optimal action, and updating the Q-table.

Algorithm 1 Q-Learning (off-policy TD control) for estimating $\pi \approx \pi$ [18].

```
Initialize action-value function Q with random weights
for episode = 1, M do
    Initialise S
    while S is not terminal do
        Choose A from S by using policy derived from Q (e.g.  $\epsilon$-greedy)
        Take action A, observe R, S'
        Q(S, A) = Q(S, A) + $\alpha$[R + $\gamma$ max Q(S', a) - Q(S, A)]
        S = S'
    end while
end for
```

As shown above, we can use Q-learning to determine the optimal policy. By storing all the q_*(s,a) values, the algorithm is able to select the action leading to the highest expected future reward. However, some environments may have many states and actions. For example, it is estimated that there are between 10^{111} and 10^{123} in Chess, and it becomes too difficult to store in memory. To handle the large amount of states from the games, we can instead use a function approximator to estimate the values of q_*(s,a). This approach will be further explained in section 3.3.

3.1.3 Exploration vs Exploitation

Exploration vs exploitation is a fundamental problem in reinforcement learning, and the ϵ-greedy method is a popular and effective approach for balancing both objectives. The basic idea behind ϵ-greedy is to select the action that is most likely to lead to the highest reward according to the current estimate of the Q-function, but with a small probability ϵ, choose a random action to explore the environment and potentially discover better rewards.

$$\text{action} = \begin{cases} \max Q_t(a) & \text{Probability } 1-\epsilon \\ \text{random action} & \text{Probability } \epsilon \end{cases}$$

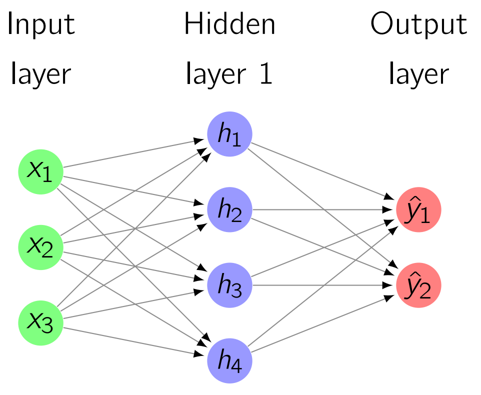
The parameter ϵ controls the balance between exploration and exploitation. As ϵ decreases over time, the agent exploits the agent and chooses the action with the highest expected reward. However, because ϵ never reaches zero, the agent will continue to explore the environment and potentially discover better policies even after it has converged to a near-optimal solution. The ϵ-greedy method is simple to implement and has been shown to be effective in a wide range of applications.

3.2 Neural Networks

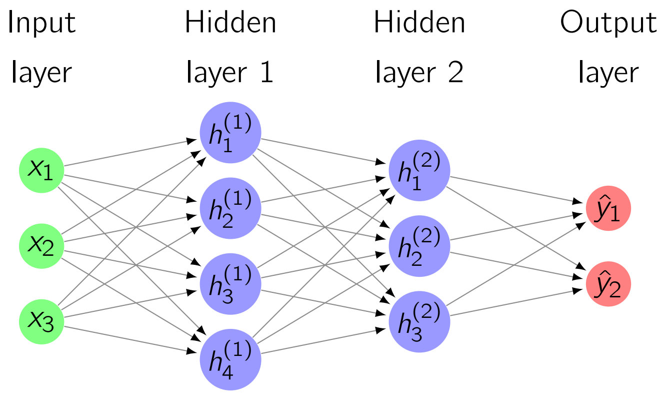
Neural networks are a class of machine learning algorithms that are designed to mimic the way the human brain works. They consist of a series of interconnected nodes, and are organized into layers. Each node receives input from the nodes in the previous layer, processes this input, and then sends its output to the nodes in the next layer. Neural networks are used to solve a wide range of problems, including image recognition, natural language processing, and speech recognition. They are useful in situations where traditional rule-based programming techniques are not effective, such as when dealing with large amounts of unstructured data. One of the key advantages of neural networks is their ability to learn from data. During the training process, the network is presented with a large amount of data, and it adjusts the weights of the connections between the neurons to minimize the difference between its predicted output and the actual output.

3.2.1 Multi-layer Perceptron (MLP)

A multi-layer perceptron (MLP) is a type of neural network that consists of multiple layers of interconnected nodes. One of the key advantages of MLPs is their ability to handle non-linear data. This makes them well-suited for tasks such as classification and regression, where the input data is often complex and difficult to model using simple statistical methods.



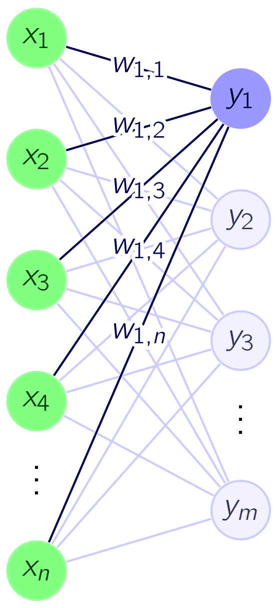
(a) MLP with one hidden layer



(b) MLP with two hidden layers

Figure 3.2: Example of a Multi-Layer Perceptron

The first layer of the MLP receives input data, and each subsequent layer performs a series of transformations on the input to produce an output. The final layer of the MLP produces the network's overall output. The output of the network is used for classification or regression tasks. In this Report, the MLP will be used to solve the Cart Pole game as it will be seen in section 5.1.



$$= \sigma\left(w_{1,0}x_0 + w_{1,1}x_1 + \ldots + w_{1,n}x_n + \mathbf{b}\right)$$

$$= \sigma\left(\sum_{i=1}^{n} w_{1,i}x_i + b_i\right)$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{1,0} & w_{1,1} & \ldots & w_{1,n} \\ w_{2,0} & w_{2,1} & \ldots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,0} & w_{m,1} & \ldots & w_{m,n} \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}\right]$$

$$y = \sigma\left(\mathbf{W}x + \mathbf{b}\right)$$

Figure 3.3: Calculating the value of a neuron in a neural network

The calculation of a node value in a neural network depends on the type of node and the network architecture being used. However, in general, the following steps are involved:

1. Inputs: Each node in a neural network receives input values from other nodes or from the network input. These input values can be raw data or the output of other nodes in the network.

2. Weights: Each input to a node is multiplied by a weight value. These weights determine the strength of the connection between the input and the node. They are learned through the process of training the network.

3. Bias: A bias value is added to the weighted sum of inputs. This bias is a parameter that helps the node output values that are more accurate for a task.

4. Activation Function: The weighted sum of inputs and bias is passed through an activation function. The activation function determines the output value of the node, based on the weighted inputs and the bias. Some of the most commonly used activation functions include sigmoid, ReLU, and tanh. In this project only ReLU is used when developing the models.

5. Output: The output value of the node is produced by the activation function and can be used as an input to other nodes in the network.

These steps are repeated for each node in the network, until the output of the network is produced. An MLP can be trained using a technique called backpropagation, which involves adjusting the

weights of the connections between the nodes in order to minimize the difference between the network's predicted output and the actual output. The mathematical background behind this algorithm is the optimisation technique known as gradient descent.
The gradient of a function gives the direction in which the function increases or decreases more rapidly. The algorithm's goal is to minimise the error between the actual value and predicted output. There are many error functions that can be used, one of which is mean squared error (MSE) loss which is described below.

Definition 3.7. *The Mean-Squared error function commonly used in a neural network is defined by*

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where (y - $\hat{y}$) is the difference between the actual value and the predicted value.

During training, backpropagation is used to update the weights of the connections between the nodes in the network, in order to minimize the error between the predicted output and the true value. This involves computing the gradient of the error with respect to the weights in the network, and then updating the weights in the opposite direction of the gradient, using a learning rate, α, to control the step size.

Specifically, we start by computing the error between the predicted output and the true output, and then work backwards through the network, computing the gradient of the error with respect to the output of each node in each layer, and then using that gradient to compute the gradient of the error with respect to the weights connecting the nodes. Once we have computed the gradients of the error with respect to the weights in the network, we update the weights using the learning rate and the gradient descent algorithm. This involves subtracting the product of the gradient and the learning rate from the current weight value, in order to move the weight in the direction that reduces the error.

By repeating this process for many iterations, adjusting the weights each time, the network gradually learns to produce more accurate outputs for a given set of inputs. In this project, the Adam optimizer will be used to update the weights, which is an optimized version of stochastic gradient descent. It is a very effective optimization algorithm and is widely used in deep learning research and applications.

3.2.2 Convolution Neural Network (CNN)

The MNIST[1] dataset is often used to evaluate image classification tasks. The image consists of 784 pixels (28x28) which can be fed into an MLP to predict which digit is in the image. For example, we can define a model with one hidden layer containing 128 neurons and an output with 10 neurons (one for each digit). The model will be able to predict which digit is in the image after training with decent results. However, in reality, images typically have high resolutions, containing a large number of pixels. For example, an image with a resolution of 1920x1080 has 2,073,600 pixels. In the case of the image consisting of 784 pixels the neural network will have 784 x 128 x 10 = 101,760 parameters (weight between each node). On the other hand, if the neural network is using an image consisting of 1920x1080 pixels, the neural network will have 2,073,600 x 128 x 10 = 2,654,208,000 paramaters. We can see how this quickly becomes a problem. To solve this, Convolutional Neural Networks (CNNs) are used.

CNNs are a specialized type of neural network that are particularly well-suited for processing and analyzing images and other types of 2D data. They are inspired by the structure of the visual cortex in the brain and use a technique called convolution to extract features from the input data. Unlike standard neural networks, the neurons in a CNN are organized into three dimensions: height, width, and depth. Furthermore, each neuron in a given layer of a CNN only connects to a small region of the layer that precedes it. As a result, the input volume for a CNN will have a dimensionality of height x width x depth (e.g. 64 x 64 x 3 for an RGB image), and the output will be a condensed version of the input. The final output layer will be the same as that of a MLP, where it will output the predicted values for each class.

CNNs consist of a series of layers that are designed to process the input data in a hierarchical manner, as seen in figure 3.4. The first layer typically consists of a series of convolutional filters, which are used to identify local patterns and features in the input data. Subsequent layers may include pooling layers, which reduce the size of the output from the previous layer, and fully connected layers, which was introduced in section 3.2.1.

One of the main features of CNNs is their ability to learn and recognize complex patterns and features in images. They are particularly effective at identifying features that are invariant to changes in scale, orientation, and other factors, which makes them well-suited for a wide range of computer vision tasks such as object recognition, image classification, and image segmentation.

[1] MNIST: A widely-used dataset in machine learning, consisting of 70,000 handwritten digits (0-9) images, with 60,000 images in the training set and 10,000 images in the test set.

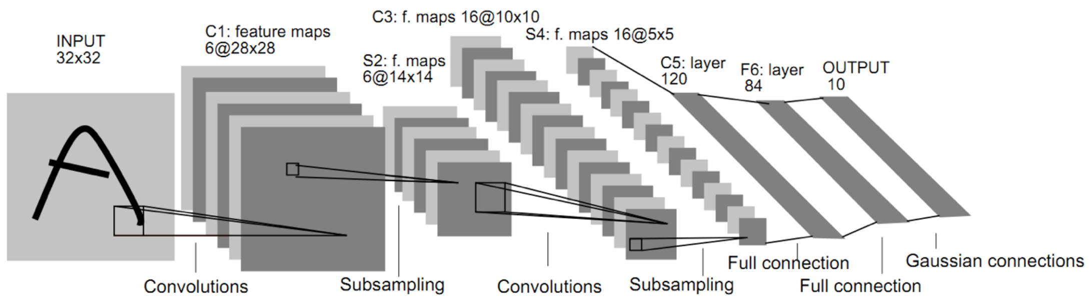


Figure 3.4: The LeNet-5 Architecture, a Convolutional Neural Network

A CNN contains four different types of layers:

- The input layer: As found in other forms of neural networks, the input layer will hold the pixel values of the image.
- The Convolutional layer: This computes the output of neurons that are connected to local regions in the input. Every one of these computes a dot product between their weights and a lesser domain to which they are linked in the input volume. The layer is responsible for detecting features or patterns in an input image. Convolution is a mathematical operation that involves sliding a filter/kernel over the input image, performing element-wise multiplication and summing up the results to produce a condensed a representation of the input image, known as a feature map.
- The pooling layer: This applies a down-sampling operation to the input image. It is typically inserted after a convolutional layer and performs a down-sampling operation by taking small patches from the input feature map and outputting a single value for each patch. This results in a smaller feature map, which reduces the number of parameters and computation required in the subsequent layers of the network.
- The fully connected layer: This layer was discussed above, in section 3.2.1, outlining the MLP.

Convolutional layer

As the name implies, the convolutional layer plays a vital role in how CNNs operate. The layers parameters focus around the use of kernels[2]. These kernels usually have a smaller dimension than the input image, but have the same depth as the input. When the data hits a convolutional layer, the layer convolves each filter across the input to produce a 2D feature map. These feature maps

[2]Kernels, filters, and receptive fields are all similar terms that are used interchangebly.

can be visualised, as depicted in figure 3.6.

By sharing the same filters across different regions of the input data, the convolutional layer can detect similar patterns regardless of their location. This property is known as translation invariance. As we glide through the input, the scalar product is calculated for each value in that kernel, as seen below.

$$y_i = \sum_{i=0}^{N} w_i * x_i$$

From this, the network will learn kernels that 'fire' when they see a specific feature at a given location in the input.

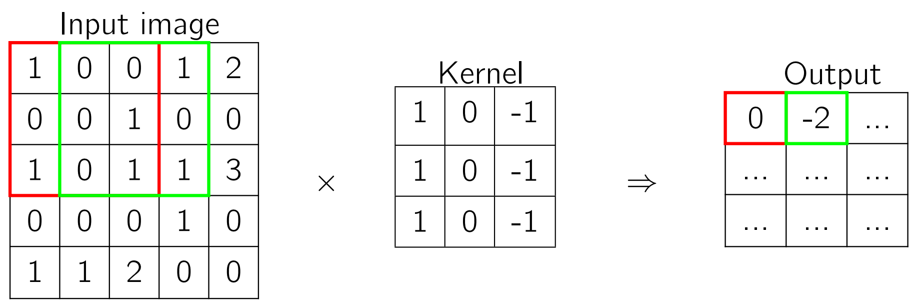


Figure 3.5: Visual representation of the convolutional layer

Each filter is a 3D matrix of size [W x H x D]. Its size is determined by the local region of the input volume that a neuron in the convolutional layer can "see" and process, known as a receptive field. The output feature maps depth is equal to the number of filters in the layer. For example, assume we have an input image with dimensions [20 x 20 x 5]. Filters can be added in any number and is a parameter set by the user. For the purpose of this example, assume the layer will use 3 filters. Since the depth of the input is 5, all filters must also have a depth of 5. Hence, there will be 3 filters each of depth 5. The width and height of the filter is also a parameter set by the user and affects the output width and height but not its depth. Finally, the output feature maps depth will be equal to the number of filters used, which is equal to 3.

As mentioned above, the output feature map has a depth equal to the number of filters used. To provide an explanation for why this is the case, lets consider a specific example. Suppose you have a convolutional layer with two filters. During the forward pass, the first filter convolves with the input image to produce a feature map, and the second filter convolves with the input image to produce another feature map. The resulting output of the convolutional layer would be a 3D volume consisting of the two feature maps stacked together. Each feature map represents a different pattern or feature learned by the network, and the combination of multiple feature maps allows the network to learn more complex representations of the input image. For example, there could be two filters - one for detecting vertical edges and one for detecting horizontal edges.

During the training process, the CNN learns the optimal values for the filter matrix that enable it to perform the task at hand, such as image classification or object. The values in the filter represent the learnable parameters of the CNN, and they are adjusted during the training process to optimize the network's performance. The specific values in the filter matrix determine the type of features that are detected in the input data. For example, a filter with positive values in the left column and negative values in the right column is used to detect vertical edges in an image, such as the one used in figure 3.5.

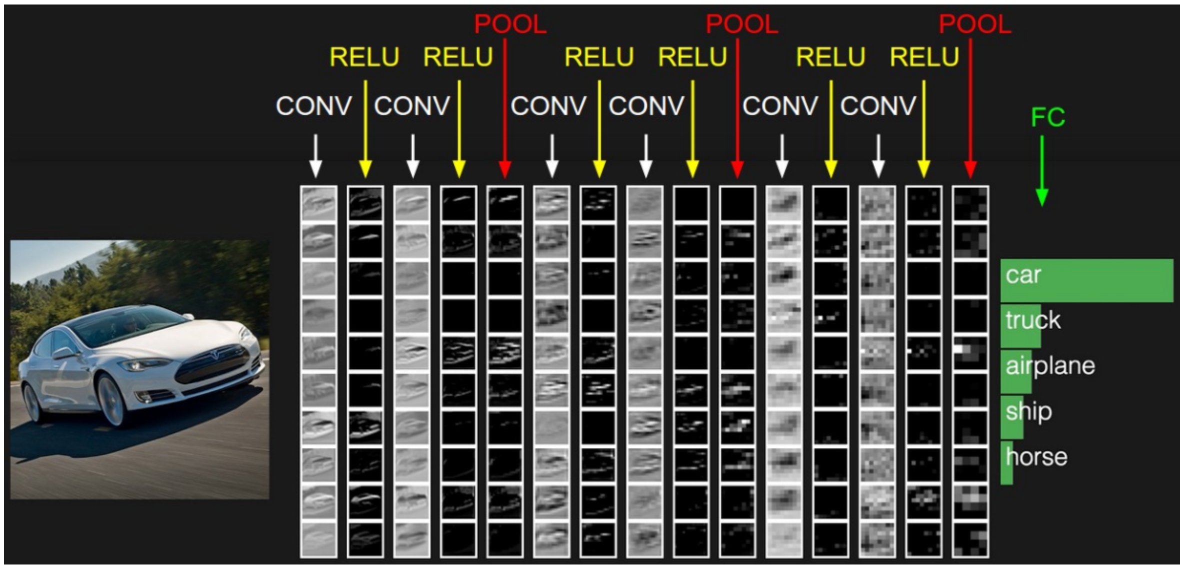


Figure 3.6: Visual representation of a convolutional Neural Network.

As we alluded to earlier, training neural networks on inputs such as images results in models of which are too big to train effectively due to the high number of parameters. To avoid this, every neuron in a convolutional layer is only connected to small region of the input volume. For example, if the input to the network is an image of size $84 \times 84 \times 4$ and we set the receptive field size as 8×8, we would have a total of 256 weights on each neuron within the convolutional layer. To put this into perspective, a standard neuron seen in other forms of neural networks would contain 28,224 weights each. Hence, convolutional layers are able to significantly reduce the complexity of the model through the optimisation of its output. These are optimised through three hyperparameters: filter size, depth, and stride.

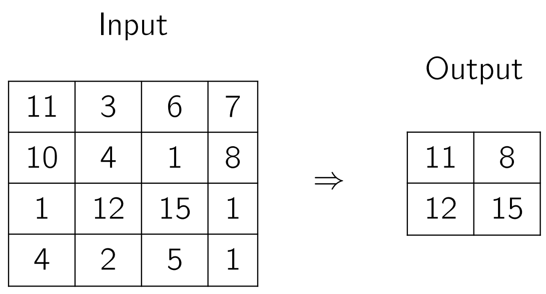
Adjusting the size of the filter is an important hyperparameter that can significantly impact the performance of the CNN. The filter size determines the size of the window that slides over the input to compute the output. If the filter size is too small, the network may not be able to capture complex patterns in the input, as the small filter only looks at a small part of the image at a time. On the other hand, if the filter size is too large, the network may capture irrelevant information from the input, leading to overfitting. Another consideration is the aspect ratio of the filter. The aspect ratio affects the type of features that the filter can detect. For example, a tall and narrow filter may help in detecting vertical edges, while a short and wide filter may help more in detecting horizontal edges. In practice, the optimal filter size may depend on the specific problem and the dataset being used, and it may need to be tuned through trial and error.

The depth of the output volume can be adjusted by changing the number of filters in the convolutional layer. Reducing the value of this hyperparameter can significantly minimise the total number of neurons of the network, but it can also significantly reduce its ability to recognize patterns.

Finally, the stride is used to determine the distance between each receptive field. The stride affects the amount of overlap between receptive fields and hence, the dimensions of the output. For example, setting a value of 1 to the hyperparameter will cause the receptive fields to overlap. Alternatively, setting the stride to a greater number will reduce the amount of overlapping and produce an output of smaller dimensions.

Pooling Layer

In CNNs, pooling layers are used to reduce the dimensions (width and height) of the feature maps produced by the convolutional layers. The pooling layer operates on each feature map independently and applies a fixed function to each part of the image. The most common pooling function is max pooling, which takes the maximum value within a sliding window of the feature map. Other pooling functions, such as average pooling or L2-norm pooling, can also be used. The main purpose of this layer is to reduce the number of parameters in the model. Pooling layers are typically inserted after one or more convolutional layers in a CNN architecture. The pooling layer convolves over each feature map in the input, and applies a function depending on which pooling layer is used. In most CNNs, these come in the form of max-pooling layers with kernels of a dimensionality of 2×2 applied with a stride of 2. This scales the feature map down to 25% of the original size - whilst maintaining the depth volume to its standard size [15].


Input

11	3	6	7
10	4	1	8
1	12	15	1
4	2	5	1

$\Rightarrow$

Output

11	8
12	15

Figure 3.7: 2 x 2 Max-Pooling with a stride of 1

Due to the destructive nature of the pooling layer, there are only two generally observed methods of max-pooling. Furthermore overlapping pooling may be utilised, where the stride is set to 2 with a kernel size set to 3. Note that, having a kernel size above 3 may greatly decrease the performance of the model [15].

Note that the project will be using a slightly modified version of the CNN developed in the DeepMind paper, which only includes convolutional layers and Fully-connected layers. The pooling layer seems promising when it comes to reinforcement learning in video games, however, it was not experimented in this project.

3.3 Deep Q-Learning

Deep Q-Learning (DQL) is a reinforcement learning technique that combines the use of neural networks and Q-learning. The basic idea behind DQL is to use a neural network to estimate the Q-values for each possible action in a given state, and then use these estimates to select the action that is most likely to lead to the highest cumulative reward. The neural network is trained using a variant of the Q-learning algorithm, in which the Q-values are updated based on the difference between the predicted Q-values and the actual Q-values obtained from the environment.

Definition 3.8. *The Mean-Squared error loss function in a Q-Network is defined by*

$$L_i(\theta_i) = E[(y_i - Q(s,a;\theta))^2]$$

where $y_i = E[r + \gamma max_{a'} Q(s',a';\theta_{i-1)}|s,a]$ *is the target for iteration i. By calculating the derivative of the loss function with respect to the weights, we get the gradient*

$$\nabla_{\theta_i} L_i(\theta_i) = E[(r + \gamma maxQ(s',a';\theta_{i-1}) \text{ - } Q(s,a;\theta_i))\nabla_{\theta_i}\ Q(s,a;\theta_i)]$$

The goal of the agent is to learn a policy that maximizes its long-term cumulative reward. In DQNs, an agent learns to make decisions by predicting the best action to take in a given state based on a Q-value function. However, there are several challenges with learning the Q-value function using standard methods. One of the main challenges that reinforcement learning agents face is that the information received by the agent may be too similar to each other, which can make the models estimates of the value of certain actions biased and unstable. Furthermore, the model may become too focused on recent experiences and have a hard time adapting to new or infrequent situations.

Experience replay is a technique that addresses these challenges by storing the agent's experiences in a replay buffer. The replay buffer is a fixed-size buffer that stores the agent's experiences as a sequence of states, actions, rewards, and next states. During training, the agent randomly samples a batch of experiences from the buffer and uses them to update its Q-value function. The main advantage of experience replay is that it allows the agent to learn from a more diverse set of experiences. By randomly sampling experiences from the buffer, the agent can learn from experiences that may be infrequent or occurred a long time ago. This helps the model to avoid overfitting to recent experiences.

Algorithm 2 Deep Q-Learning with Experience Replay [12].

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

3.4 Deep Q-Learning Optimizations

One of the key challenges in using function approximation for Q-learning is the possibility of instability or divergence, which can occur when the network is updated using its own predictions. As described above experience replay can help in reducing instability. However, there are other several techniques that have been proposed such as Double Q-Network and Dueling Deep Q-Network.

3.4.1 Double Deep Q-Network

The Deep Q-learning algorithm can sometimes overestimate the quality of actions in a state, which can result in bias. To address this issue, H. Hasselt introduced the double Q estimator function in his 2010 paper [7]. This off-policy reinforcement learning algorithm uses two Q-functions, one to evaluate the Q-value of actions and the other to select the next action.

Definition 3.9. *The Double Q-Learning update equation is defined by*

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, \arg\max_a Q\left(S_{t+1}, a; \theta_t\right); \theta_t^-\right)$$

where θ_t is the network used for selecting the action, and θ_t^- is used for evaluating the action.

Double Deep Q-Networks (Double DQN) is an extension of the Deep Q-Learning (DQL) algorithm that aims to improve its performance and stability by addressing a common problem known as overestimation bias. Overestimation bias can occur in Q-learning when the Q-values are estimated using a single neural network, which can result in an overestimation of the true Q-values. This happens because the neural network may assign high values to certain actions in a state based on the noisy or misleading data it has seen during training.

Double DQN addresses this problem by using two separate neural networks, one for selecting the optimal action and one for estimating its Q-value. Assume the first network is called the action-selection network, and is used to choose the action to take in a given state, while the second network, is called the value-evaluation network, and is used to estimate the Q-value of that action. After a number of iterations[3], the value-evaluation network is updated by copying the parameters from the action-selection network. This is known as the replace target parameter,

[3]An iteration refers to a single time-step in the training process, i.e., a batch of backward passes through the network performed to compute the gradients and update the weights.

where it will be set to 10,000 in this project. For example, if the batch[4] size is set to 32, after each iteration, the network will be updated 32 times, and after 10,000 iterations the action-selection network's parameters will be copied to the value-evaluation network. Another technique used to update the value-evaluation function is by gradually updating the parameters using a small percentage after each iteration, however, it is not used in this project.

Algorithm 3 Double DQN Algorithm [20].

```
Input: $\mathcal{D}$ -- empty replay buffer; $\theta$ -- initial network parameters, $\theta^-$ --
copy of $\theta$
Input: $N_r$ -- replay buffer maximum size; $N_b$ -- training batch size; $N^-$ --
target network replacement freq.
for episode $e \in \{1, 2, \dots, M\}$ do
    Initialize frame sequence $\mathbf{x} \leftarrow ()$
    for $t \in \{0, 1, \dots\}$ do
        Set state $s \leftarrow \mathbf{x}$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame $x^t$ from environment $\mathcal{E}$ given $(s, a)$,
            and receive reward $r$, and append $x^t$ to $\mathbf{x}$
        if $|\mathbf{x}| > N_f$ then
            delete oldest frame $x_{t_{min}}$ from $\mathbf{x}$
        end if
        Set $s' \leftarrow \mathbf{x}$, and add transition tuple $(s, a, r, s')$ to $\mathcal{D}$,
            replacing the oldest tuple if $|\mathcal{D}| \geq N_r$
        Set $s_{t+1} = s_t$, $a_t$, $x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition ($\phi_t$, $a_t$, $r_t$, $\phi_{t+1}$) in D
        Sample random minibatch of transitions ($\phi_j$, $a_j$, $r_j$, $\phi_{j+1}$) from D
        Construct target values, one for each of the $N_b$ tuples:
        Define $a^{max}(s'; \theta) = argmax_{a'} Q(s', a'; \theta)$
        Set $y_j = \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{max}(s'; \theta); \theta^-) & \text{Otherwise} \end{cases}$
        Do a gradient descent step with loss $\|y_j - Q(s, a; \theta)\|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ every $N^-$ steps
    end for
end for
```

[4]A batch is a group of tuples that include the experience, consisting of information such as the state, new state, reward, and action.

During training, the action-selection network is updated using the Bellman equation, as in regular Q-learning, but instead of using the same network to estimate the Q-value and select the action, the action-selection network is used to choose the optimal action, while the value-evaluation function is used to estimate the q-value (see Definition 3.9). This helps to reduce the overestimation bias and leads to more stable and accurate Q-value estimates.

The Double DQN algorithm implies the use of a different policy to solve the original issue of overestimation with Q-learning. It is worth noting, however, that while this method may help prevent overestimation, setting the replace target parameter too high can result in underestimation of the Q-values. Since, the value-evaluation function is 'lagging' too much behind the action-selection function.

3.4.2 Dueling Deep Q-Network

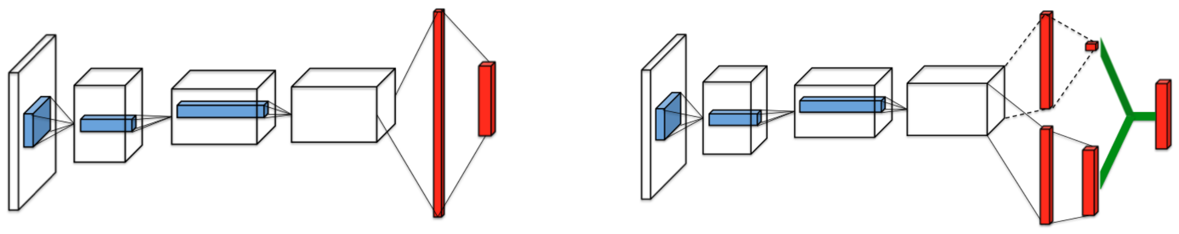
Dueling Deep Q-Networks (Dueling DQN) is a variation of the Deep Q-Learning (DQL) algorithm which aims to improve the efficiency of the algorithm by separating the estimation of state-value and action-advantage values, which can lead to more accurate Q-value values. The algorithm was introduced in 2015 by Z. Wang et al. [20].

Definition 3.10. *The Q value estimation used in the dueling Q-Learning archtitecture is defined by*

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \max_{a' \in \mathcal{A}} A(s,a';\theta,\alpha))$$

Where θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully-connected layers.

In the DQL algorithm, the Q-function is estimated for each possible action in a given state, which can lead to overestimation of the Q-values when the action values are highly correlated. On the other hand, the Dueling DQN algorithm decomposes the Q-function into two parts: a state-value function that estimates the value of being in a given state, and an action-advantage function that estimates the value of each possible action relative to the state value. The idea is that some states may have the same optimal action regardless of the state value, and by separating the estimation of state-value and action-advantage, the agent can learn to ignore the irrelevant state-value information and focus on the relevant action-advantage information.
To implement this, the Dueling DQN network architecture consists of two streams, one for estimating the state-value and one for estimating the action-advantage values. The two streams



(a) single stream Q-network (b) dueling Q-network

Figure 3.8: Comparison of single stream Q-network and dueling Q-network [20]

are then combined to obtain the final Q-values for each action in a given state as described in definition 3.10. The Dueling DQN algorithm is similar in design to the DQN algorithm (see Algorithm 2), however, the update equation is different as seen in Definition 3.10. Furthermore, the model will need to be updated to produce two outputs, the state-value and the action-advantage values. Figure 3.8 depicts the difference in the architecture between a typical DQN architecture and a Dueling DQN architecture.

Dueling DQN has been shown to achieve better performance and stability than regular DQL on a variety of reinforcement learning tasks, especially in environments with large state spaces or sparse rewards. It has been applied to many challenging problems, including playing Atari games and controlling autonomous vehicles [20].

4 Implementation

4.1 OpenAI Gym

In this project, OpenAI Gym will be used to simulate game environments. It provides a collection of environments, or "worlds", that simulate a wide range of tasks, from playing classic Atari games to controlling robots and autonomous vehicles. These environments are designed to be easy to use and provide a standardized interface for interacting with the environment, making it easier to develop and test reinforcement learning algorithms. OpenAI Gym provides an easy-to-use Python API for interacting with the environments, allowing developers to train and test their algorithms.

```
import gym
env = gym.make("Cart Pole-v0")

# Reset environment and return state values
state = env.reset()

agent = Agent()
for i in range(1000):
    # Display the environment
    env.render()

     # Predict the next action, based on the state
     action = agent.step(state)

     # Take action and update environment
     state, reward, terminated, info = env.step(action)

    # If game ended, then reset environment
     if terminated:
        state = env.reset()
env.close()
```

Listing 4.1: Source code for loading Cart Pole environment

The environments in OpenAI Gym are designed to be extensible, allowing developers to easily create new environments or modify existing ones to suit their needs. An example of this is the Super Mario environment used in this project, which is an externally developed Gym environment.

The Flappy Bird environment used in the project, however, is not an OpenAI Gym environment, but rather uses an external environment developed with Pygame[1] that has been modified to function similarly to OpenAI Gym environments.

4.2 Model

This section provides the details for implementing the models used in the experiments. There are two models, one of which is a simple neural network which will be used in the first experiment. It is a neural network which will be able to solve the Cart Pole game as it will be seen on section 5.1. The second model is a CNN developed by the DeepMind paper. It is a deep network and was used to play multiple atari games as good as human players or even better. Note that all models used in this project will use the ReLU[2] activation function.

4.2.1 Simple Neural Network

The network used in the Cart Pole environement is simple and only consists of an input layer, a hidden layer and an output layer. The input layer takes in 4 state variables and is propagated to the hidden layer which consists of 128 nodes, and finally it produces two output values.

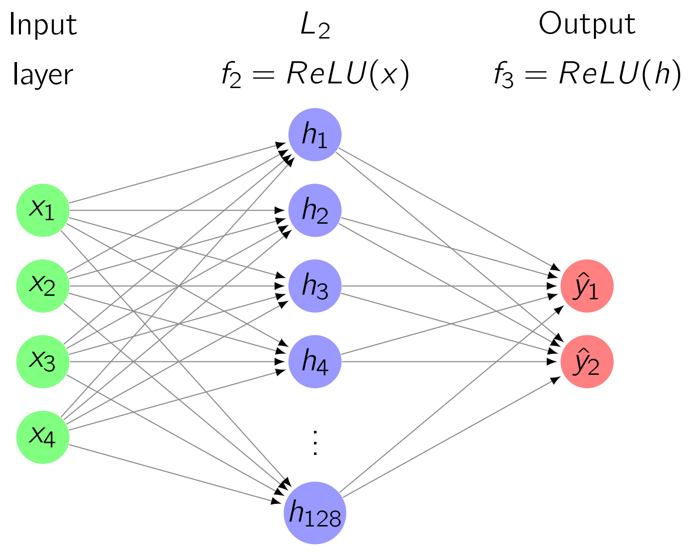


Figure 4.1: Neural network architecture used in Cart Pole experiment

[1]Pygame: a Python module that is used to create video games and multimedia applications.

[2]ReLU: It stands for rectified linear activation function and is a linear function that will output the input if it is positive, otherwise, it will output zero

4.2.2 Atari Network

As mentioned earlier, V. Mnih et al. released a paper that combined deep reinforcement learning, CNNs and experience replay. The DQN below which was used in the DeepMind paper will be used in our experiments. The output of the DQN was modified to better suit our game environments. It has also been slightly modified for the Dueling DQN to produce two different outputs, a state-value function and an action-advantage function.

Layer	Input	Filter size	Filters	Stride	Output	Activation
Convolutional	4x84x84	8x8	32	4	32x20x20	ReLU
Convolutional	32x20x20	4x4	64	2	64x9x9	ReLU
Convolutional	64x9x9	3x3	64	1	64x7x7	ReLU
FC layer	3136	-	-	-	512	ReLU
FC layer	512	-	-	-	2	Linear

Table 4.1: Parameters of DQN used in experiments

1. The first convolutional layer takes as input a stack of four grayscale images with a size of 84x84 pixels, and applies 32 filters of size 8x8 with a stride of 4. This produces 32 feature maps with a size of 20x20 pixels.

2. The second convolutional layer takes as input the 32 feature maps produced by the first convolutional layer, and applies 64 filters of size 4x4 with a stride of 2. This produces 64 feature maps with a size of 9x9 pixels. `enumerate` environment.

3. The third convolutional layer takes as input the 64 feature maps produced by the second convolutional layer, and applies 64 filters of size 3x3 with a stride of 1. This produces 64 feature maps with a size of 7x7 pixels.

4. The first fully connected layer takes as input the flattened output of the previous convolutional layer, and applies 512 linear transformations.

5. The output layer takes as input the output of the previous fully connected layer, and produces two output values, corresponding to the Q-values for the two possible actions in the game.

As mentioned previously, the activation function applied after each layer is ReLU. It stands for rectified linear unit, which simply applies the function max(0, x).

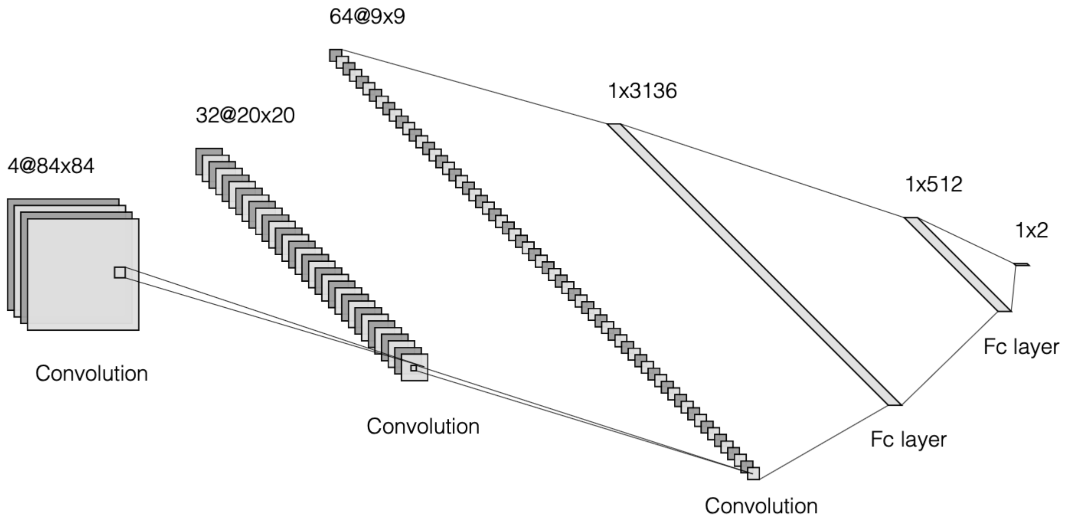


Figure 4.2: DQN architecture used in experiments

4.3 Agent

This section will provide implementation details for the agent and how it is trained. The section will be split into sections describing each agent developed.

4.3.1 Q-Network

The neural network depicted in section 4.2.1 will be used in this agent for the first experiment (Cart Pole). The agent follows algorithm 2 provided in section 3.3. The agent takes in four input states from the environment and is fed into the neural network to provide two outputs. The input states are four float values concatenated with each other. The agent also uses 'SmoothL1Loss' from the Pytorch library which is described as:

Definition 4.1. *The SmoothL1Loss equation.*

For a batch of size N, the unreduced loss can be described as:

$$\mathcal{L}(x,y)=L=\{l_1,\dots,l_N\}^T$$

with

$$l_n = \begin{cases} 0.5(x_n - y_n)^2/\beta & \text{if } |x_n - y_n| < \beta \\ |x_n - y_n| - 0.5*\beta & \text{Otherwise} \end{cases}$$

The equation creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1[3] term otherwise. It is also less sensitive to outliers than a mean-squared error loss and in some cases prevents exploding gradients. The Q-Network agent uses the default value of Beta which is a value of 1.

The "experience replay" memory is implemented using a "deque" data structure structure which we initialise with a maximum size. The memory was initialized with a maximum size of 10,000 tuples due to the limited memory available. When the memory was full, the oldest tuple was removed to make room for the newest experience. However, this implementation did not take into account that some experiences may be more important for learning than others. This issue can be addressed by prioritized experience replay [16], but it was not included in this project.

4.3.2 Deep Q-Network

The Deep Q-network agent is similar to the agent described above which follows the same algorithm as algorithm 2 described in section 3.3. However, the model used will be the Atari network. The agent will need to process the input states to be fed into the network. First, from the game environment, we receive a single frame. To speed up the training process, the frame is downsampled to 84x84 pixels and grayscaled using a function $F(s)$. Once the frame is processed, we construct a processed frame history consisting of four frames ($F(s_{t-3})$, $F(s_{t-2})$, $F(s_{t-1})$, $F(s_t)$), the past four frames are processed using the function $F(s)$. The processed frame history is then ready to be fed into the network. After each frame the oldest frame is popped out and then a new frame is appended into the list.

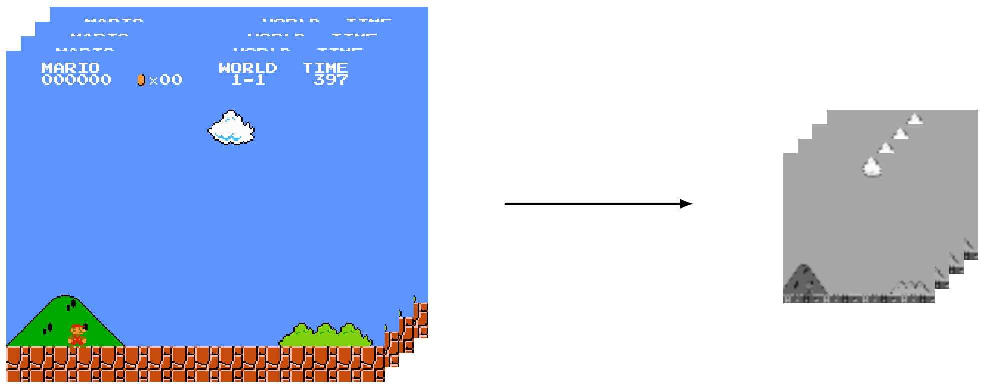


Figure 4.3: processing the frames in Super Mario

[3]L1 Regularization, also called a lasso regression, adds the "absolute value of magnitude" of the coefficient as a penalty term to the loss function

4.3.3 Double Deep Q-Network

The double DQN agent is similar to the DQN agent, however, the difference is the agent now uses two neural networks, an 'online' network and a 'target' network. The q-value of the online network is updated using the equation depicted in definition 3.9 in section 3.4.1:

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q\left(S_{t+1}, \arg\max_a Q\left(S_{t+1}, a; \theta_t\right); \theta_t^-\right)$$

Normally, in DQN the best action and its q-value is taken from the 'online' network only. However, in Double DQN the best action predicted is taken from the 'online' network, and the predicted q-value is taken from the 'target' network. After a number of iterations, the paramaters of the 'online' network is copied to the 'target' network. This usually is known as the 'replace target'. In this project we use a replace target set to 1,000. There is also another technique known as a soft update, which takes a percentage of the paramaters in the online network and copies it to the target network. However, this will not be used in this project. The double DQN will help in overestimation in q-values, which can be caused when taking too many of the same action. However, it can also be underestimating the q-values, highliting the importance of selecting a good replace target. With a good replace target value, it can produce the optimal q-values and perform the best possible actions in each state.

4.3.4 Dueling Deep Q-Network

To develop a Dueling DQN, the Atari network model will need to be modified. The final layer in the atari network will be split and produce two outputs, the advantage value and the q-values. The agent will also include two networks, an 'online' network and a 'target' network. The online network is used to predict the states and returns the advantage values and the q-values. On the other hand, the target model is used for producing the q-values for future states. The advantage values are then added to the q-values and produces the new q-values which will then be used in the update equation. Just like the Double dqn, the target network will need to be updated every 1,000 iterations.

5 Experiments

5.1 Cart Pole

5.1.1 Cart Pole Introduction

Figure 5.1 depicts the Cart Pole game environment. As it can be seen, pole is attached to a cart and follows a fixed straight-line track. The objective of the game is to balance the pole as long as possible. The reward is equal to the numbers of frames in the game. A maximum of 500 frames is possible, once 500 frames passes, the game ends. Once the pole tilts beyond fifteen degrees vertical, or the cart moves more than 2.4 units from the centre, the game is over. The Cart Pole model has four state variables:

- x - *position of the cart along the track*
- θ - *angle of the pole*
- v - *cart velocity*
- θ^{-} - *rate of change in the angle*

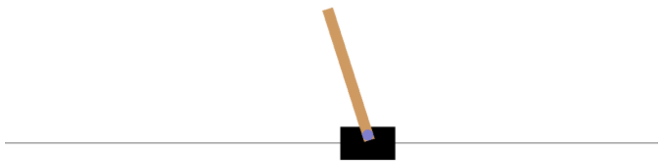


Figure 5.1: Screenshot of Cart Pole game environment

Cart Pole was chosen as the first experiment since it is a simple game, and it's easy to implement a Q-Network agent for it. Furthermore, debugging the agent in Cart Pole is straightforward because of the game's basic and uncomplicated nature. It is important to test the agents capabilities on a simple game before moving on to develop more complex DQN agents.

5.1.2 Learning Agent used in Cart Pole

The agent used in this experiment is a Q-network described in section 4.3.1. The neural network used in conjuction with the agent is mentioned in 4.2.1. The neural network takes in 4 inputs, in this case it is the input vector [x,θ,v,θ^{-}]. Since there are not many state variables it is unnecessary to use a deep neural network for this task. The model has been trained for 500 episodes, equivalent to approximately 150,000 iterations. The agent was trained using only an i7-7700K CPU for about 30 minutes. The table above displays the hyperparamters used in the

Hyperparamater	Setting
gamma	0.99
epsilon	1.0
epsilon decay	1e-3
final epsilon	0.0
replay memory size	10,000
learning rate	0.001
batch size	64

Table 5.1: Hyperparamaters used in Cart Pole's agent

agent. Notice that the epislon decay and the learning rate chosen is quite high, since this is a simple game, the agent should be able to learn and converge in a short period of time.

5.1.3 Evaluation

Figure 5.2 shows the results of training the network after 500 episodes. Since the Epsilon decay is high, the epsilon value decreases to 0 after approximately 50 episodes, meaning that the agent is always exploiting the neural network and is not making any random move.

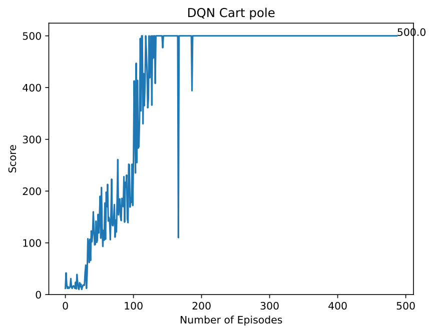


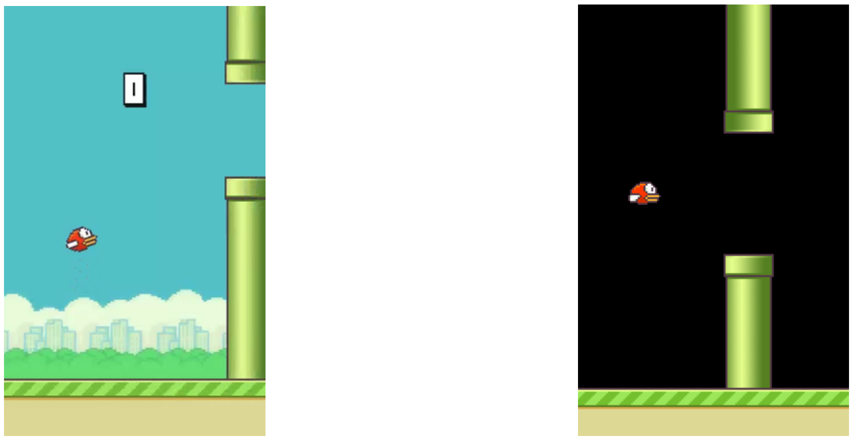
Figure 5.2: Training results for Cart Pole agent

After only 200 episodes the agent manages to solve the game and achieves a perfect score of 500. The agent continues to run for another 300 episodes to evaluate the consistency of the results. It manages to maintain a perfect score for 300 consecutive episodes, resulting in a success for the first experiment. As mentioned in section 4.3.1, the agent uses a SmoothL1Loss function to calculate the error loss, which allowed the agent to converge fast and produce a perfect score in about 200 episodes. The agent was also tested using MSELoss, however, the agent was not able to converge to a score of 500, even after training it for 1000 episodes. This shows that SmoothL1Loss is perfect for this game environment.

5.2 Flappy Bird

5.2.1 Flappy Bird Introduction

The next experiment is to develop an agent to play Flappy Bird using just the raw pixel data of the game frame, the same way a human plays and learns. The objective of Flappy Bird is to guide a bird through a series of obstacles to make the bird flap its wings and fly upwards. The game is endless, and the player's score is determined by how many obstacles they can navigate through without colliding with them. The game is over when the bird hits an obstacle. The objective is to achieve the highest score possible.



(a) Default (b) Simplified

Figure 5.3: Screenshot of default and simplified Flappy Bird Environment

Notice that the environment has been modified for reinforcement learning. The background and score has been removed to help with removing any irrelevant data in the game screen. Furthermore, in the original game, the game will terminate if the bird touches the 'ceiling', but in the simplified version the bird will not die and will continue to keep on flying. The changes were made because it is not important whether the bird dies upon touching the ceiling, since the main objective is to navigate through the tunnels.

5.2.2 Learning Agent used in Flappy Bird

This experiment will train and compare three different agents: DQN, Double DQN, and Dueling DQN. The game is 2-dimensional and consists of a 512×288×3 RGB bitmap image. To speed up the training process the image will be resized to 84x84 pixels and gray-scaled. At each state there will be 4 frames concatenated and fed into the network. The reward gained after each action is +1 if it passes through a pipe, +0.1 if it flaps, and -1 if the game is over. The reward for flapping is important, since it 'motivates' the agent to keep on flying as it is required to gain rewards for passing through a tunnel.

Hyperparamater	Setting
gamma	0.99
epsilon	0.1
final epsilon	0.01
replay memory size	10,000
learning rate	1e-6
batch size	32
replace target (for Double/Dueling DQN)	1,000

Table 5.2: Hyperparamaters used in Flappy Bird's agent

Note that the epsilon decay used in the Flappy Bird agent, is simply a linear decay over the number of iterations. To be more specific, the epsilon value starts of at 0.1 and linearly decreases over the number of iterations (2,000,000) and ending at an epsilon value of 0.01.

5.2.3 Evaluation

Each model was trained for 2 million iterations (equivalent to approximately 16,000 episodes) using an RTX 2070 GPU for about 12 hours, resulting in about 36 hours of training for all three agents. Figure 5.3 lays out the training results and it can be seen that the three agents manage to achieve an average score of about 17.5. After about 15,000 episodes The minimum epsilon value converges to 0.01, which is explains the sudden increase in the average score.

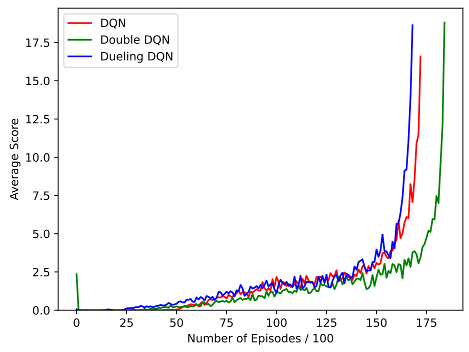


Figure 5.4: Flappy Bird agent training results

Table 5.3 lays out the test results after the training process. Each model was tested for 20 episodes and the maximum and average score was recorded. It can be seen that the best performing agent is the Double DQN, followed by the DQN agent, and the least performing is the Dueling DQN agent. While the Dueling DQN agent was the worst performing, it managed to achieve an average score of 500, which far surpasses the average human player.

Agent	Max score	Avg score
DQN	2000	837
Double DQN	4700	1200
Dueling DQN	1300	500

Table 5.3: Flappy Bird test results

5.3 Super Mario

5.3.1 Super Marion Introduction

The objective of the classic video game Super Mario is to guide the main character through various levels filled with obstacles and enemies. Along the way, Mario can collect power-ups such as mushrooms and fire flowers to help him defeat enemies and gain new abilities. In this experiment, the main objective will focusing on reaching the end goal of the first level at the fastest possible time. The game is more complex than Flappy Bird and Atari games. Super Mario's complex environment presents a wider range of obstacles and opportunities for the agent to learn from, such as avoiding enemies, jumping over holes that appear on the ground, and maneuvering over obstacles. It is interesting to see how the three DQN agents will perform in such environment.

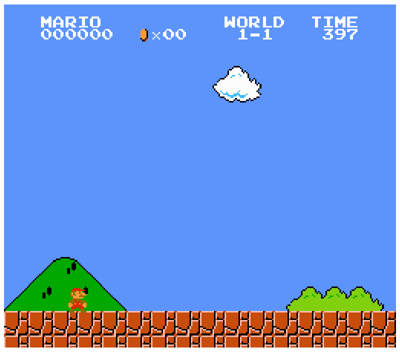


Figure 5.5: Screenshot of the Super Mario game environment

5.3.2 Learning Agent used in Super Mario

Just like the Flappy Bird environment, the game will need to process the frames, where it will be resized to 84x84 and gray-scaled. Furthermore, the frame will crop out the top of the image which contains the score and time, as it is irrelevant. The agent will be able to select only two actions in this game environment - to simply move right or to move right and jump.

Hyperparamater	Setting
gamma	0.95
epsilon	1.0
epsilon decay	0.1
final epsilon	0.01
replay memory size	10,000
learning rate	0.00025
batch size	32
replace target(for Double/Dueling DQN)	1,000

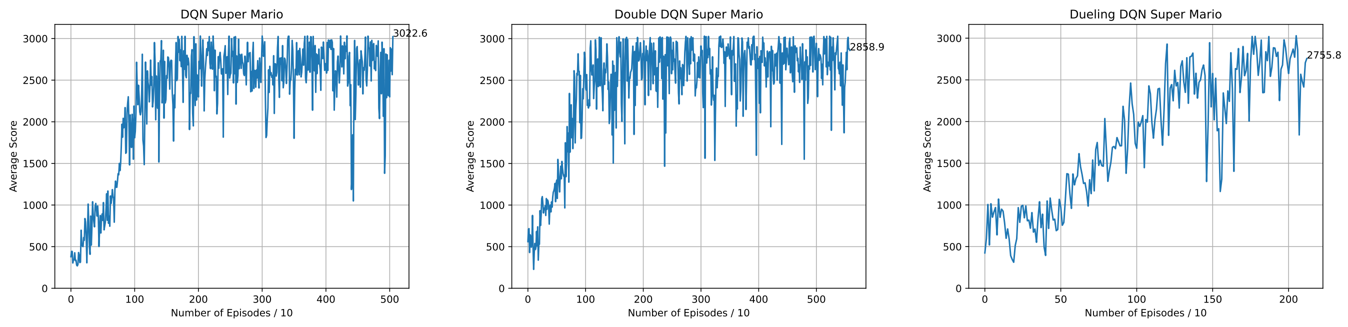
Table 5.4: Hyperparamaters used in Super Mario's agent

In this agent, we introduce the idea of a "frame-skip". A frame-skip is when we take an action in the environment, then for a series of k frames we perform the last action taken. This is very important and the frame-skip hyperparameter has been shown to be a strong determining factor in the performance of algorithms when playing Atari [3]. Using a frameskip, k = 4, we use every fourth frame from the game. When using frameskip the concatenated frames will be: (s_{t-3k}),

(s_{t-2k}), (s_{t-k}), (s_t). An important observation of this method is that the states are overlapping in consecutive forward passes. It means that we provide the information to the network to predict the movement of the ball, such that we can move the paddle under the ball in time.

5.3.3 Evaluation

Moving on, the model has been trained for 2 million iterations (equivalent to approximately 5,000 episodes). After about 100 episodes the epsilon value will be set to 0.01, due to the high epsilon decay value. Figure 5.6 compares the training results of the three models and it can be seen that the DQN agent manages to converge the fastest in this game environment.



(a) DQN Agent (b) Double DQN Agent (c) Dueling DQN Agent

Figure 5.6: Training results for Super Mario Environment

Each model was ran for 100 episodes and the average score and win rate was calculated. Table 5.5 below lays out the test results. As it can be seen, all three agents manage to gain the maximum possible score for all 100 episodes. Surprisingly, the DQN agents manage to successfully play the game perfectly, even though it is a more complex game environment.

Agent	Avg score	Win Rate
DQN	3028	100
Double DQN	3028	100
Dueling DQN	3028	100

Table 5.5: Super Mario test results

6 Conclusion

The project has come to an end, and it is crucial to reflect on all the stages of research and development process. At the beginning of the project, the focus was on comprehending Q-Learning methods for video games, and exploring techniques such ass Double DQN and Duelling DQN to improve the stability of the algorithm. However, as I was not very familiar with neural networks, I started with learning the basics of the theory, which laid a strong foundation to my knowledge of artificial intelligence. The experiments conducted during the project demonstrated that despite the limited training time and computational resources, a fully functional implementation of all the methods were successfully developed, and the results were even better than expected based on past research and theory. The design chapter's proposed improvements also had an impact on the Q-Learning algorithm's performance. For example, the Double DQN used in flappy bird managed to double the maximum score resulted in using the normal DQN agent. While the project showed the potential of deep reinforcement learning, there is still areas for improvement. Advanced algorithms such as Actor-Critic could boost the agents' performance, and transfer learning techniques could improve their ability to generalize their learning to new environments [14].

Additionally, the project was limited to video games, and it would be interesting to see how these algorithms perform in real-world scenarios. For example, reinforcement learning could be used to train robots to perform tasks in hazardous environments such as oil rigs or nuclear power plants. Furthermore, reinforcement learning could be used to develop artificial intelligence programs for autonomous vehicles that can adapt to different driving conditions and make decisions in real-time. Since the project focused on deep reinforcement learning in video games, it would be interesting to see how other approaches such as unsupervised learning and generative models can be combined with reinforcement learning to further enhance the agents' performance. This could lead to the development of more intelligent and adaptable agents that can learn from their environment in a more sophisticated way.

References

[1] Kai Arulkumaran et al. "A brief survey of deep reinforcement learning". *arXiv preprint arXiv:1708.05866* (2017).

[2] Marc G Bellemare et al. "The arcade learning environment: An evaluation platform for general agents". *Journal of Artificial Intelligence Research*. 47 (2013), pp. 253–279.

[3] Alex Braylan et al. "Frame skip is a powerful parameter for learning to play atari". In: *Workshops at the twenty-ninth AAAI conference on artificial intelligence*. 2015.

[4] Greg Brockman et al. "Openai gym". *arXiv preprint arXiv:1606.01540* (2016).

[5] Kevin Chen. *"Deep Reinforcement Learning for Flappy Bird"*. 2015. Available from: https://cs229.stanford.edu/proj2015/362_report.pdf.

[6] Vincent François-Lavet et al. "An introduction to deep reinforcement learning". *Foundations and Trends® in Machine Learning*. 11.3-4 (2018), pp. 219–354.

[7] Hado Hasselt. "Double Q-learning". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010.

[8] Fei-Fei Li et al. *"CS231n: Deep Learning for Computer Vision"*. 2015. Available from: https://cs231n.github.io/convolutional-networks/.

[9] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". *arXiv preprint arXiv:1509.02971* (2015).

[10] Long-Ji Lin. *"Reinforcement learning for robots using neural networks"*. Carnegie Mellon University, 1992.

[11] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". *Nature 518, 529–533* [online] (2015). DOI: 10.1038/nature14236.

[12] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". *arXiv preprint arXiv:1312.5602* (2013).

[13] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[14] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[15] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks". *arXiv preprint arXiv:1511.08458* (2015).

[16] Tom Schaul et al. "Prioritized experience replay". *arXiv preprint arXiv:1511.05952* (2015).

[17] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". *Neural networks*. 61 (2015), pp. 85–117.

[18] David Silver. *"UCL Course on RL"*. 2015. Available from: https://www.davidsilver.uk/teaching.

[19] Hado Van Hasselt et al. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[20] Ziyu Wang et al. "Dueling network architectures for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.

[21] Christopher Watkins. "Learning from delayed rewards" (1989), pp. 37–43.

[22] Christopher Watkins and Peter Dayan. "Q-learning". *Mach Learn 8, 279–292* [online] (1992). DOI: 10.1007/BF00992698.