



The University of Manchester

Department of Computer Science
Project Report 2020

**Reinforcement learning for a learnable agent
in classic arcade games**

Author: Sean J Parker

Supervisor: Dr. Konstantin Korovin

Abstract

Reinforcement learning for a learnable agent in classic arcade games

Author: Sean J Parker

The aim of the project is to investigate the performance of Gismos and to design and construct a super multi-functional Gismo.

The novel aspects of the new Gismo are described. The abstract should perhaps be about half a page long.

The results of testing, which show the abject failure of the Gismo, are presented.

In the conclusions proposals for rectifying the deficiencies are outlined.

Supervisor: Dr. Konstantin Korovin

Acknowledgements

I would like to thank my parents, my school teachers, my friends, my wonderful supervisor, and all my wonderful fellow students for their unswerving support during my project. Without your help none of this would have been possible.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Objectives	6
1.3	Report structure	6
2	Background	7
2.1	Reinforcement learning	7
2.1.1	Deep reinforcement learning	7
2.2	DQN on Atari 2600	8
2.3	CNN Visualisation	9
3	Design	10
3.1	Markov decision process	10
3.1.1	Markov property	11
3.2	Reinforcement learning	12
3.2.1	Exploration vs Exploitation	13
3.3	Q-Learning	13
3.4	Deep Learning	14
3.4.1	Multi-layer perceptron (MLP)	15
3.4.2	Convolutional Neural Network (CNN)	16
3.5	Deep Q-Learning	17
3.5.1	Experience Replay	18
3.6	Q-Learning improvements	19
3.6.1	Double Q-Learning	19
3.6.2	Duelling Q-Learning	19
	References	20

List of Figures

1.1	Screenshots of Pong, Breakout, Space Invaders (left to right).	6
2.1	OpenAI Robot solving a Rubik's cube	8
2.2	Representation of end-to-end RL architectures	8
2.3	Deep Q-Network architecture	9
3.1	Screenshot of single breakout frame	12
3.2	Diagram of reinforcement learning	12
3.3	Schematic the modelling of a single neuron	15
3.4	Diagram representation of MLP	15
3.5	Single convolutional layer using filters	16
3.6	Architecture of a network to predict MNIST digits	17

List of Tables

3.1	Best performing method in given environment	10
3.2	Q-Table Example	13

Chapter 1

Introduction

My project aims to replicate some of the reinforcement learning algorithms that can be used to play classic Atari 2600 games. It also compares the results of different tests with these algorithms such as varying the hyperparameters of the network. By observing the effects on the trained agents¹ when the hyperparameters are changed we can deduce a set of optimal values such that the networks can play three different Atari 2600 games. Overall, the main features of the project are the following:

- Agents with raw pixel game data as input, outputting a set of values for the best action.
- Agents attempt to find an optimal model of the environment without any prior knowledge.
- Visualization of the agent “brain” to provide insight into what information the agents is learning.

1.1 Motivation

Over the past 10 years there has been significant improvement in the RL (reinforcement learning) algorithms. One reason is that the computing power has become cheaply available by using discrete graphics cards. For example, for my project I used a Nvidia GeForce GTX 1070 that provides 1920 CUDA cores that can be used to accelerate training of neural networks. Despite this, RL algorithms are massively computationally expensive and hence take a long time to train.

Over recent years one of the pioneers in this area is DeepMind, which was acquired by Google in 2014, and they developed the DQN (deep q-network) algorithm in 2013 which they demonstrated could learn directly from the raw pixel data of games in order to achieve either human-level or super-human level performance.

This research was expanded upon by DeepMind and OpenAI which is based on the original DQN by DeepMind. This research focused on trying to approximate a Q-function and thereby infer the optimal policy. On the otherhand, there has recently been a focus on other methods such as A3C and PPO which instead seek to directly optimise in the policy space of the environment.

¹ Agent. In this case, agent refers to a trained neural network that takes actions in a chosen environment.

1.2 Objectives

Further to what was described in section 1 there was a few main objectives of the project. Firstly, I chose three games on which I decided to train the agents, Pong, Breakout, and Space Invaders which are shown below in Figure 1.1.

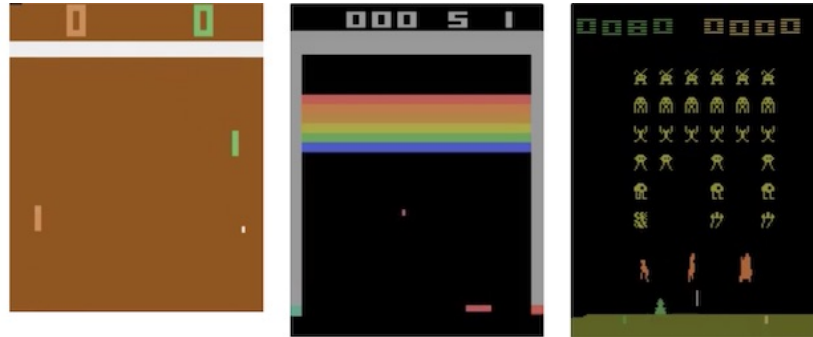


Figure 1.1: Screenshots of Pong, Breakout, Space Invaders (left to right).

Secondly, I wanted to find a way to explore the internals of a trained agent, in order to give further insight into what the agent is trying to learn. The reason for this is a researcher could use this information to determine, for example, where the agent has learnt to focus on the frame. Additionally, it provides a insight into how to optimally tune the hyperparameters which is described in section TODO: include section.

TODO: include gantt chart

1.3 Report structure

My report is divided into three main sections. Firstly, describing the background of the problem, then going onto giving details of my implementation and finally project evaluations/conclusions.

Chapter 2

Background

This chapter will cover some of the background material required for the following sections, it will cover the history of reinforcement learning (RL) and it's evolution to the current state-of-the-art. Additionally, it will cover the related work to this project and also cover some details of the past research papers for which this project has been based upon.

2.1 Reinforcement learning

Reinforcement learning is an area of machine learning has has been under active research since the late 1980s (TODO: ref watkins phd). The first defining algorithm of RL was called '*temporal-difference learning*' often referred to as TD-Learning. This algorithm learns by bootstrapping from the current value function in order iteratively converge towards a optimal policy (i.e. the agent's strategy for taking actions in the environment).

Further work by R. Sutton led to the development of TD-Lambda, an algorithm that was applied to the game of Backgammon, in 1992, by Gerald Tesauro to create TD-Gammon (TODO: ref Tesauro paper). It was a computer program that was shown to compete at expert-human level. The program also found novel strategies that were either unexplored, or dismissed in error as poor strategies. This was the first example of RL aiding in discovery or reconsideration of board game strategies. This trend of RL algorithms helping to improve human play would prove to only continue with DeepMind's AlphaZero program mastering the games of Chess (beating the strongest Chess programs such as Stockfish¹) and Go.

2.1.1 Deep reinforcement learning

Following from section 2.1 on RL, this section talks about the combination of two areas, deep learning and reinforcement learning methods, called deep reinforcement learning (DRL). Deep learning (DL) is a common class of machine learning methods that has been of much research focus over the past decade and can deal with high-dimensional sensory input; for the case of Atari this is 84x84 greyscale images after pre-processing of the raw Atari frames. On the other hand, reinforcement learning allows us to create an agent which can learn an optimal policy to navigate some environment in order to optimise its reward.

Through the combination of these methods, it has proven to provide solutions to previously intractable problems [1] in areas such as robotics, computer vision and healthcare. For example,

¹Stockfish. One of the strongest Chess programs based on the CCRL ratings list.

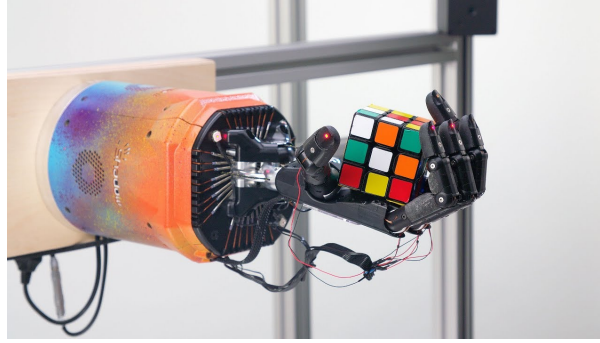


Figure 2.1: OpenAI Robot solving a Rubik's cube

in 2019 OpenAI developed a robotic hand that could solve a Rubik's cube 2.1, trained using deep reinforcement learning. As an extension to DRL, end-to-end reinforcement learning is a method for single layered neural network, trained by reinforcement learning. Figure 2.2 shows, diagrammatically, how DL and RL are used together in order to produce a single end-to-end model. In this simple architecture, there are two main components, the agent and the environment. This is a key feature of all DRL methods, an agent observes some state and reward from the environment after taking an action.

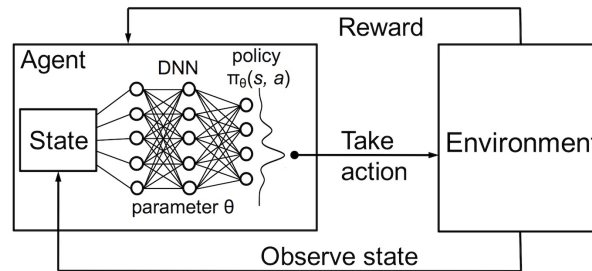


Figure 2.2: Representation of end-to-end RL architectures

As was mentioned in Section 2.1 there has been a renewed focus on the area of DRL. This research trend originated in 2013 when DeepMind showed that using a combination of Q-Learning and deep learning, it can produce agents that can compete with expert-humans in Atari.

2.2 DQN on Atari 2600

As mentioned in previous sections, one of the pioneering companies in the area was DeepMind. In 2013, V.Mnih et.al. while working at DeepMind, released a paper titled “*Playing Atari with Deep Reinforcement Learning*”[3]. This paper combined DRL, convolutional neural networks and a novel strategy called **experience replay**. DeepMind, in 2014, patented Q-Learning and it's application with deep learning on Atari games. Further, they also published further papers expanding on the idea in prestigious journals such as NIPS and Nature. Figure 2.3 shows the structure of a Deep Q-Network as used to play Atari 2600 games.

Experience replay was an important improvement that helped in stabilising the Q-Learning algorithm. It does this by removing the correlations between a observation sequence, i.e. we

don't learn from a chronological series of frames, rather we store all the experience and learn using random samples of this memory.

Over the following few years, this area would see rapid progress with many different advancements by both DeepMind and OpenAI. As was noted with the introduction of Deep Q-Learning (often referred to as '*Deep Q-Networks*' or *DQN*) the algorithm suffers from over-estimating the value of some actions. This can lead to a poor performance on more complex Atari games such as Space Invaders. Further detailed descriptions of the Q-Learning algorithm is provided in Section 3.3.

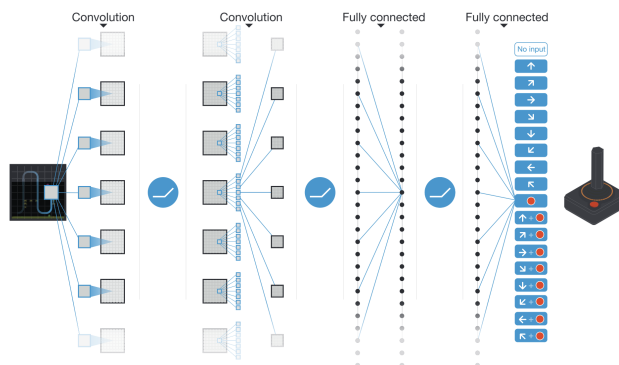


Figure 2.3: Deep Q-Network architecture that from left-to-right is showing how a single frame is processed. First passing through convolutional layers, before being flattened to a single tensor which is connected to a fully connected network. This MLP acts as the Q-function approximator which will produce an output of predicted Q-values for each action.

2.3 CNN Visualisation

Chapter 3

Design

This section will cover the details of the methods and algorithms that were used in the implementation of the project. The previous section covered the history of some of these techniques which were imperative for building the foundation of the methods described in this chapter.

Deep Q-Networks and its enhancements have the majority of focus, as it is the basis of the project. Below is a table of the different experiments and the methods that were used in order to produce the best solution for each tested environment.

Network	Algorithm	Environment
CNN + MLP	Q-Learning	Pong
CNN + MLP	Q-Learning	Breakout
CNN + MLP	Duelling Double Q-Learning	Space Invaders

Table 3.1: Best performing method in given environment

3.1 Markov decision process

This section will describe some of the basics of Markov decision processes (MDP), they are the foundation of the reinforcement learning methods that will be described later in this chapter. First, we need some mathematical definitions of MDPs, these are from David Silver's excellent lecture series on Reinforcement Learning.

Markov decision processes are just markov reward processes with decisions, i.e. At each state S_t , we have a finite set of actions to choose from in order to get to a new state S_{t+1} .

Definition 3.1. A Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$.

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

The definition above defines a Markov decision process, which we use as a basis for describing the methods in reinforcement learning. In order to illustrate the idea, let us consider the game of Pong. For simplicity, assume we can encode each frame of the game into the set of states \mathcal{S} . In order to play the game, we need to know what the best action to take would be at each frame of the game to move the paddle under the ball, hitting the ball back and (hopefully) scoring a point.

Given a frame of the game S_t and the set of actions we can choose from A_t , we are going to try and maximise our future (expected) reward using the reward function $\mathcal{R}_{S_t}^{A_t}$. We want to choose the best action a , that will result in the maximum future reward. It is important that we don't look at immediate rewards only, since, in Pong we don't get the point until we have hit the ball back to the other side.

In order to look at future rewards, we use the discount factor γ . When γ is close to zero, we are “*myopic*” in our evaluation (we only look for short-term rewards). However, as γ gets closer towards 1, we are “*far-sighted*” in our evaluation.

Overall, we need to know a strategy that provides the best action to take in a given state which maximises the expected total reward. This is called a *policy*, denoted by π .

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

$$\pi(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

The policy of an agent fully describes the behaviour of the agent (TODO: add ref) which only depends on the current state, not the history. In order to describe the optimal policy for an agent to follow, we first need some more definitions.

Definition 3.3. G_t is the total discounted reward for time-step t

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

Definition 3.4. The *action-value* function $q_{\pi}(s, a)$ is expected return starting from state s , taking action a , 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 is denoted by $q_*(s, a)$ and is the maximum action-value function over all possible policies

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

Once we have found the optimal action-value function we consider the MDP “solved”. Additionally, we know that we can, given some state, take actions that will lead to the highest possible future reward.

3.1.1 Markov property

With Markov decision processes we assume that in each state, we have all the information we require in order to produce an optimal action. However, in games such as Pong and Breakout, we may not necessarily have all the information we require at a single time. For example,

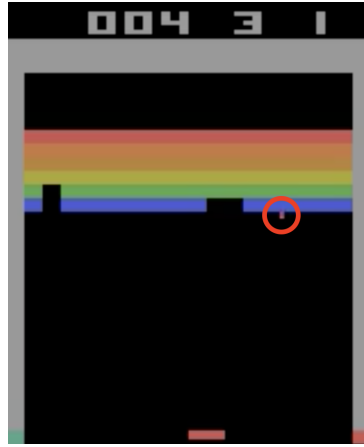


Figure 3.1: Screenshot of a breakout frame with the ball highlighted showing it just before the brick is destroyed. The ball is highlighted using a red circle.

Figure 3.1 shows a screenshot of the Atari game Breakout. In this situation, we cannot discern the optimal action given only this frame as the ball has only just hit the brick.

In order to solve this problem, we store a small rolling history of k states (where typically, $k = 4$) called the “*frame-stack*”. During forward passes through the network we pass the whole frame-stack to the network. This provides a history of frames to the network which has been shown, experimentally, to improve the learning of the algorithm.

3.2 Reinforcement learning

Following on from the previous section on Markov decision processes, this section describes Reinforcement learning and how these two methods are tightly connected to each other.

In its basic form, RL can be modelled graphically as shown in Figure 3.2. The agent gets the state from the environment, using its policy, it chooses an action to take – updating the environment. The environment then produces some new state and a reward signal which the agent uses to pick the next action.

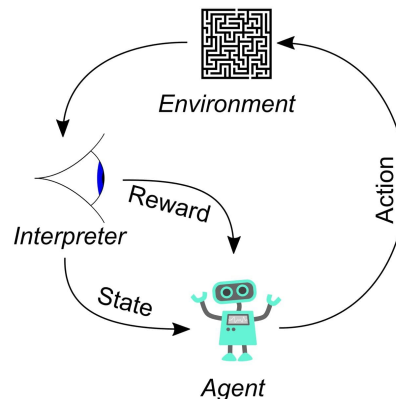


Figure 3.2: Diagram of reinforcement learning

3.2.1 Exploration vs Exploitation

A key idea in RL is the problem of exploration vs exploration. This means that if we have an environment, and a policy that dictates how we should navigate the environment, should always follow the policy, or should we deviate and try to find a better path resulting in a higher reward.

In this project we follow a method called ϵ -greedy in which we explore forever, but with a linearly decreasing probability (denoted by ϵ) of random actions, this is decreased over a predefined number of timesteps. Below is the method for choosing the actions.

- Choose random action with probability ϵ
- With probability $1 - \epsilon$ select action $= \arg \max_{a \in \mathcal{A}} \hat{Q}(a)$

Although ϵ -greedy is one of the simplest and a naïve method, it is surprisingly effective in exploring the environment and produces high scoring models. In fact, this method was used in the original DQN paper by V.Mnih et. al. in which they scored better than human averages in the majority of Atari 2600 games.

In the area of reinforcement learning, there are some more advanced methods such as Upper Confidence Bounds (UCB) and Thompson Sampling. However, these Bayesian-based methods are not implemented/explored in this project and is left for future work.

3.3 Q-Learning

Q-Learning is a model-free algorithm that is used to solve, iteratively, the Bellman equation for the MDP. By model-free we mean that the algorithm does not require a model of the environment, therefore it can easily handle stochastic rewards. The method was introduced in 1989 by Chris Watkins in his PhD thesis titled “*Learning from delayed rewards*”.

We can use Q-Learning in order to approximate $q_*(s, a)$, this produces the optimal policy π based on the Q-values. The Q-Learning algorithm stores the state-action values in a large table of values, as represented below in Table 3.2. For each possible state that the environment can produce, we store a single value (Q-value) for each action that can be performed in that state. These Q-values represents the ‘quality’ of being in a state s and taking action a .

Q-Table		Actions			
		NOOP	FIRE	LEFT	RIGHT
States	0	0	0	0	0
	1	-2.3452	-1.8375	-2.3634	-1.5463
	\vdots	\vdots	\vdots	\vdots	\vdots
	128	2.4456	1.2345	6.3462	3.8356

Table 3.2: Example of how state-action pairs are stored in a Q-Table (NOOP abbreviates ‘No action/No operation’)

Since the Q-Learning algorithm is iterative, at each timestep, we need to update the Q-value corresponding to the occupied state and the action we took. In Chapter 3.5 this idea will be expanded upon, by using neural networks as a function approximator, replacing the need to

store the whole table of Q-values. Algorithm 1 shows how the algorithm is implemented in order to both, predict the next action to take, and updating the Q-table. In order to update the Q-values for each state we use the Q-value update equation which is shown below.

Definition 3.6.

$$Q(s, a) = Q(s, a) + \alpha(R + \gamma \max_a Q(s_{t+1}, a) - Q(s, a))$$

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

A major issue with Q-Learning is that since we perform a maximisation of all future rewards when updating the Q-values, this can lead to the algorithm over-estimating the quality of actions and slowing down the learning. During training, we use the same Q-function in order to both produce the expected maximum action-value of future rewards and, select the best possible action in the current state.

Algorithm 1: Tabular Q-Learning

```

Initialize action-value function  $Q$  with random values
for  $timestep = 1$  do
    Initialise  $S$ 
    while  $S$  is not done state do
        With probability  $\epsilon$  choose random action  $a_t$ 
        Else, pick  $a_t = \max_a Q(s_t, a)$ 
        Take action  $a_t$  and observe reward  $r_t$  and state  $s_{t+1}$ 
         $Q(s, a) = Q(s_t, a_t) + \alpha(R + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$ 
        Update  $s_t = s_{t+1}$ 
    end
end

```

As shown above, we can use Q-learning to find the optimal policy by storing all the $q_*(s, a)$ values. However, in practise we run into a issue with the size of the Q-table in that it quickly becomes too big to store in memory. This issue is especially prevalent for large MDPs such as those from Atari games due to the raw pixel input. In order to handle the high-dimensional input from the games, we can instead use a function approximator to estimate the values of $q_*(s, a)$. This idea will be developed in Section 3.5 where RL is combined with Deep Learning.

3.4 Deep Learning

This section will describe the basics used in the project for Deep Learning and some of the techniques used in order to construct the Deep Q-Networks discussed in Section 3.5.

3.4.1 Multi-layer perceptron (MLP)

The main technique that underpins most of the artificial neural networks (ANN) that are used today is how we combine single neurons to form networks of neurons. Figure 3.3 shows how we model an artificial neuron with weights and activation functions, this is supposed to loosely model how neurons work inside the brain. The input connections are acting in place of synapses which connect together to form networks.

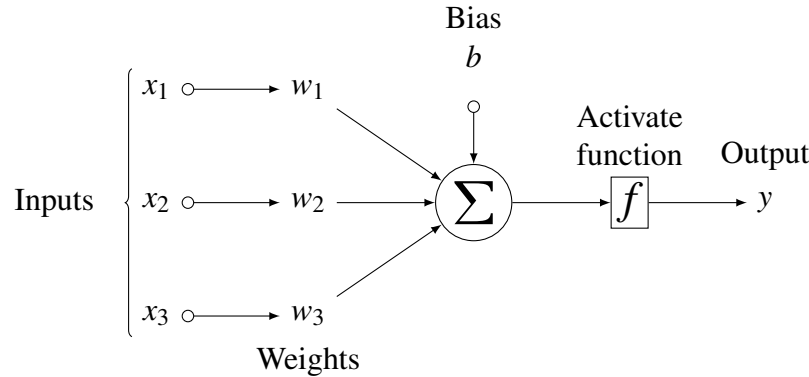


Figure 3.3: Schematic the modelling of a single neuron

The artificial neuron takes inputs $x_1 \dots x_n$ with corresponding weights $w_1 \dots w_n$. Usually, the bias b is stored in w_0 with $x_0 = 1$. During a forward pass of the network we process the input vector \mathbf{x} and producing an output y . In the center of the figure we have a single node that represents the function to calculate u which is the weighted sum of the input vector \mathbf{x} and vector of weights \mathbf{w} . The equation to calculate this quantity is $u = \sum_{i=0}^n w_i x_i$.

Finally, each neuron has an activation function associated, there are many possible options for these functions such as Logistic (Sigmoid), TanH and ReLU. Each of these functions defines a threshold for when the output u results in a 1/0 output in y .

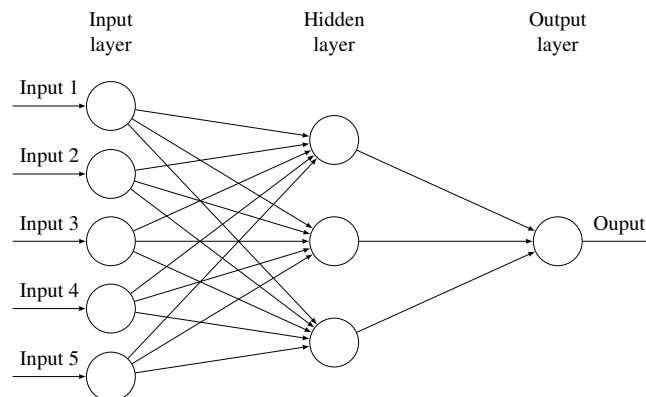


Figure 3.4: Diagram representation of MLP

Figure 3.4 shows the structure of an MLP (also called artificial neural network), we construct this network by combining multiple neurons together. The MLP consists of the n input neurons (each having a single input value), with then at least one hidden layer. The last hidden layer is connected to the output node/nodes.

In general, for each node in the ANN, it will be connected to every other node in the following layer, whether that is another hidden layer or the output layer. This network structure is referred to as “*fully connected*” as every node in a layer is connected to every other node in the next layer. One downside to this ‘fullyconnected-ness’ is that it tends to make them prone to overfitting to the input. Therefore, many techniques have been developed in order to reduce overfitting. Some examples are using regularization, dropout layers¹, early stopping and batch normalisation. This is not the only type of network consisting of artificial neurons; by connecting the neurons in different ways the networks exhibit different behaviours such as recurrent neural networks (RNNs) that have a small ‘memory’ capacity due to the network structure.

3.4.2 Convolutional Neural Network (CNN)

Neural networks are excellent approximators for complex functions that need to be learnt, however, when we have high-dimensional inputs the size of the network can become too difficult to manage. An important note is that any CNN can in theory be implemented as a neural network individual neurons. However, if we take as an example a single frame from Atari which is 210 x 160 pixels each with RGB values. This results in having over 100,000 weights on just the input layers and as such, too many weights to process in an efficient manner.

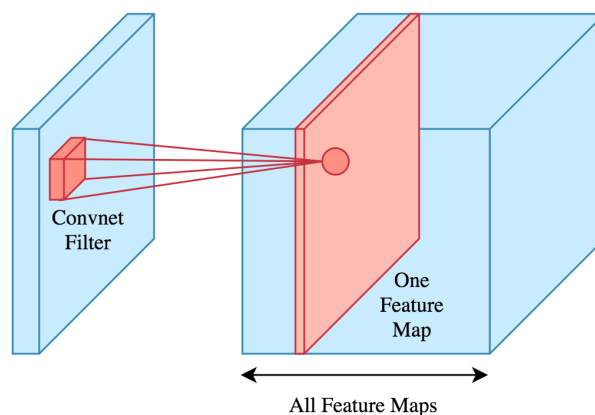


Figure 3.5: Single convolutional layer using filters

In addition, CNNs can exploit the spatial structure of the data by using different building blocks in order to construct the network. Some of the layers help to avoid problems such as overfitting which is a key problem when training these networks. Others such as ReLU layers help reduce the training time since it simplifies the activation functions with having too much impact on the overall network accuracy. Below is a list of the most common layers used in CNNs with a short explanation the role of each.

- **Convolution.** Layer with filters that have learnable weights. Computing output is dot product of each filter with the input.
- **Pooling.** Layer that acts as non-linear downsampling of the input. Helps control overfitting by reducing size of network and number of learnable parameters.

¹Dropout layer. A layer in a ANN that randomly ignores the input from the previous layer with a given probability. It aids in preventing neurons from becoming dependent on other neurons in previous layers, thereby reducing the effect of overfitting.

- ReLU. Layer which applies an activation function to the input of the layer that removes negative values from the input by setting to those values to zero. [4] Improves speed of training the network.
- Fully-connected. At the end of the network, a fully-connected layer can be added in order to perform high-level decisions such as predicting MNIST digits. This is usually an MLP (described in Section 3.4.1).

By carefully combining these different layers, we form a convolutional neural network. The main applications are in image recognition and visual processing such as images and video; in this project we a CNN is used to process frames from Atari. As previously touched upon in Section 2.3, we can view what information the network has chosen to learnt by looking at the activation in each layer, and each filter. It can provide insight into how well the network is learning and if it requires fine-tuning.

Figure 3.6 provides an example network that is used to predict the value of a handwritten digit from the MNIST² dataset. It uses a series of convolution (CONV) and pooling (POOL) layers before connecting to a fully-connected network (with 10 outputs) which is used to perform the final prediction of the number.

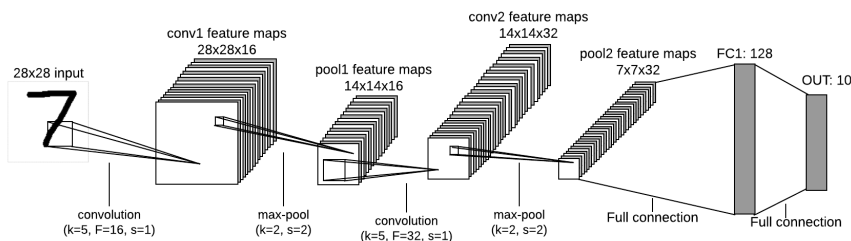


Figure 3.6: Architecture of a network to predict MNIST digits

3.5 Deep Q-Learning

The following sections will outline how the previous sections have laid the foundation for introducing Deep Q-Learning. It is the method by which the computer can learn to play the Atari games to human-level. It will use the information from the previous sections and also introduce some new concepts such as Experience replay which is key to the learning process.

In this project we use neural networks as the function approximator for $Q(s, a)$, we represent this by using θ to represent the parameterisation of the Q-function, $Q(s, a; \theta)$. Referring back to Section 3.1, the aim of Q-learning is to find a function $Q(s, a; \theta)$ that closely approximated the optimal action-value function $Q_*(s, a)$. The *value iteration* algorithm will converge to the optimal action-value function, $Q_i \rightarrow Q_*$ as $i \rightarrow \infty$ [6]. In practise however, we need to use a function approximator as it can generalise the solution, given by $Q(s, a; \theta) \approx Q_*(s, a)$.

In order to train the network, we need to define the ‘loss’ function for the Q-function. This will allow us to update the weights of the network in the direction of the gradient, bringing the Q-function closer to $Q_*(s, a)$ with every timestep.

²MNIST. Large database of 60,000 training image of 28x28 images showing handwritten digits from 0-9

Definition 3.7. The loss function of the Q-network is defined by

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

where $y_i = \mathbb{E} [r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ is the target value for iteration i and the function $L_i(\theta_i)$ is updated at every timestep. It is also important to note that we hold the parameters of the network fixed from the previous iteration θ_{i-1} in order to stabilize learning.

By differentiating the loss function defined above, with respect to the weights of the network (θ), we get the following gradient.

Definition 3.8.

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

In practise, when implementing the Q-learning algorithm, it is more computationally efficient to not compute the full expectation of the gradient, rather we optimise the loss using stochastic gradient descent. This leads us onto the Deep Q-Learning algorithm which is presented below.

Algorithm 2: Deep Q-Learning with Experience replay

```

Initialize replay memory  $D$  with maximum capacity  $N$ 
Initialize action-value function  $Q$  with random weights  $\theta$ 
Initialize target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
for  $timestep = 1, T$  do
    Initialise  $S$ 
    while  $S$  is not done state do
        With probability  $\epsilon$  choose random action  $a_t$ 
        otherwise, pick  $a_t = \max_a Q(s_t, a; \theta)$ 
        Take action  $a_t$  and observe reward  $r_t$  and state  $s_{t+1}$ 
         $Q(s, a) = Q(s_t, a_t) + \alpha(R + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$ 
        Store tuple  $(s_t, a_t, r_t, s_{t+1})$  in  $D$ 
        Randomly sample a minibatch of tuples  $(s_t, a_t, r_t, s_{t+1})$  from  $D$ 
        Every  $C$  steps copy weights  $\theta^- = \theta$ 
    end
end

```

3.5.1 Experience Replay

One of the problems with the Q-learning algorithm is that if we try and learn from consecutive frames, the algorithm will take a long time to converge. This is due to the large correlation between consecutive frames of the Atari games. By splitting up the training samples it breaks this correlation and reduces the variance of the gradient updates.

This approach is known as “*experience replay*” [2]. Every timestep, we store the tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ of experience in the replay memory which is denoted by D in Algorithm 2. Therefore, over time we accumulate a collection of experiences such that $D = e_1, e_2, \dots, e_N$.

In practise, we keep a rolling collection of experiences for a memory with capacity N ; in this project the experience is stored in a Double-Ended Queue³. It means that at timestep $N + 1$, we remove the tuple e_1 and replace it with tuple e_{N+1} .

³Double-ended queue is an abstract data type that can generalise a queue. It typically provides operations to add/remove elements from the head and tail.

During the Q-learning updates, we take a mini-batch of samples, $e D$, randomly from the replay memory in order to perform the stochastic gradient descent. This method comes with several benefits that are outlined below.

- We randomly sample the replay memory at each step, we can reuse experience samples multiple times, providing much greater data efficiency.
- As mentioned previously, there is a strong correlation between consecutive frames, experience replay provides a method to remove this correlation.

Despite the benefits that experience replay brings, this approach has limitations. The main being that each experience sample is treated with the same importance i.e. each tuple has an equal probability of being chosen in a sample e . Therefore, in 2015 Schaul et.al [5] introduced a new method called “Prioritized Experience Replay” which aimed to ensure that more important experiences are replayed with a greater frequency.

3.6 Q-Learning improvements

As discussed in Section 3.5, due to how Q-learning works, it can lead to an overestimation of the quality of some actions. Therefore, we need a way to help improve the stability of the algorithm by reducing this overestimation. This section describes two different approaches in order to improve upon the algorithm and reduce the maximisation bias.

3.6.1 Double Q-Learning

3.6.2 Dueling Q-Learning

References

- [1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866, 2017.
- [2] L. ji Lin. Reinforcement learning for robots using neural networks. 1992.
- [3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013.
- [4] V. Romanuke. Appropriate number and allocation of RELUS in convolutional neural networks. *Research Bulletin of the National Technical University of Ukraine "Kyiv Politechnic Institute"*, 0(1):69–78, Mar. 2017.
- [5] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015.
- [6] R. S. Sutton. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning series)*. A Bradford Book, nov 2018.