Playing video games using neural networks and reinforcement learning

Joseph Shihab Esmaail

May 2016

Abstract

This is a Technical Computer Science project, which aims to develop a neural network, trained with reinforcement learning, that is able to learn to play video games. This document outlines a background of the related work and theory, the management of the project itself, the technologies used, an analysis of the problem, how the problem was then implemented, and a reflection on the results as well as how it could be improved in future work.

Project Dissertation submitted to Swansea University
in Partial Fulfilment for the Degree of Bachelor of Science

Prifysgol Abertawe
Swansea University

Department of Computer Science
Swansea University

Declaration

This work has not previously been accepted in substance for any degree and is not being currently submitted for any degree.

May 5, 2016

Signed:

Statement 1

This dissertation is being submitted in partial fulfilment of the requirements for the degree of a BSc in Computer Science.

May 5, 2016

Signed:

Statement 2

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work, and pages using the bibliography/references. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure of this dissertation and the degree examination as a whole.

May 5, 2016

Signed:

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

May 5, 2016

Signed:

Contents

1 Introduction

Neural Networks are a model within the field of Machine learning which is currently a major buzzword in the Computer Science Discipline, owing a lot to the recent growth of big data. It has also gained increasing public presence through efforts such as IBM's Watson, which won against two former champions in Jeopardy in 2011, and Google's DeepDream program which displayed how convolutional neural networks interpret images.
Machine learning is also a major player in our day to day lives with Apple, Microsoft, and Google all using it for their smartphone digital assistants, Google using it for their search engine, and social media such as Facebook using it to cater it's service to each individual user. This will also extend in the coming future with autonomous vehicles such as Google's self-driving car project and motor companies such as Tesla starting to incorporate self-driving systems into their cars.

Very recently, Google DeepMind created a program known as AlphaGo which was able to play Go, a board game that originated more than 2500 years ago and has proven difficult for the application of computer players. The AlphaGo program did not just play Go however, it managed to seize a victory over a human champion play, winning four out of the five games played.

This project aims to apply machine learning, in particular neural networks, to the task of learning to play video games. Video games have been an almost constantly growing phenomenon since their inception in the 1970's with Pong, recently entering the mainstream with the growth of the eSports industry which hosts tournaments with prize pools in the region of a million US dollars. The application of machine learning to video games has the potential to enable a greater interest in machine learning, as it's current implementations are very pervasive.

This dissertation will provide the reader with a background to neural networks, reinforcement learning and relating work to these and their application to video games, as well as how this project has been organised, what technologies are utilised, how the task has been approached and implemented, as well as what results are produced.

1.1 Aims and objectives

To be able to successfully produce a neural network that can play video games, the network will need a way to interact with the video games, both to receive information about the current game state and to send actions into the video game to play. So we will need methods to obtain the outputted game screen data and current score, as well as methods to input actions as a human player would via a game controller. The neural network needs to be able to interpret

the input game state, and output a decision on what action to perform, this action hopefully being one that results in the best performance or score in the video game. The neural network will determine what action is the optimal one after performing a learning process using reinforcement learning, on the particular video game in question. After the network is capable of performing optimal actions to the best of its ability, the performance will be evaluated against human players to make a decision on how successful it was at learning to play the video game.

To summarise, the aim of this project is to develop an artificial intelligence agent, namely a neural network, that will use reinforcement learning to learn to play a large set of video games. This can be achieved by splitting the overall task into the following steps:

- Find methods to manipulate video game inputs and outputs.
- Develop the neural network.
- Develop reinforcement learning methods for the intelligence agent.
- Train and evaluate the neural network with video games.
- GPU-Accelerate the network to improve performance.

2 Background

2.1 Literature review

An early use of a neural network to play a game can be seen in the paper 'Temporal difference learning and TD-Gammon' by Gerald Tesauro. In this, Tesauro applies the neural network (known as TD-Gammon) to the game backgammon, and it trained itself using reinforcement learning, in particular *temporal-difference learning*. The neural network would train itself "by playing against itself and learning from the outcome" [20]. Tesauro concludes that the temporal difference learning proved to be "a promising general-purpose technique" [20]. TD-Gammon successfully learnt to play backgammon; successfully enough that "by version 2.1 TD-Gammon was regarded as playing at a level extremely close to equalling that of the best human player, and had even started to influence the way expert backgammon players played" [20]. This particular implementation of reinforcement learning to neural networks however would not be possible to implement across a broad selection of games as it requires a game with two players.

Neural networks and reinforcement learning have also been applied to obstacle avoidance in robots, which is a similar environment to that of video games. In the paper 'Reinforcement learning neural network to the problem of

autonomous mobile robot obstacle avoidance' by Huang et al. reinforcement learning was chosen because "a complete knowledge of the environment is not necessary" [10]. When a human player starts to learn to play a video game, they also do not have a complete knowledge of all the elements of the game. In this case the neural network needs to be learning the video game in a similar fashion to how a human player would, so reinforcement learning is applicable here. The results of the paper show that using neural networks and reinforcement learning can enhance learnability and allow task completion in a complex environment, which can also be translated across to the application to video games.

In 2007 the paper 'Reinforcement learning and neural networks for Tetris' by Nicholas Lundgaard and Brian McKee saw the application of neural networks and reinforcement learning to the popular 1984 video game Tetris. They observe that video games are well suited for the application of reinforcement learning as more often than not there is a built in system to track how well a player is doing, often known as the score or points of the player. They also make the observation that in the case of Tetris "it is nearly impossible to definitively say what was a right or wrong move" [11], which can be applied also to most video games and means that the task of learning for neural networks applied to them cannot be done through supervised learning. In practice they found that "while the learning agents fail to accumulate as many points as the most advanced AI agents, they do learn to play more efficiently"[11]. Lundgaard and McKee conclude that "compared to human performance, the learning agents appear to perform well: while they can't survive forever, they perform better during game-play"[11]. This suggests that an AI player doesn't necessarily gain an unfair advantage over a human player by being capable of surviving indefinitely, they instead play at a similar pace.

An example of an application of machine learning to video games bringing more public attention to the field of machine learning is 'MarI/O - Machine learning for video games' [16]. This learning agent developed by popular YouTube content creator 'SethBling' is featured in a video which is at the time of writing this document the eighth result to the search term 'machine learning' on YouTube, and the top result of that search term that is actually relevant to machine learning. The video game that the agent is applied to in this case is *Super Mario world* for the Super Nintendo Entertainment System, run within the Bizhawk emulator[19]. The agent in this implementation uses a genetic algorithm known as NeuroEvolution of augmenting topologies, or NEAT. This allows for both the weights and the topology of the neural network to be altered as the network learns. The way in which this agent makes decisions on the changes in the topology and weights is loosely based off of Darwins theory of natural selection and evolution. Multiple networks are created, each with varying mutations, and the most fit network for the applied task will be the one carried forward to the next generation. While I am not using NEAT for the purposes of my implementation, this related work demonstrates the ability for video games to being attention the the field of machine learning, and gives

some insight to methods other than reinforcement learning for this problem.

The most recent and relevant work in developing neural networks to play video games comes from DeepMind, who were recently acquired by Google. Initially in their paper 'Playing Atari with deep reinforcement learning' in 2013 and then expanded upon in 2015 in 'Human-level control through deep reinforcement learning', a paper which made the front page of Nature. In these papers they refer to their neural network as a Deep Q network (DQN), which is a deep convolutional neural network trained using a variant of Q-learning. This network is then applied to Atari 2600 games via the *Arcade learning environment*, which was also developed by DeepMind. A convolutional neural network was chosen here because it "mimics the effects of receptive fields"[13], so it is more equipped for the task of pattern recognition within images. A deep network was chosen because "several layers of nodes are used to build up progressively more abstract representations of the data"[13], allowing the network to "learn concepts such as object categories directly from the raw sensory data"[13]. The results of this paper show that their "DQN method outperforms the best existing reinforcement methods on 43 of the games" [13]. As well as this they report that their agent "performed at a level that was comparable to that of a professional human games tester" [13].

A selection of the DQNs performances are shown in figure 1, the values for which are "normalised with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level)." [13]. Figure 13 in the appendix displays the performance for all 43 games.

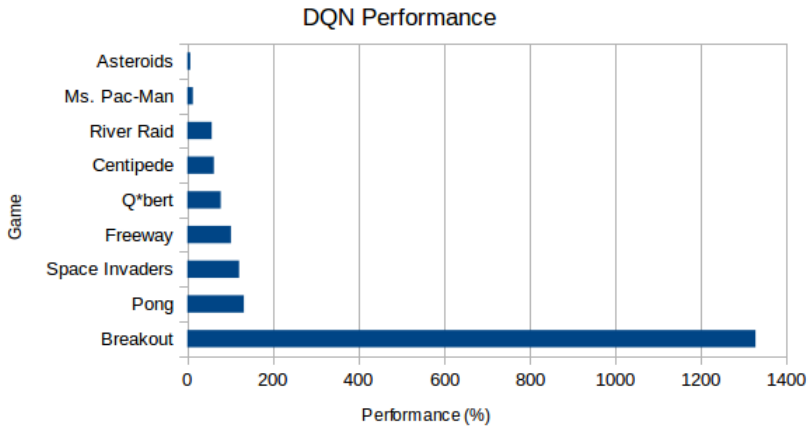

Figure 1: DQN performance with regard to human performance [13].

This related work confirmed that a neural network is a suitable model of machine learning to apply to the task of learning to play video games. In par-

ticular a deep convolutional neural network trained with a Q-learning variant has been proven as capable of learning to play multiple video games as opposed to just one, with no changes to the structure of the neural network.

2.2 Neural networks

Neural networks (also known as artificial neural networks) are machine learning models based on the structure of the brain in animals such as humans.

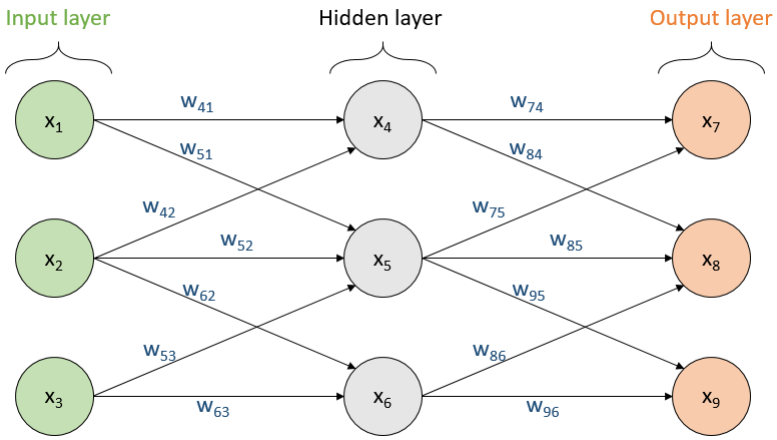

Figure 2: An example neural network.

These neural networks are, naively, a collection of connected nodes, where the nodes are referred to as neurons, and the connections between them as weights. The neurons of the network exist in layers, an input layer for inputs to the network, an output layer to give the results of the network, and potentially some hidden layers in between. If a neural network has more than one hidden layer that network is then considered a Deep neural network. A neural network is a feed-forward network if all of it's connections point towards the output layer, or a recurrent network if there exist feedback connections which point backwards in the network towards the input layer. For the purpose of this explanation I will be using a feed-forward network,

The input layer of the neural network receives its data from the input to the network, this data is then passed along weighted connection(s) to neurons in the next layer of the network. All layers of a network, with the exception of the input layer, have a bias unit as well as having a summation and an activation function performed on them before the neuron data is propagated forward through to

the next layer. The summation process for a neural is performed as follows:

$$x_i = \sum_j w_{ij} x_j + b \tag{1}$$

Where:

w_{ij} represents the weight for the connection from neuron j to neuron i.

x_j is the value of neuron j.

b is the value of the bias unit.

After this summation process is applied to a neuron an activation function is usually applied. The most commonly applied activation function, and the one i will use for this explanation, is a Sigmoid function which is visualised in Figure 3. A Sigmoid activation would be performed as follows:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2}$$

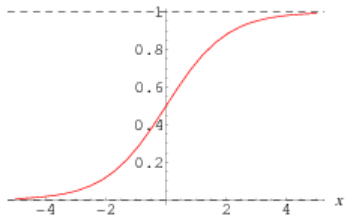

Figure 3: Graph of the Sigmoid function [3]

The summation and activation function for a neuron can be combined into one calculation for the final output value of a neuron as:

$$output(x_i) = \sigma(\sum_j w_{ij} x_j + b) \tag{3}$$

$$\sigma(z) \equiv \frac{1}{1 + e^{-\sum_j w_{ij} x_j + b}} \tag{4}$$

Once the values have been propagated to the output layer and the neurons in the output layer have had summation and activation performed on them, we have the output of the neural network.

2.3 Convolutional neural network

There are many different types of neural networks as well as just feed-forward and recurrent, for the purposes of this dissertation, I am using a convolutional neural network. Convolutional neural networks are designed to "recognise two-dimensional shapes with a high degree of invariance to translation, scaling, skewing, and other forms of distortion" [8]. These convolutional neural networks use the main concepts of shared weights and biases, local receptive field, and very often, max pooling. The input for this kind of network would be the pixels of an image, for example, a 25x25 greyscale image would have an input layer size of 25x25, a sequence of 3 25x25 greyscale images would have an input layer size of 25x25x3.

Unlike a normal neural network, convolutional neural networks have layers known as convolutional layers, which are either a 2 or 3 dimensional structure. These convolutional layers use local receptive fields which connect "the input pixels to a layer of hidden neurons. However we won't connect every input pixel to every hidden neuron. Instead we only make connections in small, localised regions of the input image"[15].

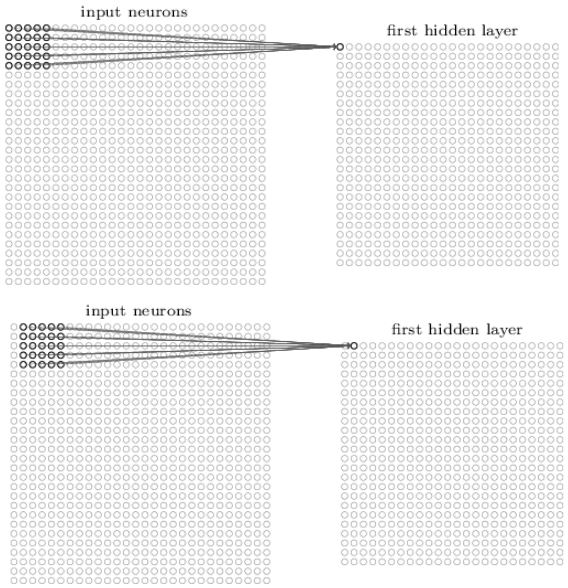

Figure 4: How local receptive fields work [15]

Figure 4 visualises how a local receptive field of size 5x5 with a stride length of 1 works. The stride length is the amount of pixels we shift the local receptive field when applying pixels to the next hidden layer neuron. So in the case of a stride length of 3, we would move 3 pixels to the right when moving the filter

across, and then 3 pixels down when moving the filter down the image. The size of the local receptive field is also then the size of the shared weights for the convolutional layer. The collection of these shared weights is known as a filter. Each filter sweeps across the pixels of the image, or neurons in the case of the network, where each weight of the filter will then be used for several of the pixels/neurons in the image/layer. Each filter is then also used on each image in the case of a multiple image input convolutional network, this will produce multiple processed images, known as feature maps, for each filter applied. For the next layer then, each filter is applied to each feature map, resulting in more feature maps being output.

The process for summation and activation for the result of a convolutional layer can expressed as [15]:

$$\sigma(b + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m} x_{j+l,k+m}) \tag{5}$$

Where a 5x5 local receptive field is being applied for the j^{th}, k^{th} hidden neuron with σ as the activation function, $w_{l,m}$ as the shared weights, and x represents the value of the input neuron [15].

Convolutional neural networks can also have pooling layers, which can be "used immediately after convolutional layers" [15]. While these layers do no actual learning, they can perform a process known as max pooling. A max pooling layer will take the output of the convolutional layer (size $N * N$), extract the maximum of a $k8k$ region, and output a $\frac{N}{k} * \frac{N}{k}$ layer [7].

A Convolutional network can be made up of a sequence of convolutional layers followed by pooling layers and then normal feed-forward neural network layers such as fully connected layers, where every neuron in the current layer is connected to every neuron on the previous layer. You can also omit the pooling layer, particularly in the case of applying to low resolution images, and just use a sequence of convolutional layers followed by normal feed-forward layers.

2.4 Backpropagation

The details covered so far only cover the forward propagation of a neural network, the network needs to also propagate backwards and tune parameters such as weights and biases to present a more meaningful output to given input, also known as learning. There are, generally speaking, three kinds of learning that can be applied in machine learning, supervised learning, unsupervised learning, and reinforcement learning.
Supervised learning requires "the availability of a teacher or supervisor who classifies the training examples into classes" [6]. For unsupervised learning the

network "must identify the pattern-class information as a part of the learning process" [6]. Reinforcement learning uses the idea of states, actions and rewards, and encourages the network to perform actions in states that produce the greatest reward.

To actually be able to utilise any of these learning methods, the neural network needs to apply an algorithm to actually perform the learning process, the most popular of which is backpropagation. The output of the neural network is highly unlikely to be correct on the first run of the network, due to the weights and biases of the network being assigned randomly at the start. We require a method to bring the actual output of the network closer to the desired output. This is backpropagation and is done using gradient descent.
Once the neural network has been run on several pieces of input data, we want to see how different the actual output is from the desired output. This error is calculated as [12]:

$$E_{total} = \sum \frac{1}{2}(target - output)^2 \tag{6}$$

$$E_{O_i} = \frac{1}{2}(target_{O_i} - output_{O_i}) \tag{7}$$

Equation 6 is for the total error of all output neurons, and equation 7 is for the error for one output neuron i.

The idea behind backpropagation is to find how much to change the weights and biases so that the actual output can be brought closer to the desired output. The process for this differs depending on if it is applied to an output layer, or to a hidden layer of the network.

When applying it to an output layer we want to find how much to change the weight w_{ij}, that connects a hidden neuron j to an output neuron i, to correct our output. So we need to calculate the change in E_{total} with regard to w_{ij}, this can be done with derivatives.

$$\frac{\partial E_{total}}{\partial w_{ij}} \tag{8}$$

$$\frac{\partial E_{total}}{\partial w_{ij}} = \frac{\partial E_{total}}{\partial out_{O_i}} * \frac{\partial out_{O_i}}{\partial net_{O_i}} * \frac{\partial net_{O_i}}{\partial w_{ij}} \tag{9}$$

Here we apply the chain rule to (8) to get (9) [12]. net_{O_i} is the net output of the output neuron i before the activation function has been applied.

Next we calculate the change in the E_{total} with regard to out_{O_i}, the change in out_{O_i} in regard to net_{O_i}, and the change in net_{O_i} with regard to w_{ij} [12].

$$\frac{\partial E_{total}}{\partial out_{O_i}} = -(target_{O_i} - out_{O_i}) \tag{10}$$

$$\frac{\partial out_{O_i}}{\partial net_{O_i}} = out_{O_i}(1 - out_{O_i}) \tag{11}$$

$$\frac{\partial net_{O_i}}{\partial w_{ij}} = out_{H_j} \tag{12}$$

$$\frac{\partial E_{total}}{\partial w_{ij}} = -(target_{O_i} - out_{O_i}) * out_{O_i}(1 - out_{O_i}) * out_{H_j} \tag{13}$$

In (11), the derivative of the activation function(sigmoid) has been calculated which is "the output multiplied by 1 minus the output" [12].

(13) can then also be written with the *delta rule* [12].

$$\delta_{O_i} = \frac{\partial E_{total}}{\partial out_{O_i}} * \frac{\partial out_{O_i}}{\partial net_{O_i}} = \frac{\partial E_{total}}{\partial net_{O_i}} \tag{14}$$

$$\frac{\partial E_{total}}{\partial w_{ij}} = \delta_{O_i} out_{H_j} \tag{15}$$

When applying backpropagation with gradient descent to a hidden layer, the process can be sped up by making use of the delta rule. We want to find the change in E_{total} with regard to w_{ij}, a weight connecting neuron j (can be in either input or hidden layer, depending on position in network) to neuron i in a hidden layer. [12]

$$\frac{\partial E_{total}}{\partial w_{ij}} = \delta_{H_i} * \frac{\partial net_{H_i}}{\partial w_{ij}} \tag{16}$$

$$\frac{\partial net_{H_i}}{\partial w_{ij}} = out_j \tag{17}$$

$$\frac{\partial E_{total}}{\partial w_{ij}} = \delta_{H_i} * out_j \tag{18}$$

Now to get our new value of w_{ij}, w_{ij}^+ we perform the following [12]:

$$w_{ij}^+ = w_{ij} - \eta * \frac{\partial E_{total}}{\partial w_{ij}} \tag{19}$$

With η being a learning rate for the neural network.

The learning method that will be used for this dissertation is backpropagation but implemented using a method known as RMSProp [9] a variant of Stochastic gradient descent. Stochastic gradient descent differs from normal

gradient descent by applying to a random subset of training data as opposed to all of it. This approach is advantageous when you have a very large training set and/or a large amount of weights to update. RMSProp then builds upon this, it will "divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight" [9]. This particular method has been chosen as it has been proven to work in the application of neural networks to video games in the paper 'Human-level control through deep reinforcement learning' [13].

2.5 Reinforcement learning

The learning method that will be utilised for the neural network in this project is Reinforcement learning. Reinforcement learning, as mentioned above, uses the idea of states, actions, and rewards to encourage the network to perform actions that produce the best possible reward for their current state. It is a learning model reminiscent of how animals learn, "learning what to do - how to map situations to actions - so as to maximise a numerical reward signal" [18]. A machine learning model using reinforcement learning is able to enter an unknown situation and "learn from it's own experience" [18].

2.6 Q-learning

The particular method of reinforcement learning that is implemented here is Q-learning. In Q-learning we attempt to find the optimal action for each state to maximise the reward gained. This kind of learning very closely matches the basic idea of how to learn to play a video game. The way that Q-learning works is that "for every possible state, every possible action is assigned a value which is a function of both the immediate reward for taking the action and the expected reward in the future based on the new state that is the result of taking that action" [21]. The value update function for this, as found in [21], is as follows:

$$Q(x,u) := (1-\alpha)Q(x,u) + \alpha(R + \gamma maxQ(x_{t+1}, u_{t+1})) \tag{20}$$

Where,

Q is the expected value of performing action u in state x,

R is the reward,

α is the learning rate,

γ is the discount factor.

This dissertation will use the neural network as a function approximator for Q-learning. This means that each input to the network (game screen data) will

be treated as a state, and the output of the network (the action to perform in the game) will be treated as an action, with an associated reward. The value update as seen in equation 20 will then be applied. The neural network used is therefore referred to as a Deep Q network.

3 Project Management

3.1 Software life-cycle

The software life-cycle used for this project is the agile development model, in particular the scrum model. The choice of agile for this project is advantageous compared to a more traditional model such as waterfall. This is because in the Waterfall model "as the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested" [17]. This would be particularly problematic and result in a prolonged development process if and when errors occur, especially in earlier stages as did occur in this project.

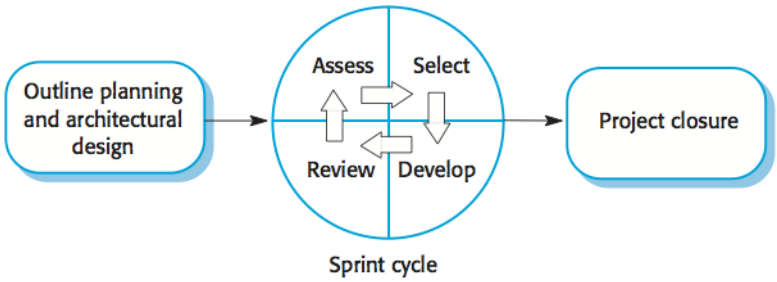

Figure 5: The Scrum development process [17]

The Scrum method is designed for small development teams, as a result many elements of scrum have been scaled down or omitted completely for application to this project. For example, there are several different roles for development team members such as a Scrum master, in this case the roles have been removed altogether.

Development in the scrum method is performed as displayed in figure 5, beginning with the outline planning stage. Here we establish the project's general objectives and design the software architecture. In this case, this stage was undertaken during the initial document. Next follows the main area of scrum, the sprint cycle. It is within this sprint cycle that the actual development is performed, and is done so in a series of fixed-length sprints. These sprints are then divided into four phases:

1. **Assessment phase**: A list of tasks to be performed on the project is reviewed, priorities and risks are assigned to them. In the review of tasks, more may be added.
2. **Selection phase**: The task or tasks to be performed this sprint cycle are chosen.
3. **Development phase**: Progress is reviewed at regular intervals, and task priorities can be potentially re-assigned.
4. **Review phase**: Tasks performed this cycle are now reviewed.

The final stage of scrum is then project closure, this wraps up the project with required documentation being produced and lessons learnt from the project are assessed. The project closure process is this final dissertation document for the project.

I feel that scrum has been beneficial to the development of this project as it broke it down into smaller digestible sprints. Each sprint had clear goals set out, prioritised and then reviewed. It has also meant that there is less chance for any delay in development as there have been regular stops to review and reassess the tasks being performed.

3.2 Project timeline

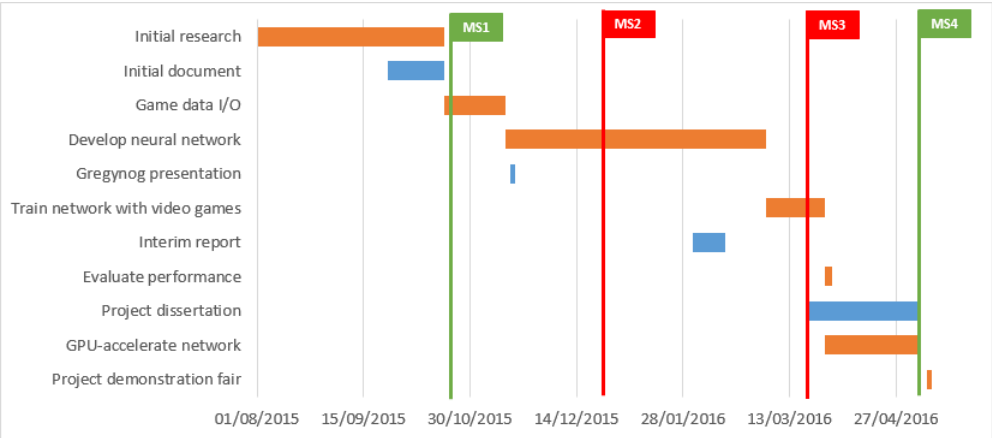

Figure 6: Planned project schedule.

Label	Milestone	Reason
MS1	End of initial phase, start of sprint cycle.	Scrum development model.
MS2	Start of Christmas break.	Allows more time dedication to project.
MS3	Start of Easter break.	Allows more time dedication to project.
MS4	End of project closure phase.	Scrum development model.

The project schedule, as seen in figure 6, has 2 kinds of tasks. Tasks set by the Computer Science department are those with blue bars, and tasks as set out by myself are those with orange bars. There are also 2 kinds of milestones, green milestones which represent the end of the intial and closure phases of scrum, and red milestones which represent the two holiday breaks in University term time. This breaks have allowed me to allocate more time to the development of the project and/or completion of documentation for the project. The bulk of the time for the project plan was dedicated to the development of the neural network in C++. This is both because it is the most integral component of this project, and because I required to learn the C++ language as I had no prior usage of it until this project.

The way that the project development actually took place can be seen in figure 7. Here the tasks of 'Train network with video games', 'Evaluate performance', and 'GPU-accelerate network' have been removed as the actual implementation of the network did not finish so these tasks could not then be performed.

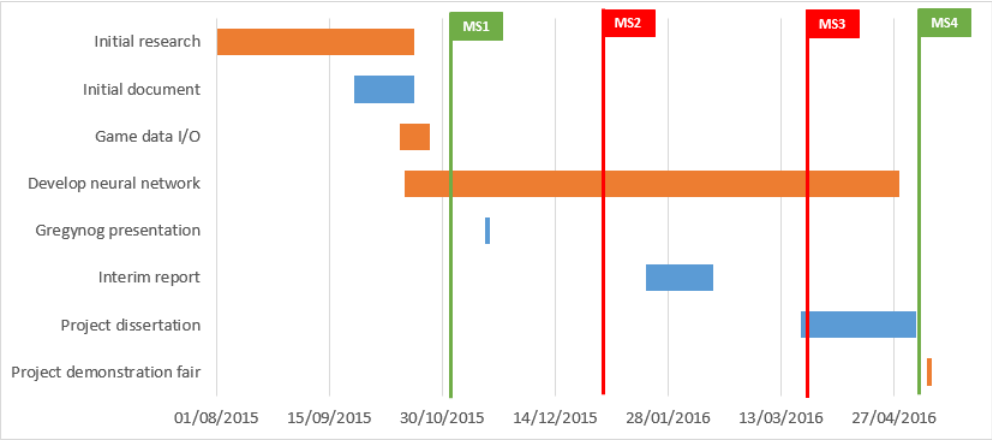

Figure 7: Actual project schedule.

3.3 Risks

Risk	P	I	Score	Mitigation
Project is too ambitious.	8	8	64	Scale back/adjust project aims.
Illness of close family member that will require my care.	5	6	30	Have good time management to afford missed days and/or catching up.
GitHub has a service failure.	3	7	21	Keep sufficient offline backups of project development
Development slowed due to Migraines.	6	3	18	Have good time management to afford missed days and/or catching up.
General illness.	5	3	15	Have good time management to afford missed days and/or catching up.
Hardware failure.	2	5	10	Keep sufficient backups of files, and spares for components.

Figure 8: Risk table

The risks involved in the development of this project can be seen in the risk table in figure 8, with 'P' representing the Probability of the risk (scaled from 1 to 10), 'I' representing the Impact of the risk (scaled from 1 to 10), and 'Score' representing the Risk score, which is calculated as the product of the probability and impact of the risk, and the risks are ordered by this risk score.

The risks of this project are mostly all relating to some kind of down time. In the case of myself getting ill, be it in general or from migraines, or a close family member becoming ill, it results in slowed progress and can be mitigated by keeping up-to date with sprint tasks. The other kind of downtime comes from hardware downtime, either my own or on the end of a service such as GitHub, on which I host my version control and backups. These issues could result in loss of project data, potentially crucial, and can be mitigated by keeping sufficient backups of files, in several places, and having backup hardware.

During the interim document stage for this project, I incorrectly decided to remove the risk 'Project is too ambitious', as at the time I was on-track to complete the aims of the project. After the interim document stage, I started work on implementing the final component to the neural network, backpropagation. This task proved a lot more difficult than I had anticipated it would have been and halted progress on the network for multiple reasons. This to me was a factor in the project perhaps being 'too ambitious'. The mitigation to this risk was to 'scale back and/or adjust project aims', however this mitigation was hard to

do as backpropagation is an integral part to a neural network. My solution to this was to change (albeit very late in the timeline) to using an existing neural network implementation for the purpose of playing the video games rather than creating my own.

The only other risk to occur during the duration of this project was the slowing of development 'due to migraines'. During the periods where I would perform a lot of implementation I would have more frequent migraines due to the amount of time spent at my computer. Early on in the implementation the mitigation of keeping 'good time management to afford missed days' worked well. However when this then became coupled with the issues involved in implementing backpropagation, the mitigation was not very effective, and productivity suffered as a result.

4 Technologies

4.1 Git

In this project, Git has been used for both version control and backups with the service GitHub being used for the repository. To perform the version control, local repositories will be created on development devices. There are also two main branches, *master* for working code, and *dev* for code that is being implemented and not yet confirmed to works. Whenever an existing file has been edited or a new file has been created, the following steps will be carried out in the operating system command line (when located in repository directory):

```
git add FILE
git commit –m ‘‘comment on changes”
git push
```

This means I can track what changes have been made for each update in the commit message, and have the ability to safely roll back if the implemented changes are no good.

4.2 C++

This project has been developed using the C++ language. This choice was made as C++ is both object-oriented, and low level. As well as this C++ is compatible with CUDA, so it would be possible to be able to GPU-accelerate the neural network.

As this was my first time using the C++ language, there were a few hiccups along the way. For example object scope, new neurons are created within loops, as many are potentially being made for each layer. These neurons are then stored in vectors for their layer, however when trying to reference these neurons

later, they no longer existed. This is because when the program left the loop, garbage collection was performed to remove the neuron objects as they went out of scope.

As I am more experienced with the Java programming language, I did not anticipate this issue. Once the error was noticed, I corrected it by using the New keyword, which allows the object to be allocated memory on the heap, but requires me to explicitly delete the object later instead of relying on the automatic garbage collection, which previously deleted the object when leaving the loop.

4.3 Python

As there were difficulties implementing the crucial backpropagation element to the C++ neural network implementation, I attempted to approach the problem of 'playing video games using neural networks and reinforcement learning' with the use of an existing library. While libraries do exist for the C++ language, they have little documentation or example usages, whereas the libraries that exist for Python are well documented and have widespread examples of implementation.

While I had no prior experience of Python, it is a much more 'simple' language to pick up than C++. As a result I was able to convert my existing intelligence agent across with ease and then progress to attempting the use of machine learning library.

4.4 The Arcade learning environment

The neural network needs a way of interfacing with video games, and this can be done using a tool known as the Arcade learning environment. The arcade learning environment is built around an Atari 2600 games console emulator known as Stella, so it is limited to only allowing interface to games for that games console. This environment was created alongside the paper 'The Arcade learning environment: An evaluation platform for general agents' [4] by DeepMind, and was used in both aforementioned DeepMind papers. It has been made freely available online for use by "researchers and hobbyists to develop AI agents for the Atari 2600" [4].

The arcade learning environment is compatible with languages such as C++, Java, and Python and presents several interfaces for these languages to communicate with it and therefore with the video games.

4.5 TensorFlow

The library I chose to use in python is known as TensorFlow [1]. This library uses data flow graphs to perform mathematical operations, and has been used extensively for machine learning, even with a few applications to video games coupled with reinforcement learning.

As I started using both Python and this very late in the project timeline, I have not got an extensive or in-depth knowledge on the library itself, however I have followed and made sense of on-site tutorials for the use of TensorFlow, that apply to using it for a Convolutional neural network.

5 Analysis of problem

5.1 Video games

To be able to apply the neural network to a video game, we will need access to control the video game, as well as the game screen data. The first option for this was the use open-source video games that specifically targeted the PC platform. These would allow access to control the video games with some modification however they may not have necessarily given us a straight forward way to access information on the game screen. The programming language to be able to interface with these games would depend entirely on the source code for the video game being chosen.

Another option I had was to use video game console emulation software, such as the 'BizHawk emulator'[19] which can emulate several consoles. Communication to video games being played within this emulator would be through the memory addresses. This can be done via scripting in *Lua*, however it would be very cumbersome, as the memory addresses are different on a game-by-game basis, and they aren't very well documented.

Building on top of the idea of using video game console emulation software, the Arcade learning environment was considered and then later chosen as the way to access video games. This is due to its use in existing work by DeepMind, and that it allows a unified interface for many different video games, albeit limited to the Atari 2600 console.

5.2 Intelligence agent

The intelligence agent in this case will be where the neural network and the interface for arcade learning environment interface will be instantiated. In this agent they will then communicate with each other, while going through the algorithm seen in algorithm 1 for performing Q-learning. This intelligence agent

is what will be called from the final programs *main* method, and it will handle the rest.

As outlined in the work from DeepMind in regard to their Deep-Q learner, the intelligence agent will use something known as *exploration-exploitation*. This means that the agent will initially perform random actions the majority of the time, and gradually begin to use the neural network more and more. This is used as "it is necessary to sometimes pick a random action to not get stuck in local reward maxima" [2], so the network is able to explore more options than just those actions it currently believes are the most advantageous.

The 2015 DeepMind paper also provides us with a table of hyperparameters and their optimal values for the Deep Q-network intelligence agent, available in the abstract of this document. Most of these will remain the same in our implementation, however some have been scaled down to allow easier implementation. For example, the minibatch size will be reduced from 32 to 16, and the replay memory size reduced from 1000000 to 1000.

Algorithm 1 Deep Q-learning with Experience Replay [13]

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

5.3 Preprocessor

Game screens are output from the Arcade learning environment as size 210x160 with 8-bit hexadecimal values for a 128-color palette specific to the NTSC version of the Atari 2600. The neural network will take greyscale images of size 84x84 (as explained in section *Network structure*) as input, therefore the image we receive from the environment needs to be preprocessed. First we need to

select a region of interest within the image, to reduce the height of the image from 210 to 160. The image will then be downscaled by a factor of 0.525 to get the correct size of 84x84. Finally the 8-bit hexadecimal values will be converted into greyscale integer values using a look-up table.

5.4 Neural network

The chosen topology for the neural network for this project is a deep convolutional neural network. This is due to both their proven ability to work well with image recognition problems, and the usage in the work of DeepMind.

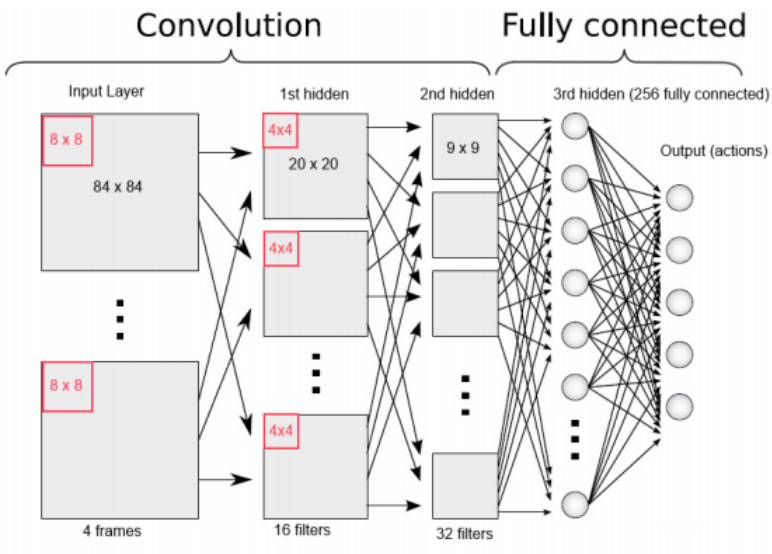

Figure 9: DeepMind original topology [2]

In the 2013 paper from DeepMind [14], the network structure seen in figure 9 was used. This was later extended to the network structure seen in figure 10 in 2015 when published in Nature.

For this implementation, I have chosen the 2013 DeepMind topology, as it has been proven to work in this application to a broad set of video games, and is less complex than its 2015 counterpart in terms of computation. This topology takes in four greyscale images of size 84x84, and then applies four filters of size 8x8, with stride of four, to produce sixteen feature maps of size 20x20 in the second layer. These filters are applied as local receptive fields, which are explained in the background section on Convolutional neural networks. The second layer then is also a convolutional layer, and applies two filters of size 4x4, with stride 2, to produce thirty-two feature maps of size 9x9 in the third layer.

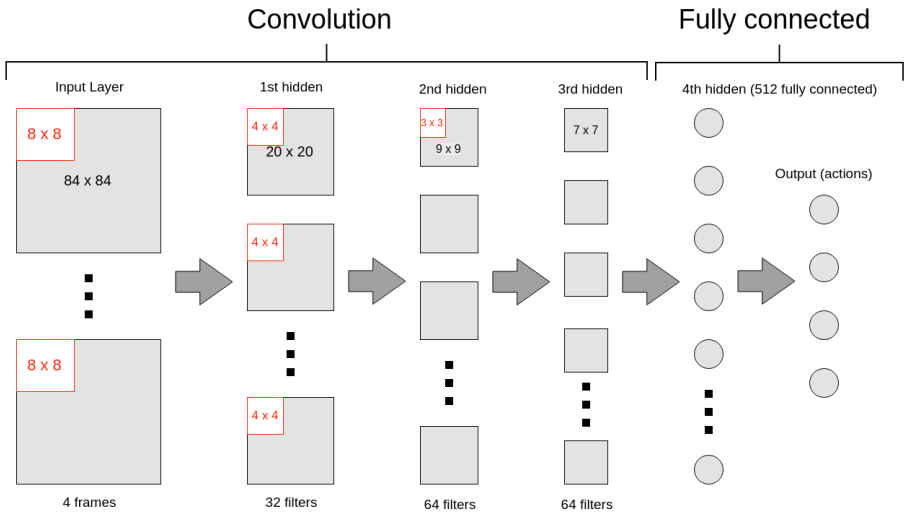

Figure 10: DeepMind extended topology

This third layer is then connected to the fourth layer, which is a fully connected layer of size 256. The fourth layer then connects to the final layer which is fully connected and is of varying size depending on the amount of actions available in the game the network is being applied to.

It's worth noting that this convolutional neural network does not have any pooling layers. This is due to the nature of the images being used for this network application, Atari 2600 games screens are low resolution images. Max pooling layers would have been used had this network been applied to images that were of a higher resolution.

The activation function to be used in the network is a non-linear rectifier function. this function works as follows:

$$f(x) = max(0, x) \tag{21}$$

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } otherwise \end{cases} \tag{22}$$

For the purposes of backpropagation, we need the derivative of this function, which is the following:

$$f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } otherwise \end{cases} \tag{23}$$

The non-linear rectifier function can also then be visualised in figure 12.

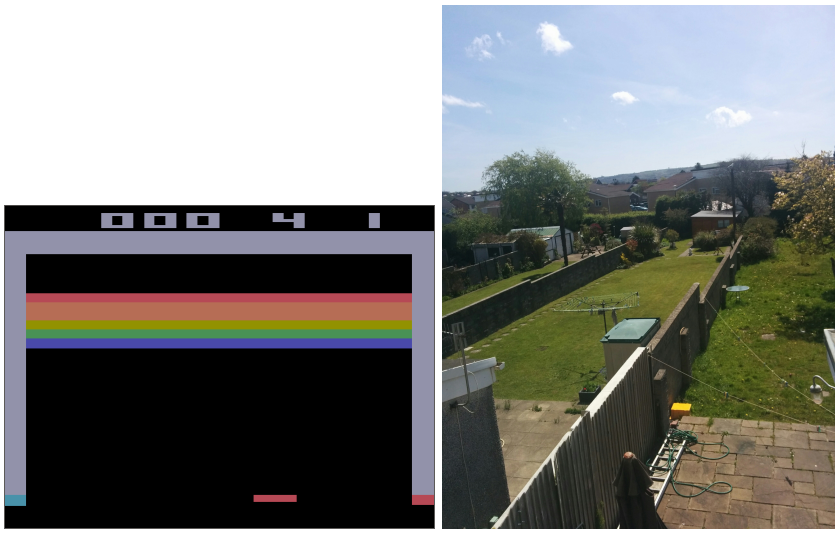

Figure 11: Screenshot of the Atari 2600 game 'Breakout' compared to a real-life image.

5.5 Learning

The network will learn overall using the reinforcement learning method of Q-learning, making it a deep q-network. The backpropagation learning for the network will use a technique known as RMSProp (Root mean square propagation) [9].

Algorithm 2 Q-Learning element [13]

1: Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

2: Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{for non-terminal } \phi_{j+1} \end{cases}$

3: Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$

From the main algorithm in algorithm 1, the learning is done in the excerpt extracted in algorithm 2. Here, after stochastically selecting samples from the minibatch in line 1, we get an estimate on future rewards from line 2, and then perform a gradient descent step in line 3.

The gradient descent step allows the network to know how much to alter the weights in the network when it performs backpropagation. The technique of RMSProp being used for the backpropagation is a variation of Stochastic gradient descent which will "divide the learning rate for a weight by a running average of the magnitudes of recent gradients for that weight" [9].

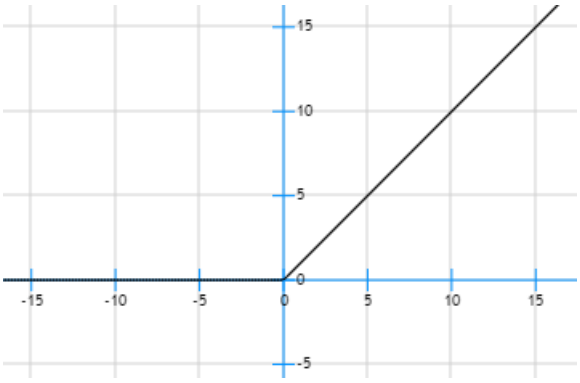

Figure 12: Non-linear rectifier

6 Implementation sprints

The development of this project was done in a series of sprints, as it was done using the agile software model (See section 5.1 - Software life-cycle).

6.1 Methods to manipulate I/O

The first sprint for the implementation was establishing a method to enable communication between C++ and the Arcade learning environment. This is accomplished by including and using a class known as the *ALEInterface*, with consistent methods across both the C++ and Python such as those seen in the example code in Code Listing 1.

The interface with ale needs to be initialised, from that we can then load in the game ROM (a .bin file specified either in the code itself or when run from the command line) and then get information about this game. We can get either the legal or minimal set of actions for the video game (both stored in a structure known as *ActionVect*). The legal action set are all actions available to the player when playing the game whereas the minimal action set is the smallest subset of the legal actions needed to play the game. In the context of the neural network, the action set will be the size of the output layer. Once we have a decision on the action to be taken, we will perform the act command with the chosen action, which will return the reward obtained in the game from the given action.

Code Listing 1: ALEInterface in C++

```
#include <ale_interface.hpp>

int main(int argc, char* argv[]) {

        //Initialise interface with environment
```

```
    ALEInterface ale;

    //Set whether game screen/sound will be displayed
    #ifdef __USE_SDL
            ale.setBool("display_screen", false);
            ale.setBool("sound", false);
    #endif

    //loads specified game
    ale.loadROM(argv[1]);

    //gets legal actions for game
    ActionVect legalAct = ale.getLegalActionSet();

    //performs action and returns reward
    int reward = ale.act(legalAct[0]);

    //returns whether game has ended or not
    bool gameOver = ale.gameOver();

    //returns game screen data
    ALEScreen screen = ale.getScreen();
}
```

We can then also get a boolean value of whether the game has reached a game over state, as well as getting game screen information. This is output as a size 210x160 *ALEScreen* structure of unsigned characters, which are 8-bit hexadecimal values representing a corresponding colour from a 128-colour palette specific to the Atari 2600.

Initially small dummy C++ classes were made to test communication with the Arcade learning environment, with it being attached to the neural network at a later sprint.

6.2 Preprocessor

As the output from the Arcade learning environment does not exactly match the expected input to the neural network, some preprocessing steps are required. Much like with the communication with the arcade learning environment, this was at first done with some dummy C++ classes before being added to the larger network later on.

The preprocessor crops the image and converts to greyscale (with use of a look-up table) using Algorithm 3.

It then downscales the resulting image by a rate of 0.525 using Algorithm 4.

Algorithm 3 Region of Interest and greyscale conversion

```
1: ALEScreen screen = ale.getScreen();
2: constant int ROI_TOP = 32; constant int ROI_BOT = 18;
3: int width = screen.width(); int height = screen.height();
4: Image img;
5:
6: int jt = 0;
7: for (int j = ROI_TOP; j <height - ROI_BOT; j++) do
8:     for (int i =0; j <width; j++) do
9:         int pix = screen.get(j, i);
10:        img[jt][i] = GREYSCALE[pix];
11:    end for
12:    jt++;
13: end for
```

Algorithm 4 Image downscaling

```
1: constant int DESIRED_IMAGE_XY = 84;
2: constant int INPUT_IMAGE_XY = 120;
3: double scaleRatio = INPUT_IMAGE_XY;(double)DESIRED_IMAGE_XY;
4: int newX, newY;
5: Image newImg;
6:
7: for (int y=0; y <DESIRED_IMAGE_XY; y++) do
8:     for (int x=0; x <DESIRED_IMAGE_XY; x++) do
9:         newY = floor(y*scaleRatio);
10:        newX = floor(x*scaleRatio);
11:        newImg[y][x] = img[newY][newX];
12:    end for
13: end for
```

The resulting data is now ready to be input to the neural network.

It is worth noting that the data type *Image* is defined as $vector < vector < int >>$ in this project.

6.3 Network generation

The neural network structure, based off the work of DeepMind and as seen in figure 9, has 3 convolutional layers, followed by two fully connected layers.

As a result, there exists two kinds of neurons, a standard Neuron to be used in fully connected layers, and a Convolutional Neuron to be used in convolutional layers. The convolutional neuron, defined as class *ConvNeuron*, inherits from the standard *Neuron* class for this Neural network project. Both kinds of

neurons hold information on their value and their connections.

The convolutional (input, first hidden, and second hidden) layers hold information on the filter(s) to be applied, that is the amount of filters, the size of the filter(s), the stride at which to apply them and the filter(s) themselves. A *Filter* data type in the case of this project is defined as $vector < vector < double >>$ and are held in the data type *Filters*, which is defined as $vector < Filter >$. The convolutional layers also hold the neurons for that layer, stored in a data type called *FeatureMap* which is defined as $vector < vector < ConvNeuron^* >>$.

The fully connected (third hidden, and output) layers hold a *Weights* data structure which is defined as $vector < double >$, as well as a *NeuronSet* defined as $vector < Neuron^* >$.

The constructor for the neural network will feed in the input image(s), weights, and game actions size). The input image(s) are received from the game after being preprocessed, the game actions size is also received from the game. The weights are randomly initialised outside of the network before being fed in. The weights are held in a structure known as *weightStruct* which holds two *Filters* data types and 2 *Weights* data types. The weights for the first pass of the neural network are initialised randomly, and will be tuned then as the network learns. The structure to hold these weights is outside of the network so that the agent can update them as it learns and then feed them into the network.

As for the creation of the neural network itself, the methods to generate the network are generalised, allowing the network structure to be altered at any time easily, granted that the input layer is convolutional and the network consists of only convolutional and fully connected layers. The pseudo code for generation of a input convolutional layer can be seen in algorithm 5.

Algorithm 5 Input convolutional layer generation pseudocode

```
1:  procedure ConvLayer(Images, filter size, number of filters, stride)
2:      for each image do
3:          for y-axis values of image do
4:              for x-axis values of image do
5:                  Create new convolutional-Neuron
6:                  Set value to pixel at [j][i] in image
7:                  Store convolutional-neuron in feature map
8:              end for
9:          end for
10:         Add feature map to feature map collection in layer
11:     end for
12: end procedure
```

The input convolutional layer differs from it's non-input counterparts in that it has no previous layer, it will take in the Images instead of the outputted feature maps of a previous layer, and does not need to take into consideration any weights, summation or activations to it's neurons. The pseudo-code for the generation of a non-input convolutional layer can be seen in algorithm 6.

Algorithm 6 Non-input convolutional layer generation pseudocode

```
1: procedure ConvLayer(Previous layer, filter size, number of filters, stride)
2:     Store pointer to previous layer
3:     for each feature map do
4:         for each filter to be applied do
5:             for y-axis values within image where filter is applied do
6:                 for x-axis values within image where filter is applied do
7:                     Create new convolutional-neuron
8:                     for y-axis of filter do
9:                         for x-axis of filter do
10:                            Add connection to previous layer using weight from filter
11:                        end for
12:                    end for
13:                    Store convolutional-neuron
14:                end for
15:            end for
16:            Add feature map to feature map collection in layer
17:        end for
18:    end for
19: end procedure
```

The amount of non-input convolutional layers can be easily specified then in the neural network, so the network is not stuck to one static structure.

The pseudocode for the creation of a fully connected layer that is connected to from a convolutional layer can been seen in algorithm 7.

These constructors are then called in the neural network to define the structure of the neural network and create it. In the case of my neural network, class variables are used in the network class to hold the layers, constants are defined for variables such as filter sizes and counts, and the generation of the layers is done in the layer class constructors. The code to generate the network topology outlined in the 2013 DeepMind paper would look like code listing 2 if no constants were used.

Code Listing 2: Network generation

```
SetInputLayer(ConvLayer(m_input, 8, 4, 4));
```

Algorithm 7 Fully connected layer generation pseudocode

```
1: procedure FullConnLayer(previous layer, layer size)
2:     for size of layer do Create new Neuron
3:         for each neuron in previous layer do
4:             add connection between new neuron and previous convolutional
   neuron
5:         end for
6:         Add Neuron to layer
7:     end for
8: end procedure
```

```
m_inputLayer.SetFilters(m_firstWeights);

SetSecondLayer(ConvLayer(m_inputLayer, 4, 2, 2));
m_secondLayer.SetFilters(m_secondWeights);

SetThirdLayer(ConvLayer(m_secondLayer, 0, 0, 0));

SetFourthLayer(FullConnLayer(m_thirdLayer, 256));
m_fourthLayer.SetWeights(m_thirdWeights);

SetOutputLayer(FullConnLayer(m_fourthLayer, GetActionSetSize()));
m_outputLayer.SetWeights(m_fourthWeights);
}
```

6.4 Network feedforward

Now that we can generate the network, the network needs to have data input and then propagate that data forward to make a decision. The forward propagation of the network is done by calling the *activateNeurons()* function of each non-input layer in sequential order.

The *activateNeurons()* function of a layer will loop through every neuron in the layer and call the neuron's *CalculateValue()* function. This function loops through all of the connections in the neuron and performs a summation on them to get the value for the neuron. After that, the *Activation* function is called on the neuron which performs the non-linear rectifier activation on the Neuron.

Once this has been performed all the way up to the output layer, the network will have it's decision values in each neuron of the output layer. To get the decision of the network in the agent, the agent calls the network's *getDecision()* function which returns the index of the neuron which has the highest value.

The feedforward of the network was however broken when trying to accommodate a new method of establishing connections between layers so that it would be able to backpropagate. This new method would, instead of storing just the value of the weight, store the index of the weight, so that it could be accessed by this index for activation and backpropagation.

The part where the feedforward started to break was trying to access the index of the weights. As the network was created with weights that were randomly initiated within the network, the addition of externally initiated weights has caused errors in accessing the weights when attempting to calculate neuron values.

As the network can not feedforward without reverting to having the weights randomly initialised every time the network is called, this network is not viable for the problem as it would only be as good as a random agent.

It is worth noting also, that even when the network was able to feedforward, the time it took to generate and propagate values forward caused massive stutters in the Atari 2600 games, as a result the network would need further optimisation to play the games at an acceptable rate.

6.5 Backpropagation

This crucial element of the neural network was sadly not completed and implemented, due to time constraints coupled with several difficulties in understanding how to implement it for a network of this kind.

One such difficulty is the first step of error backpropagation, determining the error for each of the output neurons. The target value for the network, as determined by the intelligence agent, is the index of the neuron that the network should have given the highest value to. However, in error backpropagation, each individual output neuron needs to have a an individual error calculated. I am unsure how to do so when we are only trying to determine that one of the neurons should have a higher value than the others.

Another difficulty was in implementing the propagation back through the network. The objective of backpropagation is to tune the weights of the network to be able to achieve a more accurate output decision of the network. However in the first implementation of the network, the connections just stored the values of the weights, so when propagating backwards the network was unable to tell where the weights came from to be able to update them. As well as this, in the case of convolutional layers, weights are shared across multiple neurons, so it is necessary to store either a pointer to the weights or an index value.

When implementing a pointer to the weight, the network generated much slower, which would become detrimental when it came to applying this network to the game. An attempted solution using the index values of the weights was implemented, however as mentioned in the network feedforward section, this presented issues when trying to access the weights.

As a result of this element of the network being missing, as well as the forward-propagation, the neural network being developed in C++ is not viable to be able to learn to play video games, as the fundamental elements of the network are missing.

6.6 Replay Memory

The replay memory for the intelligence agent allows us to keep a collection of previous transitions taken by the agent. These transitions consist of:

- Current state (screen) of the game.
- Action chosen by network.
- Reward received from game.
- Following state (screen) of the game.
- Whether the current state of the game is terminal.

This replay memory has two main functions, *AddTransition()*, and *GetMiniBatch()*.

The *AddTransition()* function takes in the transition as a parameter, and will store it within the replay memory data structure. The replay memory itself has a finite size, so if it is at capacity, it has will remove replace existing transitions, in a first-in-first-out fashion.

The *GetMinibatch()* function will return a $vector < transition >$ structure holding a stochastic sample of transitions known as a *minibatch*, with the size of the minibatch specified as a constant.

The replay memory of the network was implemented, however the learning process in the agent that would have utilised it was not.

6.7 TensorFlow

When creating a neural network using TensorFlow, you must first define the network structure, using a placeholder for the input of the neural network. Following a tutorial for using a convolutional neural network in TensorFlow applied to the MNIST handwritten digits problem [5] I created a slightly modified version of the 2013 DeepMind network. This can be seen in Code listing 3.

Code Listing 3: Network generation in TensorFlow

```
def init_network():
    ph_in = tf.placeholder(tf.float32, shape=(None, 84, 84, 2))
    second_layer = conv_layer(ph_in, weights['w1'], 4)
    third_layer = conv_layer(second_layer, weights['w2'], 2)
    third_reshape = tf.reshape(third_layer, [-1, 484])
    fourth_layer = full_conn_layer(third_reshape, weights['w3'])
    output_layer = tf.matmul(fourth_layer, weights['w4'])
    return ph_in, output_layer
```

Here, the weights are created as a weight structure, similar to the approach I took in the C++ neural network. The structure of the network is different to the structure of the 2013 DeepMind network in that it takes 2 input images rather than 4, and applies less filters then in the convolutional layers.

Now that the network is constructed, we need to be able to forward propagate through it. In TensorFlow this is performed by having a session ongoing (started using the *tf.InteractiveSession()* method in this implementation), and then getting the *ph_in* and *output _layer* returns from *init_network()*. Then the *net_out.eval()* function is performed, with the networks input defined as the game screens. This should then propagate forward through the network and return a result/decision.

The cost function and optimizer for the network are also defined in TensorFlow, and in this implementation they are done before the network is forward propagated, as the functions use the placeholder variables in their creation, and will therefore adapt as the values of the network change over time.

The TensorFlow network is now ready to be acted upon by the Intelligence Agent.

6.8 Artificial Intelligence Agent

The intelligence agent for this project has been implemented in both C++, for use with the neural network I created, and Python, for use with a machine learning library neural network. The intelligence agent is the implementation of the process shown in algorithm 1.

For the C++ implementation of the agent, it is possible to run the agent with preprocessed game screens for input images to the network, however there are issues present in the feedforward and generation of the network after attempting to alter the implementation to accommodate backpropagation. The process of performing actions and receiving output works as intended, however as there existed errors in the neural network itself, the agent wasn't completed. This is evident in the exploration-exploitation value never decreasing to allow more learning for the network. All the agent is capable of doing is performing

random actions in the game environment.

The Python implementation of the agent has similar functionality to the C++ implementation, however it is able to forward propagate through the network and make decisions based on the preprocessed screen inputs. It can't yet however learn, which is purely because of time constraints on the project and this implementation being used late in the project timeline as a 'last resort'. The Python agent works similarly to the C++ agent, in that it can perform random actions in the game, with the exception that some of these random actions are now coming from the network forward propagating with the randomly initialised weights. If the learning process were to be implemented to this network, it could then function as intended.

6.9 Training the network

Sadly, due to implementation issues with the C++ implementation, and time constraints with the Python implementation, no training was able to be performed on the neural network(s) against video games. If either of the networks were to have been able to have been trained against neural network, the process could have taken several days.

Existing TensorFlow implementations of Deep Q-networks applied to this problem using the Arcade learning environment state that the networks require several days of training, even when GPU accelerated. As a result, even if the late TensorFlow/Python implementation was completed on time, it may not have been trained in time to discuss the results.

6.10 Testing the network

Much like with the training of the network, this was not able to be performed. What was planned for the testing of the neural network implementation was to monitor the values being altered by gradient descent to ensure that it was being performed correctly. This would normally be done by plotting the values on a graph.

What would have been better is to have a testing method to be able to check the functionality of the neural network itself, however in regards to a deep q-network I could not find any methods. With a convolutional neural network a problem such as the MNIST handwritten digits database could have been used, however as the convolutional neural network being used was as a deep q-network, this would require substantial changes in the way that the network would have been trained.

6.11 GPU-acceleration

Sadly, another element of this project that could not be completed was the GPU-acceleration of the neural network. This was planned as the final stage of the project, purely to optimise the network and hopefully improve its performance.

As was seen in the runs of the C++ network even unfinished, the network could have benefited greatly from GPU-acceleration for the generation of the network, and perhaps even in the forward and backward propagation steps. The main place I considered applying GPU-acceleration was to the first fully connected layer, which connected the most amount of neurons with the most amount of connections, regardless of network topology scale downs.

7 Conclusion

7.1 Reflection of results

The main deliverable of this dissertation, the C++ neural network, regrettably did not come to fruition. While prior to beginning the implementation of the network I made every effort to thoroughly learn convolutional neural networks, I still found myself needing to brush up on vital information throughout. For example the network was designed and implemented to be generated every time it was called, and initially had the weights initialised within this generation process, purely for the purposes of testing. When I began work on implementing backpropagation, I attempted to change the nature of the weights in the network by externalising them, and this caused a chain reaction of errors throughout the network. As this happened very late in the project timeline, I did not have enough available time to fix the errors, let alone fix the errors and then finish the backpropagation implementation.

The network being designed to generate and then propagate when called meant that the network was slow to present it's decision, which is not ideal for the application to video games. Since working with the TensorFlow library, and implementing a convolutional neural network, I now know it would be more beneficial to have the network initialised once at the start. Then once the network has been initialised, use available methods to propagate data forward to get a decision, and propagate data backwards then to train the weights of the network.

In regards to the Python/TensorFlow, this approach could have yielded some results for the dissertation had it been used sooner in the project timeline. However as the aim of this project was to develop a neural network, I felt it best to make every effort to finalised that implementation before considering the alternative of using a library.

7.2 Suggestions for further work

In terms of further work, I feel the C++ implementation could benefit from a fundamental change in the way the network generates, as mentioned in the reflection of results section above. As well as this, implementing a better way to define connections between neurons would enable backpropagation to be better implemented when the theory is better understood.

Most of all however, I feel that the best course of action for future work would be to use an existing machine learning library such as TensorFlow for the deep q-network. The DeepMind papers, in which I have partially based this dissertation on, did not create their own neural network for their implementation, rather they used an existing library known as *torch*. This I feel is the best approach as when there are existing, open-source, libraries available, it is not worth 'reinventing the wheel'.

I will be continuing to work on the Python/Tensorflow implementation of the neural network, in hopes to have some presentable results for the Project Demonstration fair.

References

[1] Martın Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Korjus et al. *Replicating the paper playing Atari with deep reinforcement learning*. Tech. rep. University of Tartu, 2014.

[3] Wolfram Alpha. *Sigmoid Function*. URL: `http://mathworld.wolfram.com/SigmoidFunction.html` (visited on 13/10/2015).

[4] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[5] Aymeric Damien. *Convolutional Network MNIST Tutorial*. URL: `https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3%20-%20Neural%20Networks/convolutional_network.py` (visited on 25/05/2016).

[6] LiMin Fu. *Neural Networks in Computer Intelligence*. New York, NY, USA: McGraw-Hill, Inc., 1994. ISBN: 0079118178.

[7] Andrew Gibiansky. *Convolutional Neural Networks*. URL: `http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/` (visited on 14/10/2015).

[8] Simon Haykin. *Neural Networks, A Comprehensive Foundation*. 2nd ed. Prentice Hall, 1999. ISBN: 0139083855.

[9] Geoffrey Hinton. *Overview of mini-batch gradient descent*. URL: `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf` (visited on 08/01/2016).

[10] Bing-Qiang Huang, Guang-Yi Cao, and Min Guo. "Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance". In: *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*. Vol. 1. IEEE. 2005, pp. 85–89.

[11] Nicholas Lundgaard and Brian McKee. *Reinforcement learning and neural networks for Tetris*. Tech. rep. Technical Report, University of Oklahoma, 2007.

[12] Matt Mazur. *A Step by Step Backpropagation Example*. URL: `http://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/` (visited on 12/10/2015).

[13] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

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

[15] Michael A. Nielson. *Neural Networks and Deep Learning*. URL: `http://neuralnetworksanddeeplearning.com/` (visited on 12/10/2015).

[16] SethBling. *MarI/O - Machine Learning for Video Games*. June 2015. URL: https://www.youtube.com/watch?v=qv6UVOQ0F44 (visited on 10/10/2015).

[17] Ian Sommerville. *Software Engineering*. 9th ed. Harlow, England: Addison-Wesley, 2010. ISBN: 978-0-13-703515-1.

[18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. 2011.

[19] TASVideos. *Bizhawk emulator*. URL: http://tasvideos.org/BizHawk.html (visited on 09/10/2015).

[20] Gerald Tesauro. "Temporal difference learning and TD-Gammon". In: *Communications of the ACM* 38.3 (1995), pp. 58–68.

[21] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

A Appendix: Facts and figures

A.1 DeepMind 2015 network hyperparameters [13]

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in number of parameter updates) with which the target network is updated.
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times.
update frequency	4	The number of actions selected by the agent between successive SGD updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial exploration value.
final exploration	0.1	Final exploration value.
final exploration frame	1000000	The number of frames it takes to take exploration value to final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

A.2 DeepMind 2015 Results

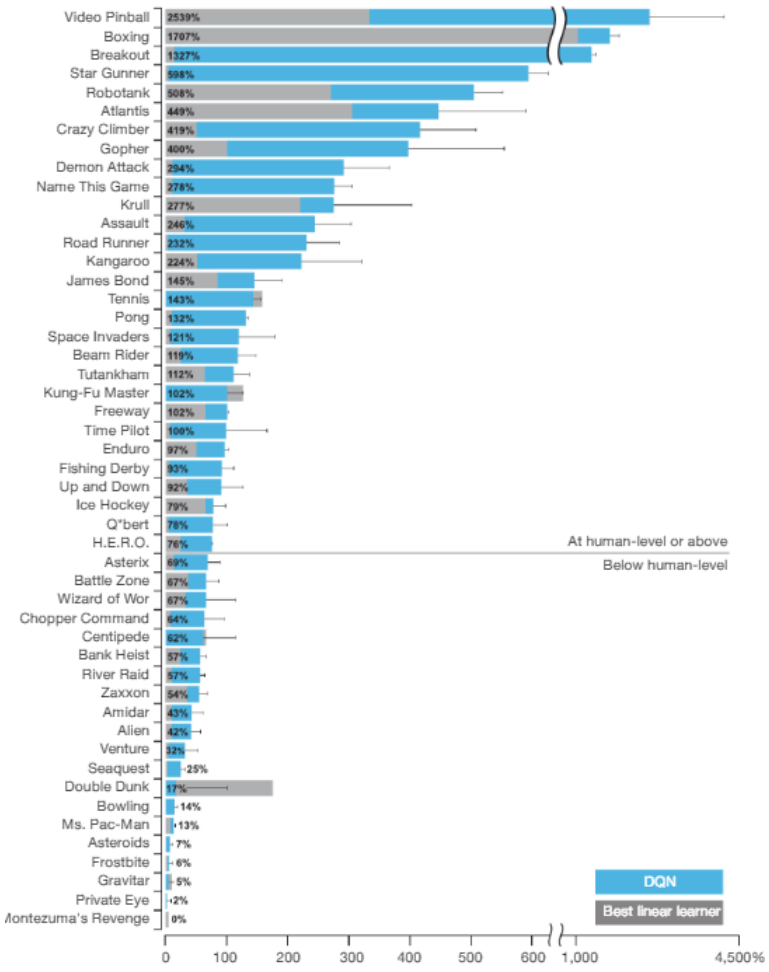

Figure 13: Observed results of DeepMind Deep Q-network normalised with respect to random play (0%), and professional human games testers (100%) [13].

B Appendix: Class diagrams

B.1 Class types

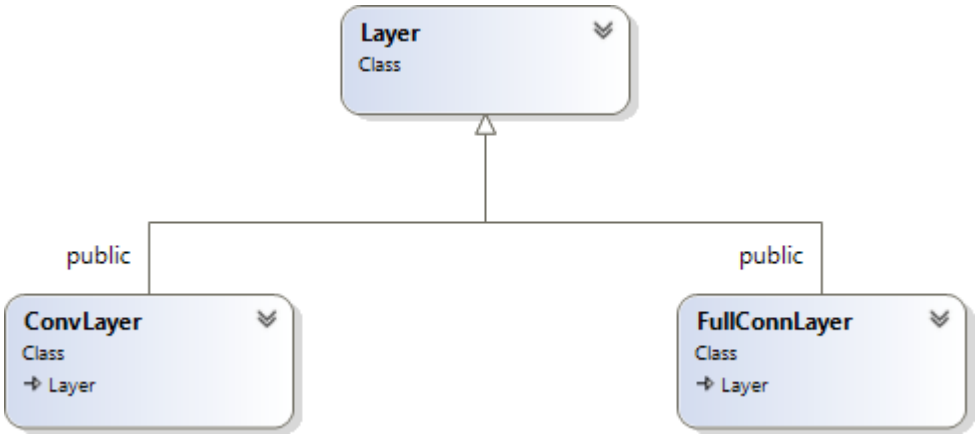

Figure 14: Types of Layer.

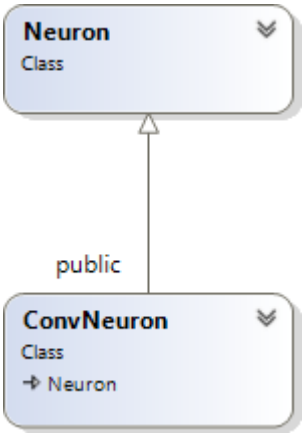

Figure 15: Types of Neuron.

B.2 Class interfaces

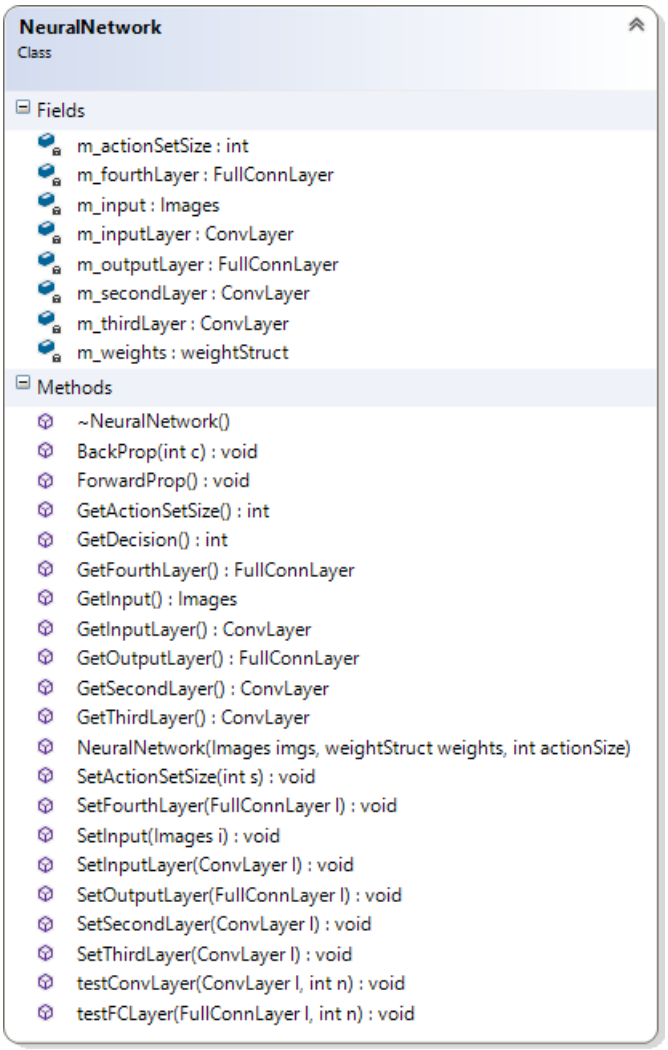

Figure 16: Neural network class.

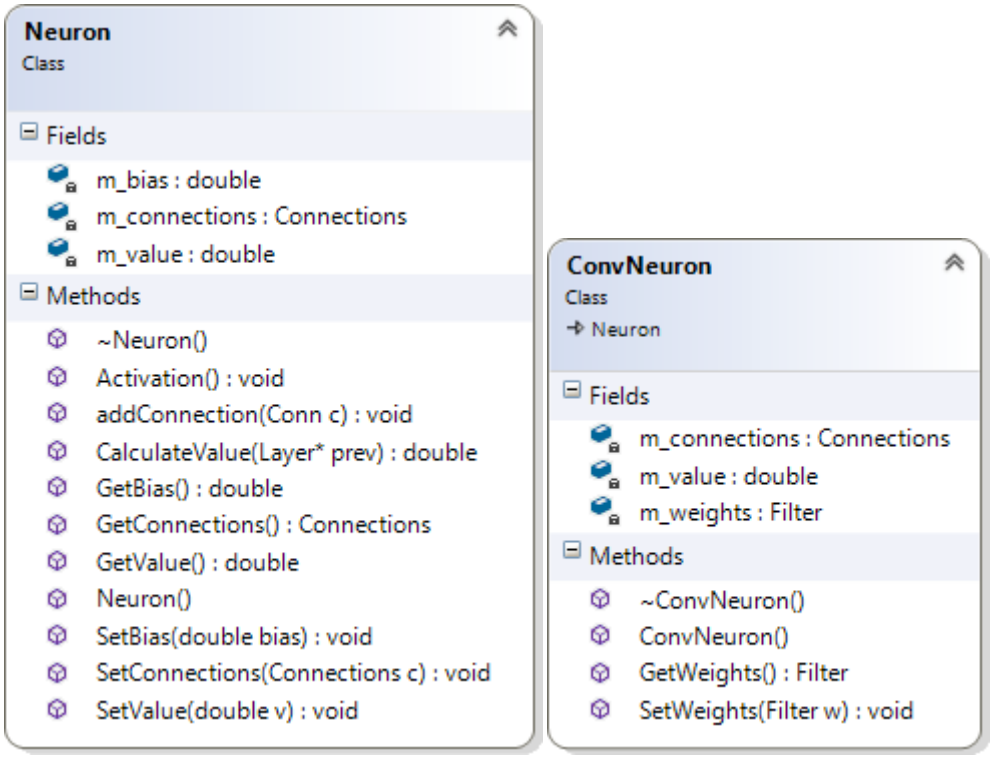

Figure 17: Neuron classes.

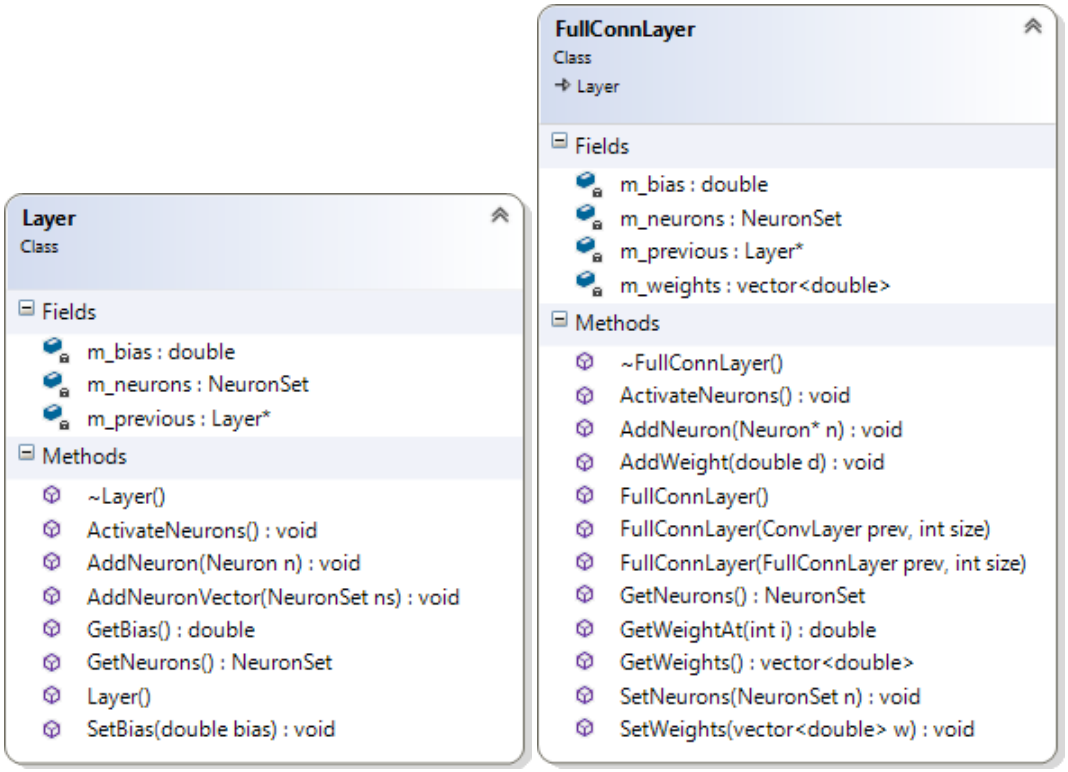

Figure 18: Layer and Fully connected layer classes.

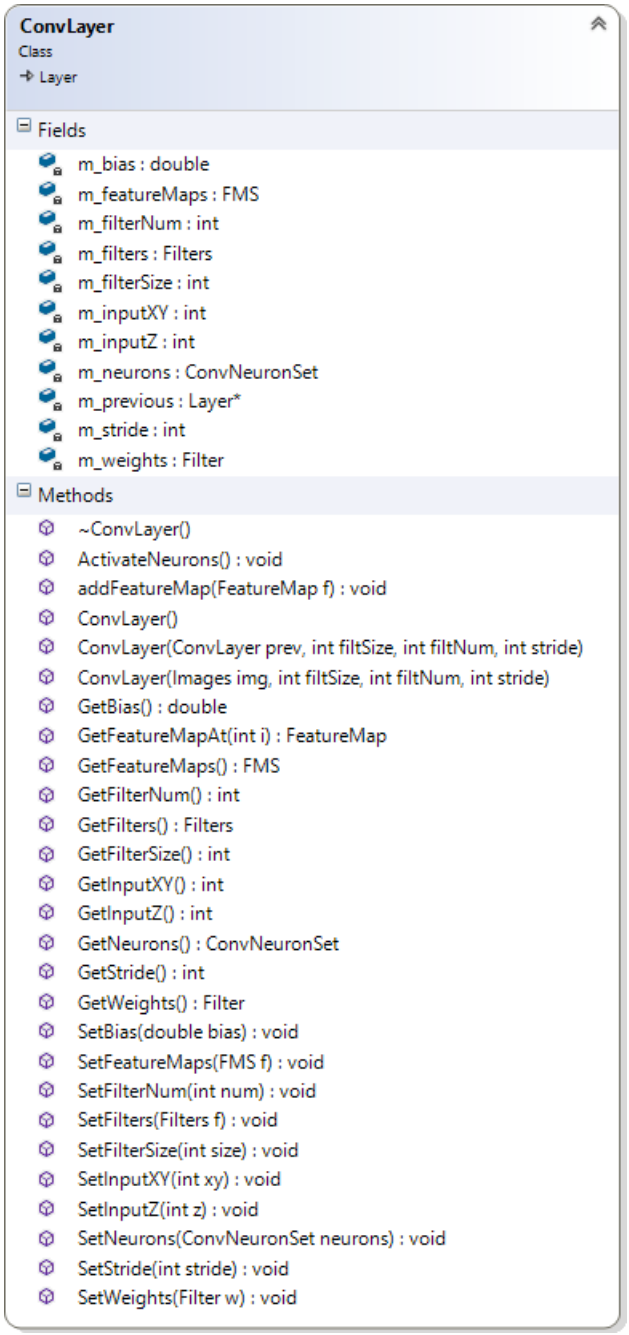

Figure 19: Convolutional layer class.