COMP532 - Assignment 2 - Deep Learning

Rob Lamprell - 201459448

Revised Due Date: 24th April 2020

As agreed with Dr Shan Luo, this is a solo project due to being a part-time student.

Contents

List of Figures

List of Tables

Assignment Details

This section is a listing of the assignment requirements. Please, refer to the contents page for direction as to which section and page the answers can be found on.

Step 1: Import an OpenAI Gym Retro game (20%)

As the starting point, you need to import a game. In your report, you need give a screenshot that you have loaded the game successfully.

Step 2: Creating a network (20%)

You need to use a deep neural network. In the report, you need to show the network structure, and show in the code how to set up different elements of the deep reinforcement learning agent, e.g., the input, output, hidden layers and reinforcement agent etc.

Step 3: Connection of the game to the network (10%)

You will need to explicitly associate the observations, actions, and rewards of the game to the network's input and output. Clearly identify this part of the code in your document.

Step 4: Deep reinforcement learning model (30%)

In this part of your report, describe your deep reinforcement learning model. You need to clearly state which model you are using, the parameters of the model, and how do you train/update the model.

Step 5: Experimental results (20%)

You may record a video demo to show what your agent can do. Alternatively, you can describe it in detail within the text.

Introduction

In this assignment we discuss how to implement a Tensorflow (v2.1) based Double Deep Q-Learning algorithm which can teach an agent how to play a game from OpenAI's Gym-Retro package.

Each of the sections build upon on one another and may not reflect the parameters or configuration of the network within the Python Code. For the certainty as to what the Python Code contains, please refer to 'Final Configuration' section at the end of the document.

The first few sections discuss both Airstriker and Sonic, however, the former is the focus of the project and Sonic will not have a game model implemented due to time constaints.

Test System Hardware

The core system specifications of the hardware utilised throughout testing are as follows:

- Intel 9900K
- 32gb RAM
- RTX 2080ti (Driver: 442.74)
- 512gb NVME (write drive)

Python Environment

The main packages used for this assignment are as follows (Keras is now inbuilt into Tensorflow, as of Ver 2.0 and does not require a separate installation):

Package	Version
gym	0.17.1
gym-retro	0.7.1
h5py	1.0.8
Keras-Applications	1.1.0
Keras-Preprocessing	1.1.0
matplotlib	3.2.1
numpy	1.18.2
tensorboard	2.1.1
tensorflow	2.1.0
tensorflow-estimator	2.1.0

A full list of packages an accompanying version numbers can be found at the end of the document.

Note: A cut-down version of **Stable-Baselines** was also incorporated into the project - specifically for the pre-processing state, reward and action information (prior to being fed into the Neural Network). Due to the current version not properly supporting Tensorflow 2.1 and edited version was implemented and can be found in **retro_wrappers.py**.

CUDA

CUDA is a technology native to Nvidia Graphics Processing Units (GPUs) and was utilised to speed up testing throughout the assignment.

The specific versions of CUDA used for this paper are as follows:

- CUDA Toolkit 10.1
- cudnn-10.1-windows10-x64-v7.6.5.32

Note: CUDA 10.2 appears to have compatability issues with the current version of Tensorflow.

Gym Retro - Step 1

This section discusses how to setup a basic game environment using Open-AI's Gym-Retro package.

Free Game Environments

Gym-Retro comes with some commercially free games already pre-loaded. These include:

- the 128 sine-dot by Anthrox
- Sega Tween by Ben Ryves
- Happy 10! by Blind IO
- 512-Colour Test Demo by Chris Covell
- Dekadrive by Dekadence
- Automaton by Derek Ledbetter
- Fire by dox
- FamiCON intro by dr88
- Airstriker by Electrokinesis
- Lost Marbles by Vantage

For the majority of this assignment, AirStriker will be used as a test bed. For developing the Neural Network and Double Deep Q-Learning Model.

One of the commercially free games was chosen as, it ensures that the code provided can easily be tested by the reader and also given that it's a relatively simple environment it may be more obvious as to whether or not the agent is improving/learning as it plays.

However, there is a potential pitfall which cannot be overlooked, the game only consists of three levels. This means the agent could potentially be configured with a large enough memory pool to remember the actions to take in all possible states rather than learning how to play the game. It also means the agent will likely experience over-training or over-fitting and when presented with a new environment would perform poorly.

Support for Other Retro Games (Sonic - Steam Version)

Gym-Retro supports thousands of games, not just those listed above. Furthermore, the assignment explicitly requests a game be imported.

The following is the import command for the Steam version of Sonic:

python -m retro.import.sega_classics (may be python3 depending on your install)

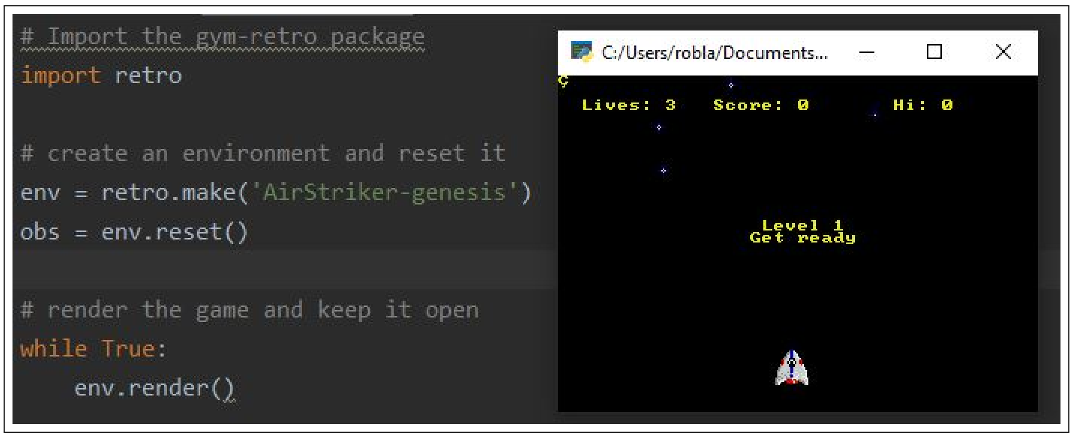
This method requires a Steam account with a valid purchase of Sonic from the sega classics catalogue - There is a bug within the import method, which will require the user to run the command twice if Steam Guard is enabled on the account.

Game Setup - Naive Agent

We will now cover how to launch a very naive environment within Gym-Retro, which will allow random actions to be taken.

Note: The code shown in this section is not in the submitted code and is only used for demonstrative purposes.

Launching Airstriker:

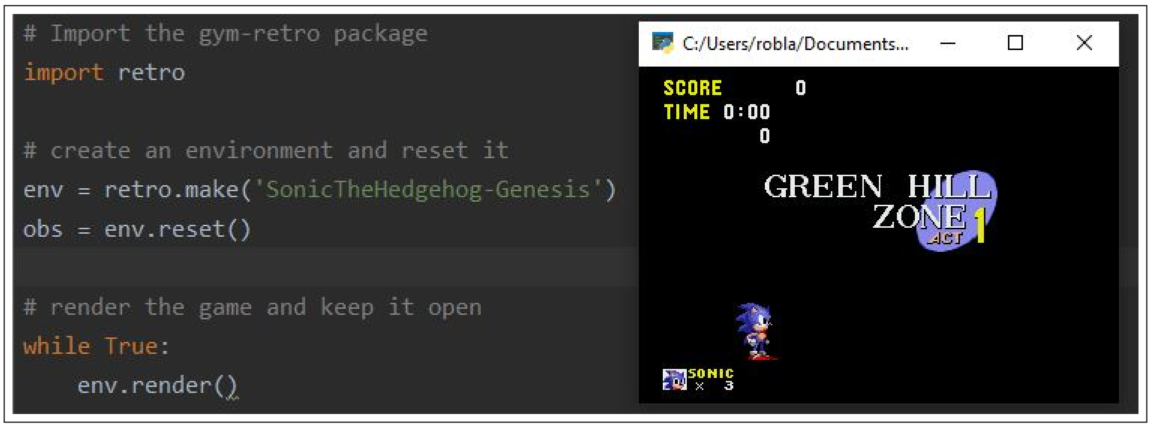


```python
# Import the gym-retro package
import retro

# create an environment and reset it
env = retro.make('AirStriker-genesis')
obs = env.reset()

# render the game and keep it open
while True:
    env.render()
```

Figure 1: AirStriker-Genesis launched using the Gym-Retro Package

As mentioned previously, AirStriker is including in the import of the Gym-Retro Package. The game will launch as a static screen as no actions have been provided to the environment.

Launching Sonic:



```python
# Import the gym-retro package
import retro

# create an environment and reset it
env = retro.make('SonicTheHedgehog-Genesis')
obs = env.reset()

# render the game and keep it open
while True:
    env.render()
```

Figure 2: Sonic The HedgeHog launched using the Gym-Retro Package

Remember, that as Sonic must be imported with the command above before attempting to create an environment with retro.

Similar to AirStriker, the screen will remain static as no actions have been provided.

Passing Actions to the Agent

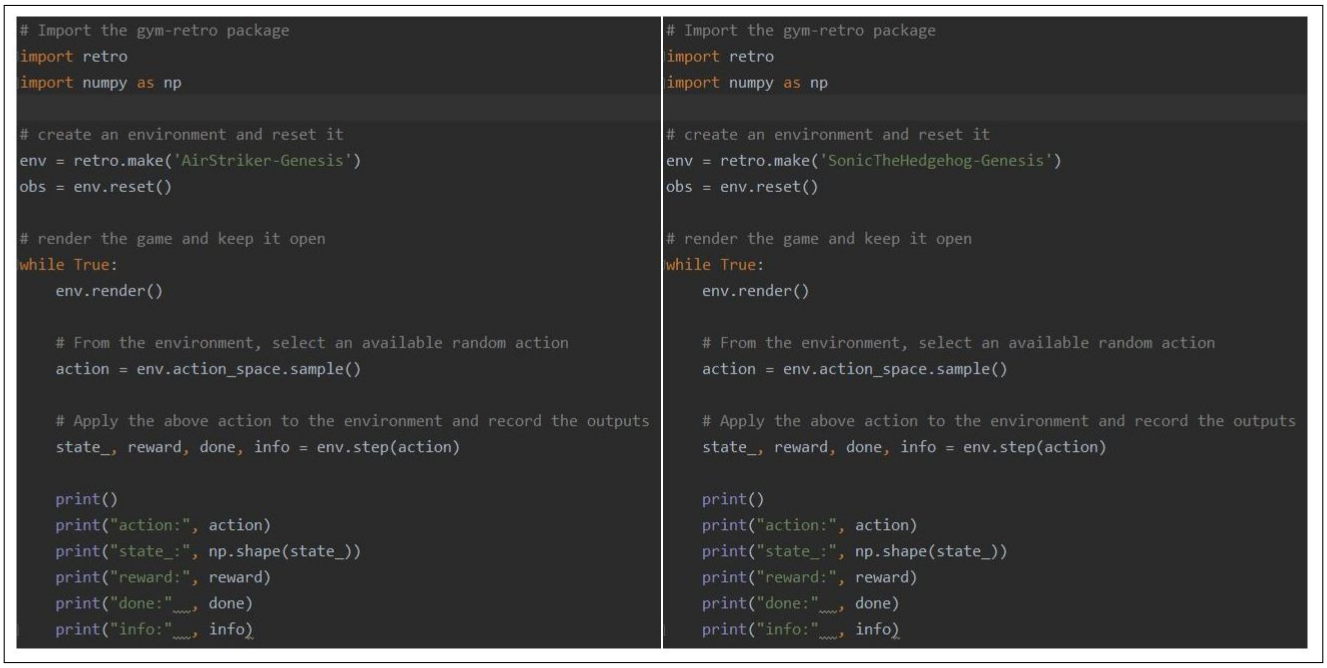
Now that we have a game session we can make our agent take random actions by taking a random sample action from the action space, and feed it back into the environment.

We can get a random action from the action space with this function:

Environment.action_space.sample()

Which will return a random twelve-length-multi-binary-Boolean array. Which refers to either a single or multi-button press input for the game. We can then feed this action back into the environment through the **step** function which will return the next state, any rewards provided, whether the game is done and any additional information provided.

Environment.step(action)

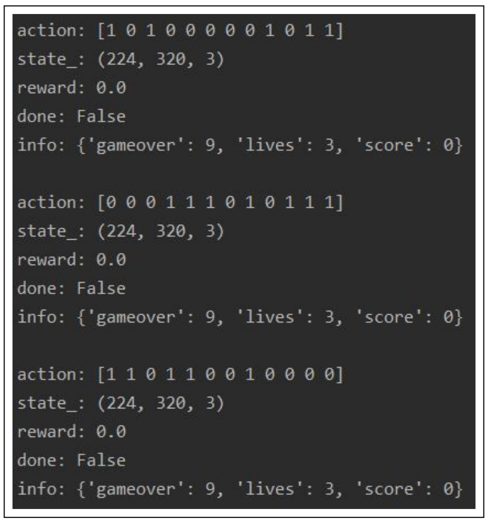


```python
# Import the gym-retro package
import retro
import numpy as np

# create an environment and reset it
env = retro.make('AirStriker-Genesis')
obs = env.reset()

# render the game and keep it open
while True:
    env.render()

    # From the environment, select an available random action
    action = env.action_space.sample()

    # Apply the above action to the environment and record the outputs
    state_, reward, done, info = env.step(action)

    print()
    print("action:", action)
    print("state_:", np.shape(state_))
    print("reward:", reward)
    print("done:", done)
    print("info:", info)
```

```python
# Import the gym-retro package
import retro
import numpy as np

# create an environment and reset it
env = retro.make('SonicTheHedgehog-Genesis')
obs = env.reset()

# render the game and keep it open
while True:
    env.render()

    # From the environment, select an available random action
    action = env.action_space.sample()

    # Apply the above action to the environment and record the outputs
    state_, reward, done, info = env.step(action)

    print()
    print("action:", action)
    print("state_:", np.shape(state_))
    print("reward:", reward)
    print("done:", done)
    print("info:", info)
```

Figure 3: Python Code to Create an Agent Who Takes Random Actions Within the Gym-Retro Environment

By using the Python code from Figure 3 above we can have our agent perform actions and print out the relevant returns.



```
action: [1 0 1 0 0 0 0 0 1 0 1 1]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'gameover': 9, 'lives': 3, 'score': 0}

action: [0 0 0 1 1 1 0 1 0 1 1 1]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'gameover': 9, 'lives': 3, 'score': 0}

action: [1 1 0 1 1 0 0 1 0 0 0 0]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'gameover': 9, 'lives': 3, 'score': 0}
```

Figure 4: AirStriker Sample Output of Actions

Observing the returns from Airstriker, we can see that:

- our actions return as a the twelve-length-multi-binary-Boolean discussed above
- our state information is an array of shape (224 × 320 × 3) - (Height × Width × Channels (RGB))
- any rewards returned as a float
- done is a Boolean value referring to whether the game is over
- info provides a gameover pointer, lives remaining and score obtained

(When gameover = 4 the game has been completed)

Below is an ordered set of reconstructed images displaying a naive game of Airstriker - passing state arrays into **image_printer.py**:

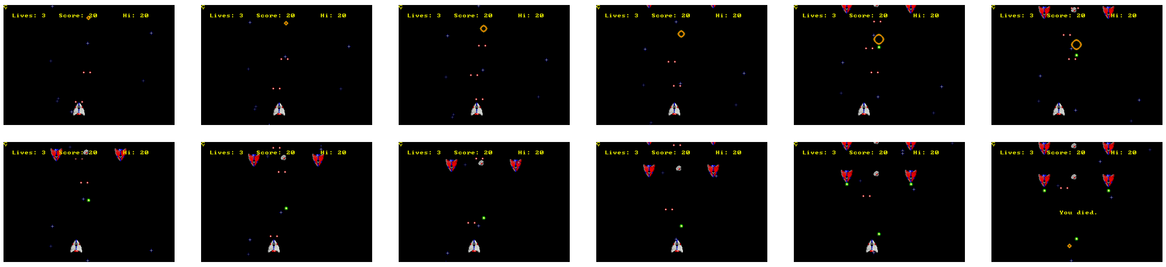
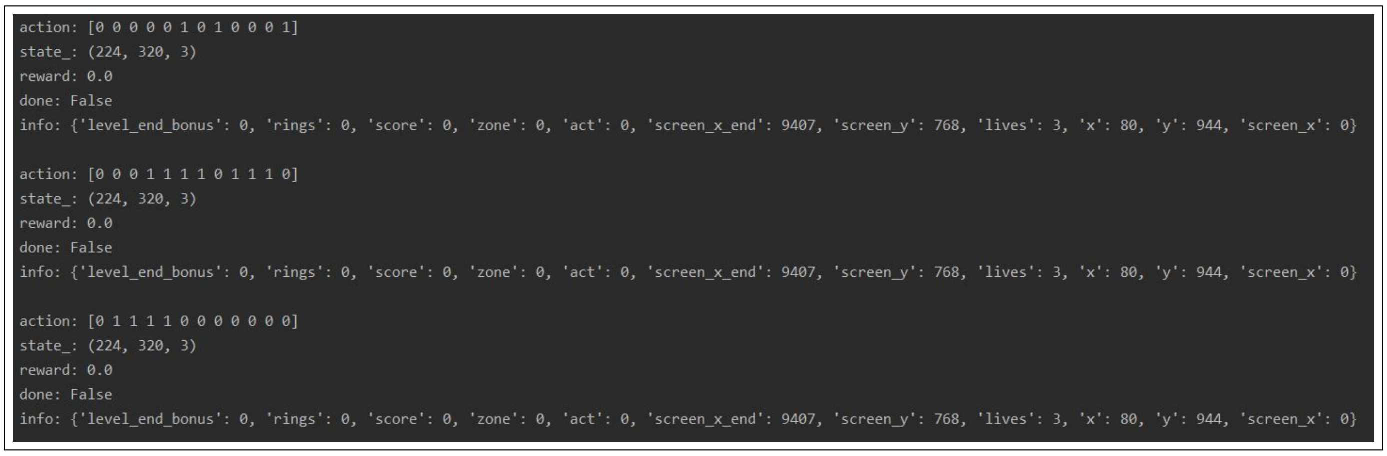


Figure 5: Airstriker-Genesis Taking Random Actions - Five Frame Spacing (Left to Right)

Now taking a look at the sonic environment we can see that different games/environments return different information.



```
action: [0 0 0 0 0 1 0 1 0 0 0 1]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'level_end_bonus': 0, 'rings': 0, 'score': 0, 'zone': 0, 'act': 0, 'screen_x_end': 9407, 'screen_y': 768, 'lives': 3, 'x': 80, 'y': 944, 'screen_x': 0}

action: [0 0 0 1 1 1 1 0 1 1 1 0]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'level_end_bonus': 0, 'rings': 0, 'score': 0, 'zone': 0, 'act': 0, 'screen_x_end': 9407, 'screen_y': 768, 'lives': 3, 'x': 80, 'y': 944, 'screen_x': 0}

action: [0 1 1 1 1 0 0 0 0 0 0 0]
state_: (224, 320, 3)
reward: 0.0
done: False
info: {'level_end_bonus': 0, 'rings': 0, 'score': 0, 'zone': 0, 'act': 0, 'screen_x_end': 9407, 'screen_y': 768, 'lives': 3, 'x': 80, 'y': 944, 'screen_x': 0}
```

Figure 6: Sonic The HedgeHog Sample Output of Actions

Note: The screen position does not change due to how fast actions are input These are the first three frames of data - though the agent manages to mash out three different button combinations, sonic has not been given enough time to move.

Sonic's **info** output provides far more information than Airstriker's. We can see that there are x and y coordinates for the player's position, the end of level position and a reference to an end of level bonus.

This suggests that different game models would likely need to be constructed for each game, especially when seeking optimal training and testing performance.

As above for Airstriker, below is an ordered set of reconstructed images displaying a naive game of Sonic - again, using **image_printer.py**:

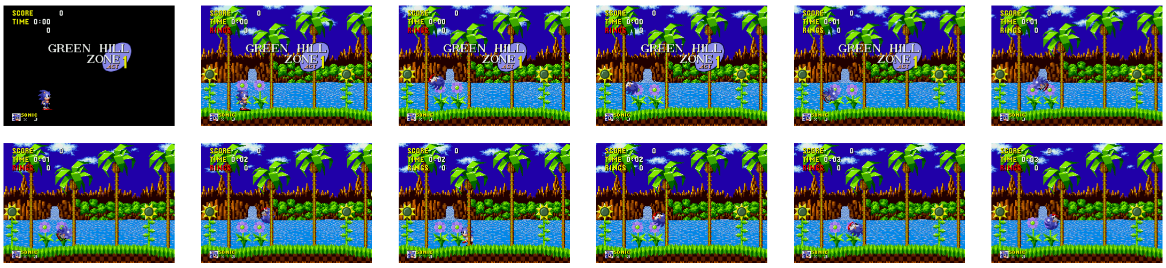


Figure 7: Sonic Taking Random Actions - Twenty Frame Spacing (Left to Right)

Neural Network - Step 2

So far we have instantiated a retro environment with an agent which who acts randomly. It is improbable that this bot would ever complete the game or achieve a high score.

So we need to implement something which can not only choose actions but, which can observe and learn from the actions it has taken in a given state. For this we need function approximators - Neural Networks are great function approximators.

$$Q(s_t, a_t; \theta) \approx Q^*(s, a)$$

Please, refer to **nerualnetwork.py** to see how the neural network was constructed using Tensorflow.

Network Configuration

Our goal is to input some data, which is reflective of our environment and then output an appropriate action based on this information.

Input

The input to the network will be the state information from the environment, which is simply the pixel data from the game. However, this will not be enough for our model to make intelligent decisions as a single frame's input provides no context as to the movement of any objects within the environment. Therefore, the input will consist of a four-stacked-frames. [7]

For the Python Code, when constructing the network the Input is listed on its own for clarity. However, it is not an layer, but instead a feed for the first Convoluational Layer (see **Hidden Layers** below).

Output

Jumping ahead to the output. These are our Q-values and should represent the action we want the network to to return after receiving an input. Therefore, the number of neurons for the the Output Layer should be the total number of relevant actions we can take - for Airstriker there are only three actions, Left, Right and Shoot.

Having more outputs than needed will likely unnecessarily reduce the rate of convergence to the optimal action for a given state. As seen above, the default action space output from retro is far greater than needed. Thus we the number of actions will be drilled-down utilising the **AirStrikerDiscretizer** class found in **retro_wrapper.py** - see below (Game Model).

Hidden Layers

There are numerous types of layers available for neural networks within the Tensorflow/Keras package. Keeping things simple, I will briefly explain the layers we will be using for our Deep Learning Retro solution.

To begin, the data we receive has too much information. All the Neural Network needs to make intelligent decisions to know where 'features' of the environment are - such as an enemy ship or asteroid to avoid. To achieve this, we can pass our state data through several Convolutional layers (**Conv2D**) of varying filter, kernal and stride sizes.

Each filter is looking to identify different features. This will provide some of the simplification desired.

MaxPooling layers could also be implemented following on from Conv2D. These would further reduce the complexity by pulling grouping several pixels, only keeping the maximum value of that grouping - the initial network used does not contain a Maxpooling layer. However, within the section 'Experiments - Step 5' Maxpooling layers are experimented with.

Following on from the Conv2D layers, the data must be transformed so that it can be passed into any fully connected layers. Therefore, between those two layer types there must be a **Flatten** layer.

Finally, before reaching the Output layer, we want to pass the data into the main decision making part of the network. These are called **Dense** layers and are fully connected to one another.

Below is a table describing how many layers should be contained within a given Neural Network:

# Layers	**Result**
none	Only capable of representing linear separable functions or decisions.
1	Can approximate any function that contains a continuous mapping from one finite space to another.
2	Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy.
>2	Additional layers can learn complex representations (sort of automatic feature engineering) for layer layers.

Table 1: Number of Layers For Neural Networks [4]

Using this information, two Dense fully connected layers of 256 neurons were chosen (the neuron count was selected arbitrarily).

Activation Functions - Rectified Linear Unit

One of the things not explored in this paper is use of different activation functions. Each layer has a Rectified Linear Unit ('ReLU') activation function applied to it. This results in any negative logits becoming zero - see Figure 8.

There are many activation functions, including three additional variations of 'ReLU' mentioned in this paper [8]. Below are the formulas and graphs describing both 'ReLU' and 'Leaky ReLU':

$$y_i = \begin{cases} x_i & \text{if } x_i \geqslant 0, \\ 0 & \text{if } x_i < 0 \end{cases}$$

$$y_i = \begin{cases} x_i & \text{if } x_i \geqslant 0, \\ x_i/a_i & \text{if } x_i < 0 \end{cases} \quad [8]$$

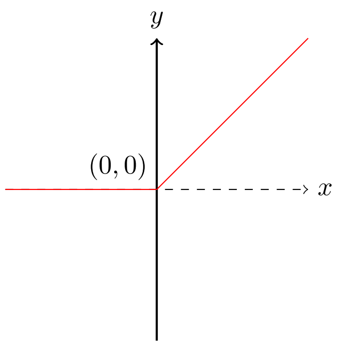


Figure 8: Activation Function: ReLU [8]

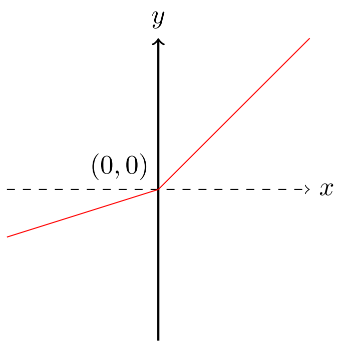


Figure 9: Activation Function: Leaky ReLU [8]

Ideally testing on 'Leaky ReLU' would have been included in this paper - exploring whether 'ReLU''s flattening of negative values has any impact on performance. However due to time constraints only 'ReLU' results were explored.

Optimizer and Loss Function

The final piece in constructing the network is to have it compile with an Optimizer and Loss Function. Although there are many options for both, this paper only covers the Adam optimizer and the Loss functions Mean-Squared-Error (initially used) and Huber-Loss (covered in 'Experiments - Step 5).

MSE is the sum of squared distance between the target variable and the predicted values. The issue with this function is that it can give a lot of weight to outliers. Huber does not suffer from this problem and also has the added benefit of being differentiable at zero [3]. See below for the formulas and graphs of both:

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^p)^2}{n}$$

$$L_\delta(y, f(x) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for } |y - f(x) \leqslant \delta, \\ \delta|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \quad [3]$$

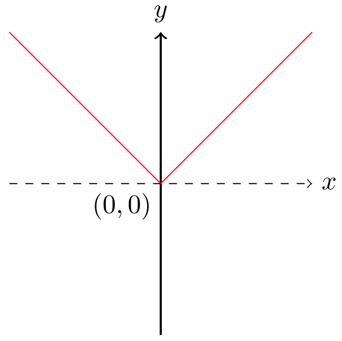


Figure 10: Loss Function: MSE [3]

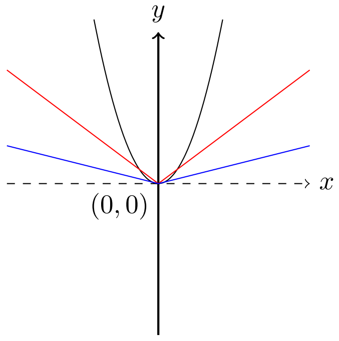


Figure 11: Activation Function: Leaky ReLU [3]

It's clear that the choice of loss function could have a significant impact on the behaviours exhibited by an agent and subsequently the results that agent obtains. 'Experiments - Step 5' and 'Replays - Step 5' both demonstrate the vastly different behaviours of these two loss functions.

Initial Network Configuration and Visualisation

Using the information from the section above, the initial Neural Network Structure choosen is as follows:

Layer Name	Type	Filters / Nodes	Kernal Size	Strides
1st_Convolution	Conv2D	16	(4, 4)	(1, 1)
2nd_Convolution	Conv2D	32	(4, 4)	(2, 2)
3rd_Convolution	Conv2D	32	(4, 4)	(1, 1)
Flattener	Flatten	-	-	-
Fully_Connected_1	Dense	256	-	-
Fully_Connected_2	Dense	256	-	-
Output	Dense	3	-	-

Compile Property	Selection
Optimizer	Adam
Loss	'mse'

Table 2: Configuration of initial Neural Network

This network closely follows the Neural Network setup in this paper [7]. However, as Airstrikers is graphically simple game with few distinct objects, a combination of 16 and 32 filter sizes was chosen instead of 32 and 64.

Additionally, lower Stride sizes were chosen, as it was feared some of the smaller objects/features may be lost - This is also the reasoning for no MaxPooling layers being included initially, please see the section 'Experiments - Step

5' below for more discussion on this.

Below is a visual representation the network - excluding the Flatten Layer:

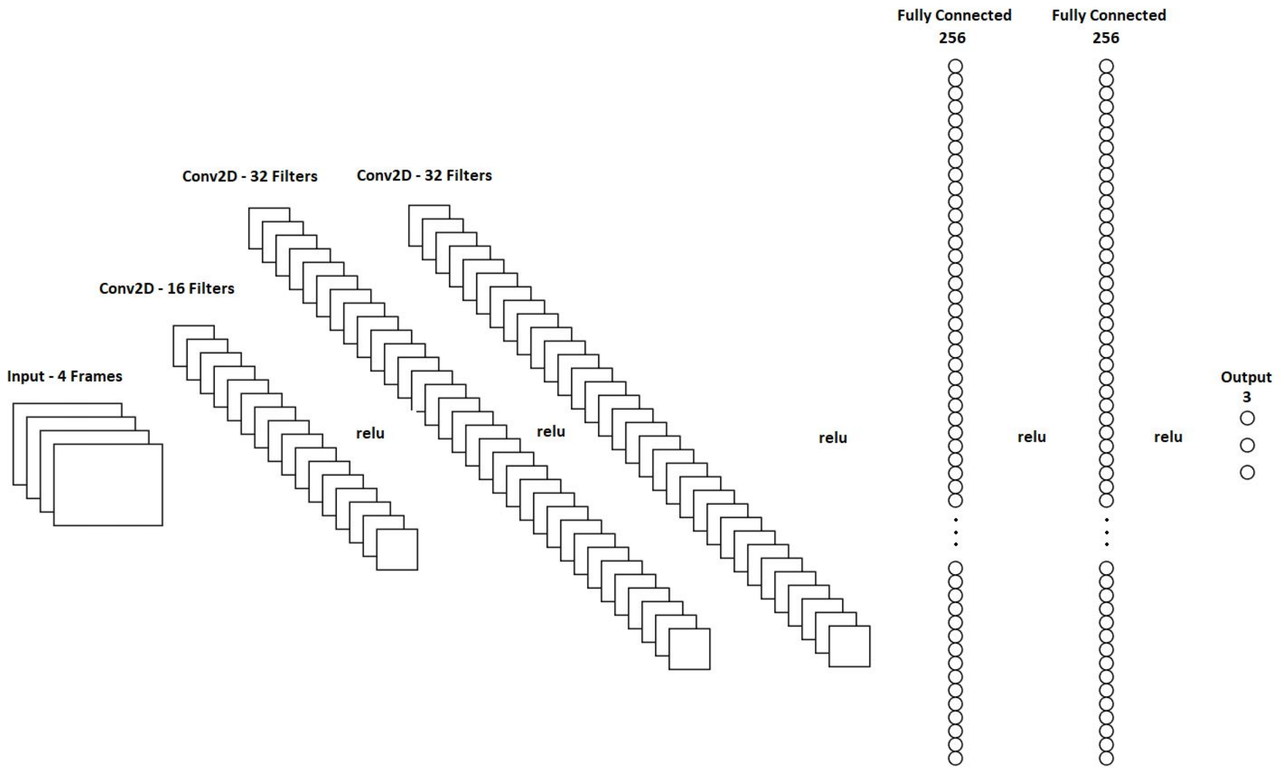
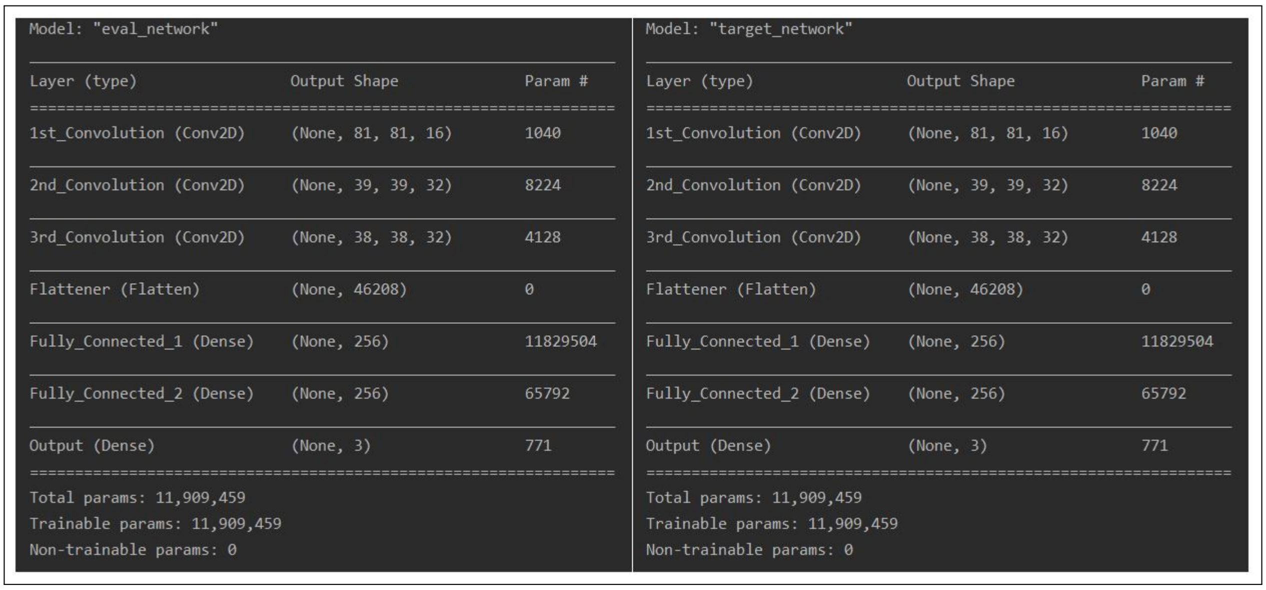


Figure 12: Basic Neural Network (Left to Right) - Flatten Layer between Conv2D and FC not shown

Below is the console output when creating two Neural Networks using the parameters described above, produced by the inbuilt **model.summary** function.



```
Model: "eval_network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_Convolution (Conv2D)     (None, 81, 81, 16)        1040
_________________________________________________________________
2nd_Convolution (Conv2D)     (None, 39, 39, 32)        8224
_________________________________________________________________
3rd_Convolution (Conv2D)     (None, 38, 38, 32)        4128
_________________________________________________________________
Flattener (Flatten)          (None, 46208)             0
_________________________________________________________________
Fully_Connected_1 (Dense)    (None, 256)               11829504
_________________________________________________________________
Fully_Connected_2 (Dense)    (None, 256)               65792
_________________________________________________________________
Output (Dense)               (None, 3)                 771
=================================================================
Total params: 11,909,459
Trainable params: 11,909,459
Non-trainable params: 0
```

```
Model: "target_network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_Convolution (Conv2D)     (None, 81, 81, 16)        1040
_________________________________________________________________
2nd_Convolution (Conv2D)     (None, 39, 39, 32)        8224
_________________________________________________________________
3rd_Convolution (Conv2D)     (None, 38, 38, 32)        4128
_________________________________________________________________
Flattener (Flatten)          (None, 46208)             0
_________________________________________________________________
Fully_Connected_1 (Dense)    (None, 256)               11829504
_________________________________________________________________
Fully_Connected_2 (Dense)    (None, 256)               65792
_________________________________________________________________
Output (Dense)               (None, 3)                 771
=================================================================
Total params: 11,909,459
Trainable params: 11,909,459
Non-trainable params: 0
```

Figure 13: Initial Neural Networks Evaluation (LHS) and Target (RHS)

Obviously, twelve-million is a huge number of trainable parameters. Again the Experiments section below explores this further.

Pre-processing, Game Model and Network Connection - Step 3

This section discusses the Game Model implementation of Airstriker-Genesis within **bot.py** along with the pre-processing prior to the environment information entering the Neural Network - the latter through the use of wrappers.

Pre-processing – Gym Wrappers

This covers the wrappers used to alter the Gym environment prior to the environment information entering the Network Neural. These wrappers include alterations to the state space (downsampling), stacking frames, stochastic frame-skipping and reward-clipping (all positives to 1).

Action Wrapper

As previously mentioned, the retro environment returns a length-twelve-multi-binary-boolean array as an action output. However, this results in a far larger number of possible actions than required. This is an issue as our action-space will also be our output layer. Meaning that convergence to optimal actions could take considerably longer. Therefore we need to reduce the action space to the minimum amount.

This can done through the a modified version of Stable-Baseline's SonicDistionizer, which takes the twelve-button input and returns only the game-relevant button combinations. Please, see the class **AirStrikeDiscretizer** within **retro_wrappers.py** - where only three actions are returned as valid inputs, 'Left', 'Right' and 'B' (where B is shoot).

Observation Wrapper, Frame-Stacker & Frame-Skipper

The default state provided by the retro environment, for Genesis games, is an array of shape (224, 320, 3). This array represents a single state snapshot from the game environment with the dimensions of height, width and RGB channels.

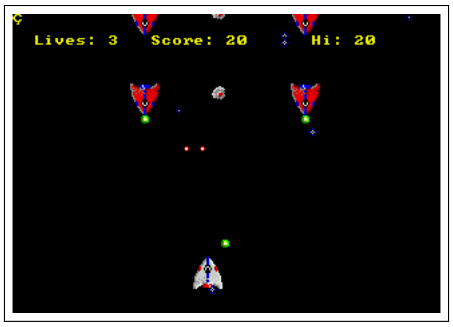


Figure 14: Airstriker-Genesis Native State Output (224, 320, 3)

Although more pixels is certainly more aesthetically pleasing, our agent does not really care about the extra detail. In fact, the additional level of detail is actually detrimental to our agent as there is a large amount of redundant information which results in a higher performance cost.

Therefore, it stands to reason that applying a pre-processing layer, before passing the state information into the neural network would be beneficial. The image should both be re-scaled and gray-scaled. Resulting in the shape of a single frame maintaining the dimensions of (84, 84, 1). These are the default down-sample settings provided by Stable-Baselines' *retro_wrapper*.

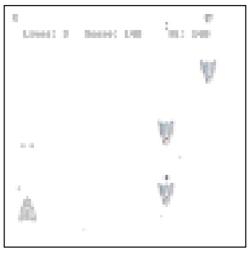


Figure 15: Airstriker-Genesis Downsampled to (84, 84, 1)

Another consideration is how our network will interpret the frames it receives. As mentioned above, a single frame provides little context as to the environment's movements. To get around this we can stack numerous frames on-top of one another and feed this into the network instead.

Again, using stable-baselines default - Stacking four frames on top of one another lets the agent see the motion of objects in the environment. Which then gives us an observation state of shape (84, 84, 4). This will be our Input for the Neural Network. This is the same as this paper [7] on Double Deep Q-Learning.

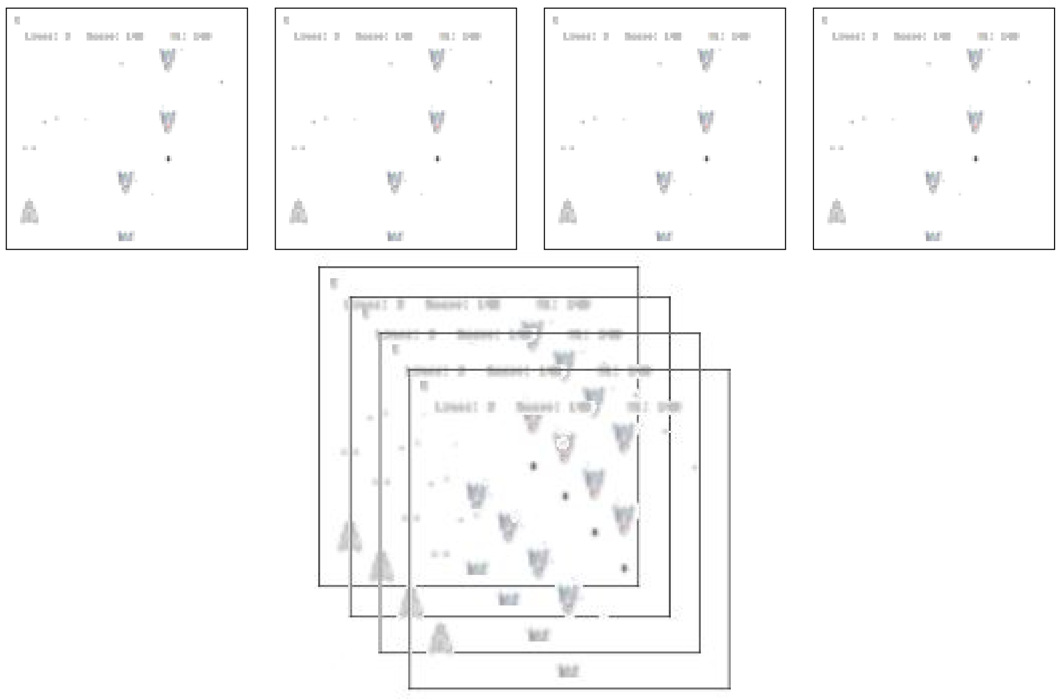


Figure 16: Four Down-sampled images stacked together (84, 84, 4)

Finally, we need to consider if every single state is necessary - Stable-Baselines' **StochasticFrameSkip** class was incorporated during the training of the model. This has two benefits:

- It significantly reduces the performance cost, and therefore training time, as the network does not need to analyse every single frame
- It ensures the environment is not fully deterministic - this should hopefully prevent the agent from brute forcing a solution from the environment as it would in traditional reinforcement learning (another way to stop this is to limit the total memory of the Deep Q-Learning model, but that also raises the question of how much memory is enough/too much)

Reward Wrapper

Stable-Baselines' **ClipRewardEnv** function was also incorporated. This limits rewards received from the environment to between {-1, 0, 1}. The main reasoning behind this is that is simplifies the model. I chose to clip only the positive rewards and heavily discourage the agent from dying by having a large -100 reward when the agent loses a life.

This implementation has the potential to encourage inefficient plays within AirStriker. As it will conceal rewards when two or more ships are shot down in the same frame. This will also lead to mismatching between the reward recorded in the **ReplayBuffer** and the actual score of the game.

Game Model - AirStriker Genesis

This section discusses the Game Model implemented for Airstriker - this is executed in **bot.py** under the functions **test** and **train**.

Game Resets - 'Done'

Airstriker's default model is to exhaust all lives before altering the 'done' value to True. However, in order to reduce complexity and the time spent on each episode, the implemented Game Model will only allow the agent to die once - resetting the game if the life count should ever decrease.

Also, it was initially thought that the information provided by the retro environment was limited in such a way that a user defined time-step would be needed, in order to ensure the model would reset correctly upon completing the game. Therefore a time-step limit of 1350 was chosen. This is the methodology used for the test results in the section below 'Experiments - Step 5'.

However, it has since been discovered that the environment return of **Info['gameover=4']** exclusively describes when the player reaches the end screen - this is now the implementation within the code and also has been used to means test various model weights in the section 'Initial Network (MSE) vs Highest Scoring (Huber-Maxpooling)'.

Negative Rewards

As mentioned in the Reward Wrapper Section above, the positive rewards have been clipped, from 20 points per ship to 1. The Game Model (outside of the wrappers) ensures the agent will encur a heavy negative reward if it dies -100 points. This is an attempt have the agent to recognise that losing is really bad. This also effectively grants a bonus reward of 100 points should the agent complete the game.

Further considerations were made as to whether or not the agent should be rewarded for completing a level. However, the additional complexity and unintentional behaviours possibly produced by the agent felt unnecessary for the scope of this project.

Linking the Game to the Network - State, Actions and Rewards

Below are two Flow-Charts describing a simplified version of the Python code for **train** and **test** found in **bot.py**:

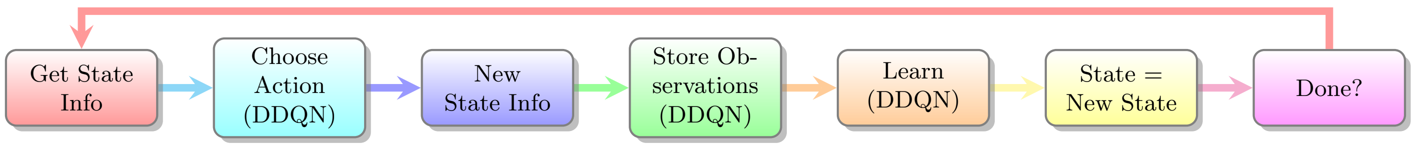


Figure 17: Training Flow-Chart

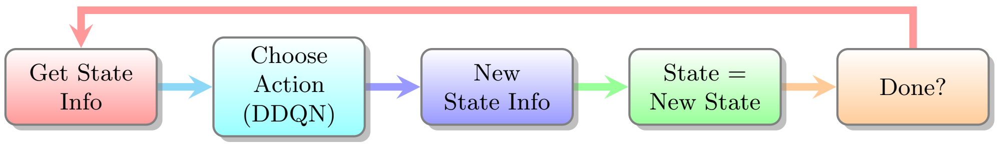


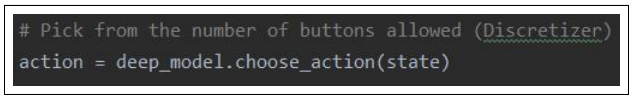
Figure 18: Testing Flow-Chart

As can be seen, the only real difference between both functions is that **test** does not store observations or learn from the actions it takes. Instead it will remain in a static state - only using the deep learning model to choose actions.

Our Game Model links to the Neural Network through the Deep Q-Learning Algorithm code found within **deep_model.py**. As mentioned above, the algorithm chosen is Double Deep Q-Learning. Because of this our game links to two different networks:

- eval_network to choose actions and act the main workhorse for the agent
- and target_network to calculate the Q-Values (and is updated only every 100 steps - by default)

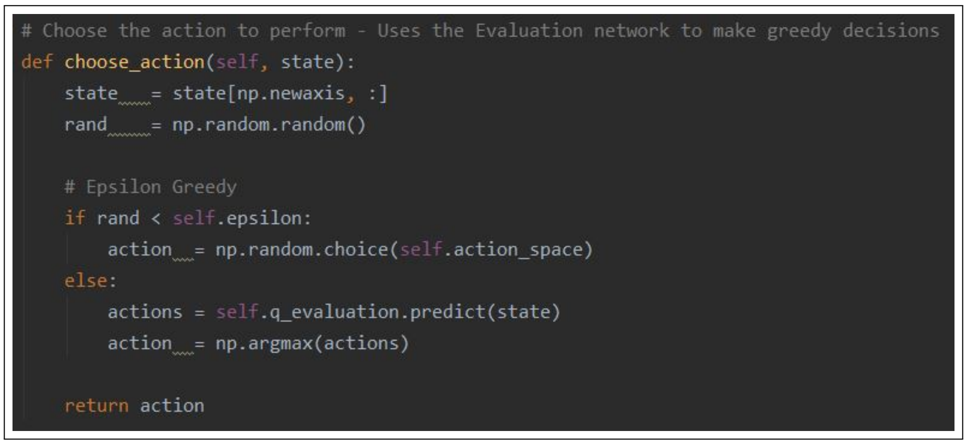
In the **train** function of **bot.py** our agent will choose an action using on this line:



```
# Pick from the number of buttons allowed (Discretizer)
action = deep_model.choose_action(state)
```

Figure 19: Have the Agent Choose an Action

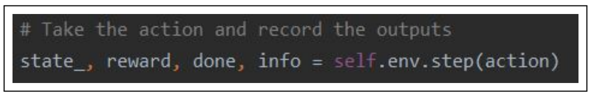
Which will then pick an action from our Evaluation Neural Network (nested within the Deep Learning Model):



```
# Choose the action to perform - Uses the Evaluation network to make greedy decisions
def choose_action(self, state):
    state   = state[np.newaxis, :]
    rand    = np.random.random()

    # Epsilon Greedy
    if rand < self.epsilon:
        action  = np.random.choice(self.action_space)
    else:
        actions = self.q_evaluation.predict(state)
        action  = np.argmax(actions)

    return action
```

Figure 20: Select Action Epsilon Greedy and Eval Network

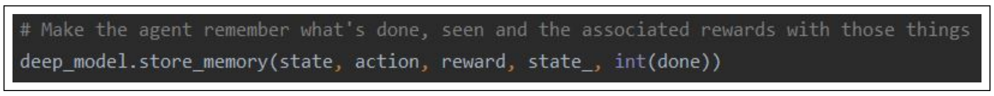
The action will be applied to the environment, returning next state, reward, done and info:



```
# Take the action and record the outputs
state_, reward, done, info = self.env.step(action)
```

Figure 21: Apply the Action to the Environment

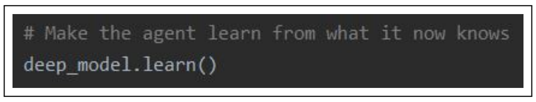
These are then recorded in the **ReplayBuffer** through the deep model:



```
# Make the agent remember what's done, seen and the associated rewards with those things
deep_model.store_memory(state, action, reward, state_, int(done))
```

Figure 22: Store the Memory in the ReplayBuffer

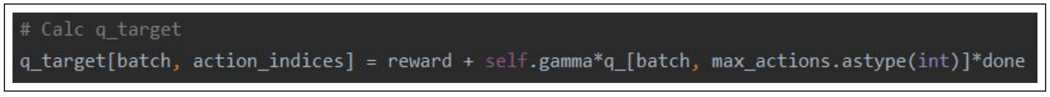
Finally the **learn** function is utilised to calculating the Q-values and update the network(s):



```
# Make the agent learn from what it now knows
deep_model.learn()
```

Figure 23: Deep Model - Learn

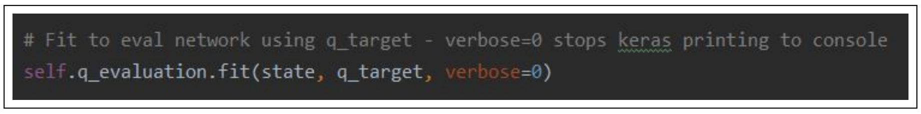
Here, we are calculating our Q(s, a) value - wrapping in our States, Actions and Rewards:



```
# Calc q_target
q_target[batch, action_indices] = reward + self.gamma*q_[batch, max_actions.astype(int)]*done
```

Figure 24: Update Q(s,a)

These are then all passed into the Neural Network through the **Model.fit** function which trains the network:



```
# Fit to eval network using q_target - verbose=0 stops keras printing to console
self.q_evaluation.fit(state, q_target, verbose=0)
```

Figure 25: DDQN Model fit Function

Double Deep Q-Learning implementation - Step 4

Here we will cover the deep reinforcement model implemented along in addition as to how to train, load weights and test it.

The relevant Python files are as follows, **nerualnetwork.py** and **ddqn_model.py**

Double Deep Q-Learning implementation (DDQN)

As mentioned previously, the Deep Learning Model chosen for this project is Double Deep Q-Learning (DDQN). DDQN is an expansion of Deep Q-Learning (DQN) and attempts to address the overestimation bias found in the latter [6].

The key difference between DDQN and DQN is that the former utilises two networks instead of one. The first Neural Network is designated as the Evaluation network and the second is the Target network. Essentially one is the workhorse, constantly interacting with the environment choosing actions and updating weights as it goes, and the other network calculates the Q-Values and then updates the Evaluation networks Q-Values every 100 steps. This delayed updating is what counter the overestimation bias found in DQN.

The implementation of DDQN is from Google Deepmind paper [7]. Below are the DQN and DDQN formulas taken from that paper:

$$Y_t^{DQN} \equiv R_{t+1} + \gamma max Q(S_{t+1}, a; \theta_t^-)$$

$$Y_t^{DoubleDQN} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax Q(S_{t+1}, a; \theta_t), \theta_t^-$$

Beyond DDQN

Although DDQN is successful in compensating for the overestimate of values issues found within Deep Q-Learning, there are still many other problems. One of which is Catastrophic Forgetting where the agent essentially forgets what bad plays look like and begins playing poorly. The agent should eventually recover given long enough - I believe this issue occurs in the testing below.

To combat these shortfalls, several other DQN extensions have been developed, including but not limited to Prioritised memory which addresses Catastrophic Forgetting. Below is a figure describing the performance of these various methods:

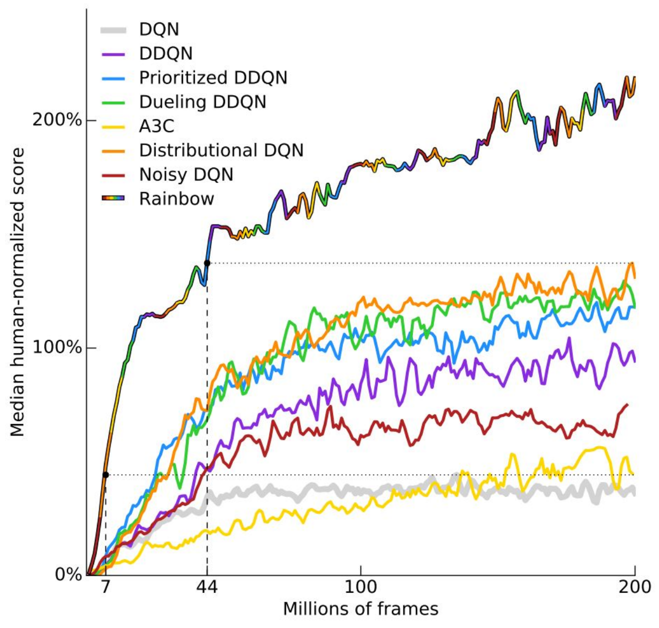


Figure 26: DQN and DDQN methods vs Rainbow DQN [6]

From Figure 19, it is clear that Rainbow DQN is significantly outperforming every other iteration of DQN. As the name implies, Rainbow combines techniques from the all the other DQN methods to produce outstanding results. This figure also provides insight as to what can be expected from the performance of this paper's DDQN model.

Model Parameters

These are the initial model parameters used:

Parameter	**Value**
Episilon	1
Episilon Decay	0.996
Epsilon Minimum	0.01
Alpha	0.0005
Gamma	0.99
Batch Size	8
Memory	20,000

Table 3: Initial Deep Model Parameters

The next section, 'Experiments - Step 5', alters the parameters somewhat as it works through examples. Hence, the parameters found in the code are aligned to the results found in that section, rather than the table above.

How to Train the model

From Scratch

You can start a basic training session by running **main_train.py**. By default this will begin running the same settings as listed in the 'Final Configuration' section below.

You can alter all the parameters listed above in Table 3, along with the number of games to be played. However, alterations to the Neural Network must be done manually.

Load a Model's Weights to Continue Training

It is possible to continue training from a model (.h5 file). When running the *train* function from the **Bot** class, enter the location and filename of the model as the parameter *model_name*. When doing this, ensure to adjust the the epsilon values accordingly.

Additionally if attempting to load weights from a model which was trained on a different Neural Network configuration an error will be generated.

How to Test a Model

A testing session can be started by running **main_test.py**. By default, this will load a model produced from the parameters found in the 'Final Configuration' section below.

You can also start your own by running the **test** function from the **Bot** class - ensuring that a model_name parameter has been passed through. Additionally, it also makes sense to set **greedy=True** for **ddqn_model.py**, as this will ensure the agent will pick, what it assumes to be, the best actions for each step.

Using the **test** function will likely produce varying results episode to episode. It will not necessarily produce a replay of the steps the agent originally took on the episode during training. This is because it will re-run using the Q-values from the model, and the agent may not have chosen highest value action for every step - due to the nature of epsilon greedy.

Experiments - Step 5

In this section we will explore how manipulating various parts of the model can impact the results. Due to the shear number of possible combinations of hyper-parameters between the DDQN algorithm and the Neural Network, this section only covers an extremely limited number possible configurations.

Batch Size

Batch Size can have a large impact on the performance provided by the DDQN algorithm. Below is a figure comparing the results of batch sizes 32 and 8:

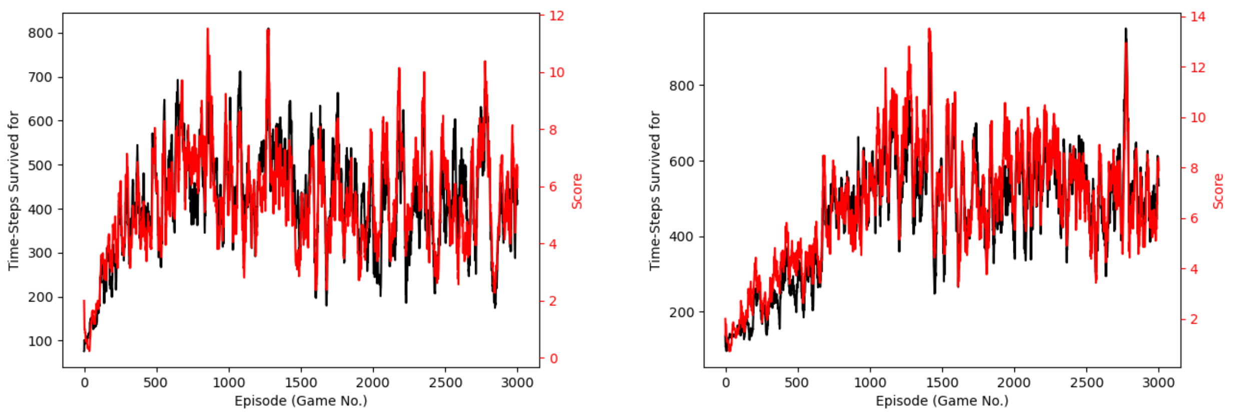


Figure 27: Batch Size 8 (LHS) vs 32 (RHS) - 3000 Episodes

Although a batch size of 8 achieved lower scores, given that the reduced size dramatically decreased training time, it was chosen as the parameter for tuning the network moving forward.

It would be preferable to experiment with further increases and decreases in batch size in combination with other hyper-parameters.

Mean-Squared-Error vs Huber-Loss

As mentioned above in the section 'Neural Network - Step 2' a second loss function was experimented upon. Figure 23, below compares the results of these two functions over the course of ten thousand episodes:

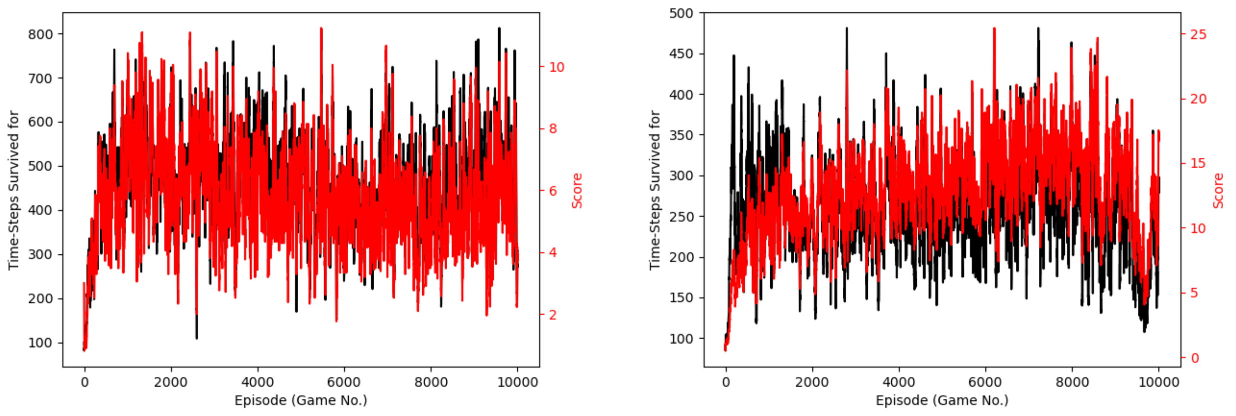


Figure 28: Mean-Squared-Error (LHS) Vs Huber-Loss (RHS) - 10,000 Episodes

The behaviours between the two are clearly very different (See 'Replays' and attached video files). MSE completed the game numerous times, sometimes in quick succession (resulting in high average time-steps survived). Huber did not see this same completion success, completing the game significantly fewer times - even when doubling the number of episodes to 20k (Figure 25).

However, Huber's scores were far higher, doubling the average scores achieved by MSE.

It's interesting that this pattern appears very early during training:

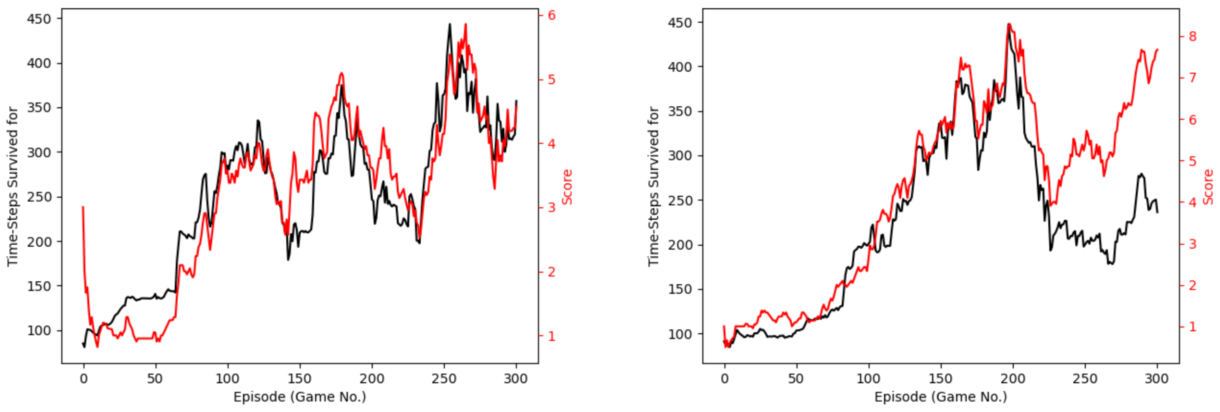


Figure 29: Mean-Squared-Error (LHS) Vs Huber-Loss (RHS) - 300 Episodes

Ignoring the start-values where the averages are till being calculated (average over 20 episodes). MSE score and time values correlate very strongly together (when the agent reaches a higher time stamp) it is likely it also receives a higher score value too. And although Huber-Loss follows the same pattern for the first 200 episodes, Score and Time begin to diverge from one another shortly thereafter.

This is clearly anecdotal, considering the small sample of episodes and given this is a single run. However, it is non-the-less interesting. It would seem that MSE is prioritising getting as far along as possible, as it reaches much higher average time-step values. Whereas, Huber-Loss prioritises achieving as high a score as possible. Part of this effect is likely due to MSE typically completing the game at much earlier episode numbers. Having not received a large negative reward, would encourage it to reach that same state each time.

The final figure in this section is a continuation of the ten-thousand episode run from above:

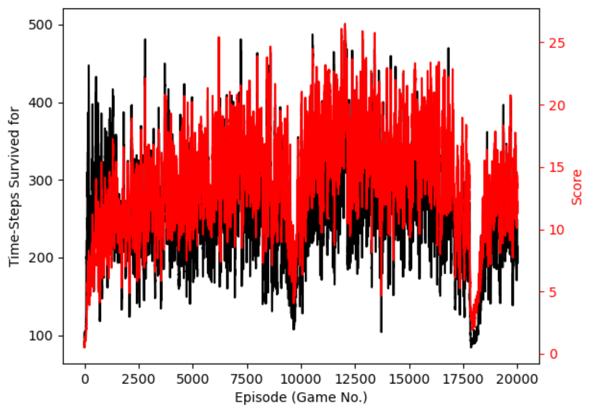


Figure 30: Huber-Loss - 20,000 Episodes

I was curious as to whether, given more time and considering the score ceiling is higher than the time ceiling if, the DDQN model would provide further rewards in this configuration. It appears however that an average score of around 25 may be the best attainable.

Furthermore, a closer look at the graph demonstrates what I believe to be an example of catastrophic forgetting. This is where the agent, after not seeing failure for awhile 'forgets' what failure looks like and begins acting in a less than favourable manner. If given long enough, the agent should eventually get back to the higher score values - which appears to be the case after episodes 10,000 and 17,500.

MaxPooling Layers

No Max pooling layers were originally added to the Neural Network. However, adding a single maxpooling2D layer following the network's final Conv2D layer (but preceding the flattening layer) led to a very interesting result:

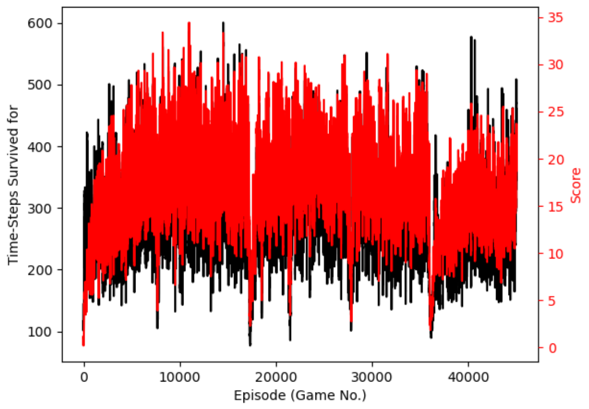


Figure 31: Maxpooling layer (2, 2) - 45,000 Episodes

Not only did adding the MaxPooling layer significantly reduce the number of parameters within the network (from 12-million to 3-million) and therefore reducing the model output size. But also, looking across 45,000 episodes we can see that the agent achieved scores far higher than the initial configuration.

Having trainable parameters could have removed a great deal of redundancy from the Network, allowing the network to converge faster. Hence, in terms of AirStriker at least, it looks as though the 12 million or so parameters available previously resulted in an over-fit. It's unlikely the number of parameters now is the optimal, but based on the results, it seems to be a better model.

Furthering the experiment above, the kernel size of the maxpooling layer was increase from (2, 2) to (4, 4):

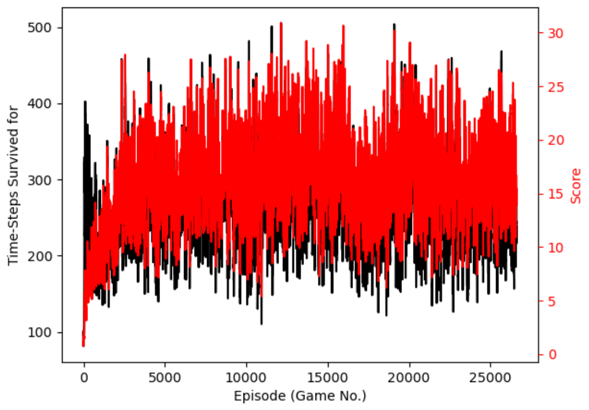


Figure 32: Maxpooling layer (4, 4) - 27,000 Episodes

Although this model does not appear to have hit the same peak of 35, interesting the catastrophic forgetting impact has been reduced substantially. There is still a dip on the max scores around 17k episodes, however the affects are far less dramatic.

Final Conv2D Strides

In the network's default state, the final Conv2D is potentially having minimal effect on the performance of the model as it does not reduce the output size by very much. Therefore, in-keeping with the idea of reducing available parameters to optimize the convergence, the last convolutional layer's strides have been increased from (1, 1) to (2, 2):

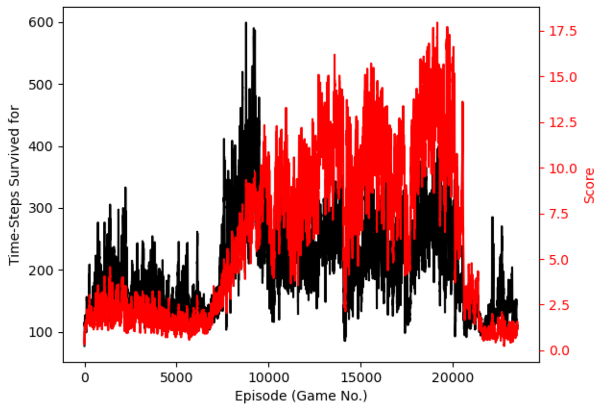


Figure 33: Maxpooling layer (4, 4) - Increased Stride of Final Conv2D Layer (2, 2)

It can be seen from the Figure 28 that the network only achieved half the score of the Figure 26. Furthermore the network took an extremely long time to get to this point - only beginning to show progress around 7.5k episodes. Though further testing within Figure 29 suggest this may be an outlier.

The most interesting part is how different this network's results look compared to the others. The same smoothing factor (average across 20 episodes) was implemented. However, the displacement from episodes to episode is far smaller than any of the other graphs. This may be a result of the significantly reduce number of trainable parameters.

Potentially, the drop-off at 20k is due to the agent reaching the end of the memory (which was set to 20k). The figures below attempted to establish if this could be remedied by a memory increase:

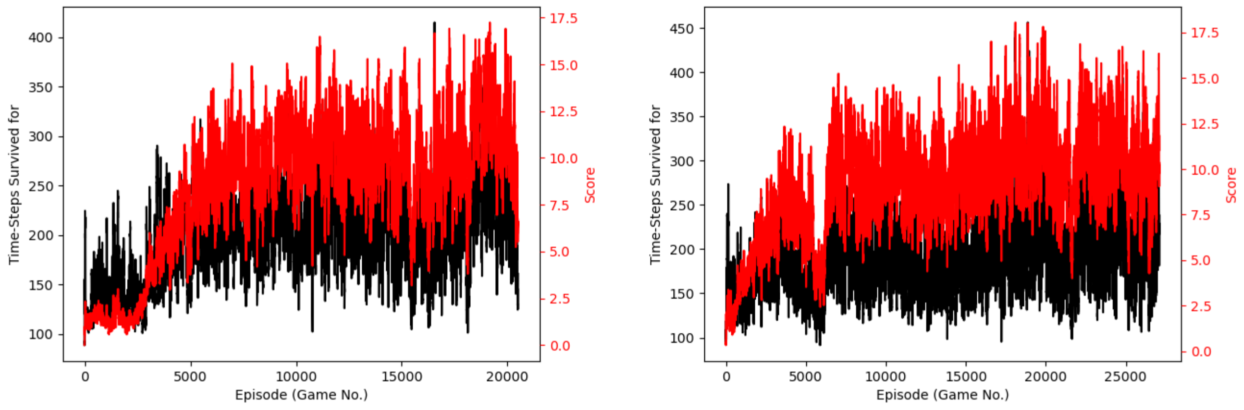


Figure 34: Maxpooling layer (4, 4) - Final Conv2D layer (2, 2) - Memory 100K (LHS) 50k (RHS)

Unfortunately, the program crashed after 20.5k episodes when using a Memory count of 100k. However 50k appears to be a stable value and does not present the same features as Figure 27. In-fact, in appearance it looks far closer to the earlier tests - with the exception of a much lower average score.

Because of this, the Final Configuration will use the parameters presented in Figure 26 - A Huber-loss function with a single MaxPooling layer of (2, 2). The third Conv2D layer will be left with a stride of (1, 1).

Initial Network (MSE) vs Highest Scoring (Huber-Maxpooling)

The Figures below show results taken from one-hundred runs utilising model weights created from the testing above. These are evaluation runs, hence, the DDQN algorithm utilises greedy=True. Which ensures the highest Q-values are always selected from the Network.

The tested models are the initial MSE network (Figure 23 - LHS) and Huber-maxpooling network (Figure 26) - the latter was shown to be the highest scoring in our testing.

NOTE: The code used to generate these results is slightly different. Instead of ending the game once a time-step of 1350 is reached, it ends when gameover=4 - this is the reason why the model is able to achieve time results above 1350.

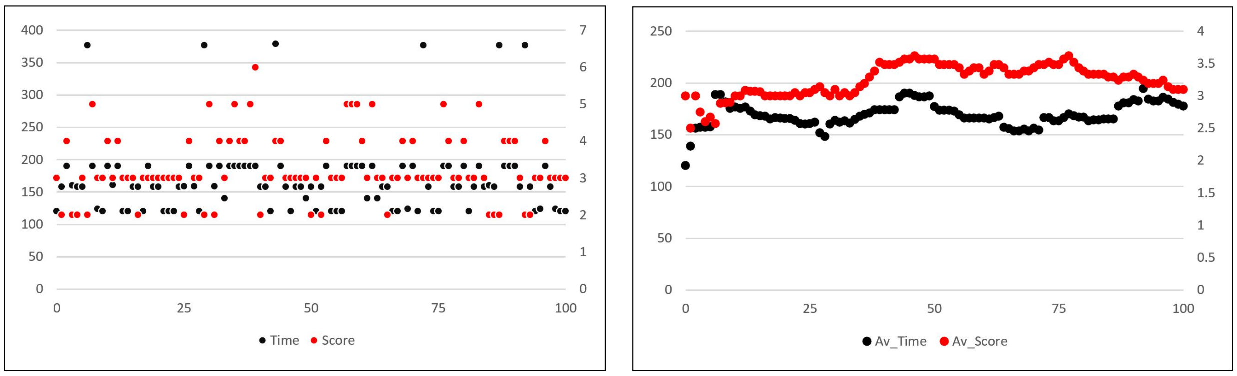


Figure 35: MSE - Actuals (LHS) and Averages (RHS) - 100 Episodes - (mse-100)

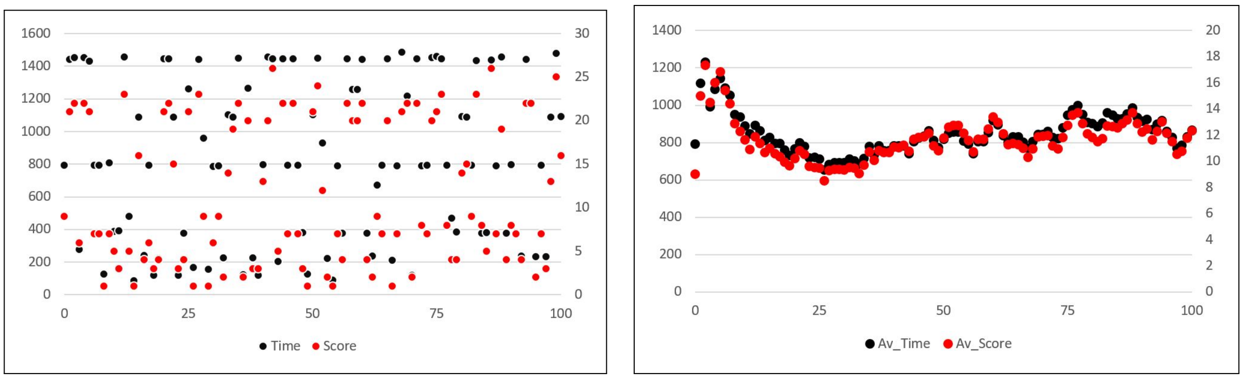


Figure 36: MSE - Actuals (LHS) and Averages (RHS) - 100 Episodes - (mse-1278)

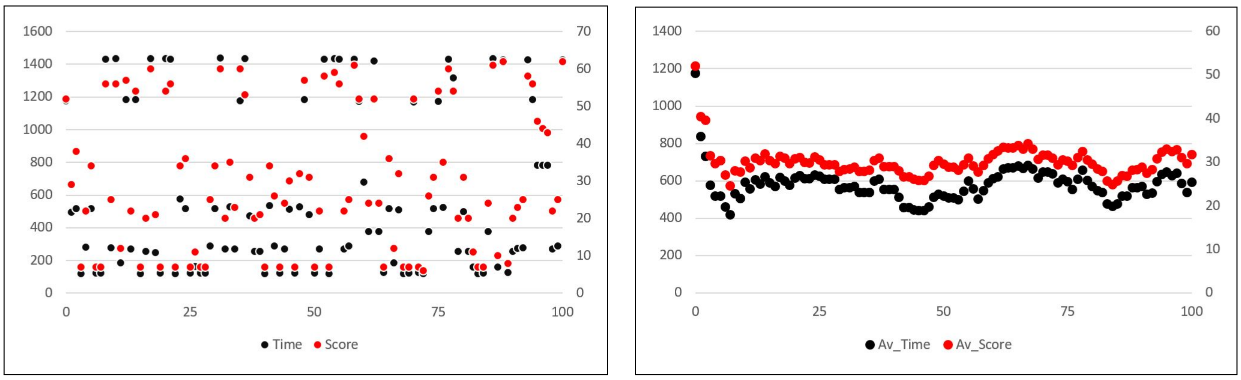


Figure 37: Huber Maxpooling (2, 2) - Actuals (LHS) and Averages (RHS) - 100 Episodes - (Huber-max 27229)

First of all, comparing Figure 30 to 31 or 32, it's very clear that that both agents improve greatly as they continue to play beyond the first 100 episodes. The weights provided by those first one-hundred episodes have the lowest average scores and never break a time-step value over 400.

Comparing Figure 31 and 32, the RHS graphs tell an expected story - producing results similar to those provided at their respective training time-stamps in the testing above.

What is interesting are the LHS charts which display the results of individual episodes. MSE's results appear well divided across the boundaries of the chart. However, Huber has a noticeable lack of data points between 800-1200 (Time). This is likely a result Huber's aggressive play style, encouraging more dangerous routes.

Further Testing

Testing time was always going to be a limiting factor on this project, as can be seen from some of the results above, a significant number of episodes would sometimes need to be run to see particular behaviours within a model's configuration.

There's also the fact that Network configuration can have huge impact on both the training speed and performance of the results provided. Time permitting, additional parameters should be explored including. Things such as the number of filters, layers, additional types of layers (such as dropouts), strides and activation types.

There are also Optimizers and Loss functions other than Adam, MSE or Huber. Additionaly, the DDQN algorithm hyper-parameters such as reducing the epsilon decay rate were not explored at all.

Finally other metrics would ideally be included, such as tracking the loss of each configuration - as shown in [8].

Replays - Step 5 Continued

Videos – .flv

Please see the attached .flv files for playback of the agent using different models - Both were provided in case of compatabilty issues. Below is a table describing the files:

File Name	**Description**
mse-100-8batches.flv	A video file using very early model weights from the initial mse Neural Network.
mse-1278-8batches.flv	A video file utilising the model weights provided by a completed run of episode 1278 using the initial mse Neural Network.
huber_maxpool-27229-8batches.flv	A video file utilising the model weights provided by a completed run of episode 27229 using the most optimal Neural Network discussed above. (This is also the submitted version of **main_test.py**)

Table 4: Replay File Listing and Accompanying Descriptions

These videos are cherry picked examples from the 100 episode run models above, and are clearly not representative of every possible episode outcome of the weights used. Regardless it's very interesting to see the different 'strategies' employed between the two. MSE takes a cautious approach, hugging the sides trying to avoid as many obstacles as possible. Whereas Huber is extremely aggressive and consistently flies across the path of oncoming objects.

Action Replay – .bk2 files

These should produce a perfect re-run of the steps the agent took, however, as stated previously Stochastic Frame skipping was enabled when training. Replay files (.bk2) produced when utilising this function result in inconsistent playback.

I am certain there is a way to make it usable. However, I was unable to come with a solution within the time constraints of the assignment - hence no .bk2 files have been provided.

Final Configuration

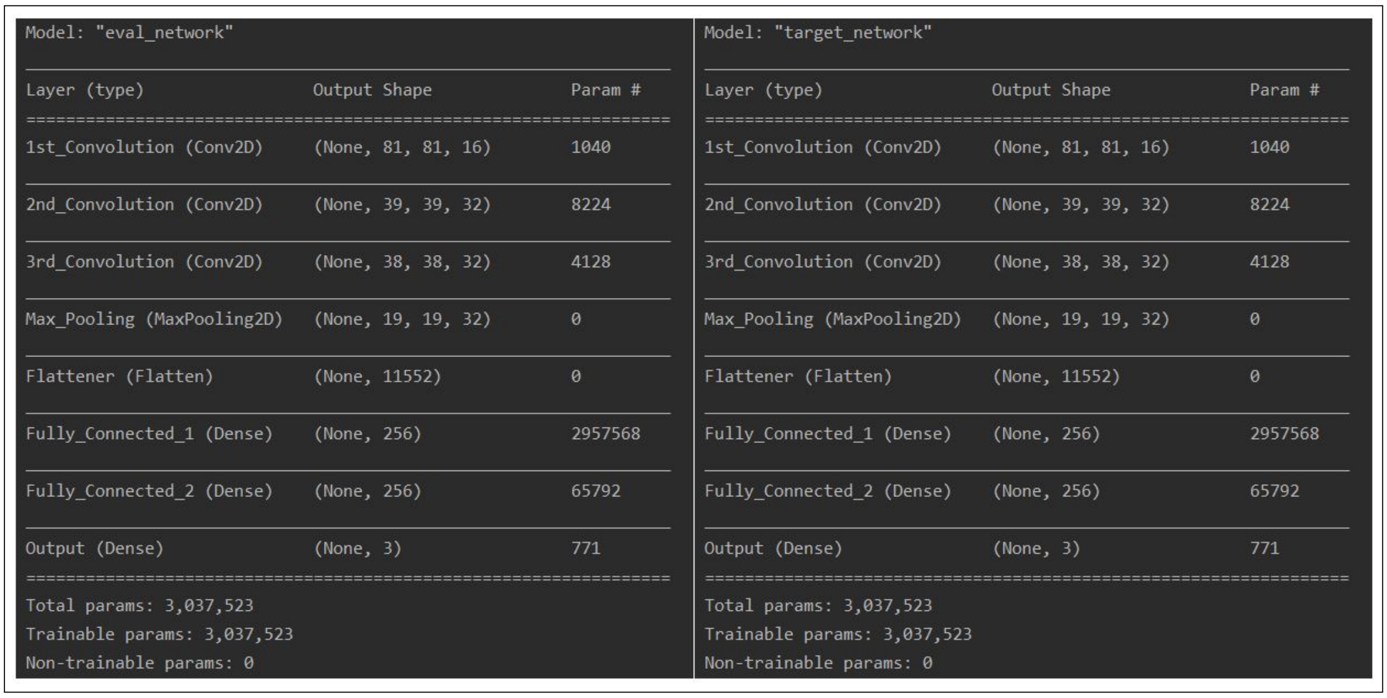
The Python code's final configuration has been left with the parameters from the highest scoring model above - Huber-Maxpooling (2,2):

Layer Name	Type	Filters / Nodes	Kernal Size	Strides
1st_Convolution	Conv2D	16	(4, 4)	(1, 1)
2nd_Convolution	Conv2D	32	(4, 4)	(2, 2)
3rd_Convolution	Conv2D	32	(4, 4)	(1, 1)
Max_Pooling	MaxPool2D	-	(2, 2)	None
Flattener	Flatten	-	-	-
Fully_Connected_1	Dense	256	-	-
Fully_Connected_2	Dense	256	-	-
Output	Dense	3 (n_actions)	-	-

Compile Property	Selection
Optimizer	Adam
Loss	Huber()

Parameter	Value
Epsilon	1
Epsilon Decay	0.996
Epsilon Minimum	0.01
Alpha	0.0005
Gamma	0.99
Batch Size	8
Memory	50,000

Table 5: Configuration of Final Neural Network and Deep Model



```
Model: "eval_network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_Convolution (Conv2D)     (None, 81, 81, 16)        1040
_________________________________________________________________
2nd_Convolution (Conv2D)     (None, 39, 39, 32)        8224
_________________________________________________________________
3rd_Convolution (Conv2D)     (None, 38, 38, 32)        4128
_________________________________________________________________
Max_Pooling (MaxPooling2D)   (None, 19, 19, 32)        0
_________________________________________________________________
Flattener (Flatten)          (None, 11552)             0
_________________________________________________________________
Fully_Connected_1 (Dense)    (None, 256)               2957568
_________________________________________________________________
Fully_Connected_2 (Dense)    (None, 256)               65792
_________________________________________________________________
Output (Dense)               (None, 3)                 771
=================================================================
Total params: 3,037,523
Trainable params: 3,037,523
Non-trainable params: 0
```

```
Model: "target_network"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
1st_Convolution (Conv2D)     (None, 81, 81, 16)        1040
_________________________________________________________________
2nd_Convolution (Conv2D)     (None, 39, 39, 32)        8224
_________________________________________________________________
3rd_Convolution (Conv2D)     (None, 38, 38, 32)        4128
_________________________________________________________________
Max_Pooling (MaxPooling2D)   (None, 19, 19, 32)        0
_________________________________________________________________
Flattener (Flatten)          (None, 11552)             0
_________________________________________________________________
Fully_Connected_1 (Dense)    (None, 256)               2957568
_________________________________________________________________
Fully_Connected_2 (Dense)    (None, 256)               65792
_________________________________________________________________
Output (Dense)               (None, 3)                 771
=================================================================
Total params: 3,037,523
Trainable params: 3,037,523
Non-trainable params: 0
```

Figure 38: Configuration of Final Neural Network Structure and No. of Trainable Parameters

Full Python Environment

This section is merely for troubleshooting purposes if the Python code is not executing.

The terminal command 'pip freeze local' provides this listing of packages and their accompanying version numbers:

Name	Version
absl-py	0.9.0
astor	0.8.1
cachetools	4.0.0
certifi	2019.11.28
chardet	3.0.4
cloudpickle	1.3.0
cycler	0.10.0
future	0.18.2
gast	0.2.2
google-auth	1.11.3
google-auth-oauthlib	0.4.1
google-pasta	0.2.0
grpcio	1.27.2
gym	0.17.1
gym-retro	0.7.1
h5py	2.10.0
idna	2.9
Keras-Applications	1.0.8
Keras-Preprocessing	1.1.0
kiwisolver	1.1.0
Markdown	3.2.1
matplotlib	3.2.1
numpy	1.18.2
oauthlib	3.1.0
opencv-python	4.2.0.32
opt-einsum	3.2.0
protobuf	3.11.3
pyasn1	0.4.8
pyasn1-modules	0.2.8
pyglet	1.5.0
pyparsing	2.4.6
python-dateutil	2.8.1
PyYAML	5.3.1
requests	2.23.0
requests-oauthlib	1.3.0
rsa	4.0
scipy	1.4.1
six	1.14.0
tensorboard	2.1.1
tensorflow	2.1.0
tensorflow-estimator	2.1.0
termcolor	1.1.0
urllib3	1.25.8
Werkzeug	1.0.0
wrapt	1.12.1

Table 6: Listing of all Python Packages Within the Development Environment

References

[1] Lecture Notes

[2] (The Book) Richard S. Sutton and Andrew G. Barton – ”Reinforcement Learning: An Introduction”

[3] https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

[4] ”Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks” ISBN: 1505714346

[5] https://pathmind.com/wiki/neural-network

[6] https://arxiv.org/pdf/1710.02298.pdf - Rainbow DQN

[7] https://arxiv.org/pdf/1509.06461.pdf - DDQN

[8] https://arxiv.org/pdf/1505.00853.pdf - Activation Functions