Machine Learning and BioInspired Optimisation (COMP532)

Assignment 2

Name: Beatrice Carroll, Daniel Fox, Sophie Hook, Devon Motte

Student ID: 201532871, 201278002, 201104221, 201145403

Student Email: b.d.carroll@liverpool.ac.uk, d.fox4@liverpool.ac.uk, sgshook2@liverpool.ac.uk, hsdmotte@liverpol.ac.uk

Date: 23/04/2021

Contents

1.0 Step 1: Import an OpenAI Gym 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.

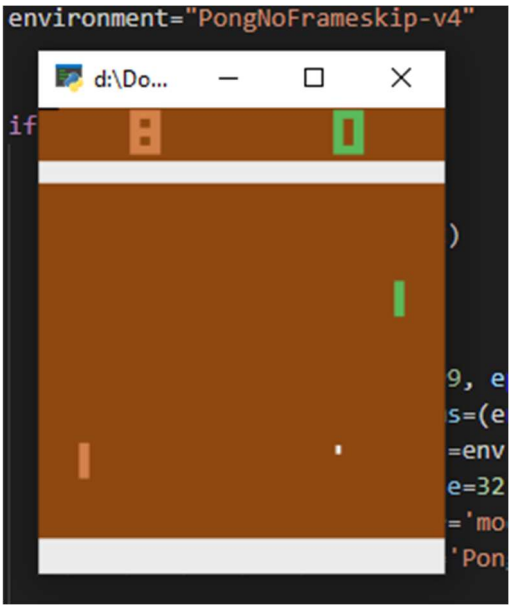

Figure 1: Imported Atari game Pong

The first step was to import an environment from the OpenAI Gym website (gym.openai.com). The environment chosen was from the Atari 2600 games as shown in Figure 1. The game used was PongNoFrameskip-v4 which loaded successfully, at the time of this report it was the most up to date version. PongNoFrameskip is often used over regular Pong because there are some performance and bug issues relating to the normal Pong environment (1).

2.0 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.

When creating the network, we modified an existing program already written by a youtuber, 'Machine Learning with Phil' (2). As a result of our inexperience with pytorch, deep learning and time constraints given it was more beneficial for our learning to implement an existing, functioning program, allowing us to achieve the desired outcome. We did first attempt to create a program of own, but realistically it was not feasible given the time taken to debug due to the run time. Furthermore, it was slightly too challenging given out experience with pytorch. It is clear in the program we submitted that we full comprehend the code and have included out own comments, explaining in detail what is achieved, in addition to making modifications which helped us to further develop our understanding.

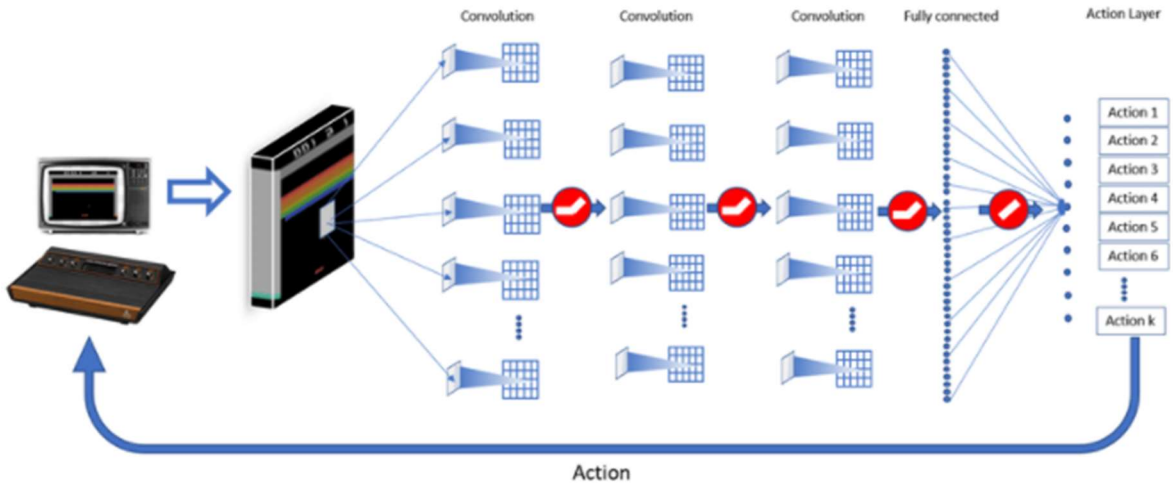

Figure 2: Architecture of Deep Q-networks, ReLU nonlinear units are emphasized by red circles (3)

Figure 2 depicts the deep neural network model used in the version of our code. The model uses three convolutional layers to process the image and then two fully connected layers. The rectified linear activation function, ReLU for short, was used because it helps scale with the CNN as described by Baeldung (4). They state that certain neuron input values can cause a negative output of that neuron, contributing negatively to the overall output of the network. Hence, it is desirable to use non-negative activation functions such as ReLU. Furthermore, ReLU helps with computation as its simple to calculate the gradient when performing learning and performing backpropagation.

CNN Is used first as it helps scale down the computation requirements when processing the Atari game. Firstly, 4 raw frames of gameplay are taken at once in RGB at scale of 84x84x1 the image is then processed and converted to greyscale as it is easier to computer greyscale since there is only one channel of size 0-255. The StackFrames function stacks the game environment so the game has four frames as shown in Figure 2. These functions were included in the open ai gym wrappers, which have premade functions including those that can be used to process the images in the Atari game. The two main wrappers utilized were gym.Wrapper and gym.ObservationWrapper.

```
def main():
    #create environment.
    env = make_env(environment)
def make_env(env_name, shape=(84,84,1), repeat=4,
             no_ops=0):
    #just defaulted shape to 84,84,1 common when researching to use this.
    #repeat 4 times like in papers.

    #calling above functions , passing env each time so we bascilly stack each change on the environment as we go.
    env = gym.make(env_name)
    env = RepeatActionAndMaxFrame(env, repeat, no_ops)
    env = PreprocessFrame(shape, env)
    env = StackFrames(env, repeat)

    return env
```

Figure 3: utils.py image processing functions

Figure 3 outlines the process order in which the environment processes the images before being sent through the convolutional network. Once the image has pre-processed it is passed into the first convolutional layer with the parameters:

- First input channels = 4
- First output channels = 32
- Kernel size = 8
- Stride = 4

The first input layer relates to the four pre-processed images created from the pre-processing. The output channel is set at 32 output neurons with 8x8 filters. The next step requires further scaling down of the convolutional layers resulting in a flattened image, so it can be passed through the fully connected layers.

- Second input channels = 32 (last layer output)
- Second output channels = 64
- Kernel size = 4
- Stride = 2

The second convolutional layer is set to produce an output of 64 neurons with kernel size of 4x4 along with a stride of two. The final convolutional layer has the following parameters:

- Third input channels = 64 (last layer output)
- Third output channels = 4
- Kernel size = 3
- Stride = 1

The final layer will fully convolve at 64 inputs and flatten the image with 64 output channels which represent at 3x3 kernel size while stride is set at one. Since the layers are flattened it can be passed into two fully linearly connected layers. An image of a similar CNN network is shown below in Figure 4 with the same processing used. Figure 5 exhibits the DQN class which implements this is in the program.

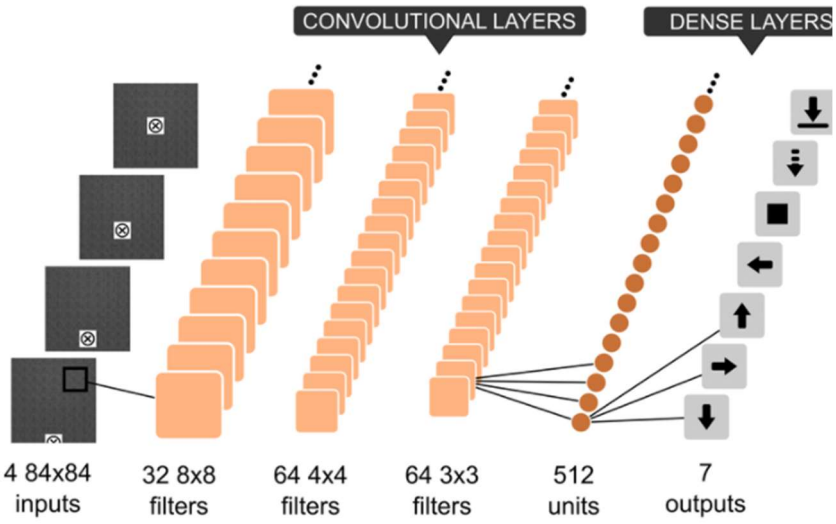

Figure 4: Close representation to the CNN network model (5)

Once the final convolutional layer is flattened the data is processed into a fully connected linear layer. The input dimensions are calculated by considering the product of the final convolutional layer the value, which is calculated to be 3136 inputs (64*7*7=3136 neurons and 512 outputs).

- Fully connected layer one inputs = 3136
- Fully connected layer one output channels = 512

After the first fully connected layer has been passed through the final fully connected layer, it will represent outputs based on possible actions the agent can choose. There will be six possible actions for the Pong game.

- Last fully connected layer one inputs = 512
- Last fully connected layer one output channels = 6 actions

Figure 5 exhibits the DQN class which implements this is in the program.

```python
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np

class DeepQNetwork(nn.Module):
    def __init__(self, lr, n_actions, name, input_dims, chkpt_dir):
        super(DeepQNetwork, self).__init__()
        """
        Creation of deepQNetwork.
        The DQN (Deep Q-Network) algorithm was developed by DeepMind in 2015

        lr=Learning rate
        n_acitons=number of actions
        name=name of checkpoint file
        input_dims=Input dimensions of environment.
        chkpt_dir=checkpoing directory.

        uses the nn.Module to inherit subclass. helps take and receive tensor inputs and outputs. Can call things like zero_grad. forward function, nn.Linear ect..

        conv = convolution layers for cnn network, image processing
        fc=full connected layers
        optimizer=helps shape and mold model to be as accurate as possible. RMSprop is a gradient based optimization technique
        loss=calculates the difference between the network output and its expected output. MSE = Mean squared error.



        """
        #device init , gpu if available otherwise use cpu.
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

        self.checkpoint_dir = chkpt_dir#checkpoint directory to save and load file
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name)#checkpoint file
        #print(input_dims)
        #input dims=num channels in our input image should be 4x1 frames  because turns greyscale 1 channel since greyscale if color it would be 4x3 for 3 rgb colors.
        self.conv1 = nn.Conv2d(input_dims[0], 32, 8, stride=4)#first conv layer 4 input dimentions so 4 frames of pong, 32filters output channels,8x8kernal, stride 4
        self.conv2 = nn.Conv2d(32, 64, 4, stride=2)#second cnn layer 64,4x4 with stride 2
        self.conv3 = nn.Conv2d(64, 64, 3, stride=1)#final conv3 layer is 64 with 3x3 and stride is equal 1. #Which is flattened out for the 2 fully connected layers in the network.


        #fully connecter layer input dimensions
        fc_InputDims = self.calc_convOutputDims(input_dims)#figure out the fully connected inputs for fc1.
        #print(fc_input_dims)#3136
```

```python
        fc_InputDims = self.calc_convOutputDims(input_dims)#figure out the fully connected inputs for fc1.
        #print(fc_input_dims)#3136

        self.fc1 = nn.Linear(fc_InputDims, 512) #512 neuron outputs
        self.fc2 = nn.Linear(512, n_actions)#final outputs are possible actions.
        #print(n_actions)#6 actions

        #optimizer RMS used in deep learning paper.
        #optimizes params of our network
        self.optimizer = optim.RMSprop(self.parameters(), lr=lr)
        #Means squared error to workout loss for backpropagation
        self.loss = nn.MSELoss()

        self.to(self.device)#init to device pytorch funtion.
```

```python
    def forward(self, state):
        """
        state of the env as input and state can be a batch size of one by input dim or a batch size of 32 by inputs dims.

        pass state through conv layers and use a activation relu through each layer apart from last fully connected as its not needed.

        we reshape the final conv layer becuase we want to pass in the something that has a shape batch size by whatever num of input features, into the fully connected layer.
        [0] element is batch size
        -1 is telling the function that it wants to flatten the rest of the functions

        the final output of the deep neural network is the action values of the given state. Note no activation function used for the final fc layer.

        """

        conv1 = F.relu(self.conv1(state))
        conv2 = F.relu(self.conv2(conv1))
        conv3 = F.relu(self.conv3(conv2))
        # conv3 shape is batchsize x num filters x Height x Width (of final conv image not the input images)
        conv_state = conv3.view(conv3.size()[0], -1)
        # conv_state shape is BS x (n_filters * H * W)
        flatten1 = F.relu(self.fc1(conv_state))
        actions = self.fc2(flatten1)

        return actions

    def calc_convOutputDims(self, input_dims):
        """
        find the input dimensions for the fully connected layer (4x84x84), taking total input dimensions
        return the product of the last layer size.
        """
        state = torch.zeros(1, *input_dims)#state matrix as batch size one with input dims.
        x = self.conv1(state)#pass into conv layers.
        x = self.conv2(x)
        x = self.conv3(x)
        return int(np.prod(x.size()))#best to return int float not needed.
```

Figure 5: DQN Class

3.0 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.

```
def storeTransition(self, state, action, reward, next_state, done):
    """
    Store transistions in the agents memory by calling the replaymemory file.
    takes state action reward next state and done.
    """
    self.memory.storeTransition(state, action, reward, next_state, done)

def sampleMemory(self):
    """
    CONVERTING TO PYTROCH TENSORS.
    Store the sample memory for agent and convert to pytorch tensors
    we samples agents memory buffer and store the resulting tensors sarsa and done.

    """
    state, action, reward, new_state, done = self.memory.sample_buffer(self.batch_size)
    #converts to torch tensors.
    states = torch.tensor(state).to(self.q_eval.device)
    actions = torch.tensor(action).to(self.q_eval.device)
    rewards = torch.tensor(reward).to(self.q_eval.device)
    next_states = torch.tensor(new_state).to(self.q_eval.device)
    dones = torch.tensor(done).to(self.q_eval.device)
    return states, actions, rewards, next_states, dones
```

Figure 6: Agent.py SARSA and Done states

```
"""
if self.memory.memCounter < self.batch_size:
    return

#first thing we want to do is zero the gradients of our optimizer when learning
self.q_eval.optimizer.zero_grad()

self.replaceNetworkTarget()#call function

states, actions, rewards, next_states, dones = self.sampleMemory()#we sample the memory sarsa and dones
index = np.arange(self.batch_size)

q_pred = self.q_eval.forward(states)[index, actions] #feed states through q eval network. giving out action values.
q_next = self.q_next.forward(next_states).max(dim=1)[0] #feed the next states throgh the network

#now work out the target values
q_next[dones] = 0.0 #type of mask
q_target = rewards + self.gamma*q_next #qtarget calcs if q dones is terminal (0) then q target is just equal rewards.

loss = self.q_eval.loss(q_target, q_pred).to(self.q_eval.device) #from dqn model
loss.backward() #backpropagation
self.q_eval.optimizer.step() #call optimizer and step
self.learnStepCnt += 1#increment counter
self.decrement_epsilon()#decrement epsilon
```

Figure 7: Agent.py Learn function

```
def step(self, action):
    """
    From gym.Wrapper class, function step takes action and steps through environment.
    """
    t_reward = 0.0
    done = False
    for i in range(self.repeat):
        obs, reward, done, info = self.env.step(action)#call step function from wrapper and produce outputs for states, rewards,info and done
        t_reward += reward #increment the reward to total reward by plus or minus 1
        idx = i % 2
        self.frame_buffer[idx] = obs
        if done:
            break

    max_frame = np.maximum(self.frame_buffer[0], self.frame_buffer[1])
    return max_frame, t_reward, done, info

def reset(self):
    """
    From gym.Wrapper class, function reset , resets the environment when done is true.
    """
    obs = self.env.reset()
    no_ops = np.random.randint(self.no_ops)+1 if self.no_ops > 0 else 0 #find the n operations we want to make which is a random number between the number of operations
    for _ in range(no_ops):#iterate over n ops
        _, _, done, _ = self.env.step(0) #determine from step function if done. other params are not needed.
        if done:#if output for done is true, then finished then reset and go again.
            self.env.reset()

    self.frame_buffer = np.zeros_like((2,self.shape))
    self.frame_buffer[0] = obs

    return obs
```

Figure 8: utils.py Wrapper functions for step and reset

```
#*************************************************************************************
#                    PLAYING EACH GAME N NUMBER OF TIMES./TRAINING AGENT
#*************************************************************************************

numSteps = 0
scores=[]
epsilonHistory=[]
steps_array = []

for i in range(NUM_GAMES):#loop through games
    done = False#done should be set at false from start of each game
    observation = env.reset() #call to reset environment with each game played
    score = 0 #score should be set at 0

    while not done:
        action = agent.chooseAction(observation) #call choose action and pass in obs
        new_observation, reward, done, info = env.step(action) #call wrapper function to step and get output params back. Take input action
        score += reward #add reward to total score for the game played.

        if RENDER_GAME: #THIS OPENS THE GAME. SO YOU CAN SEE GAME PLAY
            env.render()

        if not LOAD_CHECKPOINT: #AGENT Learns if loading checkpoint is set at false checkpoint
            agent.storeTransition(observation, action,reward, new_observation, done) #store each transaction
            agent.learn() #caall learn function
```

Figure 9: Main.py Observation state, reward, done, info

4.0 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.

4.1 Introduction

The model used in our deep learning program is the standard deep Q-learning network model which was presented in the DeepMind paper (6). The paper introduced a deep reinforcement learning model which takes several Atari games and passes them through a deep learning network. The goal achieved was achieving high score averages well above human expert level gameplay.

4.2 Deep reinforcement learning model

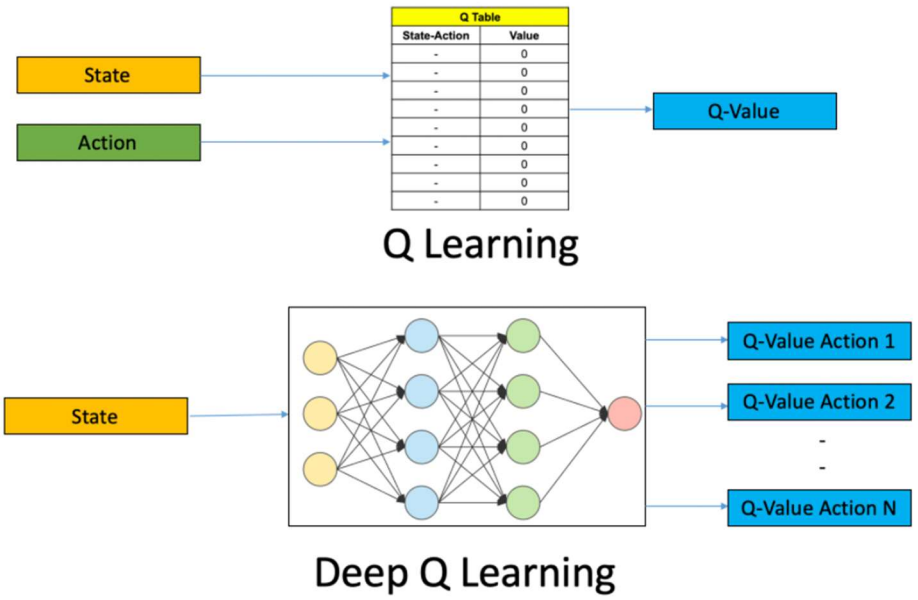

Figure 10: Comparison between Q-Learning and DQN (7)

Firstly, the difference between the usual Q-learning model and deep Q-learning model is that the deep Q-learning (DQN) model was designed to handle environments involving continuous actions and states, whereas the Q-learning model works best for small and discrete environments. When dealing with games like Atari which involve continuous state and environment changes the DQN variant was an ideal model to use as it has been tested. DQN models replace an exact value function with a function approximator, whereas Q-learning models update one state/action at each timestamp. Therefore, the DQN updates many values at once. It does this by considering the predicate model and training this, rather than checking and updating a Q-table from traditional Q-learning models.

The DQN approximates the state value function in a Q-learning environment with a neural network. The Atari games takes several frames of the game as an input (in our case 4 frames at once) and then outputs state values for each action. The DQN model uses a combination of convolutional neural networks (CNN) and fully connected linear neural networks. The CNN is necessary for processing the data produced by the Atari due to the game being in RGB and running continuously it is required to use a CNN to flatten the image and pass it through a traditional fully connected linear network which then produces possible actions as the outputs.

4.2.1 Training and updating

The DQN model uses a mix between a learning model and a target model, it is trained using an optimizer in which the online training model uses the RMSProp algorithm to update the network with gradient descent. The target network handles the calculations of the target values, and only gets updated periodically with the weights of the online network.

$$Q(s,a) = Q(s,a) + \alpha(\underbrace{R + \gamma max Q(s',a')}_{target} - Q(s,a))$$

$$Q(s,a) \rightarrow R + \gamma max Q(s',a')$$

Figure 11: DQN Target network calculation (7)

The DQN equation to update the state values using the target network can be seen in Figure 11, where alpha is the learning rate and gamma is the discount factor. Q(s,a) is the current state, Q(s',a') is the next state and R is the reward. The target model values are indicated where the max value is taken from the next state multiplied by discount factor and then adding the reward. Since we do not know the true target values, we must take estimates. The target network acts as an error measure, it attempts to try one value of the target network until it finds a better value. This gives the network more time to consider actions that have taken place recently instead of constantly updating all the time. The target network is updated after each batch has been reached. Figure 12 outlines the target Q-network update as part of a flow diagram for a DQN model.

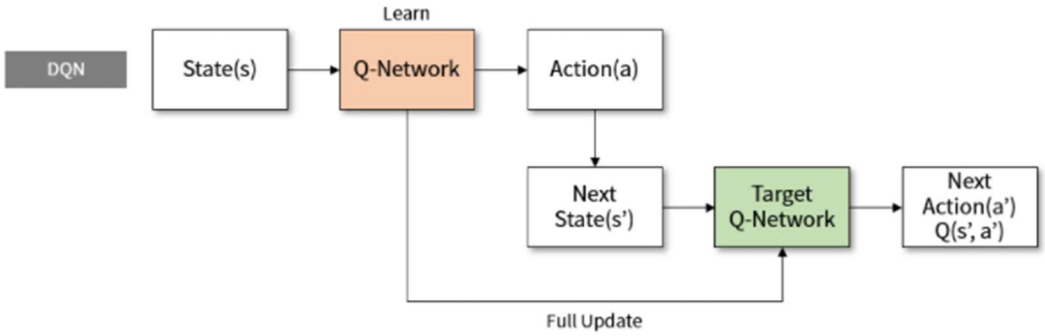

Figure 12: Structure of learning using target network in DQN (7)

DQN uses epsilon greedy to select actions. To improve training the epsilon value is set at 1 and an epsilon decay value is set to reduce the chance of exploring overtime. Meaning, once the agent has gathered enough training data it should not need to explore as often, thus the epsilon decrement is used. The epsilon value is high at first to make the agent explore as often as possible early in the episodes so it can learn. There is an epsilon minimum value set at 0.1 this is the minimum value the epsilon will reduce to (see Agent.py comments for further notes). The epsilon decay value is set at $1x10^{-5}$ and approximately after every 100,000 steps (named replace in code = 1000) epsilon will decrement.

The model also uses experience replay for storing episode steps (replace = 1000) and once the batch size has been reached it will switch to off-policy learning where the samples stored in the

experience memory are selected at random. To select the replay memory, we figure out the position of the last stored memory once the agent's memory has filled. We want to sample a range, up to agent's capacity memory size and if the memory is not at capacity, we use a memory counter to select the last position. Thus, the memory can be selected from either the minimum memory counter or the maximum memory size. The batch stored memory is then randomly selected. It is set uniformly so that the value of memory can only be selected once eliminating the change of repeating memory samples. This then produces the state action reward future state and the terminal done state out of the memory (more details are commented in the ReplayMemory.py file).

4.3 Parameters

The parameters of the model are as follows:

- Learning rate = 0.0001
- Number of actions = 6
- input dimensions = 4
- batch size = 32
- epsilon = 1 (starts at 1 and decreases with decay)
- epsilon minimum = 0.1 in the deep mind paper it uses
- epsilon decay = 1e-5 (0.00001) in about 100thousand steps epsilon will hit minimum.
- Gamma (discount factor) = 0.99
- Memory size = 50000 (set lower for ram)
- Replace = 1000
- Number of games = 500

Learning rate is set at 0.0001 which worked best along with the losses and optimizer for this model, the learning rate was set lower to accommodate the changes to other parameters as well as training time. The lower learning rate is used to slowly make tiny updates to the weights, if the learning rate was set higher the loss function is affected more.

Number of actions is set to six because in the Atari Pong game, there are six possible actions.

Input dimensions is set to four because of the four frames.

Batch size parameter is the number of samples to work through before updating the model using the experienced replay. Batch size is typically 32 or 64.

Epsilon is set to start to one before being decremented overtime.

Epsilon minimum is set at 0.1 this is the minimum epsilon value it can decrease to.

Epsilon decay is the rate in which epsilon will decrease, it decreased to epsilon minimum in around 100,000 steps.

Gamma is set at 0.99 for the discount factor this value can vary between zero and one, the higher the discount factor the more important the future reward is. If gamma is closer to zero, the agent will tend to consider only immediate rewards.

Memory size is the amount of memory to allocate 50000 seems to be typical value for this if you run on 16gb of ram and this will be used to store all the sample values from the agent. This should be lowered if the computer has less ram, as the training takes up a lot of ram.

Replace is set at 1000 so at every 1000 steps the agent will load the next Q-values with the current state value (as shown in the Agent.py file). In the DeepMind paper (6) this value was set at ten times higher, but the model was trained for days. In this version the value was lowered to save time. The model is set to run for 500 games, and each replace interval will be at every 1000 steps which was efficient enough for the agent to learn.

Number of games the agent should play is set at 500 because it reasonably plots a clear graph showing the agent learning and the points levelling off at a much higher score towards the end of the run of games. This could be lowered to 300-400 games, but we felt 500 gave clearer results to plot. The training took around 6 hours on a computer running a GTX 1660 Nvdia graphics card, 16gb of ram.

Note to marker:

These above are the important parameters needed for all deep learning models, in the code there is further comments in every file explaining further the parameters used and what each variable is doing.

5.0 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.

Video recordings are available with the other submitted documents.

5.1 Terminal results

Below shows the results from the final last games after 6 hours of training. It can clearly show that the agents score improves over time.

```
episode:  352 score:  20.0  average score 14.4 best score 14.57 epsilon 0.10 steps 800144
episode:  353 score:  20.0  average score 14.5 best score 14.57 epsilon 0.10 steps 801954
episode:  354 score:  17.0  average score 14.6 best score 14.57 epsilon 0.10 steps 804001
episode:  355 score:  16.0  average score 14.6 best score 14.57 epsilon 0.10 steps 806083
episode:  356 score:  13.0  average score 14.5 best score 14.57 epsilon 0.10 steps 808322
episode:  357 score:  18.0  average score 14.5 best score 14.57 epsilon 0.10 steps 810202
episode:  358 score:  13.0  average score 14.5 best score 14.57 epsilon 0.10 steps 812449
episode:  359 score:  19.0  average score 14.5 best score 14.57 epsilon 0.10 steps 814191
episode:  360 score:  16.0  average score 14.5 best score 14.57 epsilon 0.10 steps 816324
episode:  361 score:  19.0  average score 14.6 best score 14.57 epsilon 0.10 steps 818128
... saving checkpoint ...
... saving checkpoint ...
episode:  362 score:  15.0  average score 14.6 best score 14.60 epsilon 0.10 steps 820310
... saving checkpoint ...
... saving checkpoint ...
episode:  363 score:  15.0  average score 14.6 best score 14.61 epsilon 0.10 steps 822474
episode:  364 score:  19.0  average score 14.6 best score 14.61 epsilon 0.10 steps 824338
... saving checkpoint ...
... saving checkpoint ...
episode:  365 score:  15.0  average score 14.7 best score 14.63 epsilon 0.10 steps 826563
```

```
episode:  482 score:  14.0  average score 15.5 best score 15.70 epsilon 0.10 steps 1077362
episode:  483 score:  17.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1079418
episode:  484 score:  15.0  average score 15.5 best score 15.70 epsilon 0.10 steps 1081617
episode:  485 score:  17.0  average score 15.5 best score 15.70 epsilon 0.10 steps 1083668
episode:  486 score:  19.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1085629
episode:  487 score:  11.0  average score 15.5 best score 15.70 epsilon 0.10 steps 1087895
episode:  488 score:  19.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1090062
episode:  489 score:  13.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1092301
episode:  490 score:  17.0  average score 15.5 best score 15.70 epsilon 0.10 steps 1094223
episode:  491 score:  17.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1096092
episode:  492 score:  19.0  average score 15.6 best score 15.70 epsilon 0.10 steps 1097899
episode:  493 score:  18.0  average score 15.7 best score 15.70 epsilon 0.10 steps 1099741
episode:  494 score:  15.0  average score 15.7 best score 15.70 epsilon 0.10 steps 1101769
episode:  495 score:  19.0  average score 15.7 best score 15.70 epsilon 0.10 steps 1103723
... saving checkpoint ...
... saving checkpoint ...
episode:  496 score:  15.0  average score 15.7 best score 15.72 epsilon 0.10 steps 1105713
episode:  497 score:  16.0  average score 15.7 best score 15.72 epsilon 0.10 steps 1107854
episode:  498 score:  21.0  average score 15.7 best score 15.72 epsilon 0.10 steps 1109601
episode:  499 score:  12.0  average score 15.6 best score 15.72 epsilon 0.10 steps 1111776
episode:  500 score:  17.0  average score 15.7 best score 15.72 epsilon 0.10 steps 1113854

(machinelearning) D:\Downloads - Volume\Deep-Q-Learning-Paper-To-Code-master\Deep-Q-Learning-Paper-To-Code-master\DQN>
```

Figure 13: First run Terminal outputs while training agent

```
Episode:  489 Score:  19.0  Average Score: 16.3 Best Score: 16.32 Epsilon Value: 0.10 Steps Taken: 1038723
Episode:  490 Score:  16.0  Average Score: 16.3 Best Score: 16.32 Epsilon Value: 0.10 Steps Taken: 1040837
Episode:  491 Score:  15.0  Average Score: 16.3 Best Score: 16.32 Epsilon Value: 0.10 Steps Taken: 1043178
Episode:  492 Score:  19.0  Average Score: 16.4 Best Score: 16.32 Epsilon Value: 0.10 Steps Taken: 1045236
Saving checkpoint ...
Saving checkpoint ...
Episode:  493 Score:  18.0  Average Score: 16.4 Best Score: 16.35 Epsilon Value: 0.10 Steps Taken: 1047041
Saving checkpoint ...
Saving checkpoint ...
Episode:  494 Score:  14.0  Average Score: 16.4 Best Score: 16.36 Epsilon Value: 0.10 Steps Taken: 1049182
Saving checkpoint ...
Saving checkpoint ...
Episode:  495 Score:  16.0  Average Score: 16.4 Best Score: 16.39 Epsilon Value: 0.10 Steps Taken: 1051280
Saving checkpoint ...
Saving checkpoint ...
Episode:  496 Score:  15.0  Average Score: 16.4 Best Score: 16.42 Epsilon Value: 0.10 Steps Taken: 1053383
Episode:  497 Score:  15.0  Average Score: 16.4 Best Score: 16.42 Epsilon Value: 0.10 Steps Taken: 1055854
Episode:  498 Score:  17.0  Average Score: 16.4 Best Score: 16.42 Epsilon Value: 0.10 Steps Taken: 1057864
Episode:  499 Score:  20.0  Average Score: 16.4 Best Score: 16.42 Epsilon Value: 0.10 Steps Taken: 1059678
Saving checkpoint ...
Saving checkpoint ...
Episode:  500 Score:  19.0  Average Score: 16.5 Best Score: 16.43 Epsilon Value: 0.10 Steps Taken: 1061439
Saving checkpoint ...
Saving checkpoint ...

(machinelearning) D:\Downloads - Volume\Deep-Q-Learning-Paper-To-Code-master\Deep-Q-Learning-Paper-To-Code-master\DQN>
```

Figure 14: Second run Terminal outputs while training agent (Higher score)

5.2 Plot Results

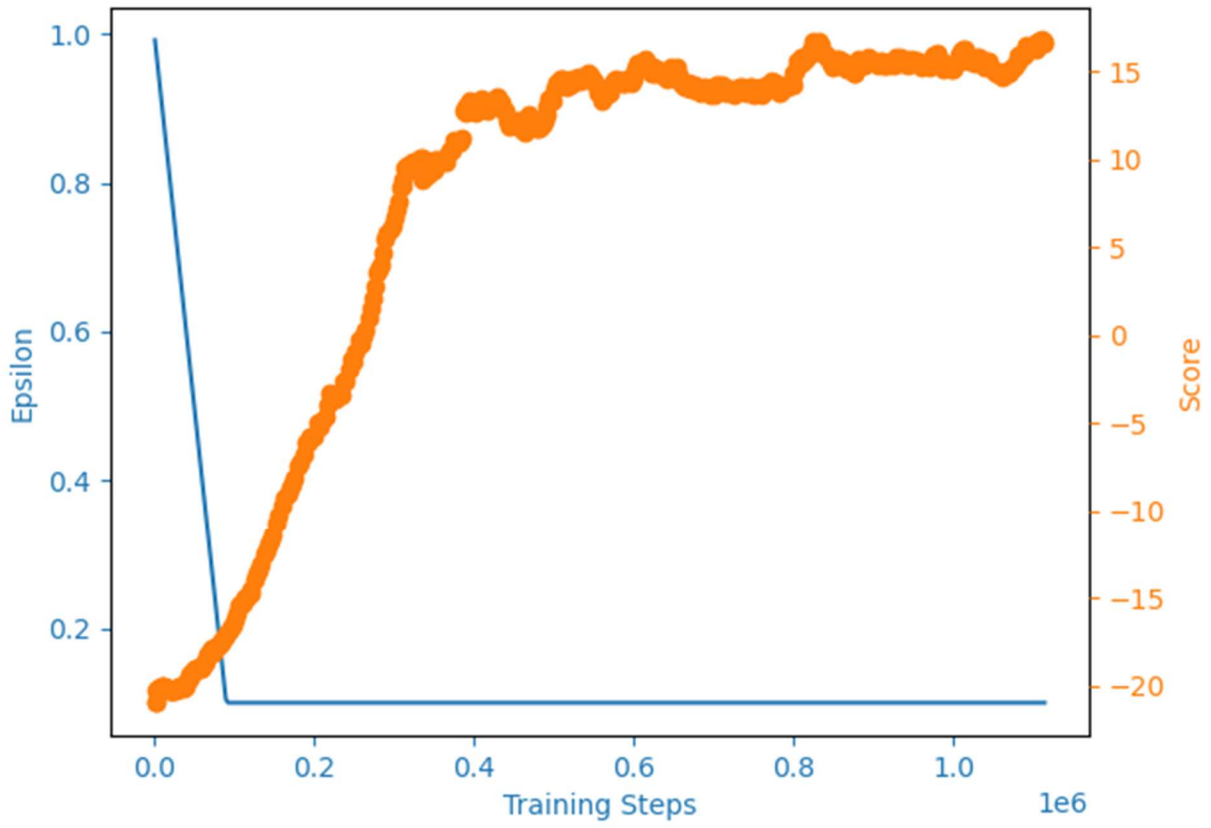

Figure 15: Plot after 6 hours of training, training steps as x axis against score and epsilon as y

Figure 15 plots the results of training, we ran 500 games and achieved an average score of ranging between 15 and 16 points. The best score is approximately 15.7 points after running for about 800 thousand learning steps.

Looking at the plot we can see that the agent starts to learn steadily as epsilon decreased over time. Most learning happens in the greedy phase. At approximately 500 thousand (0.5 x 1e6 on graph) training steps the agent scores begin to level off, with minor oscillations resulting in an average score of around 15.5 points. This graph is evidence that the agent is successfully learning using the deep Q-learning model with a few hours of training with average hardware specs running on a NVDIA GTX 1660 GPU.

The program was run for a second time, which gave slightly higher average scores due to the randomness from the agent's exploring. Overtime it learns at a very similar rate for both runs. Figure 16 below shows the same trend in learning as in Figure 15, apart from the score average score being slightly higher at approximately 16.7 when they begin to level off at around 500 thousand (0.5 x 1e6 on graph) training steps.

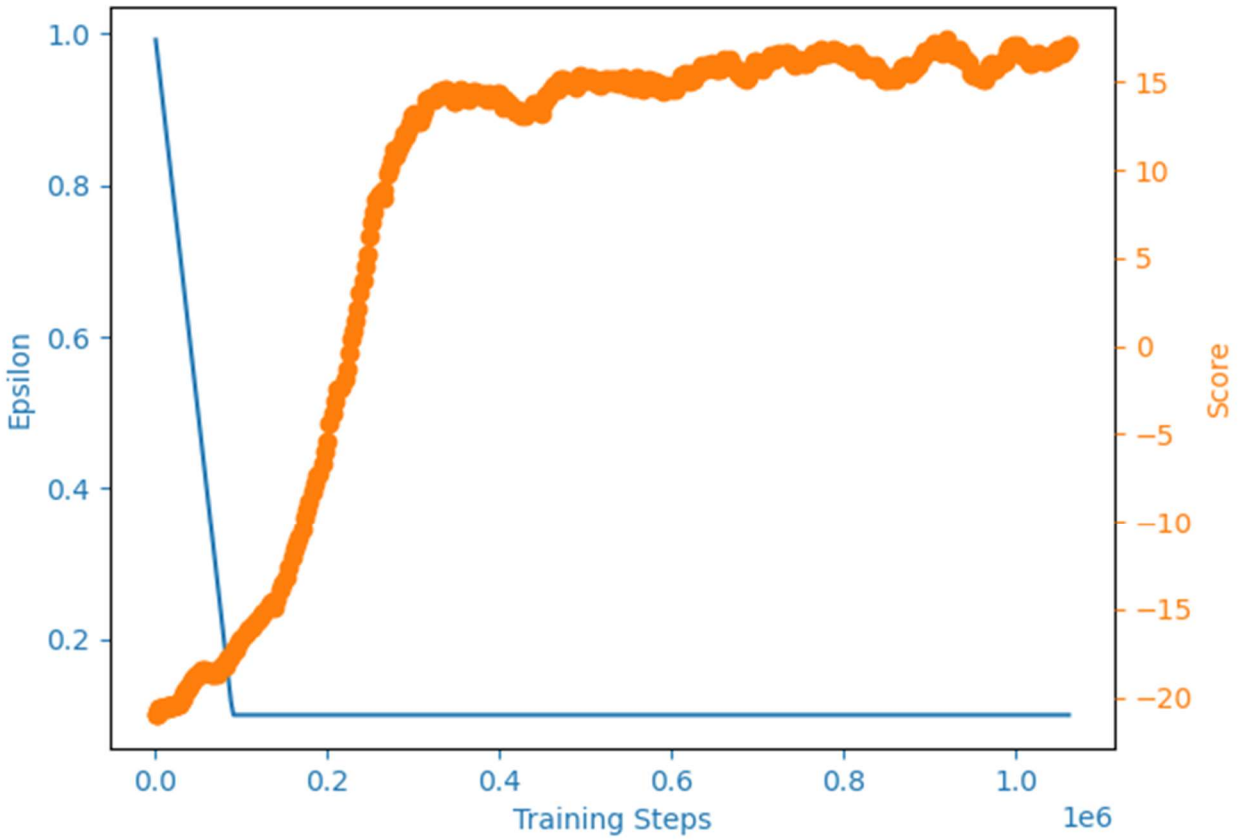

Figure 16: Second Plot after 6 hours of training, training steps as x axis against score and epsilon as y

5.3 Improvements with different models

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Figure 17: DeepMind paper results (6)

When analysing the results from the DeepMind paper (6), we can see that the average score for the pong results aims to be around 20 for Pong (Figure 17). This correlates to the average reward of the DQN we have made. The deep mind group trained the agent for a much longer period so it was expected that the average reward would be higher as the agent learned to play over a longer period.

From Figure 17, we can see that DQN has a much greater average total reward (20) when compared to Sarsa (–19) for the Pong game. This is not surprising because Sarsa is on-policy and does not use learning by experience replay to motivate its choice of action. However, it is expected that DeepSarsa, consisting of a deep neural network and utilising a Q network, would be a more effective model than Sarsa in the pong game. DeepSarsa is on-policy so does not use the max function, unlike the DQN. It would have been interesting to experiment and compare DeepSarsa with the DQN model we have implemented, had time allowed.

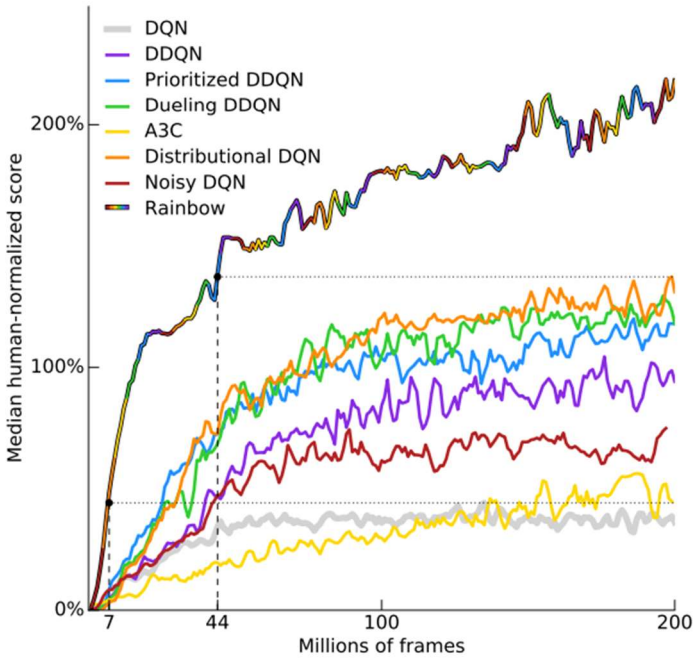

Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

Figure 18: arXiv, Deep Reinforcement Learning models (9)

The graph is taken from a conference paper 'Rainbow: Combining Improvements in Deep Reinforcement Learning'. The paper focused on whether deep reinforcement methods extending from the DQN algorithm are complimentary and can be combined. As we can see from the 'Rainbow' mapping on the graph in Figure 18, a combination of the methods results in a 'state-of-the-art performance' on Atari 2600 (9). Similarly, we can directly compare the performance of various deep reinforcement learning methods. The DQN shown in grey achieves a significantly lower median human-normalized score than most of the other deep reinforcement learning methods.

Double DQN (DDQN) shown in purple appears to yield better results than DQN because it consists of two neural network models that are identical and which function together to mediate the size of the Q-value and reduce overestimates. This means DDQN is more flexible to changes in conditions and it can learn to take different actions in response to different conditions. This contrasts DQN which instead just takes the same single max Q value action stored in a single DQN even if the action is not optimal in some specific conditions.

Dueling DDQN yields even better results than DDQN. Initially we attempted to create our model using Dueling DQN, as we had researched that the Dueling model would perform very well. In the Dueling model the learning is dependent on the advantage of performing an action. The Q-value is found through the value function and the advantage function (that expresses the advantage of performing the action) which are combined to estimate its Q-value in the final output.

Overall, while DQN produces very good results for deep reinforcement learning, there are other techniques, such as DDQN and Dueling DDQN, that perform better. The integrated ('Rainbow') agent,

consisting of an adapted DQN algorithm, performs significantly above the rest. In the future, it would be interesting to implement these two models with the Atari Pong game and compare their results with our existing DQN Pong model's results.

6.0 Conclusion

We successfully implemented a working deep learning model, using Atari games as an environment from OpenAi gym. We were able to comment on the parameters used in the program and what these mean in relation to the theory. This helped develop our understanding of deep reinforcement learning which we then were able to complete further research on and compare some other Q-learning models, from which we found that DQN is much better performing than DeepSarsa (6). Conversely it was one of the worst performing models when compared to the DQN complimentary combined algorithm models, with one such model DDQN easily outperforming DQN (9).

Overall, we found the research and implementation very rewarding. Ideally if time allowed, we would have created the models and agents from scratch as well as and implementing different models including Duelling deep Q-learning to make our own observations between these models.

References

1. CSDN. ***CSDN.*** **[Online] 01 12 2020. [Cited: 17 04 2021.] https://ask.csdn.net/questions/2719069.**

2. philtabor, "github," 19 09 2020. [Online]. Available: https://github.com/philtabor/Deep-Q-Learning-Paper-To-Code. [Accessed 14 04 2021]. 3. researchgate. ***researchgate.*** **[Online] 07 2019.**

3. [Cited: 18 04 2021.] https://www.researchgate.net/figure/Architecture-of-Deep-Q-networks-following-Mnih-et-al7-ReLU-nonlinear-units-are_fig2_335394578.

4. baeldung. ***baeldung.*** **[Online] 13 07 2020. [Cited: 18 04 2021.] https://www.baeldung.com/cs/ml-relu-dropout-layers#:~:text=As%20a%20consequence%2C%20the%20usage,adding%20extra%20ReLUs%20increases%20linearly..**

5. paperswithcode. ***paperswithcode.*** **[Online] [Cited: 18 04 2021.] https://paperswithcode.com/method/dqn#.**

6. Antonoglou, Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis. ***Playing Atari with Deep Reinforcement Learning, DeepMind Technologies.*** **2013.**

7. analyticsvidhya. ***analyticsvidhya.*** **[Online] 18 04 2019. [Cited: 18 04 2021.] https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.**

8. greentec. ***greentec.*** **[Online] 16 04 2019. [Cited: 19 04 2020.] https://greentec.github.io/reinforcement-learning-third-en/.**

9. arXiv. ArXiv.[Online] 06 10 2017. [Cited: 21 04 21] arXiv:1710.02298 [cs.AI]