

Machine Learning and Bioinspired Optimisation – Assignment II

Eamon Magdoubi, Przemysław Magiera, Ankita Malik, Seán O’Callaghan

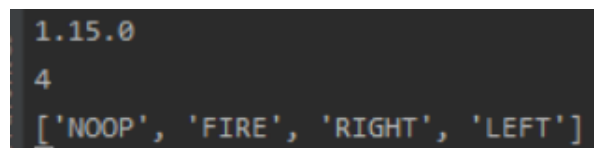
University of Liverpool — April 2020

This assignment works around using deep learning and knowledge of reinforcement learning to train an agent to get as high of a score in a game of choice as possible. An instance of the game is loaded through a Python library called Gym Atari for simulation of the game’s environments, scores and possible actions. Other libraries that were used include TensorFlow that allows data flow within a program and Keras for a Convolutional Neural Network system to be integrated.

1 Importing an Open-Ai Gym Retro game

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like pong or pinball. Gym is an open-source interface to reinforcement learning tasks. Gym provides an environment and it is up to the developer to implement any reinforcement learning algorithms. Developers can write agents using existing numerical computation libraries, such as TensorFlow or Theano. This is just an implementation of the classic “agent-environment loop”. Each timestep, the agent chooses an action, and the environment returns an observation and a reward.

The game that was decided by the group was breakout as when attempting for more complex games such as Pac-Man and PitFall the agent used too much GPU and CPU time to barely understand the concept of the game let alone achieve a reasonable score. There are four possible moves in breakout and the entire game revolves around the user (agent in our case) reacting to a pixel ball on the screen which bounces off walls. Each coloured barrier at the top signifies a point for the user in-game and will be perceived as the reward system within the game.



```
1.15.0
4
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

Figure 1: Indicating all possible movesets for environment

An Atari game was the console of choice as pixels were clearly evident and only detailed to a ratio of 210 by 160 pixels. This made convolution shorter and faster than picking a more detailed game such as Sonic for the Sega Genesis which has a ratio of 320x224 pixels.

Breakout was also chosen for the fact that when the University of Stanford tested over 40 games with a neural network in comparison to a traditional linear learner that Breakout was a game that showed a difference of 1327% increase of score.

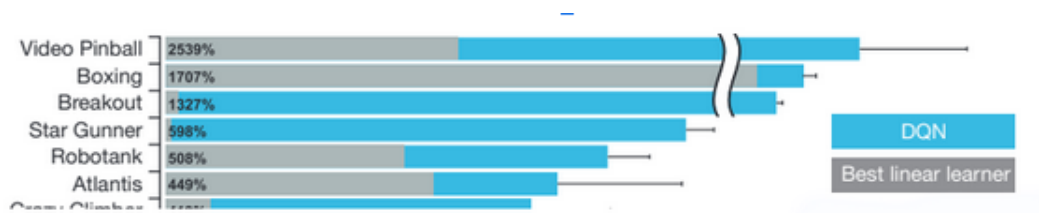


Figure 2: Display of scores for Atari games using DQN and linear learners(Mnih et all, 2015)

1.1 Version of the game

There are four base versions of breakout and the newest version was the one that we went for. As everyone's computers had a variety of computing power we did not decide on using the breakout version which stores the observations in the computer's RAM memory for a potential performance risk on some laptops. Breakout version 4 had a non-deterministic version which allowed for more effective and clear reward paths.

```
import numpy as np
import gym
- --
```

Figure 3: Importing the required environments for game simulation

Installing and importing the gym module allows us to create an environment for Breakout with the version being deterministic. To confirm this, we will print out the environment upon creation and the output is shown below. Figure 4.

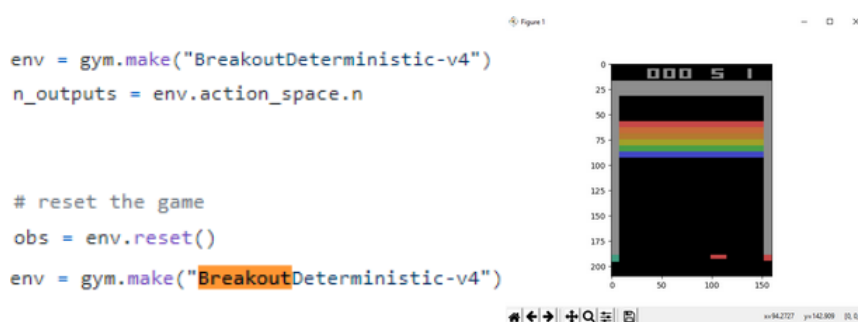


Figure 4: Creating environment and printing output

This concludes importing the Atari game environment and the next section will be with creating the network that the game states and variables will be passed in by.

2 Creating a network

The network was created using two different syntactic languages and slightly alternative approaches in TensorFlow1 and TensorFlow2, the architecture of the network has been implemented differently in both solutions, which will be explained later on.

2.1 Tensorflow 1 method

This neural network model was created and inspired from the following tutorial: **Automatic Pac-man with Deep Q-learning: An Implementation in Tensorflow by Adrian Yijie Xu**.

Firstly the first method takes into account a different sized input shape of 88 by 80 whilst having a batch size of 48. This input shape of one layer will be passed into the neural network. These are the hyper parameters for the training model and will be kept constant throughout testing. This is required because the use of 2D convolution is done on square inputs (Mnih, 2013). Figure 5.

```
batch_size = 48
learning_rate = 0.001

input_shape = (None, 88, 80, 1)
x_shape = (None, 88, 80, 1)
```

Figure 5: Setting sizes of batch and input shape for our first convolutional layer

The weights of each node within every convolutional layer has a scaled weight of two for the initialisation of the neural network. There is no activation layer within this model so there is no binary output distribution at each layer (Xu,2019). Using TensorFlow1 enforces a daisy-chaining effect on each convolutional layer being interpolated into the next. For every layer, a window of size for each convolution per pixel is estimated and gets smaller through each bypass. Strides are also set from larger variables to smaller distances between pixels as each convolution looks into a more detailed and specific section within the image. A major key with each layer is that after each convolution, the success effect is monitored via histogram values where peaks are measured for the activation to come into effect

```
def q_network(X, name_scope, env):
    n_outputs = env.action_space.n
    initializer = tf.compat.v1.keras.initializers.VarianceScaling(scale=2.0)

    with tf.compat.v1.variable_scope(name_scope) as scope:
        # three convolutional layers
        layer_1 = conv2d(X, num_outputs=32, kernel_size=(8, 8), stride=4, padding='SAME',
                        weights_initializer=initializer)
        layer_2 = conv2d(layer_1, num_outputs=64, kernel_size=(4, 4), stride=2, padding='SAME',
                        weights_initializer=initializer)
        layer_3 = conv2d(layer_2, num_outputs=64, kernel_size=(3, 3), stride=1, padding='SAME',
                        weights_initializer=initializer)
```

Figure 6: Architecture of our convolutional layers

After the three convolutions of the image are done the output for the last layer is flattened and connected to two fully connected layers. one having the number of nodes to be 128 and the last being the number of actions four. The number of actions in the final layer will be the last output and displayed in a histogram format to find the optimal action to be selected according

to the neural network. Figure 7.

```
# last output needs to be flattened in order to connect it to fully connected layers
flat = flatten(layer_3)

# define one fully connected layer and output layer
fc = fully_connected(flat, num_outputs=128, weights_initializer=initializer)
output = fully_connected(fc, num_outputs=n_outputs, activation_fn=None, weights_initializer=initializer)

# here we will store parameters of the network (weights)
vars = {v.name[len(scope.name):]: v for v in
        tf.compat.v1.get_collection(key=tf.compat.v1.GraphKeys.TRAINABLE_VARIABLES, scope=scope.name)}
return vars, output
```

Figure 7: Establishing the fully connected layers at the end

Adams algorithm is used for a stochastic approach in the deep network to find the gradient descent for backpropagation optimization. Adams algorithm is used for its straight forward nature, little memory requirements and seamlessness used within the TensorFlow1 architecture (Xu, 2019). Figure 8.

```
# for network to perform learning we have minimize loss function - we use AdamOptimizer
optimizer = tf.compat.v1.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)
```

Figure 8: Using the optimiser function for training purposes

The Adams algorithm method uses the adaptive gradient dense and not the root mean square propagation methodology.

2.1.1 Neural network diagram for TensorFlow1

The diagram here displays a scaled creation for the neural network with fully connected layers being represented by one-dimensional planes. Each layer from the permutation and convolution have been shown to use an activation via histogram so distinct peaks are recorded per value. Figure 9.

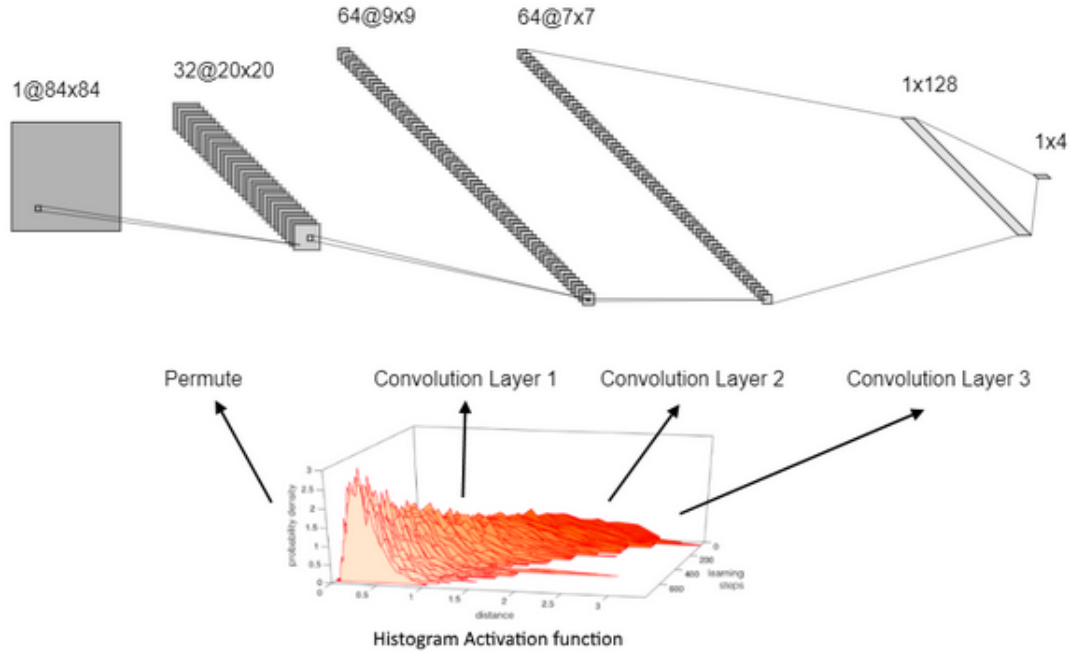


Figure 9: Network diagram for the TensorFlow1 methodology.

2.2 TensorFlow 2 method

This model was created with the base from the keras-rl2 pip install library. This library allows you to create complex base cases for deep learning environments and allows the user to further develop the scope to the user's needs. Once installed the environment playground can be integrated into a variety of algorithms such as Deep Q-Learning (Github, 2018).

The other approach tested by the group used TensorFlow2 and various modules from Keras. As an Atari game can be in the same state but from different actions, this can lead to millions of possible combinations requiring a neural network to be the backbone of the model instead of using purely a reinforcement learning model.

```
# tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten, Convolution2D, Permute
```

Figure 10: Importing required libraries for creating a networking model in TF2

To get an action function an image will first need to be dense or reshaped before being passed through multiple convolutional layers. This then determines the state and sections of the image looking for key arrangements from one of a variety of variables.

The first layer is a permute layer which enforces a given pattern and is useful to connect convolutional neural networks with recurrent neural nets. This accomplishes an output of the image which is 84 by 84 and has four layers for convolutional testing. The first convolutional layer is a two-dimensional shape that has a filter with an eight by eight ratio and a stride of four to cover as much of the image to achieve a generality of the entire image scope. This outputs a 20 by 20 by 32 2-D output that will transform and shrink the image again. This convolution

uses a filter of a quarter size and a stride of half of the previous convolution. Once these are activated the dense layer of 512 nodes (noticeably larger than the original TensorFlow1 example) are added after flattening the 64 output layers. See Figure 11.

```
class MinhModel:

    def __init__(self, shape=(84, 84), window_length=4, actions=4):

        self.input_shape = (window_length,) + shape

        self.model = Sequential()
        self.model.add(Permute((2, 3, 1), input_shape=self.input_shape))
        self.model.add(Convolution2D(32, (8, 8), strides=(4, 4)))
        self.model.add(Activation('relu'))
        self.model.add(Convolution2D(64, (4, 4), strides=(2, 2)))
        self.model.add(Activation('relu'))
        self.model.add(Convolution2D(64, (3, 3), strides=(1, 1)))
        self.model.add(Activation('relu'))
        self.model.add(Flatten())
        self.model.add(Dense(512))
        self.model.add(Activation('relu'))
        self.model.add(Dense(actions))
        self.model.add(Activation('linear'))

    def summary(self):
        return self.model.summary()
```

Figure 11: Creating network-based of research material found with Deep Q-Learning and Convolutional learning

The activation function used was ReLU, a typical activation function if a value is greater than one. Each layer used this as its activation to the next 2D convolutional layer with each stride and filter block becoming smaller and smaller. Flatten gets all image batches from the convolution and reduces them to a single layer which is passed to the densely connected network before the linear activation function which is the output. Once the last dense layer is created the analysis from this activation will indicate which action achieved the highest value with the ReLU activation function used for this neural network model. The total parameters are 1,686,180 which shows the lack of scale a reinforcement learning model could do without using neural networking.

```
dqn.compile(Adam(lr=.00025), metrics=['mae'])
```

Figure 12: With the use of the Adam algorithm for learning rate optimization at 0.0025, an effective method of calculating a stochastic gradient descent is used to allow for effective mode weight changes per backpropagation (Gu et al, 2016).

2.2.1 Neural Network Diagram for TensorFlow2 method

The diagram created was done to illustrate the complexity of the second model in comparison to the first. The permute initialiser first generates the input image into four layers to identify patterns in the construction of the image. The convolutional layers all use a simple ReLU activation function that clearly indicates a positive activation after reaching a value. Once the three convolution layers of decreased scale and filtering are done they are passed into a large 512 layered fully connected layer. This will identify 512 key patterns found from training and distribute

each activation of one to a primary action. The highest combination set that is activated from the flattening of the final layer is the selected action. See Figure 13

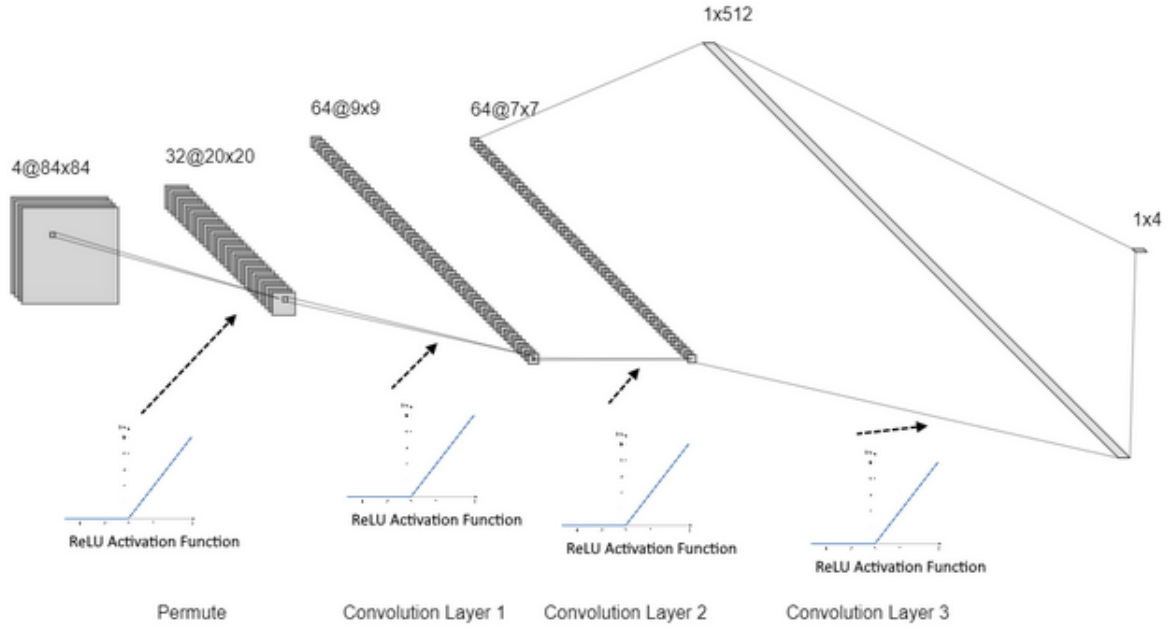


Figure 13: Diagram depiction of the neural network and activation function for the Tensorflow2 model

2.3 Comparison of Models

A major difference between the two models is the input layer from the two networking models. TensorFlow2 allows for a permute layer to interpolate and identify key patterns with each input image having four different variations of the image. The TensorFlow1 version only uses a single layer for integration in comparison so patterns are not as easily located in comparison. The convolution layers are similar but each layer differs with their activation function.

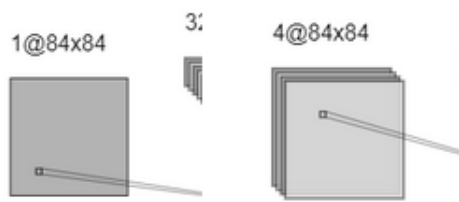


Figure 14: Comparing input layers for the TensorFlow1 version on the left and TensorFlow2 version on the right

Even though both methods utilise Adams optimizer for the weight and linear transformation the activation around them is different. TensorFlow1 emits around an entire summary of the layer which reduces its speed whilst even working on a GPU machine. Relu instead of a summary produces a shape based on the input and finds the activation value accordingly.



Figure 15: The scale of fully connected layers using diagram simulation for TensorFlow2



Figure 16: The scale of fully connected layers using diagram simulation for TensorFlow1

Another major difference between the two implementations is the fully connected layers. The two scales can be seen above with the top being the more complicated TensorFlow2 dense implementation. These two models have different methods of activation that revolve around different number values but differ in 384 nodes of pattern recognition points from the flattened third layer. This will be predicted that the TensorFlow2 version will have a more optimal simulation due to its complex input creation and connected networking layers. Not only this but the simpler activation function does not need a full summary of the entire neural net at every stage but a simple activation point. See Figure 17

	Fully Connected Nodes	Convolutional Filters	Input Layers
TensorFlow 2	516	160	1
TensorFlow 1	130	160	4

Figure 17: Complexity of neural network layers

3 Connection of the game to the network

As the agent can only observe the current situation with the screen's physical pixels it is not possible to depict what current scenario the agent is in. This is why the sequence of observations and actions $st = at, rt, xt$ are stored within the current state-action pair to produce a Markov Decision Process (MDP) (Mnih, Kavukcuoglu, Silver, Graves and Antonoglou, 2013). Since the environment is stochastic and could have millions of states the actions and rewards of the game will need to be connected to the neural network which will adapt and modify its values depending on the values of the action.

3.1 Convolutional Network

To link the input states from the game into the neural network a concept of convolution was used to determine the patterns on the screen. As Atari games are not highly detailed in pixel

density the entire game environment can be passed in without any large loss of data. The Q value in this environment is the value state action that the agent will decide to move the paddle with one of four move sets, see Figure 18.

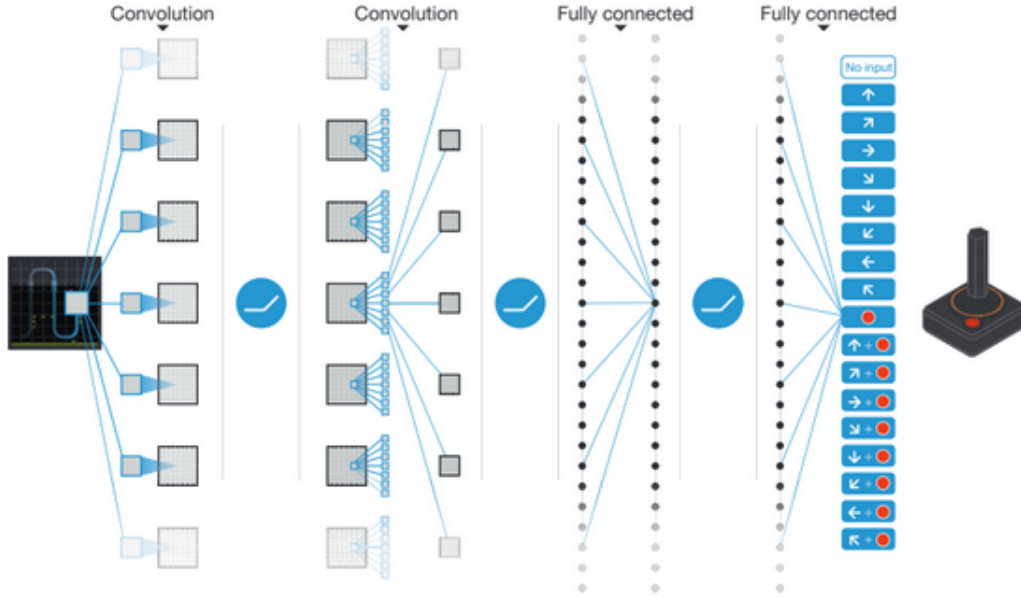


Figure 18: Neural Network for an Atari Game state through convolution (Mnih et al, 2015)

Each filter on the game is used to indicate a certain filter so the neural networking can adapt and change parameter values according to the best action with the state estimated to have the highest reward.

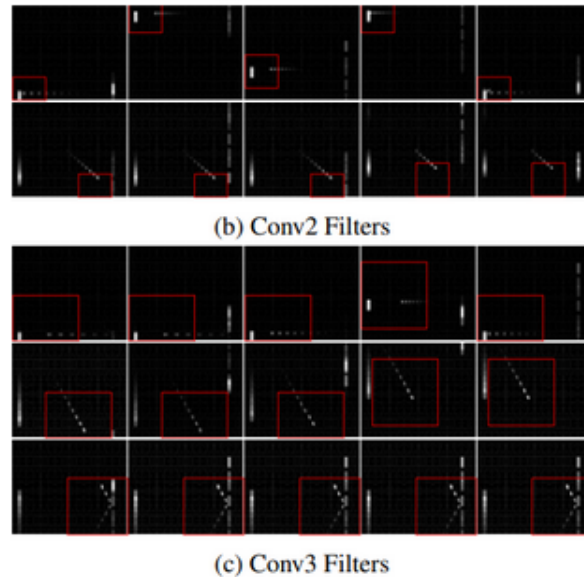


Figure 19: Behaviour if each convolutional layer against the game pong(Hausknecht et al, 2015)

Once each hidden layer is filtered it is then followed by a rectifier nonlinearity to identify the value activation it will lead to (Minh et al, 2015). Filters are then found to have an effect on the values that each state has as in the Conv3 filter above similar patterns are found with the

ball bouncing off the paddle and action can be done based on this displacement. See Figure 19.

3.2 Tensorflow 1 Connection with Atari game and Neural Network

The primary step for connecting the game screen to the neural network is to refactor and resize the game screen to be in the same proportions as the expected input. The image is grey scaled at this stage to reduce confusion and allow for objects to be clearly distinguishable after further iterations. When an image is grey scaled the level of difference between frames are reduced heavily so frame skipping up to four is possible with minimal loss (Hester et al, 2018). See Figure 20.

```
# ----- ADJUST SCREEN OF THE NN INPUT -----#
def preprocess_observation(obs):
    color = np.array([210, 164, 74]).mean()
    img = obs[1:176:2, ::2]
    img = img.mean(axis=2)
    img[img == color] = 0
    img = (img - 128) / 128 - 1
    return img.reshape(88, 80, 1)
```

Figure 20: Processing the image to the correct factor in TF1

This refactored image shape will go into the Q-network for initialization of values. Two Q-learning environments are used in conjunction with one another to generate data during the training process. Having a dual network allows for the Deep-Q learning environment to be portrayed almost as a double-Q environment where a policy is being utilised whilst the other is being optimised. Linking Q Network with the image pixel size can be seen in the Figure 21.

```
# placeholder for input
X = tf.compat.v1.placeholder(tf.float32, shape=X_shape)

# 2 networks to train and generate data (actions)
mainQ, mainQ_outputs = q_network(X, 'mainQ', env)
targetQ, targetQ_outputs = q_network(X, 'targetQ', env)
```

Figure 21: Linking Q Network with the image pixel size

The final step of linking the network is after each pass through to link an action to the output evaluation of the network. This can be done since the output of the network is one of the four actions linked to by the atari game. This action can take a greedy approach with probability $1-\epsilon$ or a random approach being ϵ . Once this step is taken it is saved to memory and appended before analysis. Integrating an action to a convolutional network can be seen in the Figure 22.

```

==== SETUP ===#

# environment
env = gym.make('BreakoutDeterministic-v4')
env.seed(123)
actions = env.action_space

# model
model = MinhModel()
model.summary()

# memory
memory = SequentialMemory(limit=1000000, window_length=4)

# atari processor
processor = AtariProcessor()

```

Figure 22: Integrating an action to a convolutional network in TF 1

3.3 TensorFlow 2 Keras

To connect the Atari gym environment to the network the entire model and processors are established with memory slot allocation for the game. The environment is linked to a variable and the actions are listed for setting up the neural network down the line. To alleviate any non-stationary distribution of actions an experience replay mechanism is established to use previous behaviors into effect to affect the choice of the next action (Mnih, 2013).

```

==== SETUP ===#

# environment
env = gym.make('BreakoutDeterministic-v4')
env.seed(123)
actions = env.action_space

# model
model = MinhModel()
model.summary()

# memory
memory = SequentialMemory(limit=1000000, window_length=4)

# atari processor
processor = AtariProcessor()

```

Figure 23: Creating the environment and processor classes

The processor function assists with interpolating the image into the correct size and adding gray-scale. As the model revolves around an Atari game, specific functions are used for this process to establish the batch size and saving image states to the memory of a given ratio. Furthermore, the agent's initialisation is created in this setup section. The action list (of four possible actions), the memory capacity and networking model are all passed into this class. Initialisation in the code can be seen in the Figure 24. The memory module was used from an example Keras repository. The memory consists of a ring buffer implementation which has a much faster random access. A deque collection was initially used, but was much too slow when training the model from the experience replay.

```

# policy
# Here we select an epsilon-greedy policy, wrapped in a linear-annealed policy. This means the value for epsilon will start high and decay
# over time. For the agent, this translates into high exploration at the beginning of training. As the training progresses, the agent's
# exploration will decrease and it's actions will be those selected by the q-network.
policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, value_test=.05, nb_steps=1000000)

# agent
#
dqn = Agent(model=model.model, actions_n=actions.n, policy=policy, memory=memory,
            processor=processor, warmup_steps=50000, gamma=.99, target_model_update=10000,
            train_interval=4, delta_clip=1.)

# learning rate
#
dqn.compile(Adam(lr=.00025), metrics=['mae'])

```

Figure 24: Initialising the agent model

When creating the network we focused on resources online to find the best combination of convolutional layers, filters and strides to achieve a successful result. As the environment is the 210 by 160 ratio it can be condensed in the gym atari environment to be 84 by 84 pixels large. This reduction in size decreased the amount of computational overhead when requiring to analyse the thousands of input screens that are passed in. This input layer is guided into the first hidden layer which primarily looks at the paddle in Pong and breakout (Hausknecht et al, 2015). The second layer begins to detect the ball's path and the third layer has a filter to detect the velocity and direction of travel. These three layers are combined to keep track of the location of the ball and having the states saved in memory through LSTM gives a representation of recent frames. See Figure 25

```

def process_state(self, state, shape=(84, 84), convert='L'):
    img = Image.fromarray(state)
    img = img.resize(shape).convert(convert) # resize and convert to grayscale
    state = np.array(img)
    return state.astype('uint8') # saves storage memory

```

Figure 25: Refactoring the image size and hue

A checkpoint and interval count are then established to periodically save the episode state every 100 intervals and save the weights for each node in the network at 250,000 episodes. This is then all put into the fit function which interns use as the reinforcement model. See Figure 26

```

checkpoint_weights_filename = 'weights_{step}.h5f'
log_filename = 'dqn_log.json'
callbacks = [ModelIntervalCheckpoint(checkpoint_weights_filename, interval=250000)]
callbacks += [FileLogger(log_filename, interval=100)]

dqn.fit(env, callbacks=callbacks, steps=1750000, log_interval=10000)

```

Figure 26: Using all the values to be fit into the Deep Q- Learning Model

Once inside the function, the model will need to first establish values such as the episodic reward as the rewards given will be linked to the current value state of the agent. The model still uses variables linked with reinforcement learning as to make the game as step-based as possible

(Figure 27). The environment works initially in an off-policy fashion with exploration across multiple actions and possible combinations. The purpose of off-policy algorithms, in this case, is that the approximation can in principle achieve better data efficiency (Gu et al, 2016).

```
# training vars
self.training = True # enable training
self.step = np.int16(0) # current step
episode = np.int16(0) # current episode
observation = None # current observation
episode_step = None # current episode step
episode_reward = None # current episode reward
abort = False # keyboard interrupt flag
```

Figure 27: Initialise training model

A loop will then form for training the algorithm using the actions, rewards and values achieved from the environment. The rewards are taken from the total point score from each page as to progress with the game and finding adequate actions to take in certain situations. Selecting in a fashion that maximises the cumulative future rewards progresses the game and results allows the agent to learn (Mnih, 2015). See the loop in Figure 28

```
# while current step is less that total number of training steps
while self.step < steps:
    # if new episode
    if observation is None:
        callbacks.on_episode_begin(episode)
        episode_step = np.int16(0) # reset episode current step
        episode_reward = np.float32(0) # reset episode reward

        self.reset_states() # reset recent observation & action
        observation = env.reset() # obtain initial observation (of reset environment)
        observation = self.processor.process_state(observation)

    ### RUN A SINGLE STEP ###
    callbacks.on_step_begin(episode_step)

    ### FORWARD STEP ###
    action = self.forward(observation) # choose an action
    reward = np.float32(0)
```

Figure 28: Begin looping through steps & episodes

Action selection using policy testing from the neural network is still used to establish real consequences for an agent’s actions for deep learning (Hester et al, 2018). The forward function uses the current neurological weights to select an action against the environment for the agent to select. Once selected the agent will perform the move and move onto the next frame of the game. After the next frame of the game is shown the agent will proceed to do this step over again until the game has no more states (Figure 29).

```

def forward(self, observation):
    """
    FORWARD STEP
    Given an observation, calculate the q-values and choose an action
    """
    state = self.memory.get_recent_state(observation) # get the most recent recorded state
    q_values = self.compute_q_values(state) # compute the q-values of this state

    action = self.policy.select_action(q_values=q_values) # select an action using the (linearly annealed) epsilon greedy

    # update recent observation & action
    self.recent_observation = observation
    self.recent_action = action

    return action

```

Figure 29: Forward step

4 Deep reinforcement learning model

A base for the neural networking that is being used is Temporal Difference Learning. This form of reinforcement learning has a goal to develop and overtime learn a policy to control an agent with a variety of states. Changing states is done through actions $u \in U$ to attempt to maximise the sum of the total of the rewards. Model-Free learning is what temporal difference is the foundations from whilst attempting to reduce the policy gradient and increase the reward count (Gu et al, 2016).

The agent will only be given the chosen action, current reward and environment so each state is a Markov Decision Process as each state is a predecessor of all the actions before it $a_x = a_1 + a_2 + a_3 + \dots + a_{x-1}$. Therefore we can use our model as a Bellman's equation from the intuition that if the optimal value $Q^*(s', a')$ as the sequence at the next time step is presented for all possible known actions which are finite. Maximising the optimal strategy in this manner will give us Q learning.

```

Initialize  $Q(s, a)$  arbitrarily
Repeat (for each episode):
    Initialize  $s$ 
    Repeat (for each step of episode):
        Choose  $a$  from  $s$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
        Take action  $a$ , observe  $r, s'$ 
         $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
         $s \leftarrow s'$ 
    until  $s$  is terminal

```

Figure 30: Base model for Q learning environment (Choudhary, 2019)

A problem with the pure usage of Q-Learning is that the assumption of every state can be attempted to eventually find an optimal solution through max selection. Games that are stochastic have a problem as every state is not captured due to frame skipping and there are far too many unique states in any Atari game for an approximate Q learning model to be used. For each state in Brick Breaker, there are multiple methods to achieve the same score or board

presence but they all do not equate to the same value and reward. An action-value function selection is unique to each sequence and will have to be generalised or estimated to be possible (Mnih et al, 2015).

Q learning would take an unreasonable amount of time to learn the projection of each ball so a more adaptive version will be used. Many concepts of Q-learning are used in Deep Q-Learning but the algorithm will adapt to estimate output nodes after a given forward pass of state inputs (Hester et al, 2018).

4.1 Deep Q-Learning

4.1.1 Reinforcement learning dictionary

Agent: an agent is the one who acts in an environment, for instance, navigating a computer game. The agent makes decisions based on the rewards obtained from the previous action. It can be a human, robot or software.

Action: An action is a set of different moves which an agent takes after receiving a reward while interacting with the environment at discrete time steps. An agent has a set of actions from which it chooses an action which then afterwards is sent to the environment.

Discount factor(γ): a discounted factor is denoted as γ , it is a factor multiplying the supplementary reward, and varies on the range of $[0, 1]$. Lower the discount factor is, less important the supplementary rewards are.

Environment: The thing from which an agent can interact with directly or indirectly. It changes when an agent performs a particular action this change is considered as state-transition. The environment returns two types of information to agent i.e., Reward(R) and State (S)

State (S): the state describes the current situation. The agent goes through various states, states can vary from initial state to terminal state.

Rewards (R): the reward is received immediately after an action is executed. For example, during a computer game, when a player kills the enemy, the player gains points. From any given state, an agent sends output within the sort of actions to the environment, and therefore the environment returns the agent's new state also like rewards.

Policy (π): The policy is denoted as π and defined as a "map" from states to actions, An optimal policy can be derived from a Q function. It is the reinforcement learning objective to find the optimal policy. The policy is the strategy that the agent employs to determine the next action based on the current state.

Episode: all the phases which come in between initial state and terminal state are episodes. For example, in CSGO (Counter Strike Global Offensive) the main goal of a player is to gain maximum points so that the player wins the game so, all those states from which the player goes are the episodes which are used to maximize the reward.

4.1.2 Our model

With real-world environments that are not controlled requires us to enforce a Deep Q-Learning model. Using Q learning would be the basis but the action will be decided through a neural network that will change its estimates for states given the outcome. Learning and

changing the weights based on the reward is how the neural network will learn as the target variable is constantly changing (Choudhary, 2019).

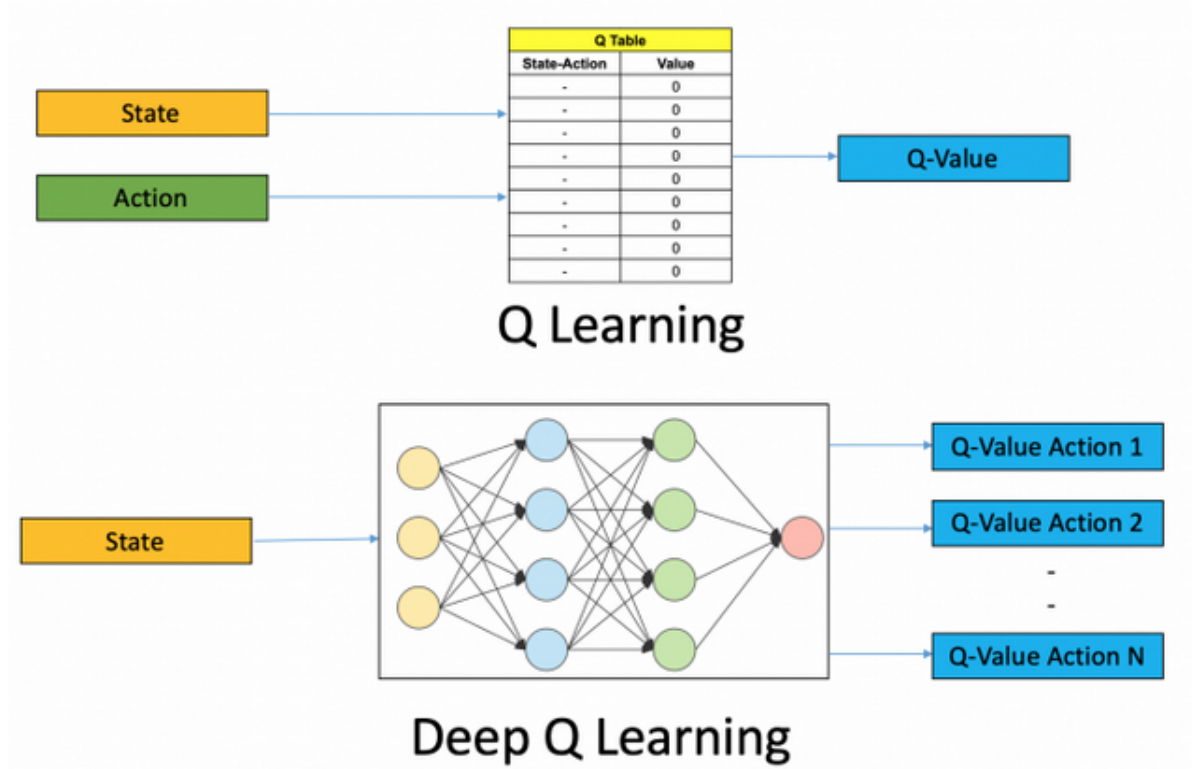


Figure 31: How actions are taken with q learning in comparison to deep q learning(Choudhary, 2019)

When the agent attempts an episode it uses the previous rewards and activation function for a pattern of states found from the three hidden convolutional layers. Action will be done as the output and the reward is monitored from this action set. Once done the Adam optimiser will modify each ReLU activation function depending on the said reward. The goal is then changed from finding the $Q(S, A)$ state action to achieve the highest value to lowering the degree of error per neural connection. These will be perceived as weights following a gradient descent but done after each time step in the Q-Learning environment (Mnih et al, 2015).

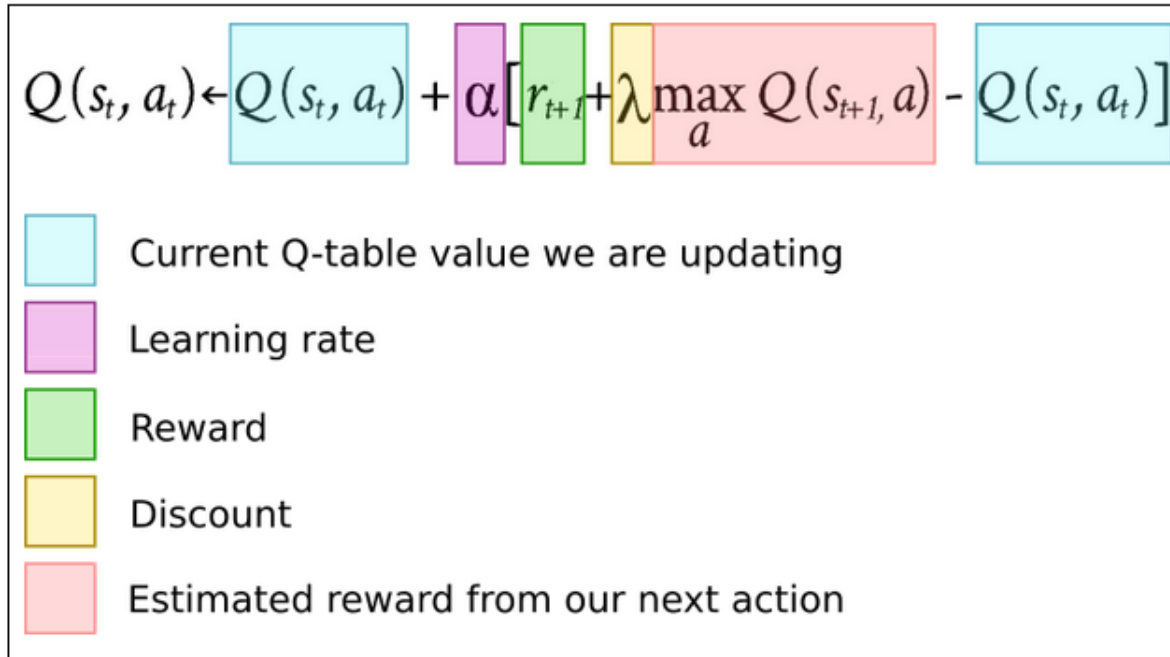
$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

$$\mathcal{L}(s, a|\theta_i) \approx (r + \gamma \max_a Q(s', a|\theta_i) - Q(s, a|\theta_i))^2$$

Figure 32: Change of goals from Q-Learning to Deep Q-Learning(Hester et al, 2018)

Q-Learning is a basic form of Reinforcement Learning which is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state uses Q-values (also called action values) to iteratively improve the behaviour of the learning agent. when we perform Q-learning we make create a Q-table which includes state and action $Q(s, a)$.

In Q-learning we also use a learning rate which is referred as alpha α which is set between 0 and 1, setting it to 0 means that the Q-values are never updated, hence nothing is learned.



$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Current Q-table value we are updating
 Learning rate
 Reward
 Discount
 Estimated reward from our next action

Figure 33: Q-learning algorithm(Kilii, 2019)

Values within the model are gained when the agent is in a state that is beneficial. For example, the value in state one and two have similar values even though the second state has more of a reward. This is due to the fact that the agent hit a section that would not benefit in the wall being broken and thus raining the most score. This is why in the third state that the value climbs up astronomically and goes even higher when a gap is formed in the wall. The process can be seen in the Figure 34

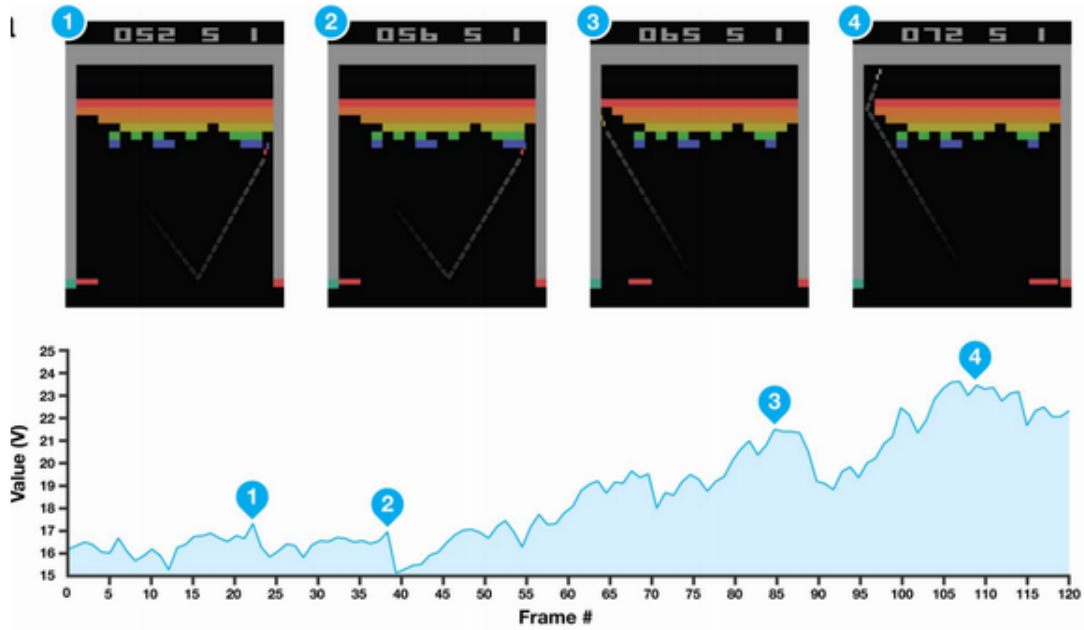


Figure 34: How the value function output is changed between frames even though they are in similar states (Mnih et al, 2015)

4.2 Training algorithm for Deep Q-network

When the agent is going through each episode it selects an action execution based on the ϵ -greedy policy. During a Q-learning update steps the maximisation of each function is not what is selected because the agent could get stuck at a local minimum but by using experience and reply the behaviour is averaged out over many previous states.

lqn_BreakoutDeterministic-v4_weights_14000.h5f.data-00000-of-00001	05/04/2020 18:39	DATA-00000-OF-0...	6,588 KB
lqn_BreakoutDeterministic-v4_weights_14000.h5f.index	05/04/2020 18:39	INDEX File	1 KB

Figure 35: Saved state for training weights at a given state

A wide variety of variables are found thanks to the random nature of a ϵ -greedy action selection which allows for exploration so average reward values are across a multitude of probabilistic differences.

4.3 How to Train a Deep Q-learning Network?

While training the deep Q-learning network we don't need current Q table values and learning rate, we can simplify it. We won't be needing the learning rate, as our back-propagating optimizer will already have that. Once the learning rate is removed, we can also remove the two $Q(s, a)$ terms, as they cancel each other - Figure 36.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha [r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Current Q table value we are updating
 Learning rate
 Reward
 Discount
 Estimated reward from our next action

Figure 36: Components need for training the deep Q learning Network (Juhu kiili,2019)

Training the model with a single experience:

1. Let the model estimate Q values of the old state
2. Let the model estimate Q values of the new state
3. Calculate the new target Q value for the action, using the known reward
4. Train the model with input = (old state), output = (target Q values)

4.4 Training Models - code description

4.4.1 TensorFlow 1 implementation

The agent starts with an exploration rate of 5% but will slowly decay with time when the results become more struct worth (Xu, 2019). All starting parameters are set as seen in the Figure 37.

```
# ----- PARAMETERS -----#
epsilon = 0.5
eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 500000
```

Figure 37: Parameters for the algorithm

Next thing is to implement what was mentioned in the previous paragraph - ϵ -greedy decay policy.

```
# ----- DECAP EPSILON-GREEDY POLICY -----#
def epsilon_greedy(action, step, env):
    n_outputs = env.action_space.n
    epsilon = max(eps_min, eps_max - (eps_max - eps_min) * step / eps_decay_steps)
    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs)
    else:
        return action
```

Figure 38: Epsilon-greedy decay implementation

In each step we select an action and get reward for it in the classic Q-Learning manner.

```
# choose best action
action = np.argmax(actions, axis=-1)
actions_counter[str(action)] += 1

# choose action with eps-greedy policy
action = epsilon_greedy(action, global_step, env)

# take a step
next_obs, reward, done, _ = env.step(action)

# append experience buffer
exp_buffer.append([obs, action, preprocess_observation(next_obs), reward, done])
```

Figure 39: Collecting the reward after eps-greedy action choice

In the main loop, during learning process, the agent processes its memories (the experience replay buffer concept described in the previous paragraphs), then process current observation, which are presented by Neural Network outputs in the form of available actions. Then we just calculate episodic loss and append it to memory.

```

# we start training network after "start_steps" number of steps
if global_step % steps_train == 0 and global_step > start_steps:
    # get last memories
    o_obs, o_act, o_next_obs, o_rew, o_done = sample_memories()

    # take states and next states
    o_obs = [x for x in o_obs]
    o_next_obs = [x for x in o_next_obs]

    # get next action
    next_act = mainQ_outputs.eval(feed_dict={X: o_next_obs, in_training_mode: False})
    #
    y_batch = o_rew + discount_factor * np.max(next_act, axis=-1) * (1 - o_done)
    # calculate train loss
    train_loss, _ = sess.run([loss, training_op], feed_dict={X: o_obs, y: np.expand_dims(y_batch, axis=-1), X_action: o_act, in_training_mode: True})

# after every "copy_steps" we copy parameters to target network
if (global_step + 1) % copy_steps == 0 and global_step > start_steps:
    copy_target_to_main.run()

```

Figure 40: Experience replay buffer implementation

4.4.2 TensorFlow 2 implementation

Similarly, we start with select action method which is analogical to the previous implementation, we still use the same epsilon greedy policy.

```

def select_action(self, q_values):
    """Return the selected action

    # Arguments
        q_values (np.ndarray): List of the estimations of Q for each action

    # Returns
        Selection action
    """
    assert q_values.ndim == 1
    nb_actions = q_values.shape[0]

    if np.random.uniform() < self.eps:
        action = np.random.randint(0, nb_actions)
    else:
        action = np.argmax(q_values)
    return action

```

Figure 41: Select action method implementation

In our second implementation the main loop breaks each step into two main sections; a forward step and backward step. We run a single step forward choosing a single action sticking to the ϵ -greedy policy. The backward step saves the experience of the current step into memory, and samples previous experiences to train the network. Finally, we collect the reward for our metric callbacks.

```

# while current step is less that total number of training steps
while self.step < steps:
    # if new episode
    if observation is None:
        callbacks.on_episode_begin(episode)
        episode_step = np.int16(0) # reset episode current step
        episode_reward = np.float32(0) # reset episode reward

        self.reset_states() # reset recent observation & action
        observation = env.reset() # obtain initial observation (of reset environment)
        observation = self.processor.process_state(observation)

    ### RUN A SINGLE STEP ###
    callbacks.on_step_begin(episode_step)

    ### FORWARD STEP ###
    action = self.forward(observation) # choose an action
    reward = np.float32(0)

    callbacks.on_action_begin(action)

    observation, reward, done, info = env.step(action) # perform action and make observation, collect reward
    observation, reward, done, info = self.processor.process_step(observation, reward, done, info) # process ob

    callbacks.on_action_end(action)

    ### BACKWARD STEP ###
    # in training, train the agent
    metrics = self.backward(reward, terminal=done)
    episode_reward += reward

```

Figure 42: Main loop for the TensorFlow 2 method

With the forward pass of the neural network, the function uses a linear annealing function to determine which action it will select. This is done using an epsilon greedy policy of 1%. This computes a threshold for the current state value which is transferred to a policy. This means that, initially, a high epsilon value favours exploration over exploitation. The agent is therefore much more exploratory over the initial episodes. As the value for epsilon decays, the agent starts to exploit its knowledge of the environment. This ensures the more comfortable the algorithm gets the less it demands on searching for sub-routes and is an effect that the first TensorFlow1 model does not take into consideration which results in scores being low after thousands of episodes. As epsilon decreases, the average reward per interval increases. This is shown in the results section.

```

# Compute Q values from the experience batch
# predict q-values using 'next_state' batch, using the target-network
# the target-network's weights are updated less often compared to the 'live' network
# this is governed by the target_model_update interval
# the target-network is a more stable version of the live network
target_q_values = self.target_model.predict_on_batch(state1_batch)
q_batch = np.max(target_q_values, axis=1).flatten() # select the highest q-values from each experience

# Compute r_t + gamma * max_a Q(s_t+1, a)
discounted_reward_batch = self.gamma * q_batch # calculate the discounted q-values for each experience
discounted_reward_batch *= terminal1_batch # for the terminal states, set the discounted reward to 0
Rs = reward_batch + discounted_reward_batch # for each experience, calculate the total reward (current + discounted)

```

Figure 43: Experience replay batch implementation

During the backward step, the current transition/experience (state, action, reward, next state) is saved to the experience replay memory. During the warm-up phase, no training occurs, meaning experience is built-up from exploratory actions. Once the warm-up phase ends however, the network begins to sample random batches of experiences from the memory, during the set training intervals (we don't train every single step). These experience batches are broken

down into the experience parameters. For each experience, the q-values are calculated using the Bellman equation. Using the q-values, the target network is trained and its weight's are updated. Note: during testing, the network is not trained; no experiences are saved nor learned from.

4.5 Training Values

We chose following training values, which were the result of our own research and the values that were used in the referenced articles.

Tensorflow 1

Training Intervals	Gamma	Learning Rate	Epsilon Greedy Value
60000	0.97	0.001	0.5

Tensorflow 2

Training Intervals	Gamma	Learning Rate	Epsilon Greedy Value
100000	0.99	0.00025	0.01

Figure 44: Training parameters for both implementations

5 Experimental results

Although using the first method with TensorFlow 1 did not achieved satisfactory results, our TensorFlow 2 agent did well. We have tested the algorithms with the parameters that can be seen in the Figure 44.

The group configured our computer systems to allow for simulation through TensorFlow GPU. Nvidia Nsight and Nvidia NN were downloaded which allowed for simulations to be faster as visualisations can be rendered and calculated faster. Screen of a training session is available in the Figure 45.

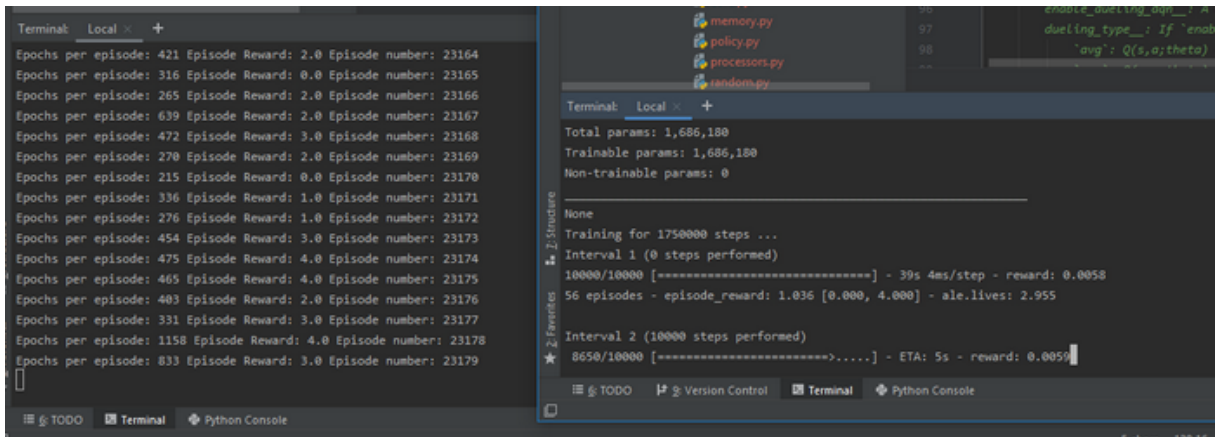


Figure 45: Training through TensorFlow 1 and TensorFlow2 methods placed respectively

5.1 Tensorflow 1 results

We have performed 2 long learning sessions with the TensorFlow 1 agent, first with 20k episodes, which took 27 hours on a Nvidia GTX 960m, and the second one with 60k episodes on GTX 1080. None of them brought successful results in terms of average score of an agent. It can suggest that our first neural network was just not appropriate for this game. So it is probable that it had too few parameters (weights), or just our method was not good enough. Results of 20k learning sessions can be seen in the Figure 46, results of the second (60k) sessions are available in the Figure 47

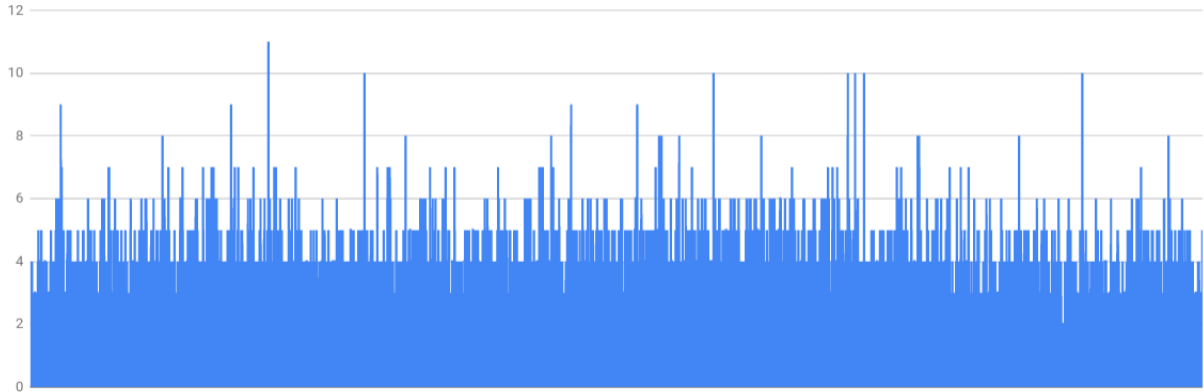


Figure 46: Training session results - 20k episodes

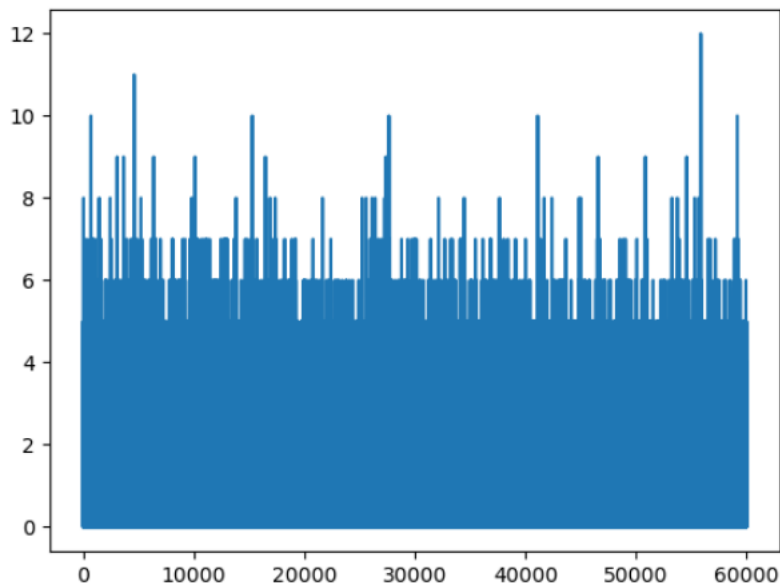


Figure 47: Training session results - 60k episodes

5.2 Tensorflow 2 results

The agent was trained for a total of 1.75M steps which equated to 4,055 episodes. This took between 12 and 18 hours on a Nvidia GTX 1660Ti. The resulting agent was able to play breakout

at an impressive level, obtaining a score of between 50 and 120 points in each game.

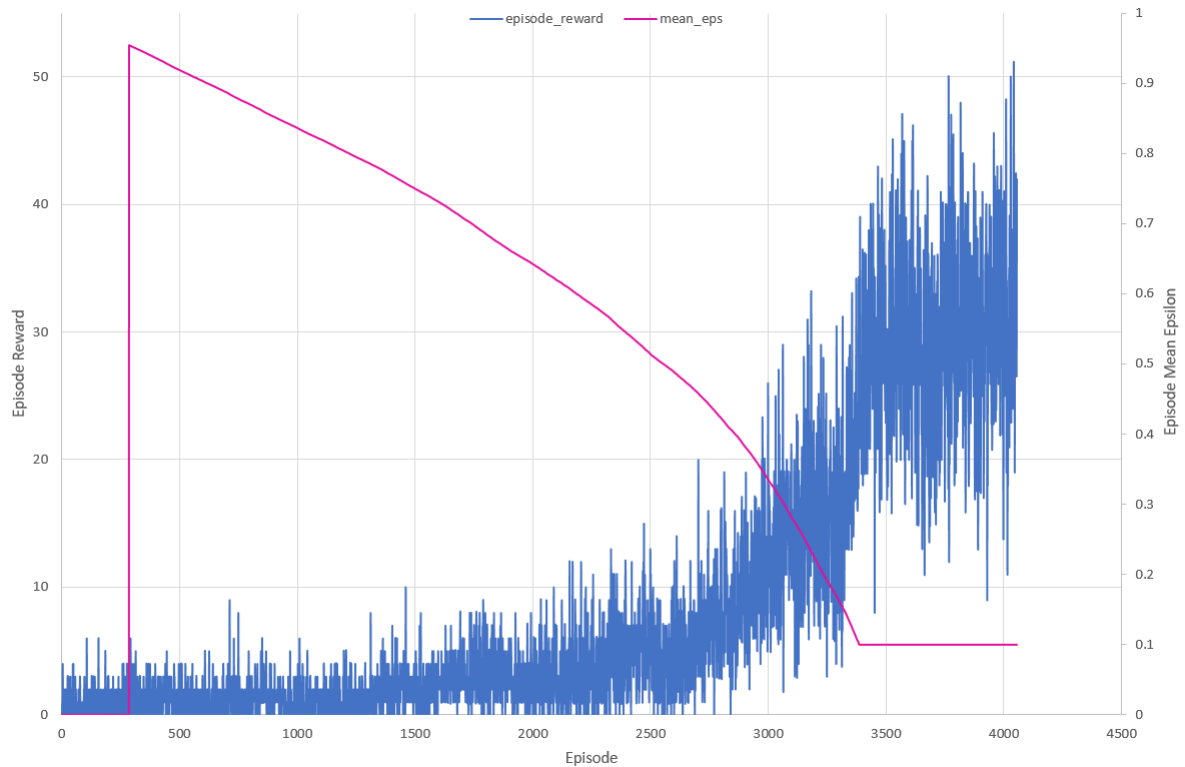


Figure 48: Reward and mean epsilon value during training over 4055 episodes (1.75M steps)

The figure above showcases some logged metrics captured during training. As mentioned previously, training was initiated with a warm-up phase (50,000 steps/283 episodes) of random actions in order to build a foundation of exploratory experiences for the agent’s memory. Training began at episode 284, with an initial mean value for epsilon of 0.95. The final value for epsilon was 0.1. Further reductions in epsilon would increase the reward, a result of the increased exploitation. This model was then rendered into a .mp4 video which can be seen here for a visual representation of the results: <https://youtu.be/K29Q04bNtwI>

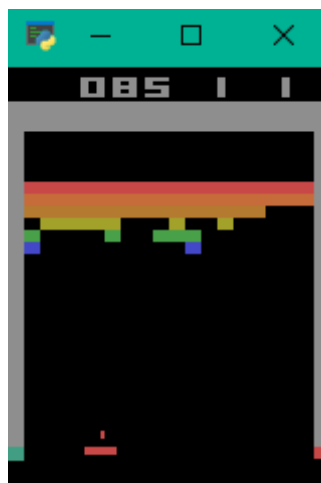


Figure 49: Testing the agent

6 Conclusion

Overall, the assignment was a success as a model of an Atari game was created with its actions being connected to a neural network. The neural network worked with the screen as an input and used convolutional layers to interpolate the screen to identify various patterns to learn to play Breakout for the Atari 2600. Two models were used and eventually a model which contained a larger input field and wider fully connected networking layer showed progression. The scores were getting larger and larger and more consistent the more iterations the agent had. Learning positive reinforcement from the reward score and attributing action selection to this guided the agent to respectable reward output.

7 References

- A.Y. Xu. Automating Pac-man with Deep Q-learning:** An Implementation in Tensorflow. Fundamentals of Reinforcement Learning. Medium [online] <https://towardsdatascience.com/automating-pac-man-with-deep-q-learning-an-implementation-in-tensorflow-ca08e9891d9c>. 23 December 2019 (accessed 14 April 2020).
- A. Choudhary. A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python. Analytics Vidhya.** <https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/> 8 April 2019 (accessed 2 April 2020). GitHub. wau/keras-rl2: Reinforcement learning with tensorflow 2 keras. <https://github.com/wau/keras-rl2>. 29 January 2019 (accessed 14 April 2020).
- Juha Kiili. Reinforcement Learning Tutorial Part-3: Basic Deep Q-Learning.** <https://towardsdatascience.com/reinforcement-learning-tutorial-part-3-basic-deep-q-learning-186164c3bf4> (accessed 7 April 2020).
- Juha Kiili. Reinforcement Learning Tutorial Part-1: Q-Learning.** <https://blog.valohai.com/reinforcement-learning-tutorial-part-1-q-learning> (accessed 7 April 2020).
- S. Gu, T. Lillicrap, I. Sutskever and S. Levine. Continuous Deep Q-Learning with Model-based Acceleration,** v.1, pp. 1-10, 2 March 2016, Proceedings. Cornell University. <https://arxiv.org/abs/1603.00748> (accessed 4 April 2020).
- T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, G. Dulac-Arnold, J. Agapiou, J.Z. Leibo and A. Gruslys.** The Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), 29 April 2018, Presentations. <https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16976/16682> Deep Q-learning From Demonstrations. (accessed 3 April 2020).
- V. Mnih, K. Kavukcuoglu, D. Sliver, A.A. Rusu, J. Rusu, M.G. Bellemare, A. Grave, M. Riedmille, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, A. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg and D. Hassabis.** Human-level control through deep reinforcement learning. Nature: Macmillan Publishers Limited, v.518, pp. 529- 541, doi:10.1038/nature14236. 26 February 2015 (accessed 4 April 2020).
- V. Mnih, K. Kavukcuoglu, D. Sliver, A. Graves, L. Antonoglou, D. Wierstra and M. Riedmiller.** Playing Atari with Deep Reinforcement Learning. NIPS Deep Learning Workshop: Cornell University, v.1, pp. 1-9, <https://arxiv.org/pdf/1312.5602.pdf>. 19 December 2013 (accessed 4 April 2020).