# Deep Reinforcement Learning

**COMP341 - Robot Perception and Manipulation - Assignment 2**

**Report documentation**

| **Team Members** | | |
|---|---|---|
| **Student Name** | **Student ID** | **Contributions** |
| Conor O'Sullivan | 201411716 | Demonstration Video, Report Step: 1, 2, 5 |
| Elena Manoli | 201426062 | Report Step: 1, 3 |
| Hadi Aljishi | 201387758 | Report Step: 2, 4 |
| Mihai Tranca | 201437789 | Report Step: 1, 2, 3, 5 |
| Mike Semple | 201352715 | All Code, Training, Report Step: 1, 2, 4, 5 |

# Table of contents:

## Step 1. Import an OpenAI Gym environment

First, we must set up the environment. We can learn about the available actions, the rewards, maximum episode steps, observation space, and action space by executing the **query_environment** function.

```
def query_environment(name):
  env = gym.make(name)
  spec = gym.spec(name)
  print(f"Action Space: {env.action_space}")
  print(f"possible actions: {env.unwrapped.get_action_meanings()}")
  print(f"Observation Space: {env.observation_space}")
  print(f"Max Episode Steps: {spec.max_episode_steps}")
  print(f"Nondeterministic: {spec.nondeterministic}")
  print(f"Reward Range: {env.reward_range}")
  print(f"Reward Threshold: {spec.reward_threshold}")
```

*Figure 1.1 - Snippet of Code*

The environment's name is then passed. We went with Atlantis in our case.

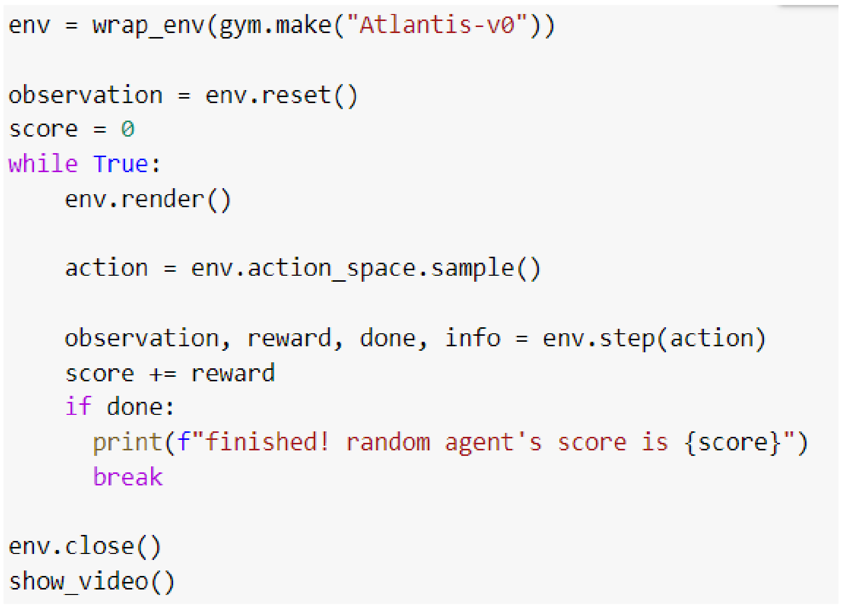
```
query_environment("Atlantis-v0")
```

*Figure 1.2 - Snippet of Code*

```
Action Space: Discrete(4)
possible actions: ['NOOP', 'FIRE', 'RIGHTFIRE', 'LEFTFIRE']
Observation Space: Box(0, 255, (210, 160, 3), uint8)
Max Episode Steps: 27000
Nondeterministic: False
Reward Range: (-inf, inf)
Reward Threshold: None
```

*Figure 1.3 - Result of environment query*

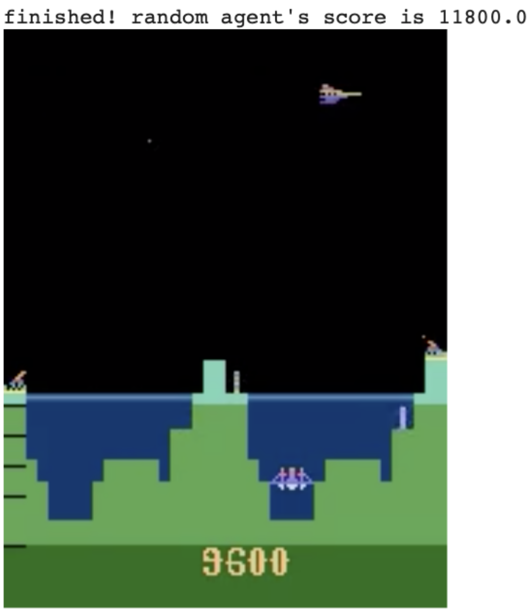
To enable video recording of the gym environment and then display it we created the wrap_env function and the show_video function as shown in Figure 1.4.



```
env = wrap_env(gym.make("Atlantis-v0"))

observation = env.reset()
score = 0
while True:
    env.render()

    action = env.action_space.sample()

    observation, reward, done, info = env.step(action)
    score += reward
    if done:
      print(f"finished! random agent's score is {score}")
      break

env.close()
show_video()
```

*Figure 1.4 - Code for Random Agent*

By running the aforementioned code we were able to import and run the environment using a random agent as displayed in Figure 1.5. We found the random agent would get a mean score of around 12,000 - the benchmark to beat. Out of our group, we found our human average after 30 mins playing was about 10,000-15,000 - not too dissimilar from a random agent. We hypothesise that a random agent is actually fairly good at this game since a random agent sampling uniformly from the action space will have a ¾ probability of shooting one of the 3 guns, as well as a ¼ probability of not shooting. Since there an action is taken every 2-4 frames, and the framerate is 30hz, an action is taken approximately 10 times per second, and approximately 7 of those actions could be shooting. This means that a random agent shooting randomly actually does quite well, since it can shoot 5-6 times per second. We will aim to beat this, using artificial intelligence.

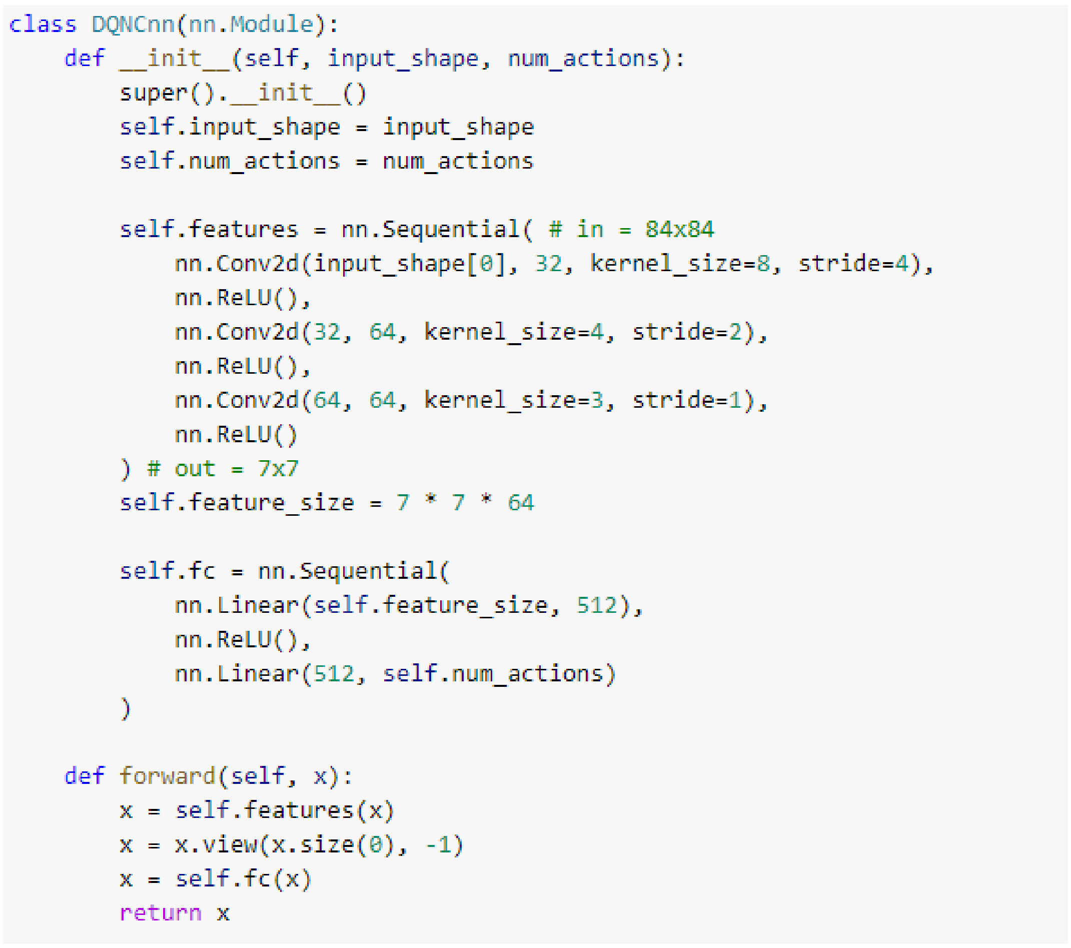


```
finished! random agent's score is 11800.0
```

*Figure 1.5 - Screenshot of game running*

## Step 2. Creating a network

While there are many approaches one can take when tackling a reinforcement learning problem, our approach involved the creation of a DQN or a ‘Deep Quantitative Network’. This takes the best of both Deep Learning and Q Learning. A snippet of code can be found in Figure 2.1.

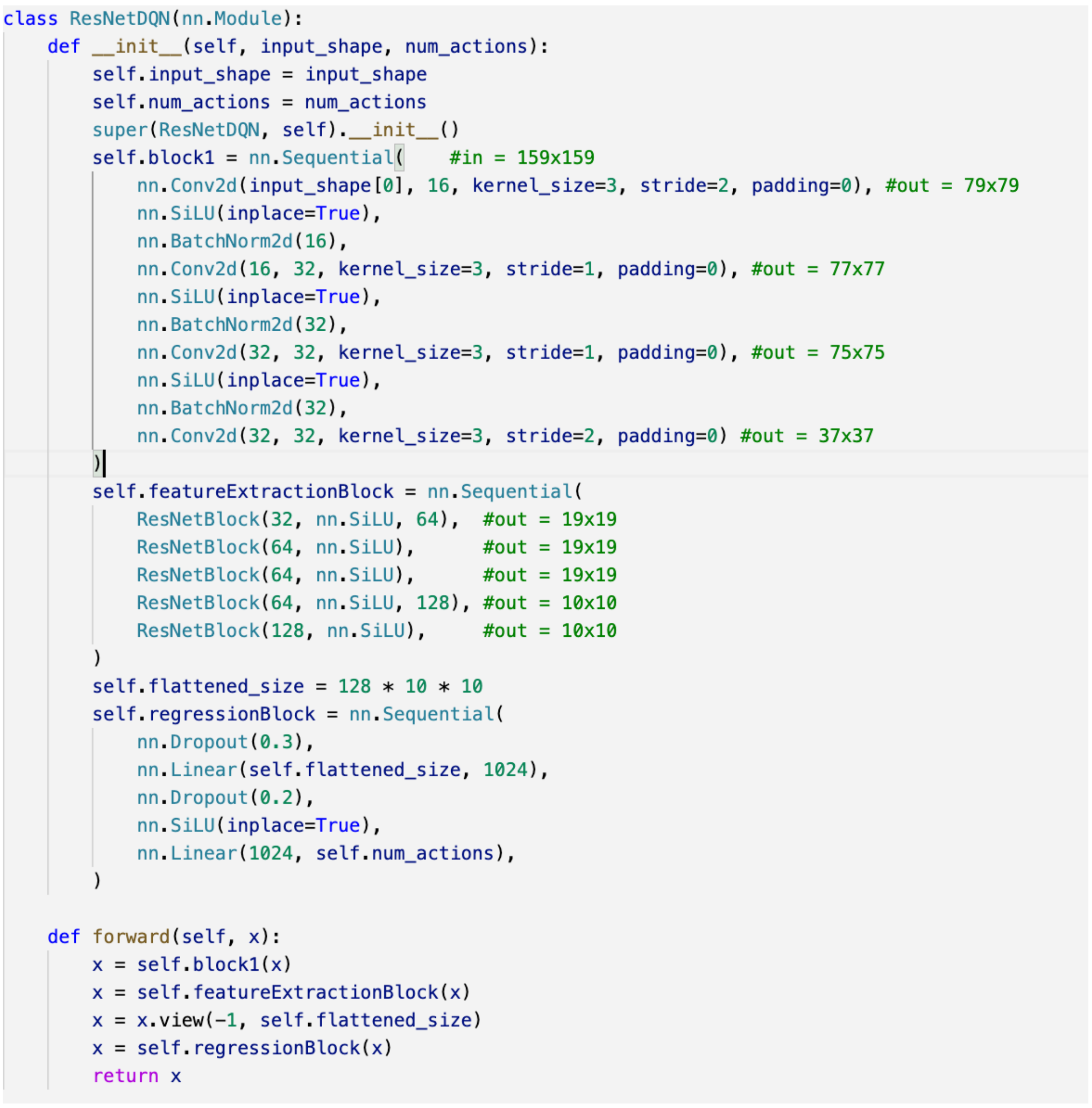


```python
class DQNCnn(nn.Module):
    def __init__(self, input_shape, num_actions):
        super().__init__()
        self.input_shape = input_shape
        self.num_actions = num_actions

        self.features = nn.Sequential( # in = 84x84
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        ) # out = 7x7
        self.feature_size = 7 * 7 * 64

        self.fc = nn.Sequential(
            nn.Linear(self.feature_size, 512),
            nn.ReLU(),
            nn.Linear(512, self.num_actions)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

*Figure 2.1 - The DQN model*

The architecture of the network we have decided to put in place has an 84x84 input that is produced by our preprocessing functions.

The DQN CNN network we have created is the same as the one used in the original DQN paper and the “Rainbow: Combining Improvements in Deep Reinforcement Learning” paper published in 2017. It has 3 convolutional layers: 32, 64, and 64 channels. The layers use 8 × 8 filters with stride 4, 4 × 4  filters with stride 2, and 3 × 3 filters with stride 1. The hidden layer is 512 units and the output of the network is 7 x 7. The agent uses the network to find the action that it should take. It is initialised as shown in Figure 2.4.



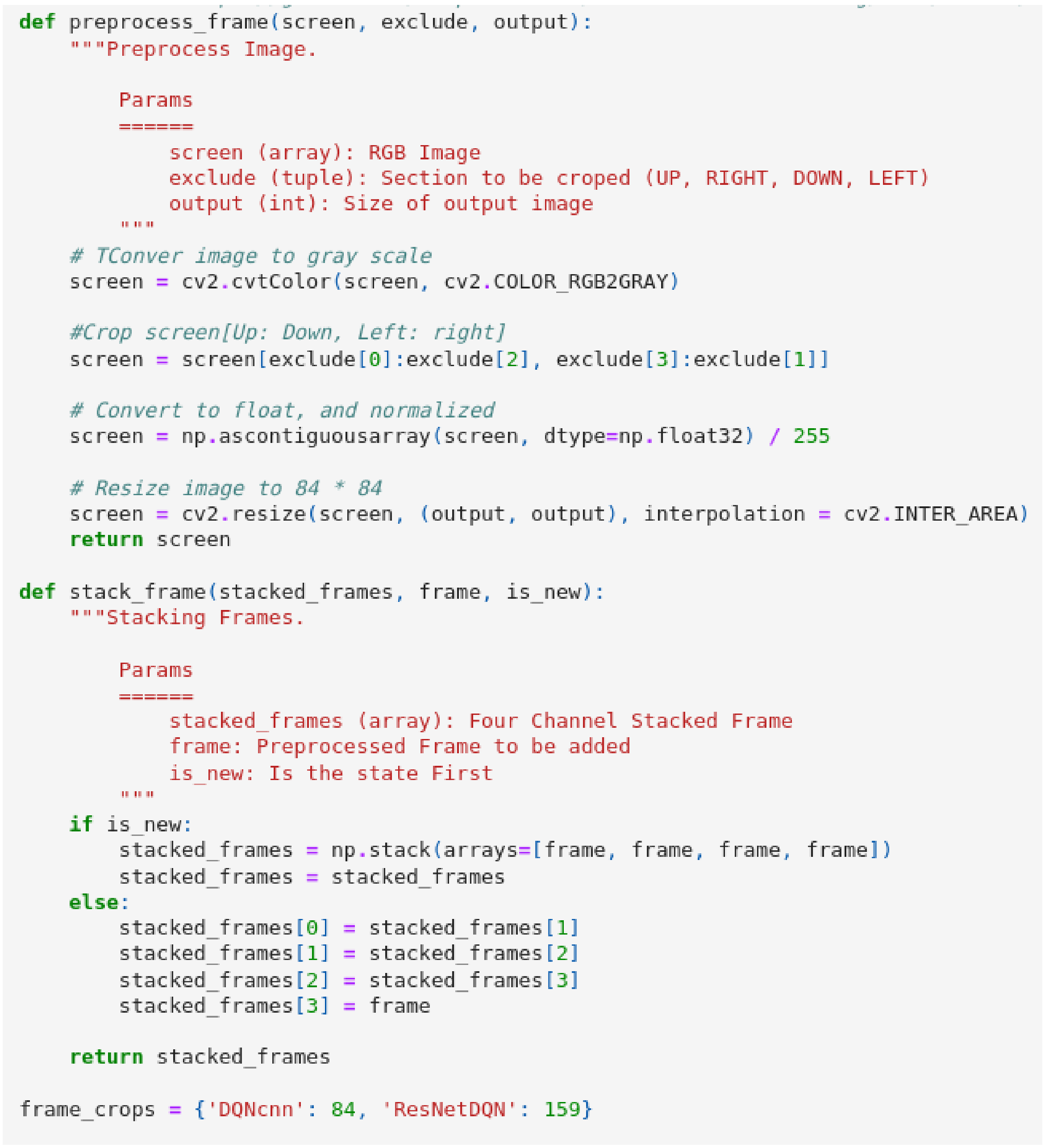
```python
class ResNetDQN(nn.Module):
    def __init__(self, input_shape, num_actions):
        self.input_shape = input_shape
        self.num_actions = num_actions
        super(ResNetDQN, self).__init__()
        self.block1 = nn.Sequential(    #in = 159x159
            nn.Conv2d(input_shape[0], 16, kernel_size=3, stride=2, padding=0), #out = 79x79
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(16),
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=0), #out = 77x77
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 32, kernel_size=3, stride=1, padding=0), #out = 75x75
            nn.SiLU(inplace=True),
            nn.BatchNorm2d(32),
            nn.Conv2d(32, 32, kernel_size=3, stride=2, padding=0) #out = 37x37
        )
        self.featureExtractionBlock = nn.Sequential(
            ResNetBlock(32, nn.SiLU, 64),  #out = 19x19
            ResNetBlock(64, nn.SiLU),      #out = 19x19
            ResNetBlock(64, nn.SiLU),      #out = 19x19
            ResNetBlock(64, nn.SiLU, 128), #out = 10x10
            ResNetBlock(128, nn.SiLU),     #out = 10x10
        )
        self.flattened_size = 128 * 10 * 10
        self.regressionBlock = nn.Sequential(
            nn.Dropout(0.3),
            nn.Linear(self.flattened_size, 1024),
            nn.Dropout(0.2),
            nn.SiLU(inplace=True),
            nn.Linear(1024, self.num_actions),
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.featureExtractionBlock(x)
        x = x.view(-1, self.flattened_size)
        x = self.regressionBlock(x)
        return x
```

*Figure 2.2 - ResNetDQN*

The ResNet DQN network aims to use recent discoveries in deep learning to improve the backbone network of the DQN algorithm. We use a residual connection block to improve the performance of the network, whilst increasing the depth of the network to 14 convolutional layers and 2 dense linear layers. We hypothesise that the residual connections will allow for better learning by trying to eliminate the vanishing gradient problem, whilst allowing for a deeper network to learn more complex features.

One way we could have improved this would have been to create a dual DQN network, with 2 network heads - one to estimate the Q value of a given state and one to estimate the advantage of each action in the state.

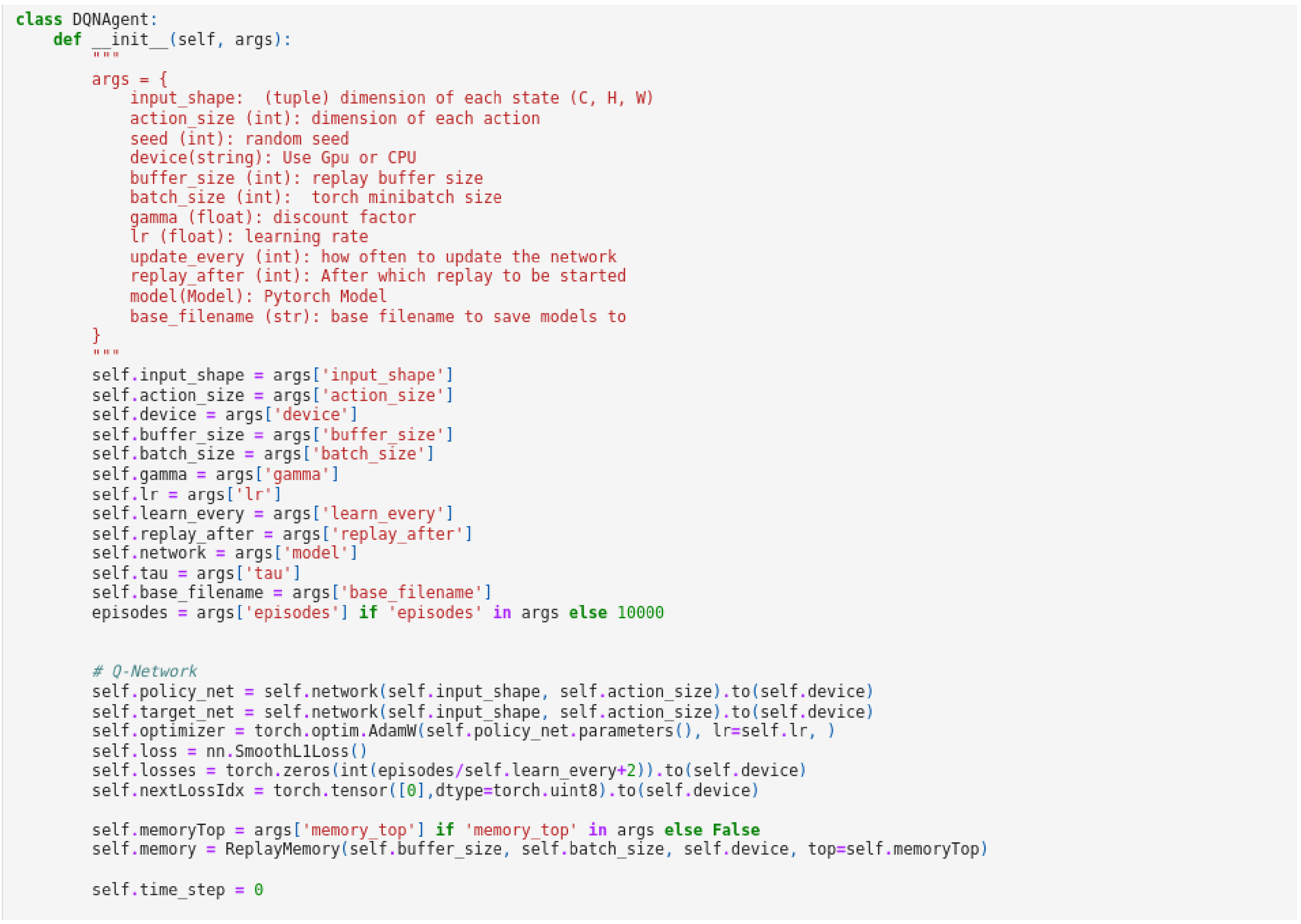
As displayed in Figure 2.3 the pre-processing starts by stacking 4 frames and then converting the image to grayscale, cropping and rescaling the image. We crop the image to only the area on the screen that the enemy ships actually occupy. Then for The DQNCnn we rescale the image to 84x84 and for the ResNetDQN we rescale the image to 159x159. Then the 4 consecutive frames are concatenated as each state's representation.



```python
def preprocess_frame(screen, exclude, output):
    """Preprocess Image.

        Params
        ======
            screen (array): RGB Image
            exclude (tuple): Section to be croped (UP, RIGHT, DOWN, LEFT)
            output (int): Size of output image
        """
    # TConver image to gray scale
    screen = cv2.cvtColor(screen, cv2.COLOR_RGB2GRAY)

    #Crop screen[Up: Down, Left: right]
    screen = screen[exclude[0]:exclude[2], exclude[3]:exclude[1]]

    # Convert to float, and normalized
    screen = np.ascontiguousarray(screen, dtype=np.float32) / 255

    # Resize image to 84 * 84
    screen = cv2.resize(screen, (output, output), interpolation = cv2.INTER_AREA)
    return screen

def stack_frame(stacked_frames, frame, is_new):
    """Stacking Frames.

        Params
        ======
            stacked_frames (array): Four Channel Stacked Frame
            frame: Preprocessed Frame to be added
            is_new: Is the state First
        """
    if is_new:
        stacked_frames = np.stack(arrays=[frame, frame, frame, frame])
        stacked_frames = stacked_frames
    else:
        stacked_frames[0] = stacked_frames[1]
        stacked_frames[1] = stacked_frames[2]
        stacked_frames[2] = stacked_frames[3]
        stacked_frames[3] = frame

    return stacked_frames

frame_crops = {'DQNcnn': 84, 'ResNetDQN': 159}
```

*Figure 2.3 - Preprocessing functions*

In Figure 2.4, the learning agent connected to the network is defined. It takes a variety of different inputs, which can be tweaked depending on which machine the learning process is happening. For instance, Google Colab has a limit for the amount of time any given training can last, so we can alter the batch size or the number of episodes to accommodate for this. We also pass in the network we are using, either the DQNC or the ResNetDQN, which allows us to better compare the results of each method once training is complete

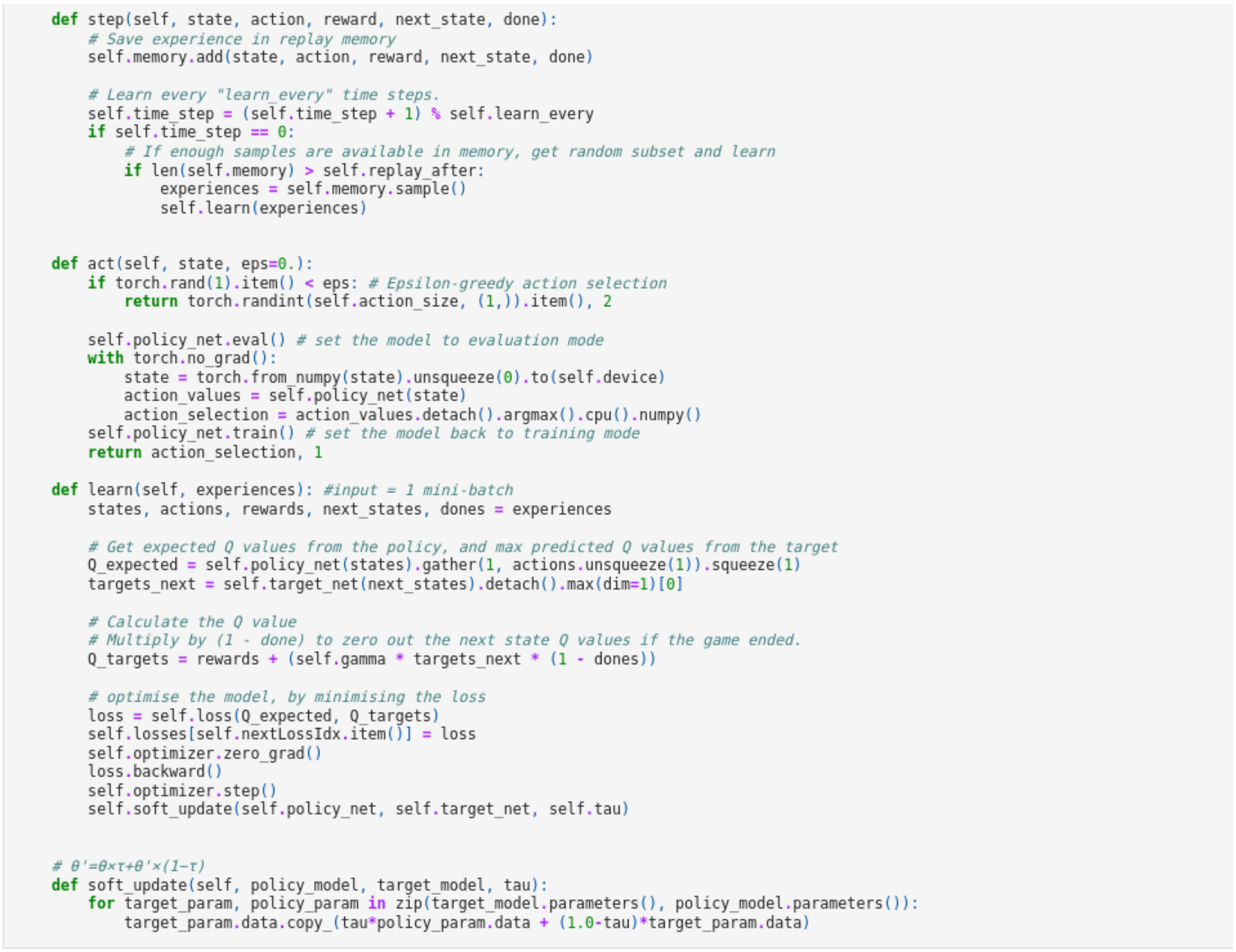


```python
class DQNAgent:
    def __init__(self, args):
        """
        args = {
            input_shape:  (tuple) dimension of each state (C, H, W)
            action_size (int): dimension of each action
            seed (int): random seed
            device(string): Use Gpu or CPU
            buffer_size (int): replay buffer size
            batch_size (int):  torch minibatch size
            gamma (float): discount factor
            lr (float): learning rate
            update_every (int): how often to update the network
            replay_after (int): After which replay to be started
            model(Model): Pytorch Model
            base_filename (str): base filename to save models to
        }
        """
        self.input_shape = args['input_shape']
        self.action_size = args['action_size']
        self.device = args['device']
        self.buffer_size = args['buffer_size']
        self.batch_size = args['batch_size']
        self.gamma = args['gamma']
        self.lr = args['lr']
        self.learn_every = args['learn_every']
        self.replay_after = args['replay_after']
        self.network = args['model']
        self.tau = args['tau']
        self.base_filename = args['base_filename']
        episodes = args['episodes'] if 'episodes' in args else 10000


        # Q-Network
        self.policy_net = self.network(self.input_shape, self.action_size).to(self.device)
        self.target_net = self.network(self.input_shape, self.action_size).to(self.device)
        self.optimizer = torch.optim.AdamW(self.policy_net.parameters(), lr=self.lr, )
        self.loss = nn.SmoothL1Loss()
        self.losses = torch.zeros(int(episodes/self.learn_every+2)).to(self.device)
        self.nextLossIdx = torch.tensor([0],dtype=torch.uint8).to(self.device)

        self.memoryTop = args['memory_top'] if 'memory_top' in args else False
        self.memory = ReplayMemory(self.buffer_size, self.batch_size, self.device, top=self.memoryTop)

        self.time_step = 0
```

*Figure 2.4 - DQN Agent*

The function used by the agent for the loss function is the smooth l1 loss or huber loss. Huber loss works by calculating the l2 norm until a certain threshold where it will go back to l1. This combines the advantages of both MSE and MAE, as outliers don't dramatically affect the loss, whilst the loss is stable in the most important area.

The DQN Agent class shown in Figure 2.5 makes the agent able to act, step, and learn from the environment.



```python
def step(self, state, action, reward, next_state, done):
    # Save experience in replay memory
    self.memory.add(state, action, reward, next_state, done)

    # Learn every "learn_every" time steps.
    self.time_step = (self.time_step + 1) % self.learn_every
    if self.time_step == 0:
        # If enough samples are available in memory, get random subset and learn
        if len(self.memory) > self.replay_after:
            experiences = self.memory.sample()
            self.learn(experiences)


def act(self, state, eps=0.):
    if torch.rand(1).item() < eps: # Epsilon-greedy action selection
        return torch.randint(self.action_size, (1,)).item(), 2

    self.policy_net.eval() # set the model to evaluation mode
    with torch.no_grad():
        state = torch.from_numpy(state).unsqueeze(0).to(self.device)
        action_values = self.policy_net(state)
        action_selection = action_values.detach().argmax().cpu().numpy()
    self.policy_net.train() # set the model back to training mode
    return action_selection, 1

def learn(self, experiences): #input = 1 mini-batch
    states, actions, rewards, next_states, dones = experiences

    # Get expected Q values from the policy, and max predicted Q values from the target
    Q_expected = self.policy_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
    targets_next = self.target_net(next_states).detach().max(dim=1)[0]

    # Calculate the Q value
    # Multiply by (1 - done) to zero out the next state Q values if the game ended.
    Q_targets = rewards + (self.gamma * targets_next * (1 - dones))

    # optimise the model, by minimising the loss
    loss = self.loss(Q_expected, Q_targets)
    self.losses[self.nextLossIdx.item()] = loss
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    self.soft_update(self.policy_net, self.target_net, self.tau)


# θ'=θ×τ+θ'×(1−τ)
def soft_update(self, policy_model, target_model, tau):
    for target_param, policy_param in zip(target_model.parameters(), policy_model.parameters()):
        target_param.data.copy_(tau*policy_param.data + (1.0-tau)*target_param.data)
```

*Figure 2.5 - DQN Agent class methods*

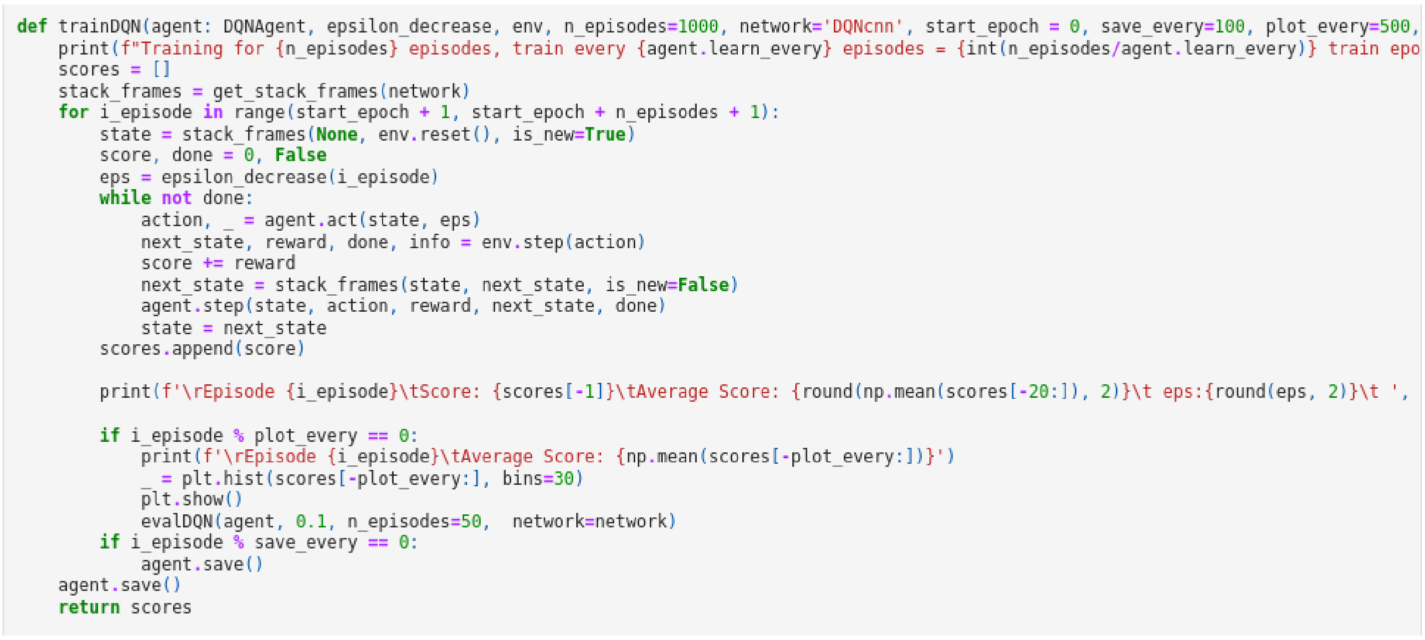
It is worth noting that taking a closer look at Figure 2.3 it can be noticed our DQN model is a double DQN model using two identical networks, one network is being updated more slowly than the other so that the model does not chase its own tail.

The training is done by letting the agent choose an action and adding the reward and every frame to the memory every 10 episodes we run the learning function that takes a subset of stacked frames and uses these to train the network. We then evaluate the agent for 100 episodes.

## Step 3. Connecting the environment to the network

We connect the network to the environment by stacking the frames into groups of 4 and sending them to the DQN agent.
The DQN agent after analyzing returns an action that should be taken. This is done until the game is over. A code example of this can be seen in Figure 3.1 where we train the agent.

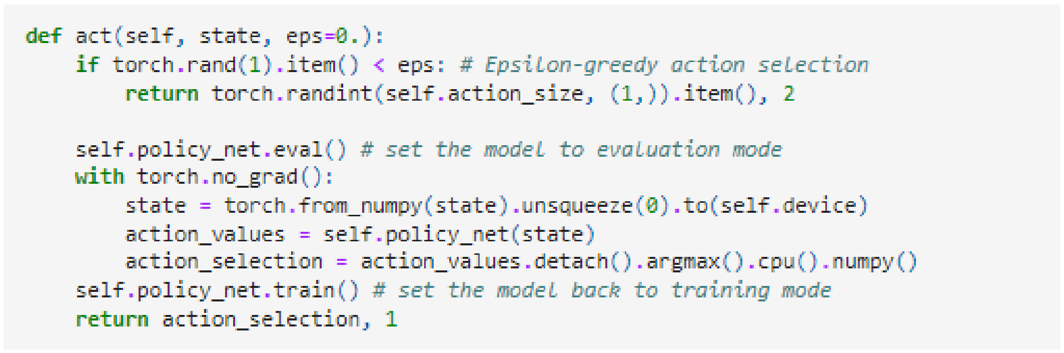
Throughout the training, we want to collect data from the environment and use it to teach the agent. The action will be chosen in accordance with an epsilon greedy policy. To begin, we'll reset the environment and initialise the state. To train the agent, we want to run each training with as many training episodes as possible. Then we take a sample of action, perform it, and observe the following screen and reward. The act function can be seen in Figure 3.2. When the episode is finished, the loop is restarted.
We use matplotlib to plot the duration of episodes along with an average over the last 100 episodes. The plot will get updated after every episode.



```
def trainDQN(agent: DQNAgent, epsilon_decrease, env, n_episodes=1000, network='DQNcnn', start_epoch = 0, save_every=100, plot_every=500,
    print(f"Training for {n_episodes} episodes, train every {agent.learn_every} episodes = {int(n_episodes/agent.learn_every)} train epo
    scores = []
    stack_frames = get_stack_frames(network)
    for i_episode in range(start_epoch + 1, start_epoch + n_episodes + 1):
        state = stack_frames(None, env.reset(), is_new=True)
        score, done = 0, False
        eps = epsilon_decrease(i_episode)
        while not done:
            action, _ = agent.act(state, eps)
            next_state, reward, done, info = env.step(action)
            score += reward
            next_state = stack_frames(state, next_state, is_new=False)
            agent.step(state, action, reward, next_state, done)
            state = next_state
        scores.append(score)

        print(f'\rEpisode {i_episode}\tScore: {scores[-1]}\tAverage Score: {round(np.mean(scores[-20:]), 2)}\t eps:{round(eps, 2)}\t ',

        if i_episode % plot_every == 0:
            print(f'\rEpisode {i_episode}\tAverage Score: {np.mean(scores[-plot_every:])}')
            _ = plt.hist(scores[-plot_every:], bins=30)
            plt.show()
            evalDQN(agent, 0.1, n_episodes=50,  network=network)
        if i_episode % save_every == 0:
            agent.save()
    agent.save()
    return scores
```

*Figure 3.1 - The Train Function of the DQN Agent*

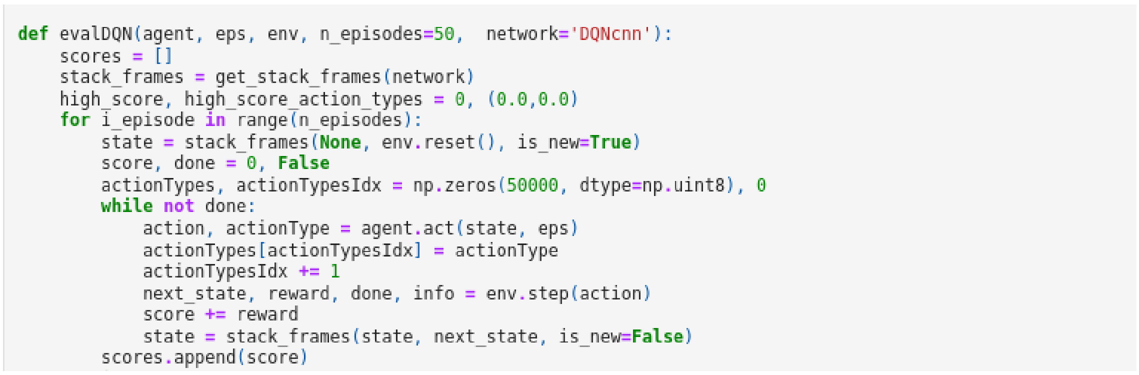
Using the Epsilon-Greedy Exploration technique, we choose an action. The agent chooses a random action with probability epsilon and utilizes the best-known action with probability 1 - epsilon. Then we set the model to the evaluation model which is covered in the next step.



```
def act(self, state, eps=0.):
    if torch.rand(1).item() < eps: # Epsilon-greedy action selection
        return torch.randint(self.action_size, (1,)).item(), 2

    self.policy_net.eval() # set the model to evaluation mode
    with torch.no_grad():
        state = torch.from_numpy(state).unsqueeze(0).to(self.device)
        action_values = self.policy_net(state)
        action_selection = action_values.detach().argmax().cpu().numpy()
    self.policy_net.train() # set the model back to training mode
    return action_selection, 1
```

*Figure 3.2 - The Action Function of the DQN Agent*

The expected return after taking action ***a*** in state ***s*** is represented by Q-values. As a result, the Q-values of better actions will be higher. The average reward in the last timesteps would be a better performance metric. If our agent improves, its average reward should rise as well.
We return the sum of rewards gained while running a policy for an episode in our evaluation. A total of several episodes are shown, resulting in an average return.



```
def evalDQN(agent, eps, env, n_episodes=50,  network='DQNcnn'):
    scores = []
    stack_frames = get_stack_frames(network)
    high_score, high_score_action_types = 0, (0.0,0.0)
    for i_episode in range(n_episodes):
        state = stack_frames(None, env.reset(), is_new=True)
        score, done = 0, False
        actionTypes, actionTypesIdx = np.zeros(50000, dtype=np.uint8), 0
        while not done:
            action, actionType = agent.act(state, eps)
            actionTypes[actionTypesIdx] = actionType
            actionTypesIdx += 1
            next_state, reward, done, info = env.step(action)
            score += reward
            state = stack_frames(state, next_state, is_new=False)
        scores.append(score)
```

*Figure 3.3 - The Evaluation Function of the DQN Agent*

## Step 4. Deep reinforcement learning algorithm

The DQNAgent class shown in figure 2.3 describes our Double DQN model. The class contains several parameters specified on initialization, including two Q-networks. The first network, *policy_net*, is used to estimate the Q-function for the current state and represents the brain of our reinforcement learning model. The second network, *target_net*, estimates the Q-function for future states, which will be used to calculate the temporal difference error and update the policy_net parameters. The agent also uses a simple queue replay memory buffer of size 1,000,000 to store its past experiences in the form of state transitions.

Three functions allow the agent to interact with the environment: act(), step(), and learn(). The first function, act(), takes the current state as input and outputs one of the four actions allowed by the environment. The action is selected using the Epsilon-Greedy method. This means that, according to some parameter epsilon, the agent will either perform a greedy action determined by the output of the policy_net, or it will instead perform a completely random action. Epsilon is sampled using *epsilon_decay()*, which starts at a value of 0.9 and decays after every episode at a rate of number_of_episodes/2 until it reaches a minimum value of 0.02.

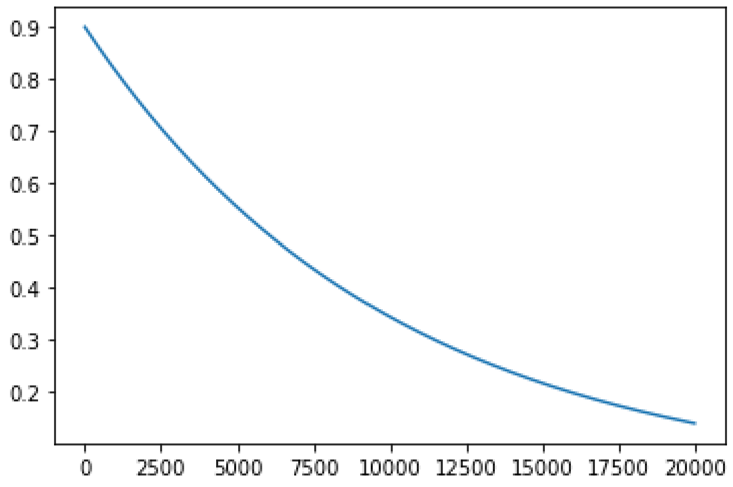


*Figure 4.1 - Epsilon function plot*

The second function, step(), simply observes the new state and the reward after performing an action, and stores them in the replay memory. It also keeps track of how many steps the agent has completed to decide when to update the network parameters using the learn() function.

The *learn()* function takes a batch sample from the replay memory and uses it to calculate the loss and then perform the *optimizer.step()* operation. First, we feed the sampled states and next_states to the policy and target net respectively to find $Q(s_t,a_t)$ and $V(s_{t+1}) = \max_a Q(s_{t+1},a)$. We then calculate the expected Q-values using $r + \gamma V(s_{t+1})$, where r is the reward and $\gamma$ is the discount parameter set to 0.99. If $s_{t+1}$ is a terminal state, $V(s_{t+1})$ will be set to 0 and the expected value will be the reward itself only. Finally, we feed the real and expected Q-values to our Huber Loss function, and update the policy net parameters. After each update, we copy the parameters of the policy net to the target net, scaled by a factor of $\tau$, set to 0.0001. This is equivalent to copying the parameters completely every 1,000 episodes.

Initially, the model was trained for only 2,000 episodes in Google Colab. This took a considerable amount of time and produced subpar scores averaging around 2,000 points. To speed up the training and increase the number of episodes, we opted to abandon Google Colab and train the model using the Barkla cluster instead.

Our final model was trained for 20,000 episodes. In the first 10,000 episodes of training, we used a learning rate of 0.001 and sampled from the memory uniformly. However, for the final 10,000 episodes, the learning rate was decreased to 0.0003, and a different sampling method was used. As the memory contains a disproportionately low amount of positive experiences compared to negative ones (about 1%), in the second half of training, we would sample twice the specified batch size, sort the samples decreasingly by reward, then take only the top half of the sampled entries. This made our training process much more balanced.

Our evaluation process consisted of running the model for 100 episodes using a constant epsilon value of either 0.05 or 0.1. We found that epsilon 0.1 worked better for models where the neural network did not train well, but if the neural network trained well, epsilon 0.05 would produce better results.
For each episode, after each action was performed, we would store not only the resulting reward but also whether the action performed was random or made deliberately by our agent. This allows us to gain a better insight into how the agent operates.

## Step 5. Experimental results

Our ResNet DQN agent was able to train rather well. In Figure 5.1 and Figure 5.2 it can be seen how the network trained. These graphs show the score distribution of the previous 1000 episodes, and how the network got progressively better.

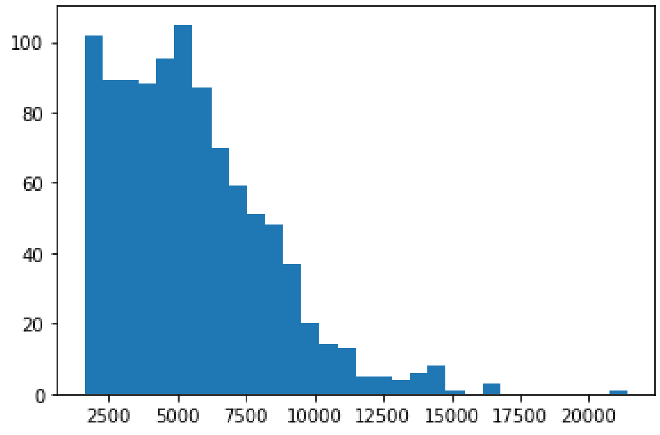


*Figure 5.1 - Snapshot of training results between episodes 5000-6000 using the ResNet DQN*

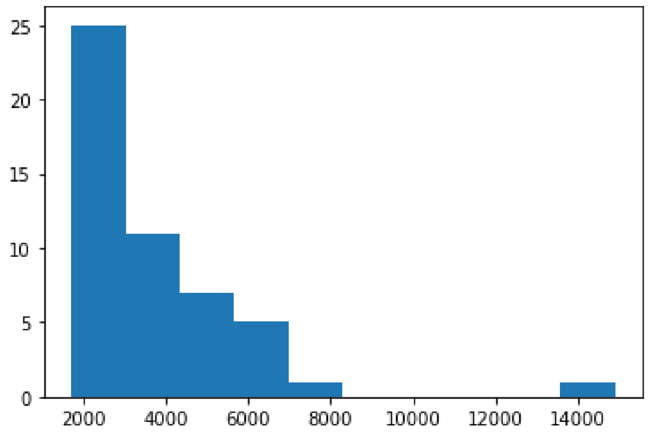


*Figure 5.2 - Snapshot of evaluation of 100 episodes after 1000 episodes*

As mentioned above, the ResNet DQN agent training performance increases significantly if it runs for more episodes and this can be seen by comparing the results in Figure 5.1 with the results in Figure 5.3. The score distribution has significantly fewer low scores and many more higher scores.

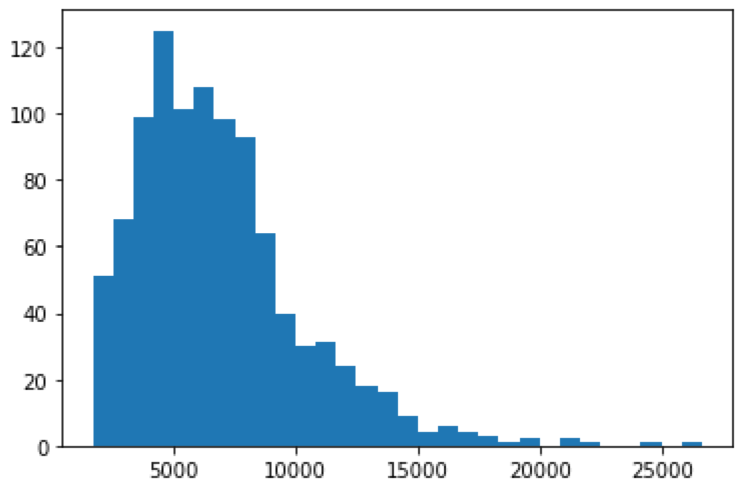
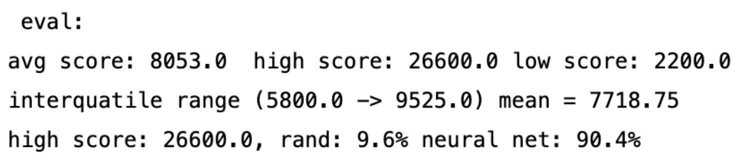


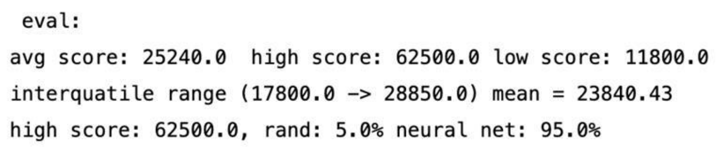
*Figure 5.3 - Snapshot of the ResNet Agent between 14000-15000 episodes*

By taking a glance at Figure 5.4 it quickly becomes apparent that our DQN agent also evaluation performs rather well.



```
 eval:
avg score: 8053.0  high score: 26600.0 low score: 2200.0
interquatile range (5800.0 -> 9525.0) mean = 7718.75
high score: 26600.0, rand: 9.6% neural net: 90.4%
```

*Figure 5.4 - DQN Agent evaluation for 100 episodes, eps = 0.1*



```
 eval:
avg score: 25240.0  high score: 62500.0 low score: 11800.0
interquatile range (17800.0 -> 28850.0) mean = 23840.43
high score: 62500.0, rand: 5.0% neural net: 95.0%
```

*Figure 5.5 - ResNet DQN evaluation for 100 episodes, eps = 0.05*

Comparing the results of the DQN Agent in Figure 5.6 with the results of the ResNetDQN Agent evaluation shown in Figure 5.7.

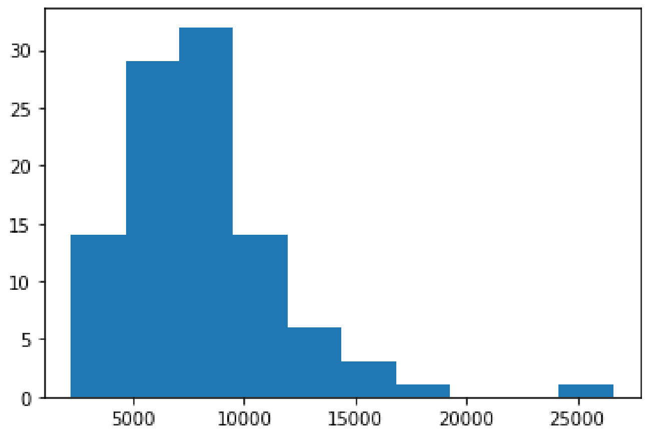


*Figure 5.6 - DQN Agent evaluation results over 100 episodes.*

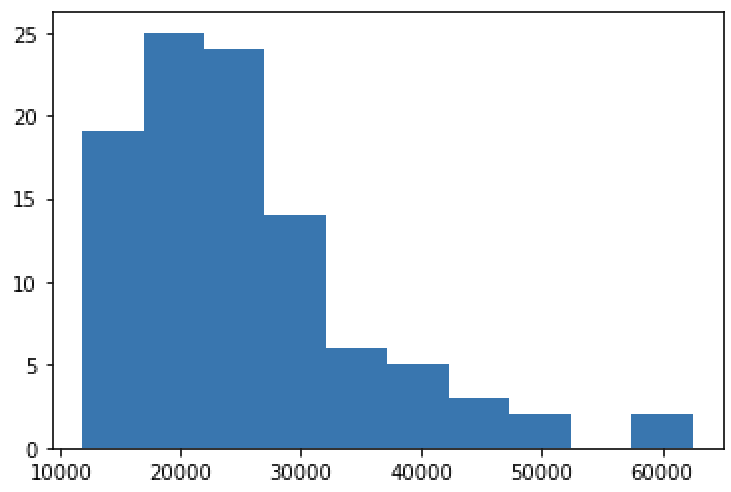
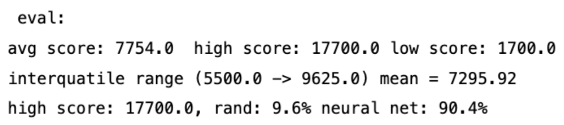


*Figure 5.7 - ResNet DQN evaluation results over 100 episodes.*

As you can see, the ResNet DQN Agent managed to outperform the DQN Agent massively - the average score of the ResNet DQN was nearly as large as the high score from the DQN Agent.
Our ResNet DQN was able to achieve performance equal to that achieved in the paper "Deep Reinforcement Learning with Double Q-learning" [Ziyu Wang et al.]. Using the double DQN architecture, the authors managed to achieve a high score of 64,758 - whereas our high score was 62,500. This is very good compared to their 'human average' score of 29,028.
Also, note that they used the RMSProp optimisation procedure compared to our model using AdamW, so we would expect our implementation could have been improved when using RMSProp with a bayesian hyperparameter search, but this would have been unnecessary for this assignment. Our implementation could have also been improved by using a learning rate scheduler, such as 1Cycle, which has shown to improve both the speed of learning and the quality of the learned model.

*Figure 5.8 - ResNet DQN Agent final demo*
*average score = 28,620, high score = 51300*



```
 eval:
avg score: 7754.0  high score: 17700.0 low score: 1700.0
interquatile range (5500.0 -> 9625.0) mean = 7295.92
high score: 17700.0, rand: 9.6% neural net: 90.4%
```

*Figure 5.9 - DQN architecture evaluation score after 100 episodes at eps = 0.1*

**Please find the video link to several attempts made by our ResNet DQN model below:**

https://liverpool.instructuremedia.com/embed/a98e5b8a-dedf-44e2-a7d1-6c905eff2cdb

As a result of our work, we were able to develop an agent that is a competent player of Atari's Atlantis. Reinforcement learning as a whole is been an incredibly powerful tool for this kind of project, in some cases, being able to produce an agent that can achieve scores unreachable by a human player. Clearly, our agent isn't quite as adept from our brief attempt in this assignment, given that the current high score for Atlantis by a machine is the GDI-H3, which achieved a score of 3,837,300 - a far cry from our own attempt. This goes to show what can be done at the higher end of machine learning. GDI's approach involved looking at 200 Million frames to learn, which clearly would not have been feasible in this project,

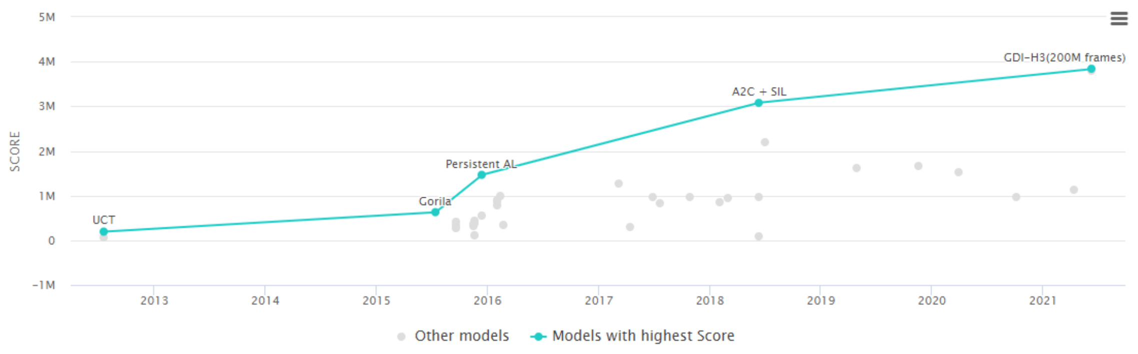


*Figure 5.11 - Performance of other algorithms on Atlantis*

In the wider world, reinforcement learning can be one of the most powerful machine-learning methods, with examples like natural language processing and self-driving cars being just two clear-cut examples of its incredible use in the world today. Our results today barely scratch the surface of the myriad applications for reinforcement learning, but it is certainly the beginning of having a comprehensive understanding of everything this approach is capable of.

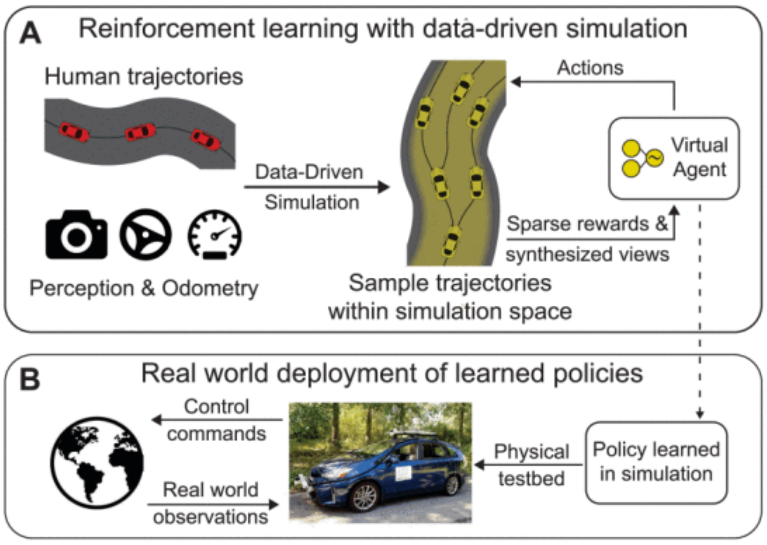

A Reinforcement learning with data-driven simulation
Human trajectories
Data-Driven Simulation
Perception & Odometry
Actions
Virtual Agent
Sparse rewards & synthesized views
Sample trajectories within simulation space
B Real world deployment of learned policies
Control commands
Real world observations
Physical testbed
Policy learned in simulation

*Figure 5.12 - Self-Driving Approach with Reinforcement learning*

One of the primary lessons learned from this project was the amount of time it is necessary to train an agent in order to achieve worthwhile results. It took thousands of episodes before the taught agent approached the same level as a casual attempt made by a human player, while an agent choosing its moves entirely at random had the potential to perform much better. Clearly, this was not an ideal outcome at the beginning of the project, but with further tweaks to our method and more time spent training, the taught agent gradually began to outpace its randomised competitors.

Another lesson learned came in choosing the right number of episodes to train with, especially considering we were using Google Colab. A smaller number like 2000 was able to train but did not produce worthwhile scores that could surpass even randomised agents, while if the number of episodes was too large, like 15,000, it would've theoretically yielded a better result, but the training time meant that colab would time-out before training was complete. Obviously, choosing a suitable number between these two extremes was vital to the success of the project, a number high enough to produce a competent agent while still being able to complete its training.

Something that greatly impacted the agent's training was the selection of the epsilon-greedy curve selection. The proportion to actions that maximise reward and randomised actions can be difficult to balance and is the cornerstone of the classic problem of RL, exploration vs exploitation. We had to choose a curve that allowed for random action often enough that the agent would get a good estimate of the environment, before tending towards the actions that should maximise reward at each given timestep. For our purposes, an epsilon value of at least 0.5 yielded better results as, when values remained lower, such as ~ 0.1, for an extended period, it led to an agent that would perform significantly worse than predicted. A higher epsilon value granted greater rewards, and thus the agent learned a more beneficial pattern to gain a higher score during training.

Similarly, as with regular deep learning, it was important not to overfit to any one action based on the training data. In the case where the agent appears to perform better in training by shooting only the right-hand gun, for instance, the agent would likely perform worse, during testing on unseen episodes. Training would need to be monitored rigorously to prevent agent over-specialisation.

## References:

Jiajun Fan, Changnan Xiao, Yue Huang; 2021; Accessed 18/04/2022; <https://paperswithcode.com/paper/gdi-rethinking-what-makes-reinforcement>

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas; 2016; Accessed 21/04/2022; <https://arxiv.org/pdf/1511.06581.pdf>

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller; 2013; Accessed 21/04/22; <https://arxiv.org/pdf/1312.5602.pdf>

DQN Experiment with Atari Breakout; n.d; Accessed 21/04/22; <https://nn.labml.ai/rl/dqn/experiment.html>

Deep Q Networks (DQN); n.d; Accessed 21/04/22; <https://nn.labml.ai/rl/dqn/index.html>

Deep Q Network (DQN) Duelling Model; n.d. Accessed 22/04/22; <https://nn.labml.ai/rl/dqn/model.html>