REINFORCMENT LEARNING

Atari - Asteroids

Arnas Juravicius – 18257305

Oisin McNamara – 18237398

Dylan Kearney – 18227023

Cyiaph McCann - 17233453

Why is Reinforcement Learning the machine learning paradigm of choice for this task?

To answer this question, we first need to look at what Reinforcement Learning is and why it is best suited to this project. Reinforcement Learning is the area of machine learning that focuses on how an agent navigates an environment to maximize the cumulation of reward. This method of machine learning is one of the three main paradigms, the other two being Supervised learning and Unsupervised learning. Reinforcement learning introduces the concept of an agent which does not need any dataset to work off. When exploring, it generates its own data as it tries to interpret its environment. This approach brings a range of possibilities to the table in terms of automation.

Going into the architecture in more detail, we shall interpret how the agent operates. The agent in our Atari game will wonder about the environment attempting to play the game. Using trial and error the agent will learn how to play the game. This type of learning is achieved using a policy. A policy is a set of instructions to take the agent in the path with respect to gaining points. The Reward mechanism is what will award points to the agent if a desired path is chosen. Otherwise, the reward punishes or provides the agent with negative points based on a poor decision.

We can see how reinforcement learning is the optimal choice for automating an Atari game as we want our agent to learn how to play the game by itself. In the later sections, we will explore the environment space and the interesting architecture components used in our model.

The Environment

The Atari game selected

The Atari game we selected is Asteroids. Asteroids is a space-themed multidirectional shooter released in 1979 by Atari inc. The basic idea of the game is the player controls a spaceship trapped in an asteroid field which is also complemented with enemy flying saucers. The game's objective is to shoot down the incoming asteroids and saucers while trying not to collide with any of them. The game increases in difficulty as the number of items on screen increases. We chose this game because we wanted to observe what strategy the model would identify to beat it as it is a free-flowing environment (Asteroids (video game), 2021).

The input received from the OpenAI Gym environment

To create this project, we utilized the OpenAI Gym. This is a toolkit for developing and comparing reinforcement learning agents. This toolkit made it convenient for us to implement the game with our model (Gym, 2021). We utilized various inputs from the Gym environment. First, we imported the game environment using gym.make. Next, we needed to utilize the height and width of the environment so our model could interpret the world it is in. The input our model required was the channel. The application we are working with requires multiple channels. This represents the RGB color channels of the Atari game (Francis, 2021). Finally, the actions specify the different variations the AI can take in the problem space.

```
1 # Here we create a new enviroment using gym make function
2 # Asteroids is our game of choice, this can be changed easily to any atari game
3 env = gym.make('Asteroids-v0')
4 # We get the height, width and channels from the observation space of the game enviroment
5 height, width, channels = env.observation_space.shape
6 # We get the joystick actions from the action space of the game enviroment
7 actions = env.action_space.n
```

Figure 1 Gym Environment

The control settings for the joystick

The asteroid game has a large set of controls compared to a lot of other Atari games due to its freedom to navigate anywhere in the environment space. The set of controls includes ['NOOP', 'FIRE' (Allows the ship to fire a laser at targets), 'UP'(flies the rocket in an upward position on the y-axis), 'RIGHT'(Flys the rocket right along the x-axis), 'LEFT'(flies the rocket left along the x-axis), 'DOWN'(flies the rocket down along the y-axis), 'UPRIGHT'(moves the rocket up-right in a diagonal position along the plane), 'UPLEFT' (moves the rocket up-left in a diagonal position along the plane), 'UPFIRE' (aims the rockets bullet in an upward direction to shoot), 'RIGHTFIRE' (aims the rockets bullet in to the right to shoot), 'LEFTFIRE' (aims the rockets bullet in to the left to shoot), 'DOWNFIRE' (aims the rockets bullet in to the down to shoot), 'UPRIGHTFIRE' (aims the rockets bullet in to the up right to shoot), 'UPLEFTFIRE' (aims the rockets bullet in to the down

left to shoot)]. The large set of controls allows the player to develop many different strategies to achieve a higher score.

Implementation

Capture and pre-processing of the data

Before we could implement the model, the data needed to be preprocessed. This preprocessing would allow us to fit the data correctly to the model resulting in more optimal outputs. The first aspect of data preprocessing we carried out was cropping the image. Not all of the image was useful when training the model, cropping the image extracted the useful information making the image smaller thus increasing the speed of training. We did this by making a slice of the image array cropping from 20 pixels down from the top eliminating the score from the image and 5 pixels from the bottom removing empty space. Furthermore, we also resized the image. Resizing the image will reduce the quality of the image but will also greatly increase the training speed of the model. We just resized the image to 84 pixels in height and width reducing from the original image of 210 pixels by 160 pixels. Lastly, we translated the image to grayscale as the model only needs this to be trained. This greatly increased training speed as the model no longer processed three images as one, one image for each color in the RGB color scale.

This image on the left below is image before cropping. This became this image to the right showing only the valuable information. As you can see, the image has been cropped just below the score as the score is not deemed useful when training. Furthermore, we removed 5 pixels from the bottom since this is empty space.

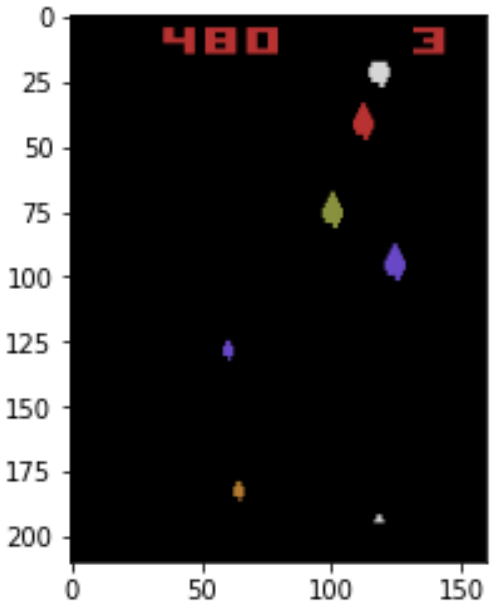

Figure 2 Asteroids before

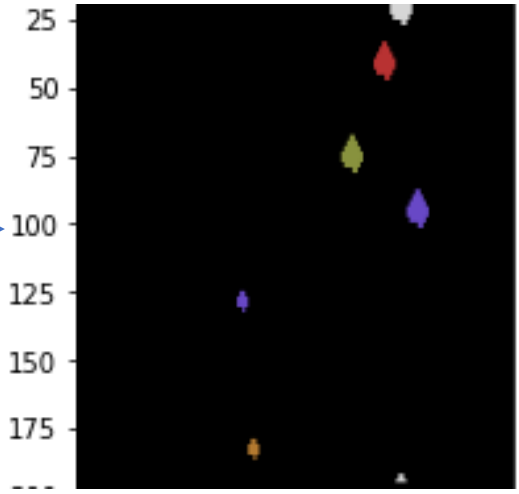

Figure 3 Asteroids after

When preprocessing, we had to create an Atari Processor class which inherited from the keras.rl Processor class. In this class, the preprocessing took place. Here is the processor below. We can see things in the code such as cropping, rescaling, and converting to gray scale.

```python
# We inherit from the keras.rl Processor
# This is used to preprocess the images
class AtariProcessor(Processor):
    def process_observation(self, observation):
        # Do a check
        assert observation.ndim == 3  # (height, width, channel)
        # First we crop the image leaving only the useful information
        observation = observation[20:195]
        # We create an image from the Image library
        img = Image.fromarray(observation)
        # We resize our image to 84 by 84 and convert the image to gray scale
        img = img.resize((84,84)).convert('L')
        # We convert the image back to a numpy array
        processed_observation = np.array(img)
        processed_observation = np.expand_dims(processed_observation, axis=-1)
        # We return the image
        return processed_observation.astype('uint8')  # saves storage in experience memory
```

Figure 4 Atari Processor

The processor is called in the DQN agent itself and can be seen below being used. We initialize the processor first and then call it the processor variable. The game environment which gets passed into the agent as images will then be processed by the processor we defined. Cropping and resizing the images greatly increased training speed, speed before processing for training was around 4-5 hours for 10000 steps, after processing the images, training for 10000 steps takes in-between 40 minutes and an hour.

```python
# Here we inialize the Atari processor
processor = AtariProcessor()
# Here we build the agent using the model and actions
def build_agent(model, actions):
    # Here we create the policy for our agent
    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_
    # This is the memory for the agent
    memory = SequentialMemory(limit=50000, window_length=3)
    # Here we create the DQN agent using things like the model, processor and policy
    dqn = DQNAgent(model=model, processor=processor, memory=memory, policy=policy,
                   enable_dueling_network=True, dueling_type='avg',
                    nb_actions=actions, nb_steps_warmup=1000
                   )
    # Return the DQN agent
    return dqn
```

Figure 5 DQN Agent

The network structure

Our Convolutional Neural Network model is made up of the following:

Convolution layer with 32 filters of size 8x8 with a stride of (4, 4). The activation function is ReLU. The input shape is (3, 84, 84, 1). Since the image is resized to an 84 by 84-pixel image with grayscale. What we are doing in the convolutional layer is training our model to detect different things in the images. We need to be able to detect the obstacles and enemies in the game environment. Following this we stacked more convolutional layers on this one. We finally then use the function Flatten to flatten these layers on top of each other to pass to the dense layer. So, the next obvious step is to add the dense layer also known as a fully connected layer. This layer is connected to every single layer in the next. We have three sets of Dense layers two with an activation of ReLU and one with a linear activation function. These layers compress down as the first layer has 512 units followed by the second dense layer with 256 units and finally a dense layer with the number of actions we have. So, in the last layer whichever one is activated will decide what action our model is going to take.

```
# Here we build the model passing in the number of actions
def build_model(actions):
    # We intialize the HeNormal weight intializier
    weights = tf.keras.initializers.HeNormal()
    # Build a sequential model
    model = Sequential()
    # Add the first layer
    model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu', input_shape=(3, 84, 84, 1), kernel_initializer=weights))
    # Add a second layer
    model.add(Convolution2D(64, (4,4), strides=(2,2), activation='relu', kernel_initializer=weights))
    # Add a third layer
    model.add(Convolution2D(64, (3,3), activation='relu', kernel_initializer=weights))
    # Flatten the three layers
    model.add(Flatten())
    # Add a dense layer
    model.add(Dense(512, activation='relu', kernel_initializer=weights))
    # Add another dense layer
    model.add(Dense(256, activation='relu', kernel_initializer=weights))
    # Add an output layer
    model.add(Dense(actions, activation='linear'))
    # Finally we return the model
    return model

```

Figure 6 RL Model

A summary of the model is shown below, we have a total of 5,024,174 parameters and 0 non-trainable parameters. The last layer has 14 nodes, this represents the number of actions available in the Atari game of Asteroids. We avoid the usage of pooling layers, as pooling layers can significantly alter the state of the game for our DQN agent.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 3, 20, 20, 32)     2080
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 3, 9, 9, 64)       32832
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 7, 7, 64)       36928
_________________________________________________________________
flatten (Flatten)            (None, 9408)              0
_________________________________________________________________
dense (Dense)                (None, 512)               4817408
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_2 (Dense)              (None, 14)                3598
=================================================================
Total params: 5,024,174
Trainable params: 5,024,174
Non-trainable params: 0
_________________________________________________________________
```

Figure 7 Model Summary

The Q learning update applied to the weights

Q learning has the ability to find the optimal policy by maximizing the expected value of the total reward. We used a deep Q network (DQN) learning agent that is supplied by the Keras-RL library. As a result, we were able to create a DQN agent in just a couple of lines.

```
# Here we inialize the Atari processor
processor = AtariProcessor()
# Here we build the agent using the model and actions
def build_agent(model, actions):
    # Here we create the policy for our agent
    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1., value_min=.1, value_test=.2, nb_steps=10000)
    # This is the memory for the agent
    memory = SequentialMemory(limit=50000, window_length=3)
    # Here we create the DQN agent using things like the model, processor and policy
    dqn = DQNAgent(model=model, processor=processor, memory=memory, policy=policy,
                   enable_dueling_network=True, dueling_type='avg',
                    nb_actions=actions, nb_steps_warmup=1000
                   )
    # Return the DQN agent
    return dqn
```

Figure 8 DQN Agent

The policy we used for our agent is the Eps greedy Q policy which balances exploitation and exploration which is supplied by the Keras-RL library also. The policy uses the Keras-RL Linear

Annealed Policy to decay epsilon which stands for exploration vs exploitation as the agent steps forward in the world. The value_max value of 1 represents the value of we want epsilon to start with and the value_min means to go no smaller than 0.1.

Memory is provided by the Keras-RL library again, which is a data structure that allows us to store the agent's experiences quickly and efficiently. The memory limit size must be specified and is also known as a hyperparameter. As new experiences are added to memory, it fills up and old ones fade away.

The DQN agent is built then using the model, processor for processing the images, the memory, and the policy. Furthermore, the nb_steps_warmup parameter determines how long to wait before starting to conduct experience replay, this is when we start training the network. The warmup parameter allows us to get sufficient experience to create a decent minibatch.

```
1 # Here we build the DQN agent
2 dqn = build_agent(model, actions)
3 # We compile the agent using Adam as the optimizer
4 dqn.compile(Adam(learning_rate=0.0001))
```

Figure 9 Building the agent

We then build our agent using the model and number of actions. Furthermore, we compile the DQN agent using the Adam optimizer and a learning rate of 0.0001. A learning rate that is low may allow the model to acquire a more optimum set of weights, but training will take much longer. A higher learning rate helps the model to learn more quickly, but at the cost of a sub-optimal final set of weights. We decided to go for a lower learning rate as we cropped and resized the images in the preprocessing which greatly increased the learning rate, so we allowed for a more accurate yet time consuming learning rate (Brownlee, 2019).

When it came to our DQN agent, the last thing we were required to do was to fit the game environment to our DQN agent. This is where the training took place. We fitted the model with 10000 training steps and a verbose 1 to see the progress. As you can see, training the model for 10000 steps took 38 minutes. We got an average reward of 0.6590.

```
1 # Finally, we fit out game enviroment to the DQN agent
2 dqn.fit(env, nb_steps=10000, visualize=False, verbose=1)
```

```
Training for 10000 steps ...
Interval 1 (0 steps performed)
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/en
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied automatically.
10000/10000 [==============================] - 2284s 228ms/step - reward: 0.6590
done, took 2283.794 seconds
<tensorflow.python.keras.callbacks.History at 0x7f901b599dd0>
```

Figure 10 Fitting DQN agent

Once training had finished, we could then test our DQN agent. We would test the agent for 10 games and print the result for each episode and the average score at the end. Here we can see the average reward for each episode is much higher compared to random actions. The average reward is also much higher 1227 compared to 710 for the random actions.

```
1 # Here we test our DQN agent
2 scores = dqn.test(env, nb_episodes=10, visualize=False)
3 print(np.mean(scores.history['episode_reward']))
```

```
Testing for 10 episodes ...
Episode 1: reward: 1440.000, steps: 1853
Episode 2: reward: 1320.000, steps: 2175
Episode 3: reward: 2000.000, steps: 3653
Episode 4: reward: 1460.000, steps: 1778
Episode 5: reward: 1080.000, steps: 1272
Episode 6: reward: 780.000, steps: 862
Episode 7: reward: 830.000, steps: 762
Episode 8: reward: 580.000, steps: 1025
Episode 9: reward: 1650.000, steps: 2018
Episode 10: reward: 1130.000, steps: 1488
1227.0
```

Figure 11 DQN testing

Other concepts

Our model uses HeNormal weight initializing function. This method is similar to the Xavier initialization but instead, the factor is multiplied by two. In He-normal function, the weights are initialized keeping in mind the size of the layer before. This helps to keep the global minimum of the cost function faster and more effective. The weights in this are also random and differ in

range depending on the size of the previous layer. This in return provides a controlled initialization and a faster gradient descent (Kakaraparthi, 2018). Using HeNormal initialization can speed up training considerably and it is one of the tricks of the trade that led to the success of Deep Learning.

```
# We intialize the HeNormal weight intializier
weights = tf.keras.initializers.HeNormal()
# Build a sequential model
model = Sequential()
# Add the first layer
model.add(Convolution2D(32, (8,8), strides=(4,4), activation='relu', input_shape=(3, 84, 84, 1), kernel_initializer=weights))
```

Figure 12 Model weights

Results

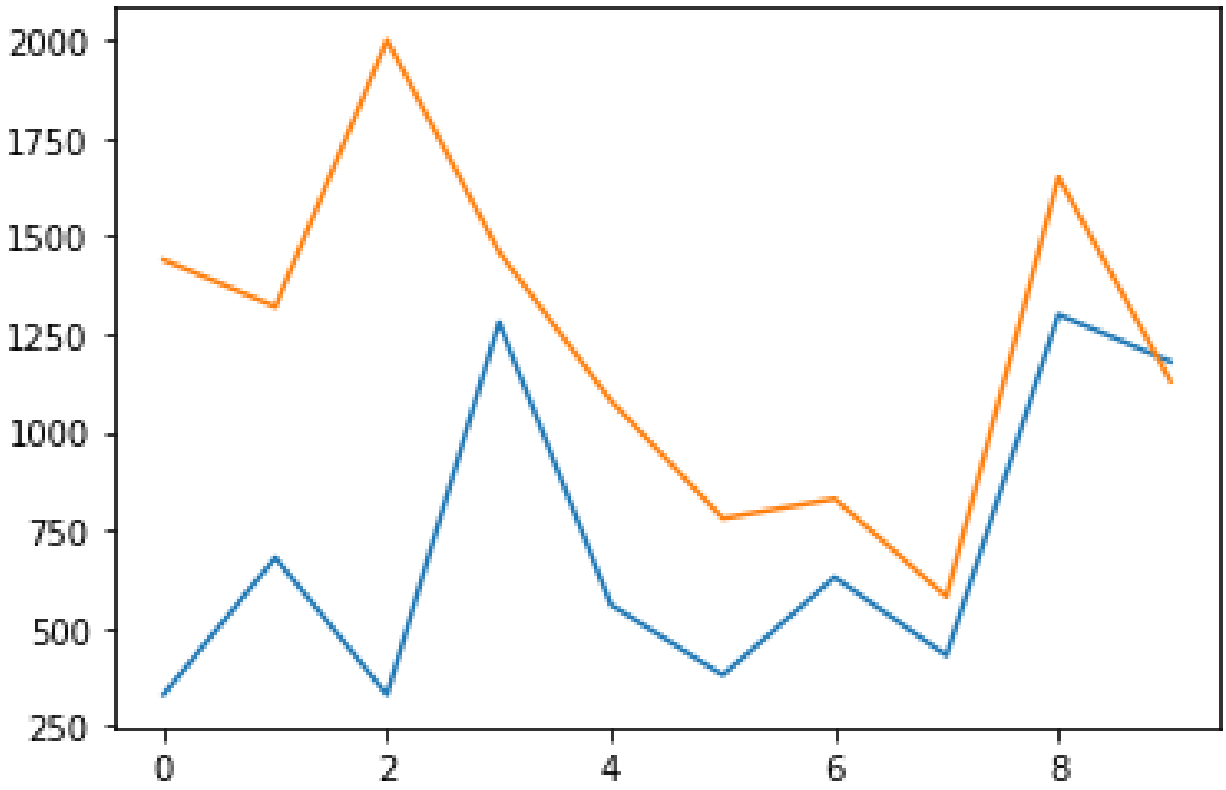

Figure 13 DQN Results

The plot here displays the scores of our trained model in orange; compared to randomly selecting actions in blue. On the X axis, the number of episodes is displayed, we tested both for 10 episodes. On the Y axis, the game reward is displayed. It is clear that the trained model achieves much higher scores and a much higher score on average.

Evaluation of the results

How does one evaluate the performance of the RL agent?

To effectively evaluate our agent's performance playing our Atari game we will compare each episode and the points scored by the agent to the previous run. This way we can visually represent the progress of the agent in this environment.

We will also compare the results of a model that has been untrained and compare its results to our trained model. This should give us a comparison of how our model has progressed since its first episode. On the left is the random actions, with an average reward of 710. We can see here that the trained model achieves much higher peak rewards and a higher reward on average being at 1227. We can say our model is much better than random actions.

```
Episode:1 Score:330.0
Episode:2 Score:680.0
Episode:3 Score:330.0
Episode:4 Score:1280.0
Episode:5 Score:560.0
Episode:6 Score:380.0
Episode:7 Score:630.0
Episode:8 Score:430.0
Episode:9 Score:1300.0
Episode:10 Score:1180.0
710.0
```

Figure 14 Random action rewards

```
Testing for 10 episodes ...
Episode 1: reward: 1440.000, steps: 1853
Episode 2: reward: 1320.000, steps: 2175
Episode 3: reward: 2000.000, steps: 3653
Episode 4: reward: 1460.000, steps: 1778
Episode 5: reward: 1080.000, steps: 1272
Episode 6: reward: 780.000, steps: 862
Episode 7: reward: 830.000, steps: 762
Episode 8: reward: 580.000, steps: 1025
Episode 9: reward: 1650.000, steps: 2018
Episode 10: reward: 1130.000, steps: 1488
1227.0
```

Figure 15 Trained model rewards

Is the agent learning?

The agent is learning as we can see the performance of our trained model above constantly achieving a higher score compared to random actions. This shows that the agent has learnt from previous episodes. One issue to highlight however is that the model we ran was for around 10,000 steps which only resulted in a minor improvement. We then ran the model for 50,000 steps and it also showed minor improvements to the model. To see drastic improvements in the model, we would have to train the model for 1 to 10 million steps to see a vast improvement. This would take hundreds of hours to train on our personal computers as 10000 steps takes around 40 minutes, this was not feasible for the time constraints on this project.

References

wikipedia. 2021. *Asteroids (video game)*. [online] Available at: <https://en.wikipedia.org/wiki/Asteroids_(video_game)> [Accessed 11 December 2021].

gym.openai. 2021. *Gym*. [online] Available at: <https://gym.openai.com/> [Accessed 11 December 2021].

Francis, J., 2021. *Introduction to reinforcement learning and OpenAI Gym*. [online] OReilly. Available at: <https://www.oreilly.com/radar/introduction-to-reinforcement-learning-and-openai-gym/> [Accessed 11 December 2021].

Kakaraparthi, V., 2018. *Xavier and He Normal (He-et-al) Initialization*. [online] medium.com. Available at: <https://prateekvishnu.medium.com/xavier-and-he-normal-he-et-al-initialization-8e3d7a087528> [Accessed 13 December 2021].

Brownlee, J., 2019. *How to Configure the Learning Rate When Training Deep Learning Neural Networks*. [online] machinelearningmastery. Available at: <https://machinelearningmastery.com/learning-rate-for-deep-learning-neural-networks/> [Accessed 14 December 2021].