MACHINE LEARNING AND BIOINSPIRED OPTIMIZATION

ASSIGNMENT 2

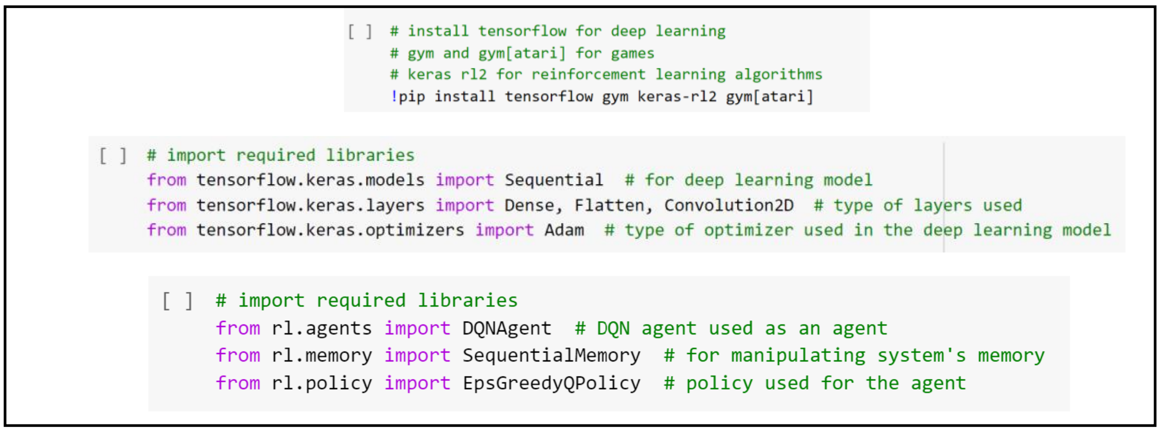
DEEP REINFORCEMENT LEARNING

Group Members	Email	Student ID
Huy Pham	huycam1997@gmail.com	201534475
Pamela Mercado	p.mercado@liverpool.ac.uk	201309392
Utsav Dihingia	u.dihingia@liverpool.ac.uk	201599488
Saeth Wannasuphoprasit	s.wannasuphoprasit@liverpool.ac.uk	201585689

Date: 27th April 2022

Atlantis

Step 1: Importing an OpenAI Gym game



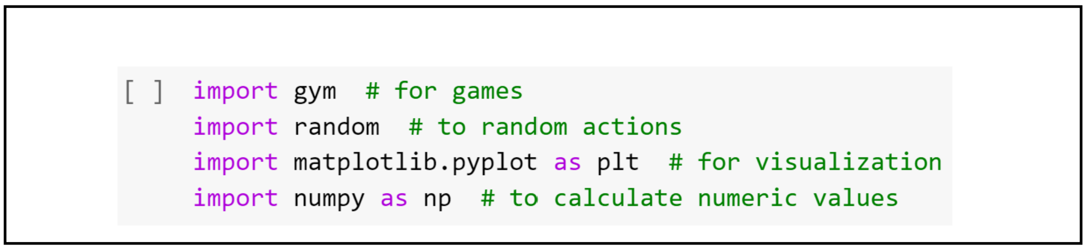
```
[ ] # install tensorflow for deep learning
    # gym and gym[atari] for games
    # keras rl2 for reinforcement learning algorithms
    !pip install tensorflow gym keras-rl2 gym[atari]
```

```
[ ] # import required libraries
    from tensorflow.keras.models import Sequential  # for deep learning model
    from tensorflow.keras.layers import Dense, Flatten, Convolution2D  # type of layers used
    from tensorflow.keras.optimizers import Adam  # type of optimizer used in the deep learning model
```

```
[ ] # import required libraries
    from rl.agents import DQNAgent  # DQN agent used as an agent
    from rl.memory import SequentialMemory  # for manipulating system's memory
    from rl.policy import EpsGreedyQPolicy  # policy used for the agent
```

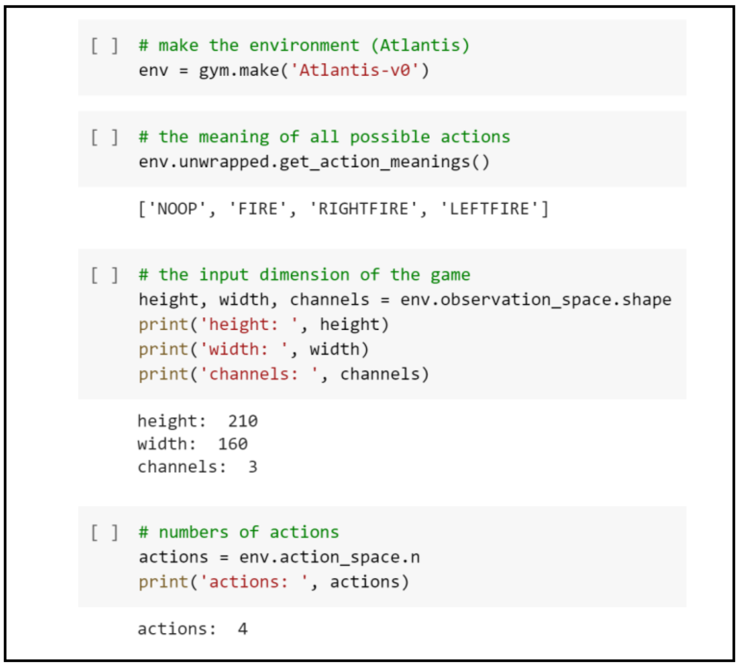
Figure 1. Install required libraries

As shown in Figure 1, we installed the libraries required to install the Atari game, build the deep learning model and the agent.



```
[ ] import gym  # for games
    import random  # to random actions
    import matplotlib.pyplot as plt  # for visualization
    import numpy as np  # to calculate numeric values
```

Figure 2. Import required libraries

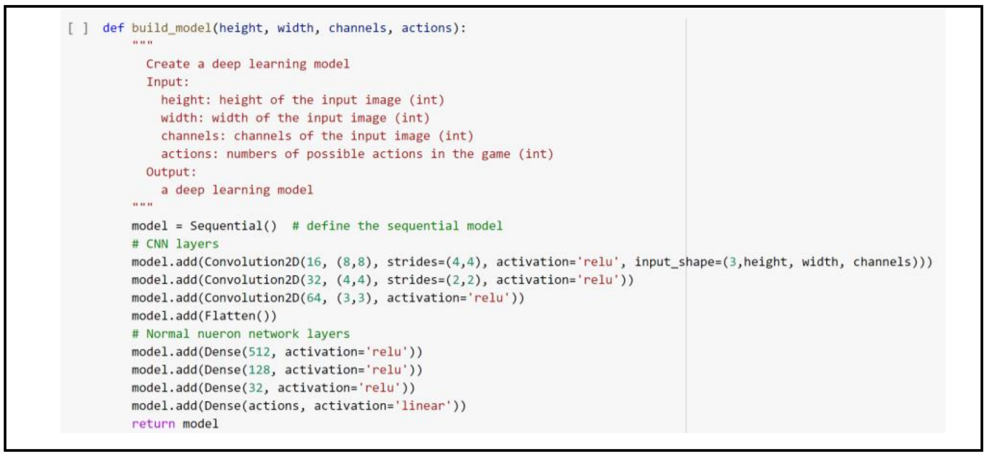


```
[ ] # make the environment (Atlantis)
    env = gym.make('Atlantis-v0')

[ ] # the meaning of all possible actions
    env.unwrapped.get_action_meanings()

    ['NOOP', 'FIRE', 'RIGHTFIRE', 'LEFTFIRE']

[ ] # the input dimension of the game
    height, width, channels = env.observation_space.shape
    print('height: ', height)
    print('width: ', width)
    print('channels: ', channels)

    height:  210
    width:  160
    channels:  3

[ ] # numbers of actions
    actions = env.action_space.n
    print('actions: ', actions)

    actions:  4
```

Figure 3. Setting up Atlantis environment

Figure 3 shows the import of the Atlantis game.

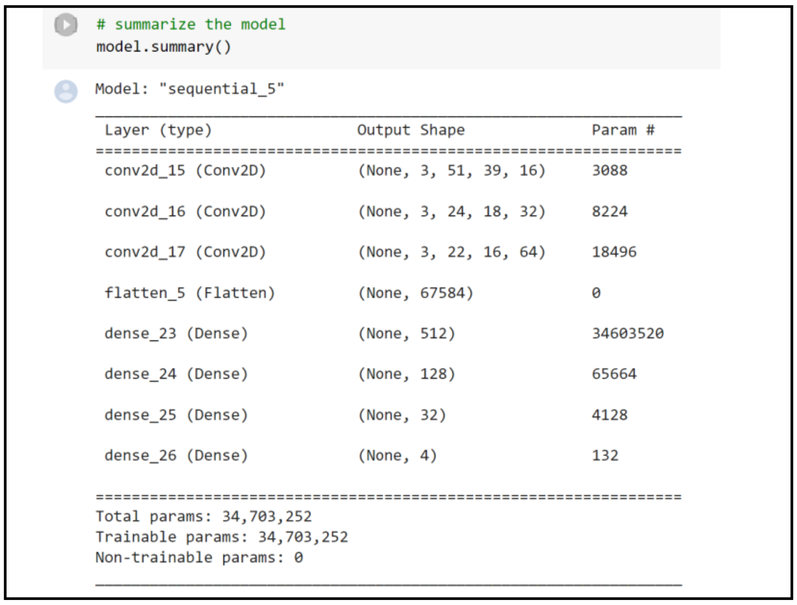
Step 2: Creating a network



```
[ ] def build_model(height, width, channels, actions):
        """
          Create a deep learning model
          Input:
            height: height of the input image (int)
            width: width of the input image (int)
            channels: channels of the input image (int)
            actions: numbers of possible actions in the game (int)
          Output:
            a deep learning model
        """
        model = Sequential()  # define the sequential model
        # CNN layers
        model.add(Convolution2D(16, (8,8), strides=(4,4), activation='relu', input_shape=(3,height, width, channels)))
        model.add(Convolution2D(32, (4,4), strides=(2,2), activation='relu'))
        model.add(Convolution2D(64, (3,3), activation='relu'))
        model.add(Flatten())
        # Normal nueron network layers
        model.add(Dense(512, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(32, activation='relu'))
        model.add(Dense(actions, activation='linear'))
        return model
```

Figure 4. Network created

Figure 4 shows the deep learning model created



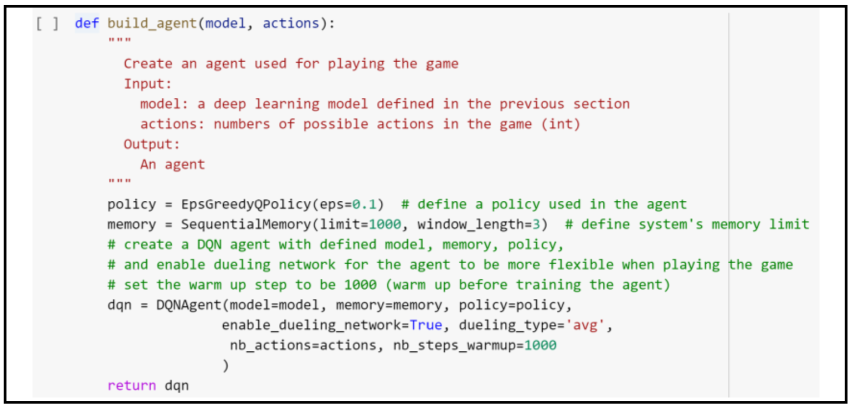
```
# summarize the model
model.summary()

Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_15 (Conv2D)          (None, 3, 51, 39, 16)     3088

 conv2d_16 (Conv2D)          (None, 3, 24, 18, 32)     8224

 conv2d_17 (Conv2D)          (None, 3, 22, 16, 64)     18496

 flatten_5 (Flatten)         (None, 67584)             0

 dense_23 (Dense)            (None, 512)               34603520

 dense_24 (Dense)            (None, 128)               65664

 dense_25 (Dense)            (None, 32)                4128

 dense_26 (Dense)            (None, 4)                 132

=================================================================
Total params: 34,703,252
Trainable params: 34,703,252
Non-trainable params: 0
_________________________________________________________________
```

Figure 5. Model summary

We chose the Atlantis game as the DQN agent performs better than the human level according to an article 'https://www.datahubbs.com/deepmind-dqn/.' We first played the game randomly for 10 episodes in order to compare the results (rewards) with the trained agent later. We then utilized the CNN model (Figure 4) as it is suitable for detecting features in images and used normal neuron networks for extracting all probability distribution of actions. The model inputs the height, width and channels of the input image, along with the number of possible actions in the game to create the deep learning model.

As shown in Figure 5, the total parameters are 34,703,252. We used 3 CNN layers and 4 normal neuron network layers with the relu activation function and the linear activation for the last neuron network layer.

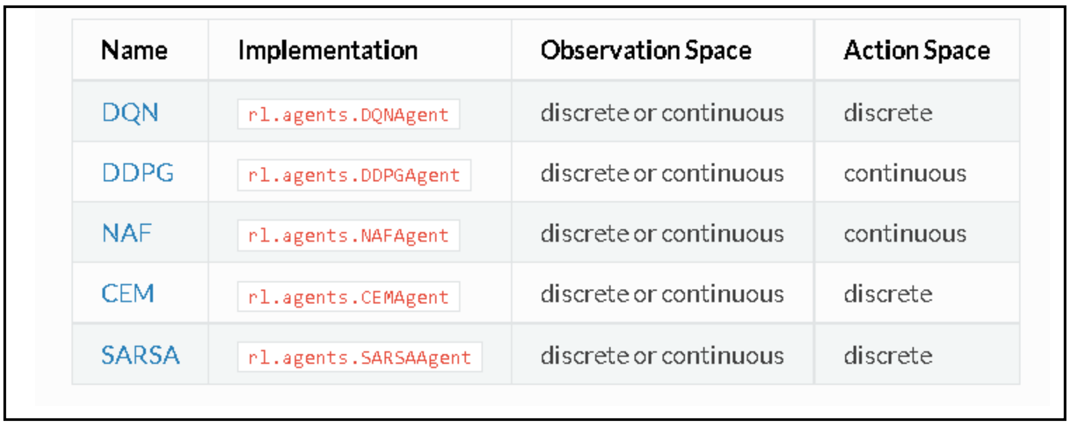
Step 3 and 4: Connecting the game to the network and implementing the deep reinforcement learning model



```
[ ]  def build_agent(model, actions):
         """
           Create an agent used for playing the game
           Input:
             model: a deep learning model defined in the previous section
             actions: numbers of possible actions in the game (int)
           Output:
             An agent
         """
         policy = EpsGreedyQPolicy(eps=0.1)  # define a policy used in the agent
         memory = SequentialMemory(limit=1000, window_length=3)  # define system's memory limit
         # create a DQN agent with defined model, memory, policy,
         # and enable dueling network for the agent to be more flexible when playing the game
         # set the warm up step to be 1000 (warm up before training the agent)
         dqn = DQNAgent(model=model, memory=memory, policy=policy,
                       enable_dueling_network=True, dueling_type='avg',
                        nb_actions=actions, nb_steps_warmup=1000
                       )
         return dqn
```

Figure 6. deep reinforcement learning agent

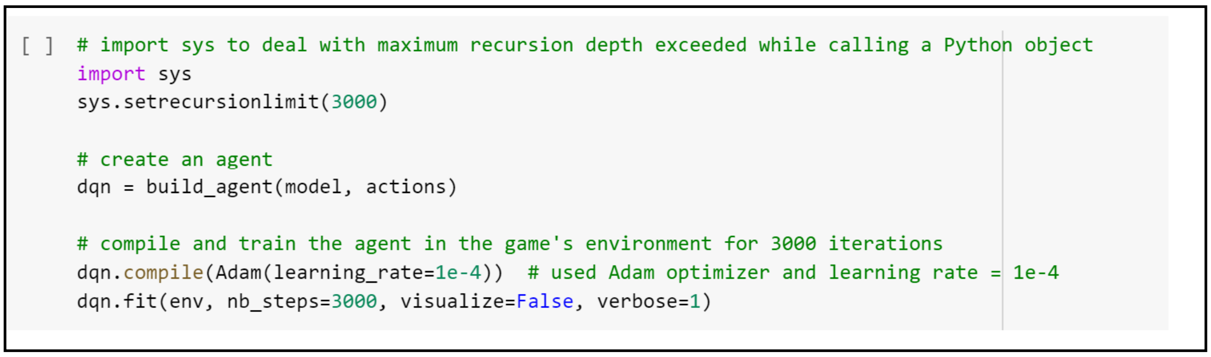
The next step was creating the agent (Figure 6) that will play the game, in which we used a DQN agent as our action space is discrete as shown in Figure 7, using the following reference 'https://keras-rl.readthedocs.io/en/latest/agents/overview/.'



Name	Implementation	Observation Space	Action Space
DQN	`rl.agents.DQNAgent`	discrete or continuous	discrete
DDPG	`rl.agents.DDPGAgent`	discrete or continuous	continuous
NAF	`rl.agents.NAFAgent`	discrete or continuous	continuous
CEM	`rl.agents.CEMAgent`	discrete or continuous	discrete
SARSA	`rl.agents.SARSAAgent`	discrete or continuous	discrete

Figure 7. Available Agents

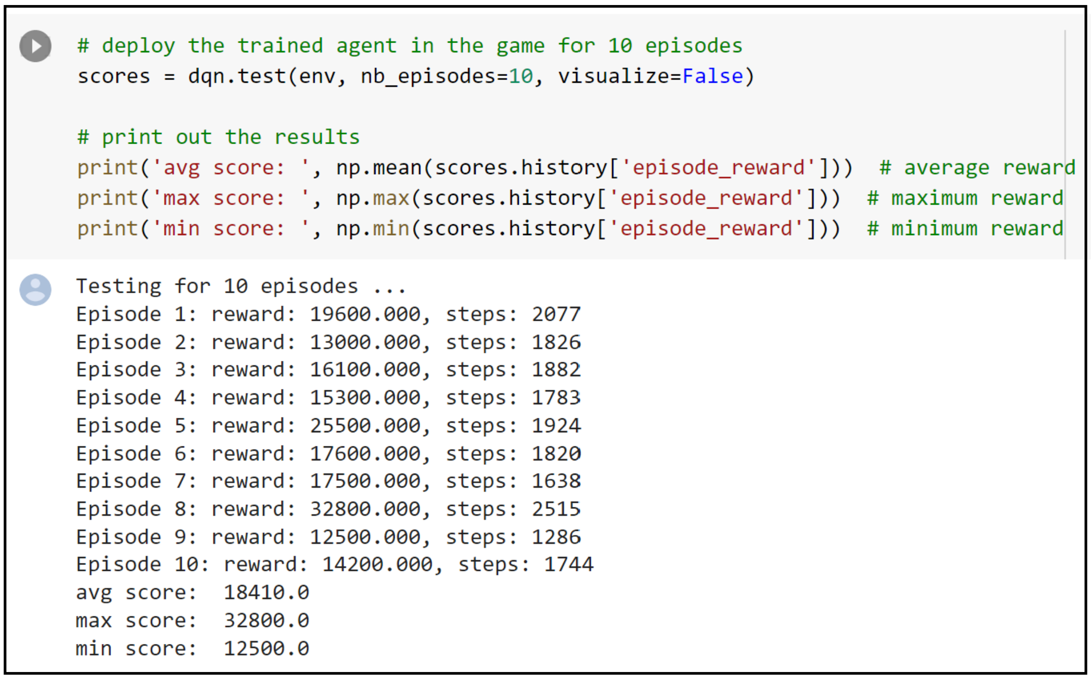
Since our model is dealing with images, Convolutional Neuron Networks is suitable to apply. The Deep Q-Learning (DQN) model was chosen to implement. DQN aims to approximate the Q function, which can be defined as the best policy given a set of Q values for state-action pairs. The network calculates Q-values for each state-action pair and fine-tunes weights in terms of loss, resulting in epsilon greedy phase performance that is facilitated. The main benefit of Deep Q-learning is that it allows Q-values to be determined in situations where creating a Q-table for each state-action pair is impractical. As shown in Figure 6, We used epsilon of 0.1, which means the agent spends 10% of the time for exploring and the rest for exploiting. The learning rate affects the rate at which the system moves toward the minimum error; if it is too high, it can cause the system to jump past the minimum. However, if it is too little, it may result in slow convergence. We chose 0.0001 as the learning rate, the Adaptive Movement Estimation (ADAM) as the optimizer and the RELU function as the activation function.



```
[ ]  # import sys to deal with maximum recursion depth exceeded while calling a Python object
     import sys
     sys.setrecursionlimit(3000)

     # create an agent
     dqn = build_agent(model, actions)

     # compile and train the agent in the game's environment for 3000 iterations
     dqn.compile(Adam(learning_rate=1e-4))  # used Adam optimizer and learning rate = 1e-4
     dqn.fit(env, nb_steps=3000, visualize=False, verbose=1)
```

Figure 8. Training the agent

The next step was training the agent (Figure 8) in which we implemented the deep reinforcement learning model and connected the network to the game by training the agent in the game environment for 3000 iterations as it is a sufficient training time without overfitting the model.



```
# deploy the trained agent in the game for 10 episodes
scores = dqn.test(env, nb_episodes=10, visualize=False)

# print out the results
print('avg score: ', np.mean(scores.history['episode_reward']))  # average reward
print('max score: ', np.max(scores.history['episode_reward']))  # maximum reward
print('min score: ', np.min(scores.history['episode_reward']))  # minimum reward
```

```
Testing for 10 episodes ...
Episode 1: reward: 19600.000, steps: 2077
Episode 2: reward: 13000.000, steps: 1826
Episode 3: reward: 16100.000, steps: 1882
Episode 4: reward: 15300.000, steps: 1783
Episode 5: reward: 25500.000, steps: 1924
Episode 6: reward: 17600.000, steps: 1820
Episode 7: reward: 17500.000, steps: 1638
Episode 8: reward: 32800.000, steps: 2515
Episode 9: reward: 12500.000, steps: 1286
Episode 10: reward: 14200.000, steps: 1744
avg score:  18410.0
max score:  32800.0
min score:  12500.0
```

Figure 9. Testing the agent in the game

Finally, we tested the trained agent for 10 episodes and calculated the maximum, minimum and average scores of those 10 episodes.

Step 4: Results and Discussion

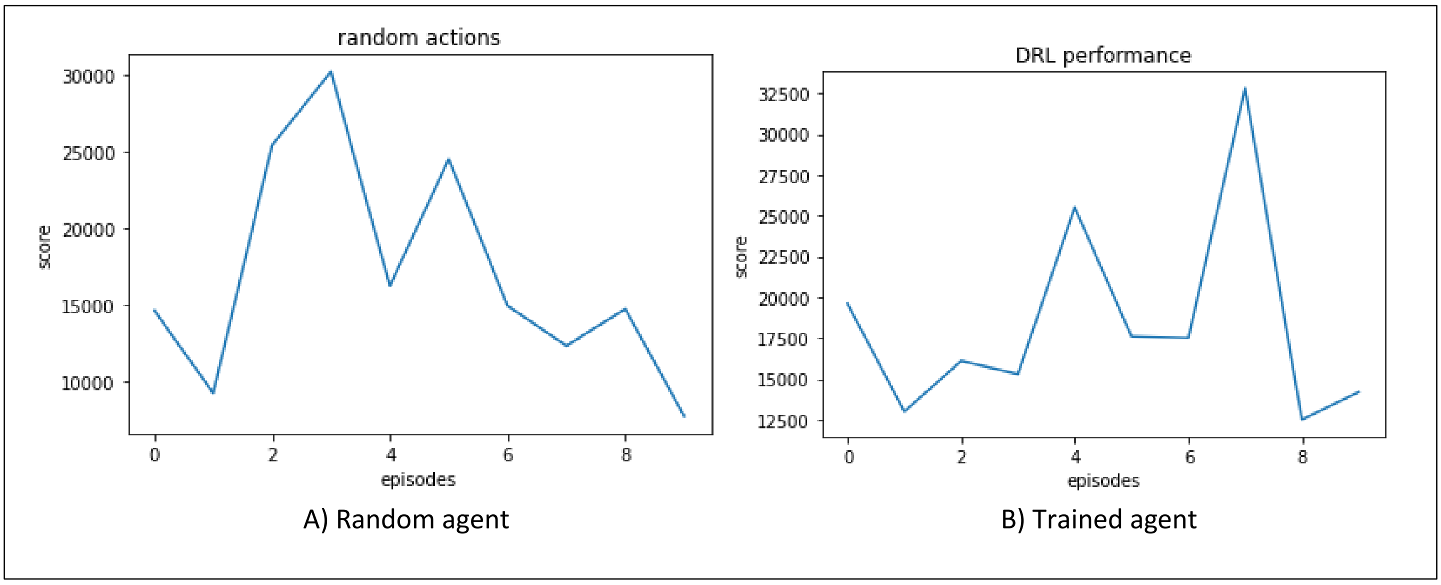


Figure 10: Random agent vs Trained agent performance graph

We can see from the above Figure 10, the difference in performance between an agent with random actions and a trained agent (we used 10 episodes for comparison). The random agent takes random actions without caring about maximizing the reward it receives on the other hand the trained agent aims for maximizing its reward which results in a higher average. Moreover, a trained agent tries to find a pattern and extract features from the game to find a path where it gets more positive rewards. Unlike random agent it doesn't take actions randomly and chooses what to do next only through its policies.

In Figure 10 we can also see that in some episodes the random agent performs scores higher than the trained agent, we must note that it is possible for an agent to have a positive return, but still not finishing the task it is required to solve. So, we need to check how many episodes the agent can solve or return a higher score. To find this we need to have a look at the average scores for both the agents.

Agent	Average Score	Max Score	Min Score
Random	16970.0	30200.0	7700.0
Trained	18410.0	32800.0	12500.0

Table 1: Score comparison for random and trained agent

From Table 1, we can see the trained agent has a higher average, max and min score which is a good sign that the agent is learning.

Future Improvements:

- We could have experimented with different agents and find out more optimal agents.
- Train the agent for more duration for better performance. But we should be careful not to overfit the model.
- Trying different values of epsilon, to find out its optimal value. We should also note that too much exploration is not good neither is not exploring enough.
- Use a different deep learning model and then compare performances to evaluate which gives better results.