Jahan Ulhaque
ID: 201272455
A2 – Deep reinforcement learning

3 Deep Reinforcement Learning

OpenAI Gym CartPole - DQN

Checklist: All complete: 7/7 And Extra

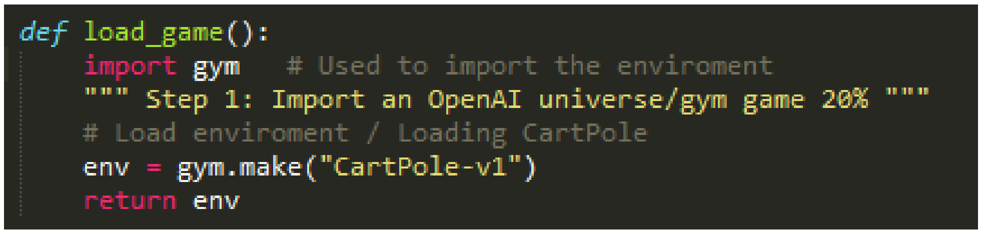
- ✓ Step 1: Import an OpenAI universe/gym game 20%
- ✓ Step 2: Creating a network 20%
- ✓ Step 3: Connection of the game to the network 10%
- ✓ Step 4: Deep reinforcement learning model 30%
- ✓ Step 5: Experimental results 20%
- o How to run:

Please read the "README" file contained within the folder.

Step 1: Import an OpenAI universe/gym game 20%

I have used OpenAI Gym and the game "CartPole-v1". To do this, I have cloned the GitHub repository created by OpenAI Gym into my Python 3.5 virtual environment (as this is the latest version they support), which allows me to use their toolkit for developing reinforcement learning algorithms.

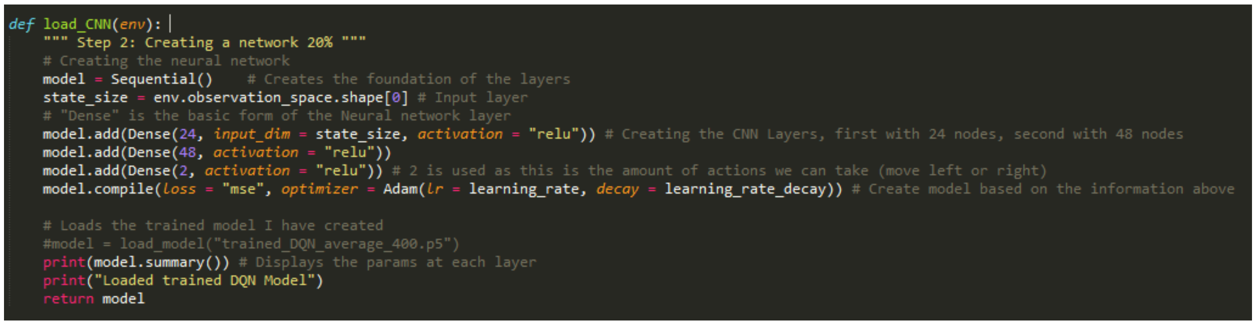
The following code explains how to import and load a game:



```
def load_game():
    import gym   # Used to import the enviroment
    """ Step 1: Import an OpenAI universe/gym game 20% """
    # Load enviroment / Loading CartPole
    env = gym.make("CartPole-v1")
    return env
```

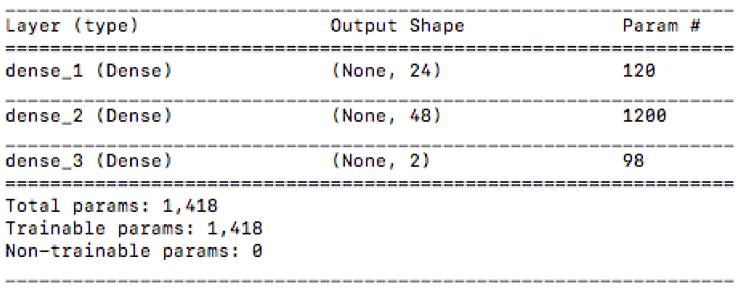
Step 2: Creating a network 20%

I also implemented a deep neural network via Keras, as shown here:



```
def load_CNN(env):
    """ Step 2: Creating a network 20% """
    # Creating the neural network
    model = Sequential()    # Creates the foundation of the layers
    state_size = env.observation_space.shape[0] # Input layer
    # "Dense" is the basic form of the Neural network layer
    model.add(Dense(24, input_dim = state_size, activation = "relu")) # Creating the CNN Layers, first with 24 nodes, second with 48 nodes
    model.add(Dense(48, activation = "relu"))
    model.add(Dense(2, activation = "relu")) # 2 is used as this is the amount of actions we can take (move left or right)
    model.compile(loss = "mse", optimizer = Adam(lr = learning_rate, decay = learning_rate_decay)) # Create model based on the information above

    # Loads the trained model I have created
    #model = load_model("trained_DQN_average_400.p5")
    print(model.summary()) # Displays the params at each layer
    print("Loaded trained DQN Model")
    return model
```

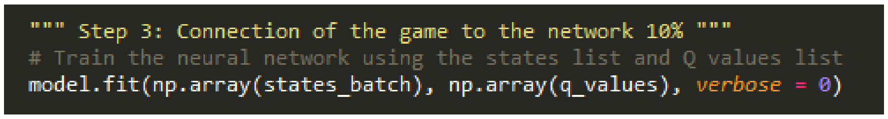
The image below shows a summary of the neural network model, further I have 4 input layers, which is the “state_size” (which I further explain in step 3) and the two CNN layers, and the parameters associated with them. We lastly have 2 output layers, as that is the final amount of actions our agent can decide. In CartPole, the agent can pick to either go left or right, and option has a higher predicted value will be chosen as this can give the agent the highest reward. (I have comments in my program explaining this more).



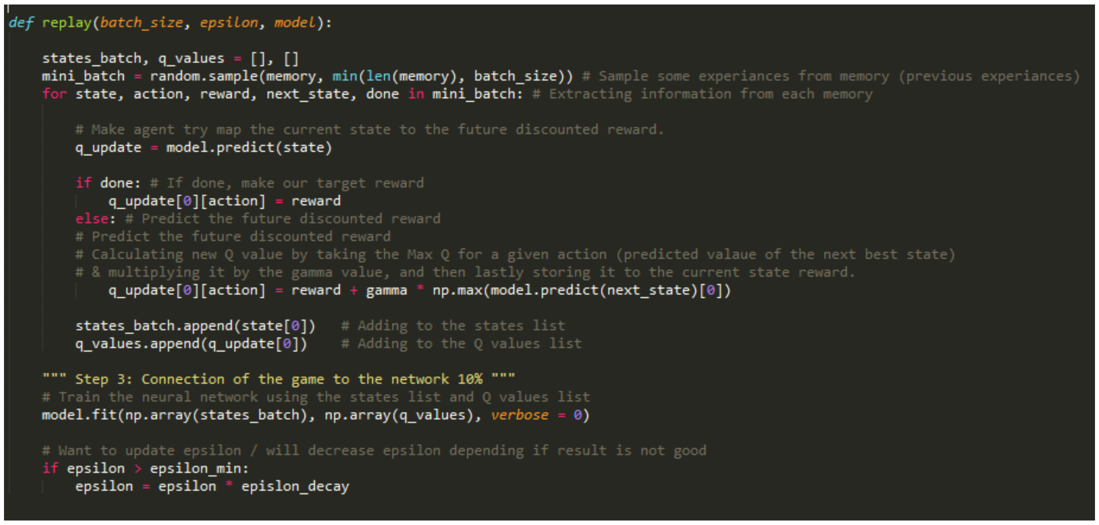
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 24)                120
_________________________________________________________________
dense_2 (Dense)              (None, 48)                1200
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 98
=================================================================
Total params: 1,418
Trainable params: 1,418
Non-trainable params: 0
_________________________________________________________________
```

- ## Step 3: Connection of the game to the network 10%

Once we have created the neural network model, we must connect the DQN reinforcement algorithm to the neural network so that it can learn, remember and improve. The main part of code that does this is:



```
""" Step 3: Connection of the game to the network 10% """
# Train the neural network using the states list and Q values list
model.fit(np.array(states_batch), np.array(q_values), verbose = 0)
```

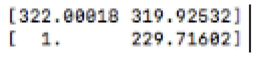


```
def replay(batch_size, epsilon, model):

    states_batch, q_values = [], []
    mini_batch = random.sample(memory, min(len(memory), batch_size)) # Sample some experiances from memory (previous experiances)
    for state, action, reward, next_state, done in mini_batch: # Extracting information from each memory

        # Make agent try map the current state to the future discounted reward.
        q_update = model.predict(state)

        if done: # If done, make our target reward
            q_update[0][action] = reward
        else: # Predict the future discounted reward
        # Predict the future discounted reward
        # Calculating new Q value by taking the Max Q for a given action (predicted valaue of the next best state)
        # & multiplying it by the gamma value, and then lastly storing it to the current state reward.
            q_update[0][action] = reward + gamma * np.max(model.predict(next_state)[0])

        states_batch.append(state[0])   # Adding to the states list
        q_values.append(q_update[0])    # Adding to the Q values list

    """ Step 3: Connection of the game to the network 10% """
    # Train the neural network using the states list and Q values list
    model.fit(np.array(states_batch), np.array(q_values), verbose = 0)

    # Want to update epsilon / will decrease epsilon depending if result is not good
    if epsilon > epsilon_min:
        epsilon = epsilon * epislon_decay
```

To further expand, we are fitting the model to the two arrays called “states_batch” and “q_values”.
“States_batch” – Stores the states of the agent. As shown here:

```
State: [-0.05952681  0.14573108  0.0508777  -0.06561154]
State: [-0.04524687 -0.73761605 -0.19875357 -0.24806196]
```

This is storing what the environment is observing, this includes the following:
Cart position, cart velocity, pole angle, pole velocity at tip.
These values are all taken into account when deciding which is the most optimal action to do (which will give the agent the most reward).
”q_values” – stores the actions the agent can pick from:

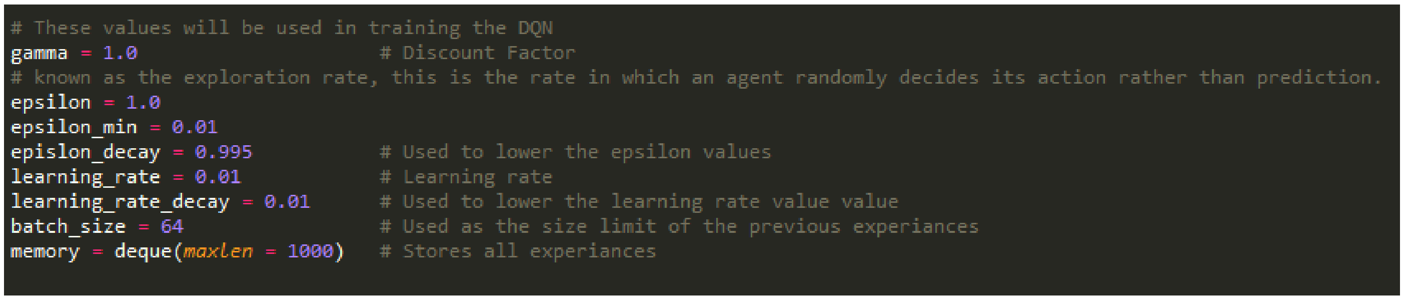


```
[322.00018 319.92532]
[  1.      229.71602]
```

For example, we have two actions, to turn left or right. And the DQN wants to survive as long as it can to achieve the most reward, so based of the observations we have collected, we pick the action that will help the agent survive as long as it can. In this instance, we can image that the cartpole is turning to the right, and the value 1 is the action that telling the agent to turn to the left. The reason this is done is that the agent can gain more reward, and thus is decreasing the gap between prediction and the target loss (the loss is a value representing how far the gap between the agent's prediction and the actual target).

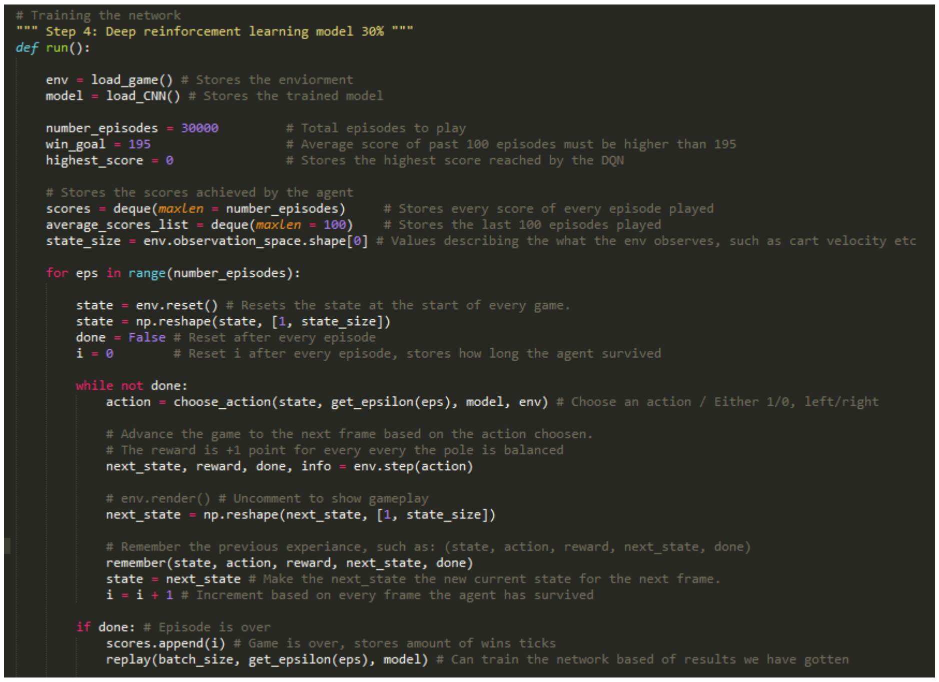
○ Step 4: Deep reinforcement learning model 30%

The network architecture I used was "Deep Q-network" (DQN). For DQN I use a range of hyper parameters, such as:



```python
# These values will be used in training the DQN
gamma = 1.0                     # Discount Factor
# known as the exploration rate, this is the rate in which an agent randomly decides its action rather than prediction.
epsilon = 1.0
epsilon_min = 0.01
epislon_decay = 0.995           # Used to lower the epsilon values
learning_rate = 0.01            # Learning rate
learning_rate_decay = 0.01      # Used to lower the learning rate value value
batch_size = 64                 # Used as the size limit of the previous experiances
memory = deque(maxlen = 1000)   # Stores all experiances
```

I have already explained parts of how the "replay" function works, as we can see from step 3, we take a sample batch of past experiences (which contain the state, action, reward, next predicted state, and done). For the agent to perform well, we consider immediate rewards as well as long term rewards. To calculate this, we have used the "gamma" (discount factor) value. This allows the agent to learn and maximize the discounted future rewards based on the current state.



```python
# Training the network
""" Step 4: Deep reinforcement learning model 30% """
def run():

    env = load_game() # Stores the enviorment
    model = load_CNN() # Stores the trained model

    number_episodes = 30000         # Total episodes to play
    win_goal = 195                  # Average score of past 100 episodes must be higher than 195
    highest_score = 0               # Stores the highest score reached by the DQN

    # Stores the scores achieved by the agent
    scores = deque(maxlen = number_episodes)     # Stores every score of every episode played
    average_scores_list = deque(maxlen = 100)    # Stores the last 100 episodes played
    state_size = env.observation_space.shape[0] # Values describing the what the env observes, such as cart velocity etc

    for eps in range(number_episodes):

        state = env.reset() # Resets the state at the start of every game.
        state = np.reshape(state, [1, state_size])
        done = False # Reset after every episode
        i = 0        # Reset i after every episode, stores how long the agent survived

        while not done:
            action = choose_action(state, get_epsilon(eps), model, env) # Choose an action / Either 1/0, left/right

            # Advance the game to the next frame based on the action choosen.
            # The reward is +1 point for every every the pole is balanced
            next_state, reward, done, info = env.step(action)

            # env.render() # Uncomment to show gameplay
            next_state = np.reshape(next_state, [1, state_size])

            # Remember the previous experiance, such as: (state, action, reward, next_state, done)
            remember(state, action, reward, next_state, done)
            state = next_state # Make the next_state the new current state for the next frame.
            i = i + 1 # Increment based on every frame the agent has survived

        if done: # Episode is over
            scores.append(i) # Game is over, stores amount of wins ticks
            replay(batch_size, get_epsilon(eps), model) # Can train the network based of results we have gotten
```

This is how the agent trains. After loading the model and environment. We loop through the number of episodes, and we reset the environment (since it's a new game) and we store the states in a 1-dimensional array.

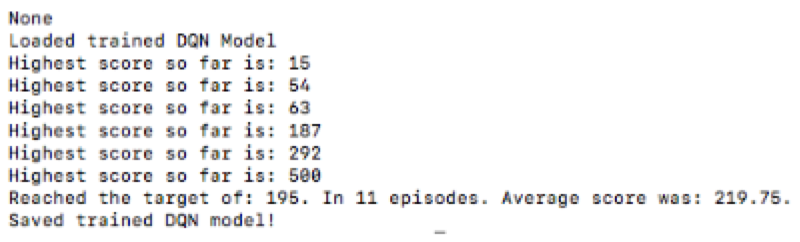
While the episode has not finished, we allow the agent to choose an action, based on the current state and epsilon value. We then advanced the game to the next state, and for every second the cartpole is balanced we gain +1 reward.

We then store the states every experience the agent has, and then move onto the next state. Once the game is over, we use the replay function as discussed above. And this is how the agent uses DQN architecture and CNN to learn, remember and play the game.

o STEP 5: EXPERIMENTAL RESULTS 20%

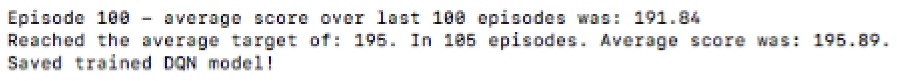
I have attached a video displaying the agent abilities.

I also have 2 different models which can be loaded, one model is when the agent was trained to just achieve an average of 195, and another model where the agent was trained to achieve an average of 400. When applying the better model to the original task (achieving an average of 195) the new improved agent can do it in a lot faster time, as shown here:



```
None
Loaded trained DQN Model
Highest score so far is: 15
Highest score so far is: 54
Highest score so far is: 63
Highest score so far is: 187
Highest score so far is: 292
Highest score so far is: 500
Reached the target of: 195. In 11 episodes. Average score was: 219.75.
Saved trained DQN model!
```

Compare to the agent trained to just achieve an average:



```
Episode 100 - average score over last 100 episodes was: 191.84
Reached the average target of: 195. In 105 episodes. Average score was: 195.89.
Saved trained DQN model!
```

It is worth noting that results can vary a lot, sometimes the agent can finish in 30 episodes and others it can take 100+. However, the trained agent who averages 400 can finish it a lot faster every time, as expected.

Cart Pole is considered solved after getting an average of 195 over 100 consecutive episodes. Because the trained model can do it under 100 episodes, I felt it would be unfair to divide the result by 100 to work out the average, so for the first 100, the average is worked out by the number of episodes played. So if the agent solved the game in 20 episodes, the calculation would be the “total score” / 20.