Network Architecture Description:

The learning algorithms described in the attached code are implementations of the Deep Q Network and Dueling Deep Q Networks. The Deep Q Network implementation includes 3 fully connected layers that take in input ray-based perception of objects and the network is able to output a chosen action. The Dueling Deep Q Network involves taking in input data and running it through a few fully connected layers and then splitting into a value and advantage stream. Finally, these stream outputs are used to update final predicted Q values. Both networks utilize a local and target network in order to stabilize and improve training.

Learning Algorithm Description:

The learning process utilizes 3 main objects: a UnityEnvironment, an Agent (with a ReplayBuffer) , and a DQNTrainer. The environment is initialized and is then passed actions predicted by either Deep Q Networks or Dueling Deep Q Networks and provides feedback of observations, rewards, and states.

The Agent initializes the Q Networks for evaluation and training, an optimizer to fine-tune the networks and a memory buffer, from which it will randomly choose a subset of experiences and learn from its past decisions and subsequent rewards. After every update_frequency timesteps, the Agent initializes an update step from which it updates the local network according to the optimizer and triggers a soft update of the target network. The agent also is able to "act" from which it evaluates the action values from the outputs of the local Q Network and makes a decision randomly with probability epsilon or a decision based on largest action value with probability 1- epsilon.

The DQNTrainer contains all the methods for the training processes. It runs a for loop for every episode from which it resets the environment and runs a for loop for every timestep. Here it evaluates predicted actions from the agent and pushes the action to the environment. From here it is able to get the data for the next states, rewards, observations and whether the training is done. If done, it is able to store the scores, update the epsilon utilizing the specified decay, and move on to the next episode. If the desired score of 13 is achieved, DQNTrainer initializes the saving of the successful network weights and creates and saved plots showing the training progress throughout the whole training process.

Hyperparameters:

Parameters were chosen based on what values these variables normally hold in other implementations and what intuition would suggest might be most optimal, but only some were fine-tuned based on training performance. The hyperparameters for agent creation shown below seemed to be around the optimal values for training performance.

```
def create_agent(state_size, action_size, buffer_size=int(1e5), batch_size=64,
gamma=0.99, tau=1e-3, lr=5e-4, update_frequency=4, duel=False):
```

The hyperparameter of highest interest in initializing the trainer was the epsilon decay parameter. Usually, this parameter is set to a value close to 1, as one would not want the agent to rely too heavily on past experiences and stop exploring too early. However, since this task may be somewhat more simplistic that previously thought, an epsilon decay of around 0.5 actually improved performance significantly!

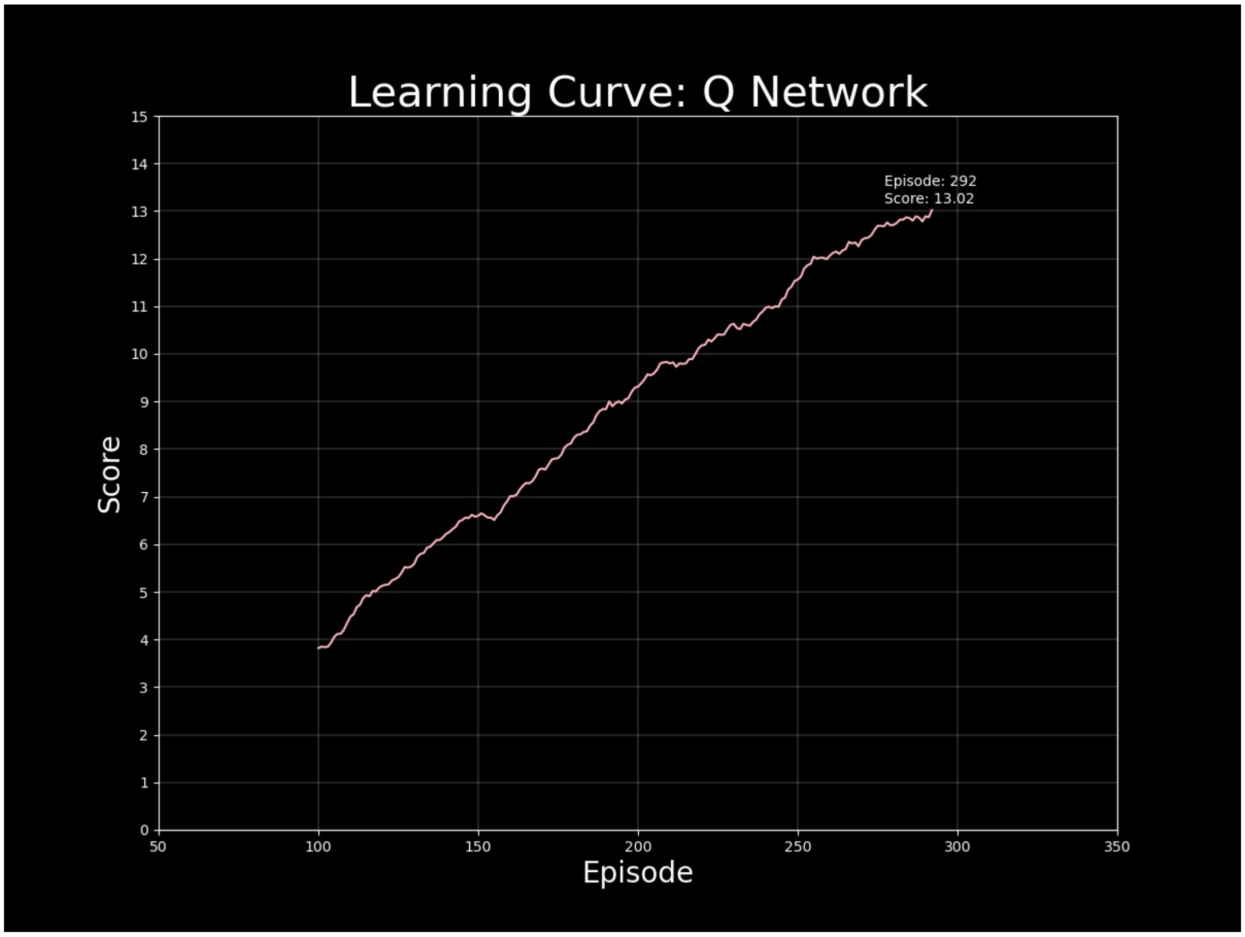
```
def create_trainer(agent, env, end_score, max_t=1000, eps_start=1.0,
eps_end=0.01, eps_decay=0.5, save_dir=r'./final_model'):
```

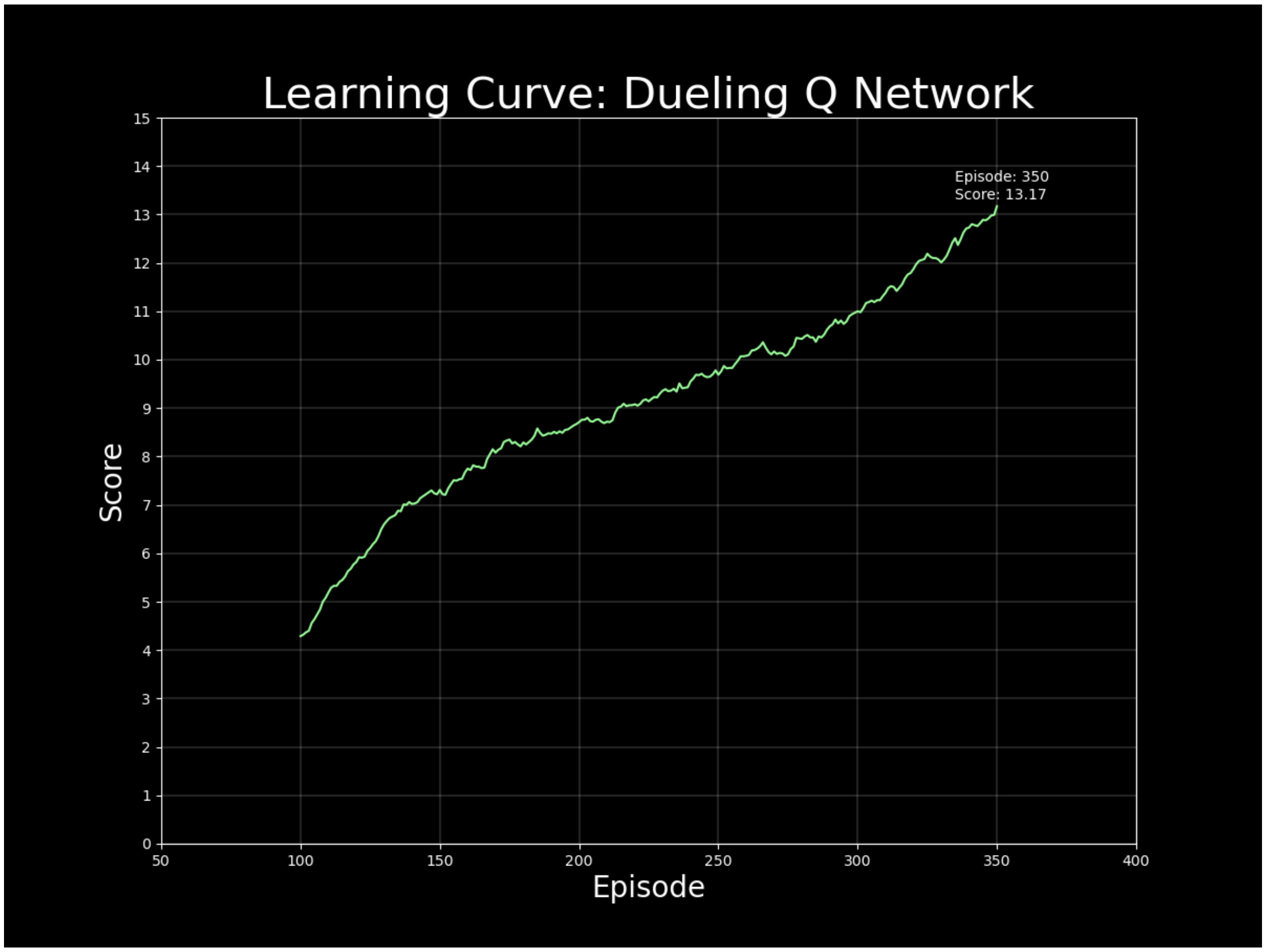
Future Ideas:

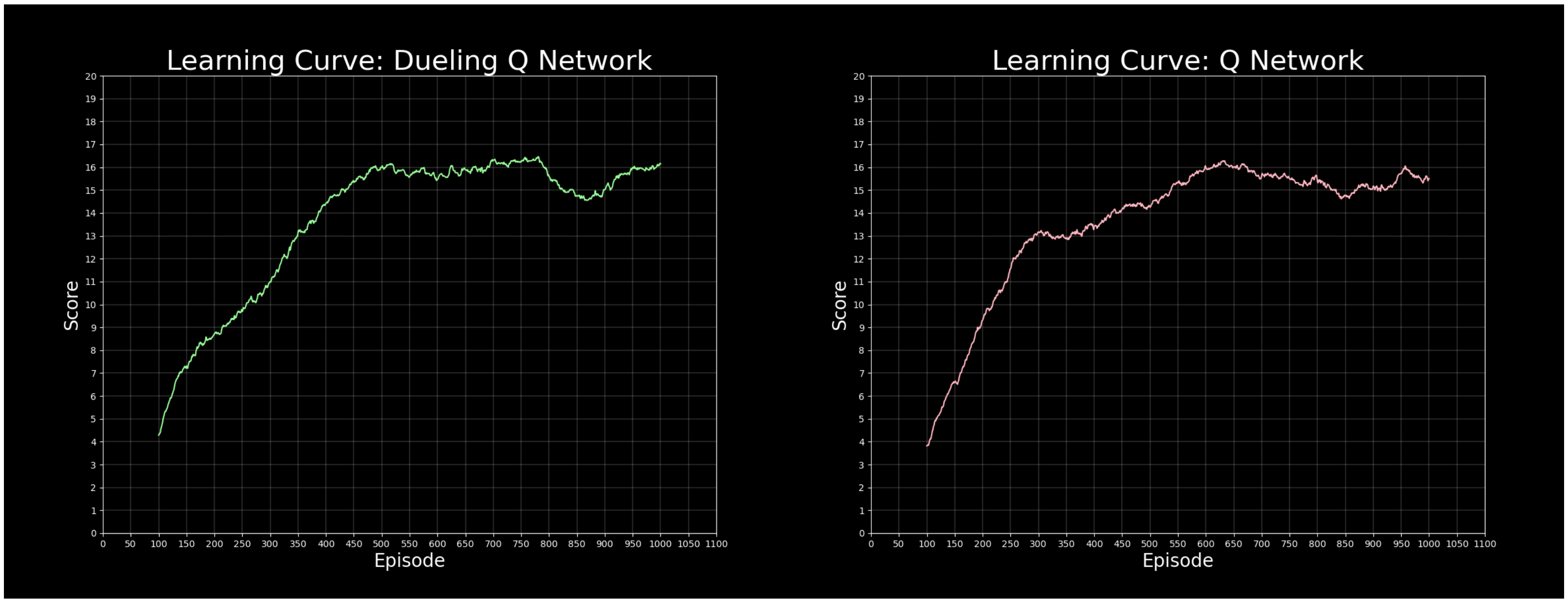
In order to improve the model performance further, I would like to fine-tune the hyperparameters to see if improvement can be made on performance based on the type of Q Networks used. Furthermore, different types of model architectures may improve performance even further. Perhaps a deeper network, more input/output nodes or different activations. Finally, training utilizing pixels would be something interesting to explore further to see if this alternative may improve the time or episode number needed to solve the environment.

Environment Solutions:

Below you may see the solutions of the Dueling Deep Q Network and Deep Q Network. The graphs below plot the average score over the past 100 episodes and start at x=100 (due to the fact that any number less would not have 100 values to average). The Deep Q Network solves the environment (achieves an average score of 13 over 100 episodes) in 292 episodes! The Dueling Q Network achieves this feat in 350 episodes. Interestingly, the regular Deep Q Network outperforms the Dueling Deep Q Network in order to solve the environment at a score of 13. If the training continues after the score of 13, however, the Dueling Deep Q Network then starts to outperform the regular Deep Q Network. At around 500 episodes the Dueling Deep Q Network achieves a score of around 16 while the Deep Q Network only achieves a score just above 14.




Learning Curve: Dueling Q Network
Episode: 350
Score: 13.17
Score
Episode


Learning Curve: Dueling Q Network
Score
Episode
Learning Curve: Q Network
Score
Episode

This repository contains the following files:

1. learn_banana.py: The main file which initializes the **env**ironment, **agent** and **trainer** and carries out the training process.

Functions include:

- *load_env*: Initializes UnityEnvironment (**env**) based on operating system and initializes 'state_size' and 'action_size' parameters.
- *create_agent*: Initializes **agent** object for training.
- *create_trainer*: Initializes **trainer** object for training agent in environment.
- *train_agent*: Trains agent in env with the **trainer.train** method.
- *restore_agent*: If desired, restores parameters for the local network of a **past successful agent**.

2. dqn_agent.py: Initializes methods and attributes of **Agent** and **ReplayBuffer** objects.

Agent methods include:

- *__init__*: Initializes relevant attributes, double Q networks (dueling or otherwise), optimizer and memory queue.
- *step*: Adds experiences to memory and triggers learn() method if applicable.
- *act*: Returns action based on highest reward with probability 1-eps or random action with probability eps.
- *learn*: Evaluates local model and triggers soft_update() of target model.
- *soft_update*: Performs a soft update on the target model based on local model.

ReplayBuffer methods include:

- *__init__*: Initializes relevant attributes for memory.
- *add*: Adds experiences to memory.
- *sample*: Randomly samples experiences for agent learning!

3. dqn_algo.py: Initializes methods and attributes of **DQNTrainer** object.

Methods include:

- *__init__*: Initializes relevant attributes for training.
- *run_episode*: Resets environment and carries out episode for max_t timesteps utilizing agent and env objects.
- *step_train*: Evokes run_episode() while keeping track of scores and updating epsilon.
- *train*: Evokes the training process while monitoring progress. Evokes save() and plt_rolling_avgs() when successful model is achieved.
- *save*: Saves successful local model weights.
- *plt_rolling_avgs*: Creates a plot for the rolling averages of scores over the past 100 episodes.

4. model.py: Initializes **QNetwork** and **DuelingQNetwork** architectures for mapping state to action values!

QNetwork architecture:

- *__init__*: Initializes network layers.
    - *fc1*: Fully connected layer with state_size unit input and 64 unit output.

fc*2*: Fully connected layer with 64 unit input and 64 unit output.
fc*3*: Fully connected layer with 64 unit input and action_size unit output
forward: Builds QNetwork that maps state to action values.

DuelingQNetwork architecture:
__*init*__: Initializes network layers.
feature_layer: Fully connected layer with state_size unit input and 64 unit output & another fully connected layer with 64 unit input and 64 unit output.
value_stream: Fully connected layer with 64 unit input and 64 unit output & another fully connected layer with 64 unit input and 1 value output.
advantage_stream: Fully connected layer with 64 unit input and 64 unit output & another fully connected layer with 64 unit input and action_size unit output.
forward: Builds DuelingQNetwork that maps state to action values.