Network Architecture Description:

The algorithm described in the attached code is an implementation of the Deep Deterministic Policy Gradient algorithm. This network consists of an Actor (which generates actions according to its perceived optimal policy) and a Critic (which evaluates the Actor's actions based on estimated value and suggests improvements to the Actor).

The Actor network consists of 3 Fully Connected layers. The first layer has an input of state_size which corresponds to the dimensions of states offered by the environment and its output of 400 units is activated by ReLU. The second layer has an input of 400 units, is also activated by a ReLU activation and has an output of 300 units, while the final layer has an input of 300 units, an output dimension corresponding to the action dimensions needed for the environment and is followed by a Tanh activation to force the predicted action values between -1 and 1.

The Critic network consists of a fully connected layer with input of state_size, output of 400 units and an activation of ReLU. Then the concatenation of state and action values in order to map them to Q values. Next, another fully connected layer with 400 units + action_size input and 300 unit output activated by ReLU and finally the third layer has an input of 400 units and an output of 1 and is returned.

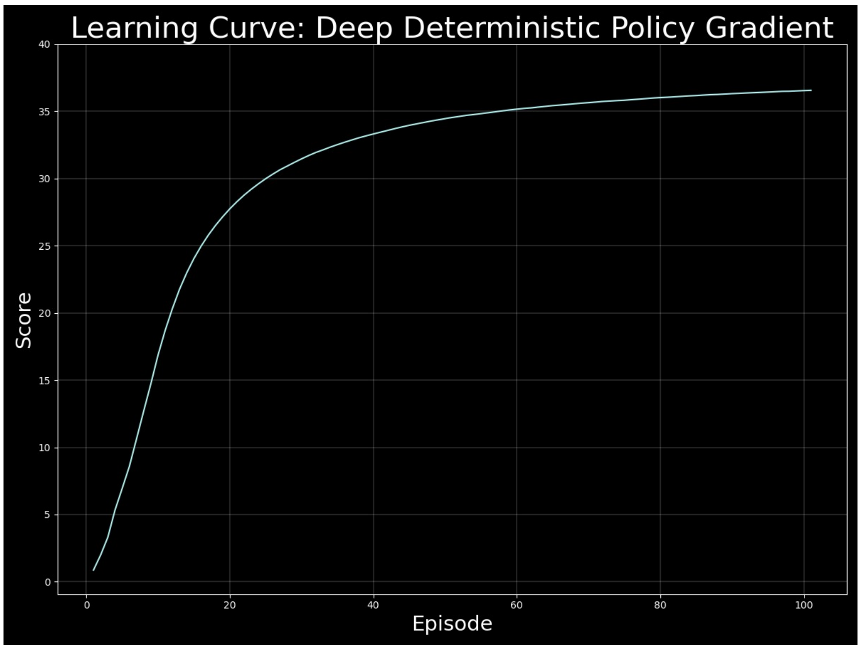
Learning Algorithm Description:

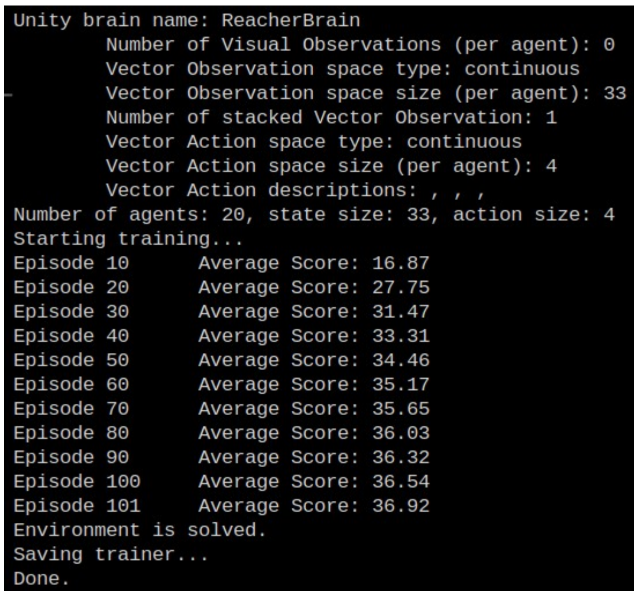
The training process consists of iterating over episodes until the average reward of 20 agents over 100 episodes reaches the value of +30. Action values consist of 4 numbers between -1 and 1 that correspond to the torque applied to two joints on each of the arms, hopefully forcing them into the designated location which offers a reward of +0.1 for every timestep inside the sphere.

Actions are generated by a model which tries to optimize policy (the Actor) and the action predictions are evaluated by a model trying to optimize the value function (the Critic). Each are instantiated with local and target networks in order to stabilize training.

After each timestep, the states, actions, rewards, next states and dones are stored in experiences in a ReplayBuffer object from which the local Actor and Critic networks utilize a random batch to learn after a predefined number of timesteps. The target networks are then updated by taking a percentage, Tau, of the local network weights and 1-Tau percentage of the target network weights. To reduce generalization bias, a GaussianNoise object adds noise to the action values prior to evaluation.

As experiences are gained, and the Actor and Critic networks optimize themselves jointly, the agents begin to enact better and better policies. Below you may see the learning curve graph for the average score of 20 agents for the current and all prior evaluated episodes. In the case below, the environment was solved well before the 100 episode evaluation window was even completed. It was solved within 30 episodes!





```
Unity brain name: ReacherBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 33
        Number of stacked Vector Observation: 1
        Vector Action space type: continuous
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
Number of agents: 20, state size: 33, action size: 4
Starting training...
Episode 10      Average Score: 16.87
Episode 20      Average Score: 27.75
Episode 30      Average Score: 31.47
Episode 40      Average Score: 33.31
Episode 50      Average Score: 34.46
Episode 60      Average Score: 35.17
Episode 70      Average Score: 35.65
Episode 80      Average Score: 36.03
Episode 90      Average Score: 36.32
Episode 100     Average Score: 36.54
Episode 101     Average Score: 36.92
Environment is solved.
Saving trainer...
Done.
```

Hyperparameters:

The most influential hyperparameters were initialized in the *create_agent* and *create_trainer* functions in the *run_reacher_main.py* file.

```
def create_agent(state_size, action_size, seed=0, actor_fc1_units=400,
actor_fc2_units=300, actor_lr=1e-4, critic_fc1_units=400,
critic_fc2_units=300, critic_lr=1e-4, weight_decay=0,
buffer_size=int(1e5), batch_size=128, gamma=0.99, tau=0.1,
noise_dev=0.3):
```

```
def create_trainer(env, agent, update_frequency=1, num_updates=5,
max_episode_length=1000, save_dir=r'./saved_files',
score_window_size=100):
```

All of the chosen hyperparameters above were chosen in order to optimize the training process and increase the score as soon as possible. Fc1 and fc2 units of 400 and 300 for both the Actor and Critic networks seemed to be ideal, as well as the learning rate of around 1e-4 for both. A weight_decay value of 0 seemed to also help the model perform most optimally. Gamma at 0.99 and a noise_dev of 0.3 also helped the models train optimally.

Unusually, a larger than expected Tau value of around 0.1, rather than a smaller value of 0.001, increased performance 7 fold after the first ten episodes. This means that we can learn from the local models much more quickly than initially expected. Furthermore, a num_updates of 5 rather than 1 doubled the initial training speed.

Future Ideas:

To further improve this project, I would like to be able to further evaluate different Neural Network architectures for the Actor and Critic Networks. I would like to further finetune the hyperparamters used and also attempt to utilize different types of Noise algorithms to see if it may improve model training performance!