Deep Reinforcement Learning – Continuous Control

Overview

The goal of this project is to achieve an average reward of at least +30 from all agents (20) over 100 consecutive episodes in the environment.

The environment is a double-jointed arm that can move to a target.

A reward of +0.1 is given at every step when the agent hand is in the target location. The goal of the agent is to maintain its position on the target for as long as possible.

The state space is a vector of 33 variables corresponding to the position, rotation, velocity and angular velocities of the arm.

The action space is a vector with four numbers between -1 and 1 corresponding to torque applied to two joints.

Implementation

The project was solved using Deep Reinforcement Learning, more specifically using Deep Deterministic Policy Gradient or DDPG. The code was based upon the ddpg-bipedal and ddpg-pendulum example from the Udacity Deep Reinforcement Learning GitHub repo. This was modified and updated to work with the Unity-ML environment and extended with new model architecture. Some of the code was separated out to modules, for example ReplayBuffer class in memory.py.

The Jupyter notebook results.ipynb contains the implementation for training the agent in the environment.

Agent.py contains a DDPG agent which interacts with the environment to optimize the reward.

Memory.py contains the Replay Buffer which is used by the Agent to record and sample a (state, action, reward, next_state) tuples for training the model.

Model.py contains the Actor and Critic Neural Network Class which takes in the input state and outputs the desired Q-values

Noise.py contains the OUNoise Class which contains the Ornstein-Uhlenback process which adds noise to actions.

Learning algorithm

The Deep Deterministic Policy Gradient algorithm (https://arxiv.org/abs/1509.02971) was chosen to solve the environment. It is an Actor-Critic method but could be called an approximate DQN. This is because it is different to normal Actor Critic algorithms in that the Critic is used to approximate the maximizer over the Q-Values of the next state and not a learnt baseline. It uses a Replay Buffer and is Off-Policy and Model-Free. At its core, DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn

Replay Buffer

One of the break throughs with the DQN architecture was the introduction of Experience Replay. This has been ported to the Actor-Critic method for the DDPG algorithm and works in the same way. It stores experiences in a Replay Buffer so that learning is separated and can take place on both the current experience and past experiences. This allows us to use Supervised Deep Learning techniques. This is done by randomly sampling the stored tuples in the Replay Buffer for training the model. The agent imports this class from memory.py as ReplayBuffer

Noise

OUNoise Class is used to add noise to actions to promote exploration. It uses the Ornstein-Uhlenback process to achieve this.

Model Architecture

Two Neural Network models are used in the DDPG algorithm. An Actor and a Critic network both with a Local and Target network each.

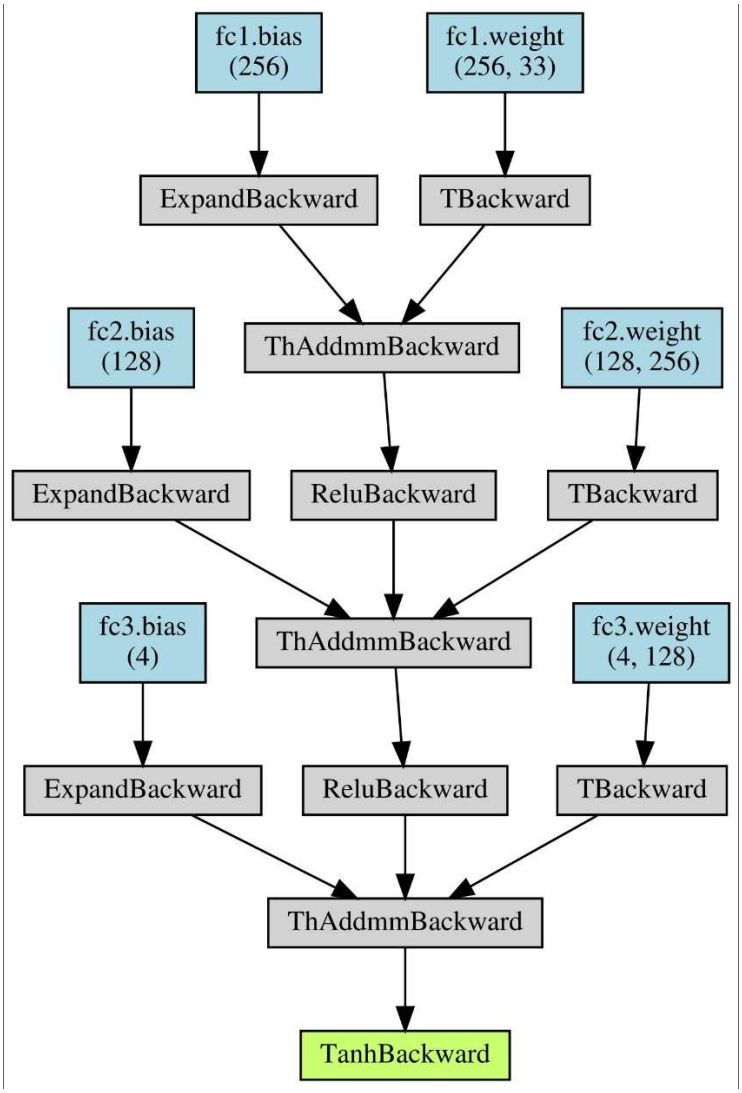
Actor Network

The Actor or Policy network maps states to actions

- The model has 3 fully connected layers
- The first layer takes in the state passes it through 256 nodes with Relu activation
- The second layer take the output from the first layer and passes through 128 nodes with Relu activation

- The third layer takes the output from the previous layer and outputs a single real value representing an action chosen from a continuous action space
- Adam optimizer is used.

The following diagram displays the Actor Network architecture



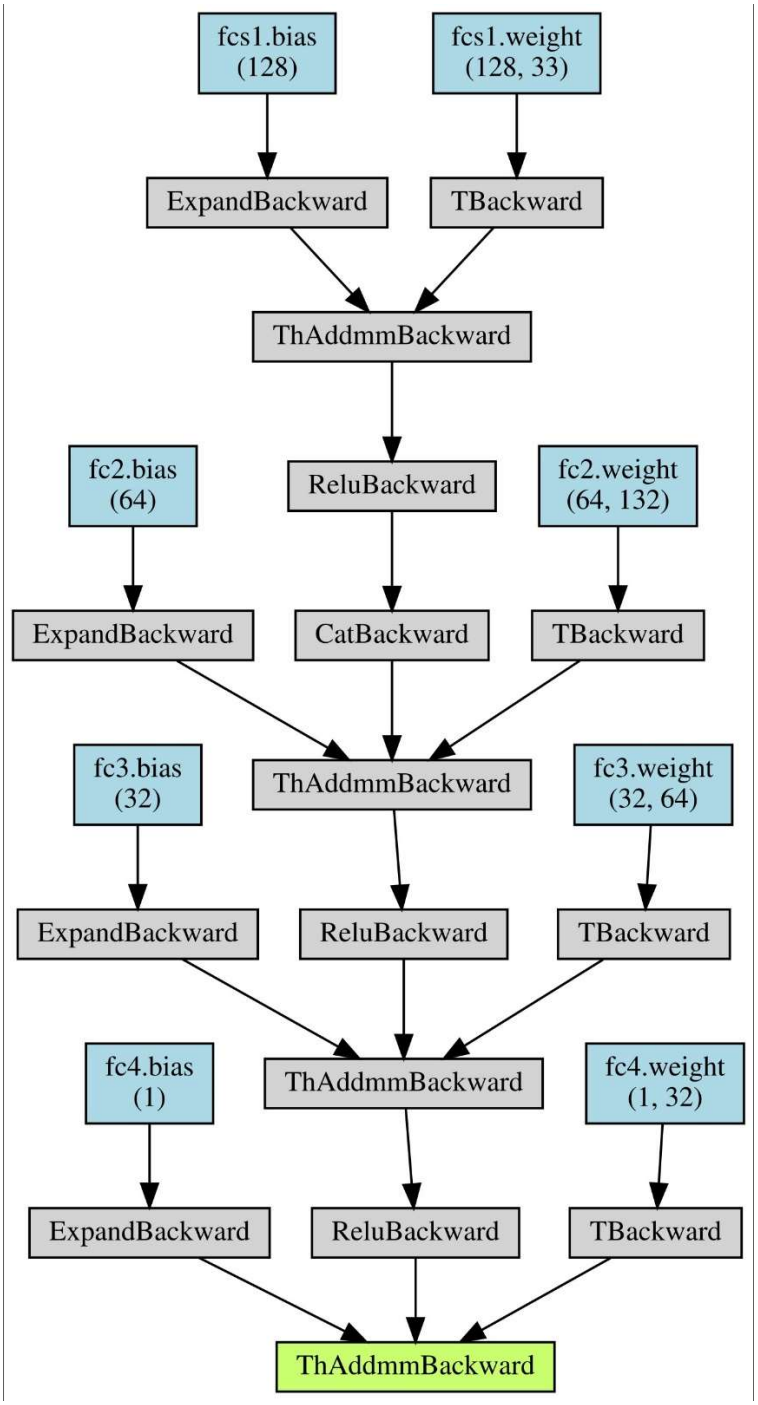
Critic Network

The Critic or Value network maps (state, action) pairs -> Q-values.

- The model has 4 fully connected layers
- The first layer takes the state and passes through 128 nodes with Relu activation
- Then we take the output from the first layer and concatenate it with the action size
- We then forward this to a second layer which passes it through 64 nodes with Relu activation

- The third layer takes the output from the previous layer and passes it through 32 nodes with Relu activation
- The Critic's output is estimated Q-value of the current state and of the action given by the actor
- Adam optimizer is used.

The following diagram displays the Critic Network architecture



Agent

The agent class in agent.py contains the DDPG algorithm. The algorithm interreacts with the environment to maximize reward. It has four networks, an Actor and Critic with both a Local and Target network which is imported from model.py. Replay memory is imported from memory.py and OUNoise is import from noise.py. It consists of five methods and hyperparameters. The methods are described below.

Step: Saves experiences in Replay memory. And uses random sample from Replay buffer to learn

Act: Returns actions for the given state as per current policy

Learn: Updates the policy and value parameter using given batch of experience tuples.

Q_target = r+y * Critic_target(next_state, Actor_target(next_state)where actor target(state) -> action and Critic_target(state, action) -> Q-values

The Critic network gets the predicted next state action and Q values from the target models. Then it computes the Q targets for the current state, computes the Critic loss and then minimizes that loss

The Actor computes the loss from the Critic and then minimizes the loss

Finally, it performs a soft update on the target networks

Reset: Resets the internal state (= noise) to mean (mu)

Soft_update: Performs a soft update of model parameters based on the TAU hyperparameter. The local model is updated based on TAU while the target model is updated based on 1.0 – TAU.

DDPG

The DDPG function in results.ipynb brings all these components together to train the network. It is basically a runner for the DDPG algorithm in the Agent class. The agent interacts with the environment by viewing the state and performing an action based on the policy from the DDPG. The rewards and next state are then sent back to the agent. The agent records a tuple (s,a,r,s') into replay memory. It then samples these to updates the Actor and Critic networks to optimize rewards. This process is done until the desired average score is achieved over all agents which is 30+, it then discontinues training and finishes the loop.

Hyperparameters

The following hyperparameters were used for training the agent.

BUFFER_SIZE = 1000000 (replay buffer size)

BATCH_SIZE = 1024 (minibatch size)

GAMMA = 0.99 (discount factor)

TAU = 0.001 (soft update of target parameters)

LR_ACTOR = 0.0001 (Actor learning rate)

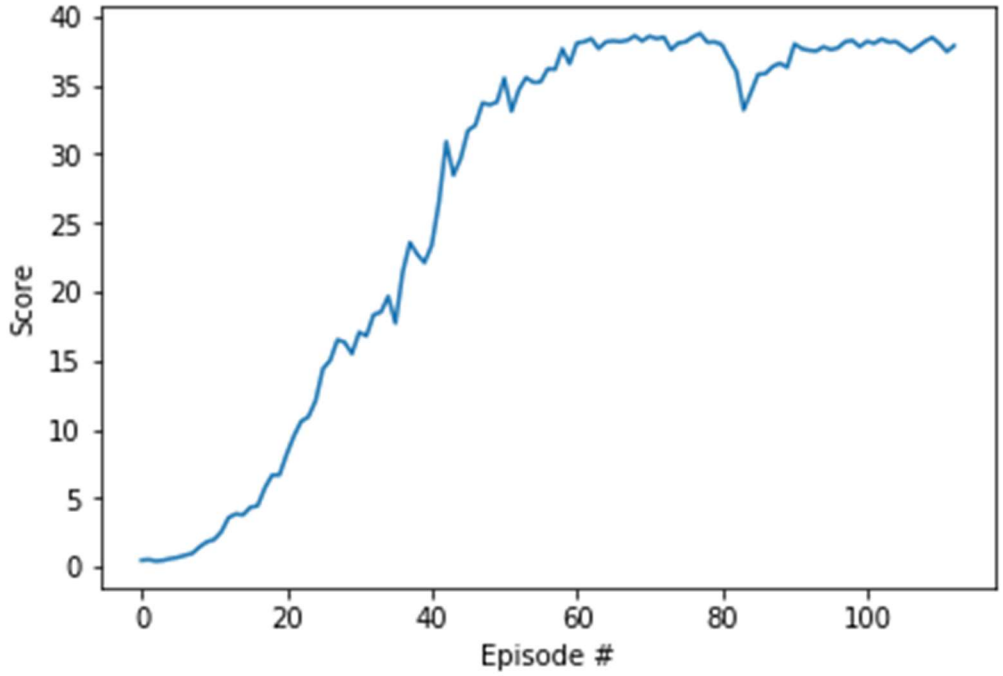
LR_CRITIC = 0.001(Critic learning rate)

WEIGHT_DECAY = 0.0 (L2 weight decay)

Results

The results from the Deep Deterministic Policy Gradient were incredible. The final Hyperparameters achieve fast and stable training.

Episode:113, Low Score:35.50, High Score:39.20, Score:37.86, Best Score:38.76, Average Score: 30.18, Best Avg Score:30.18
Average score of 30 achieved



Improvements

While the results achieved were great, there is always room for improvement. To try and achieve faster training or improve final scores I would implementing the following.

- Modifying the Hyperparameters. Changing the Hyperparameter could potentially speed up training or increase the final score
- Implement a different algorithm like PPO https://arxiv.org/pdf/1707.06347.pdf or D4PG https://openreview.net/pdf?id=SyZipzbCb
- Modifying the model architecture by changing the number of layers or neurons
- Change the update frequency of the networks in the step function

Conclusion

The project has given me a good understanding of the difficulties of a continuous control problem in Reinforcement Learning.

Training the single agent version of this environment was simple but the 20-agent brought new challenges.

One problem I faced early was that I could achieve the desired scores but not able to maintain it for long enough to average it over 100 episodes

Another problem was steady training that failed to achieve a high score

Changing the update rate to eg 2 or 4 increased stability but slowed down training. Finding a happy medium with this was hard so I set it to 1

In the end, I modified the noise and found a great model and hyperparameters that trained quickly, steady overall learning curve and average over all agents well over each episode.

My next challenge is to implement the PPO algorithm to this environment