# Report – Continuous Control

## Introduction

The project uses the Deep Deterministic Policy Gradient (DDPG) algorithm to train an agent to control a double-jointed arm to reach for target locations in Udacity's version of the Unity ML Reacher environment. There are 20 agents, and each is rewarded with +0.1 for every step that the arm's hand is in the target location. The goal is to achieve an average score of at least 30 over all agents and over 100 consecutive episodes.

The implementation is based on the ddpg-pendulum example in Udacity's Deep Reinforcement Learning Nanodegree program.

## Environment

20 agents (double-jointed arms) reach for moving target locations (transparent spheres). Agents are rewarded with +0.1 for every step that the hand is in the target location.

### States

Each agent observes 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm.

### Actions

The action space consists of four continuous variables for the torque at the two joints.

### Helper functions

The implementation uses a helper function `open_brain_surgery` to reshape outputs from the environment. The function unpacks the next state, rewards, and episode termination information.

## Learning Algorithm

The implementation uses a version of the Deep Deterministic Policy Gradient algorithm, an extension of Deep Q-Learning to continuous state and action spaces. The algorithm uses two neural networks (each with a local and a target network): an Actor Network to map states to actions, and a Critic Network to estimate Q-values for different state-action pairs.

### Agent Class

The DDPG agent is implemented in the `Agent` class. In addition to the usual inputs (the size of the state and action spaces and a random seed), the number of hidden

layers (at least one) and their neurons for the actor and the critic networks can be passed in lists to the `hidden_sizes_actor` and `hidden_sizes_critic` parameters.

## Hyperparameters

The hyperparameters for training in all settings were set to the following values:

```
BUFFER_SIZE = int(1e5)

BATCH_SIZE = 128

GAMMA = 0.99

TAU = 1e-3

LR_ACTOR = 1e-4

LR_CRITIC = 1e-3

WEIGHT_DECAY_AC = 0

WEIGHT_DECAY_CR = 0
```

## Model Architectures

The local and the target networks for both the actor and the critic are initialized with the same random weights. The Adam optimizer is used for the learning rates in both networks, and soft-updates are used to update the values of the target networks. To stabilize learning, the critic also uses gradient clipping.

The actor networks (depicted below) determines the policy to map states to actions and consist of three fully connected layers. There are two hidden layers with 64 and 32 neurons, respectively, using RELU activation, and a final layer that maps outputs from the hidden layers to the action space using the hyperbolic tangent function.

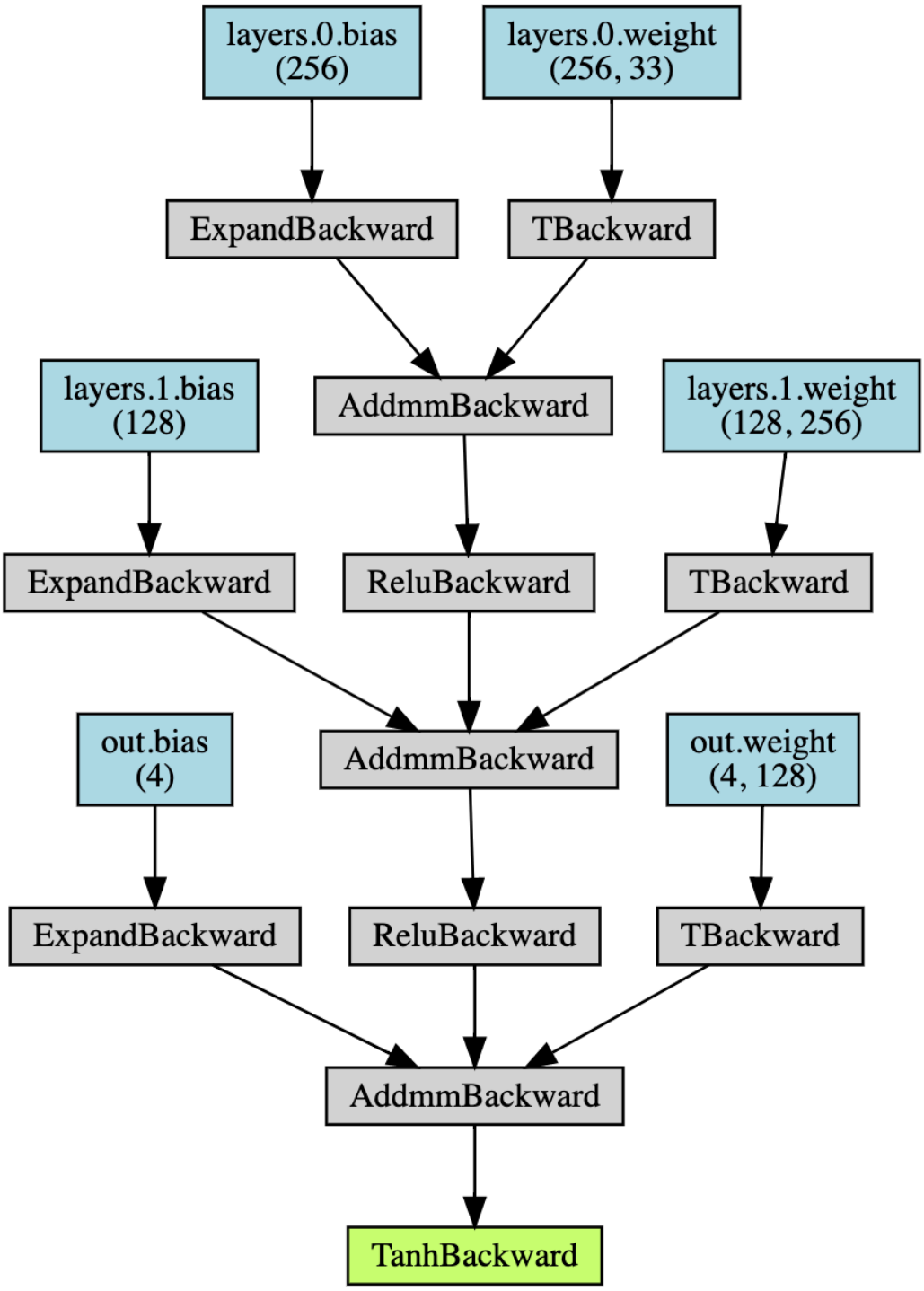


**Figure 1: Actor Network**

The critic networks (depicted below) estimate Q-values for different state-action pairs and consist of four fully connected layers. There are three hidden layers with 128, 64 and 32 neurons, respectively, using RELU activation, and a final layer that maps outputs from the hidden layers to scaled Q-values, using the hyperbolic tangent function.

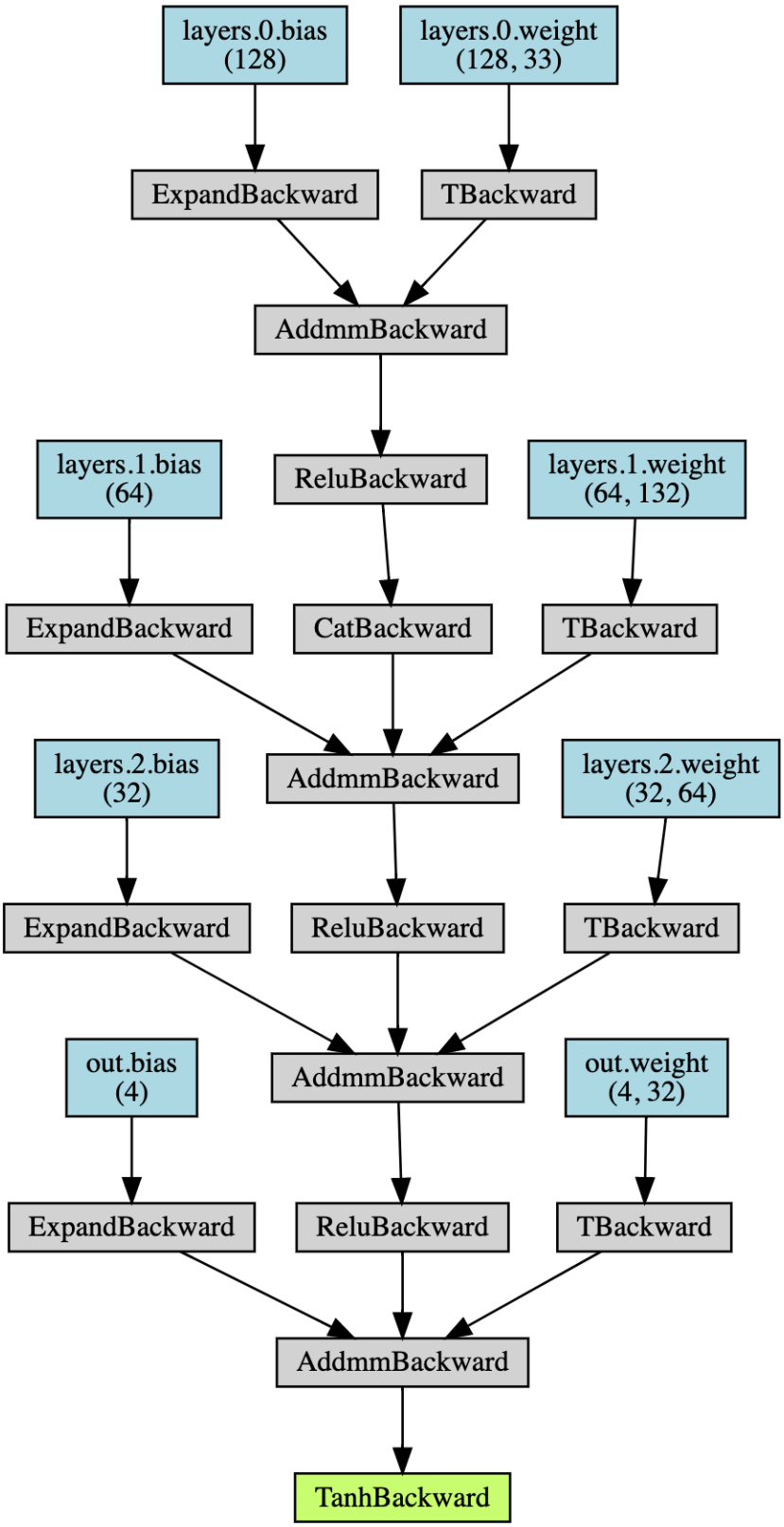


**Figure 2: Critic Network**

### Replay Buffer

The implementation uses a Replay Buffer to store past experiences (tuples of the state, action, reward, next state, and episode termination signal) and use random batches of past experiences to learn.

### Noise

Ornstein-Uhlenbeck noise is added to the action recommended by the actor network to facilitate exploration.

## Training

The algorithm is learning relatively quickly, solving the environment in just 126 episodes.

```
Environment solved in 26 episodes!      Average Score: 30.01
```

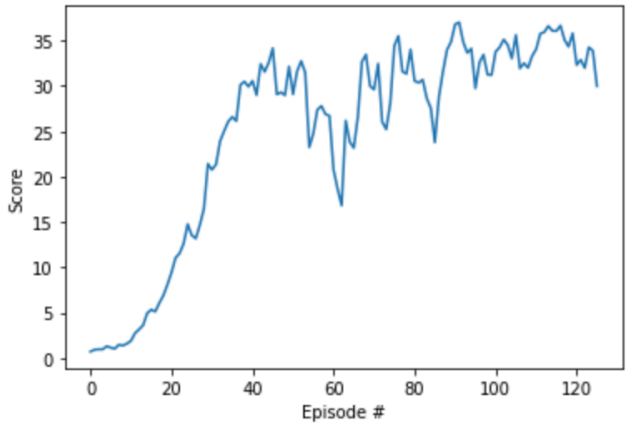


**Figure 3: Plot of Rewards**

## Conclusion and Possible Extensions

The algorithm is able to solve the environment relatively quickly. However, visual inspection of the trained agents reveals that their performance tends to deteriorate towards the end of the episodes. It is possible that more extended training provides a remedy for this behavior.

In addition, performance and training speed could potentially be increased further:

1. The hyperparameters used are likely not optimal. Additional tuning or grid search could be used to make the training more efficient.

2. The ReplayBuffer class could be appended to support prioritized experience replay.

3. Switching to more advanced algorithms, such as D4PG, A3C, TPRO, or PPO might yield better performance.