The algorithm used is an adaptation of the deep deterministic policy gradient algorithm developed in Continous Control with Deep Reinforcment Learning Article. For implementation I adapted the code for the pendulum example from udacity repository.

Description of Algorithm

DDPG method (the algorithm from the article):

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i \quad Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

The algorithm is a further development of the deep gradient policy algorithm developed in this article with a combination of ideas explored when developing DQN algorithm. The algorithm proved successful in extending the previous version applied to finite state spaces to spaces with infinite or large state space problems. The main novelty in extending the DPG algorithm is to use ideas from the DQN algorithm to add a replay buffer when training two networks called actor and critic for estimating value and policy. This proved essential for stabilising the learning procedure. The other important point was using Ornstein-Uhlenbeck process for adding noise to exploration of actor.

Hyperparameters changes:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-4         # learning rate of the actor initially 1e-3
LR_CRITIC = 1e-4        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay, initially 0.01
```

The initial hyperparameters that were set as in the pendulum example, which corresponds to initial parameters set in paper with only difference in the weight decay of 0.01 for the actor.

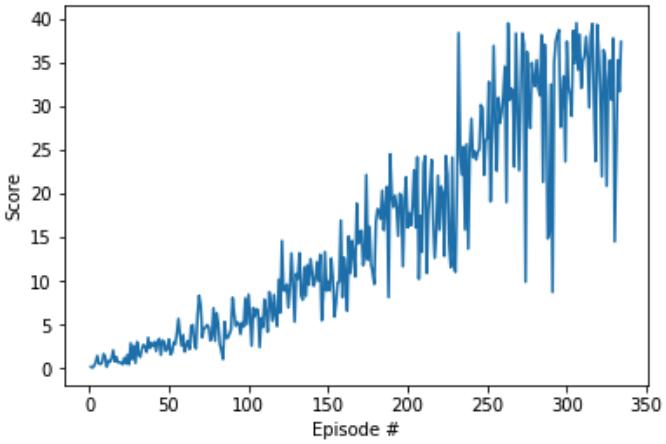
I started with using the default parameters but without success, then I increased the buffer size and experimented with reducing the critic learning rate. Finally after checking discussions in the program collaboration space I figured out that the noise was too large and reduced the sigma parameter to 0.1, it was then that the first progress started appearing.

After that I reduced the networks size to two fully connected layers with 128 units. This showed some success. I played with weight decay and increasing buffer size but without success. Finally I have checked the internet and found a very good resource in https://github.com/fdasilva59/Udacity-DRL-ContinuousControl where the whole training process is very well documented. I figured that I should try batch normalization (it is also mentioned in the article that this was a key point in stabilising the learning procedure) so I have added them after the first layer and the whole process started converging faster. I have tried adding the batch normalization after the second layer but it did not provide me with any success.

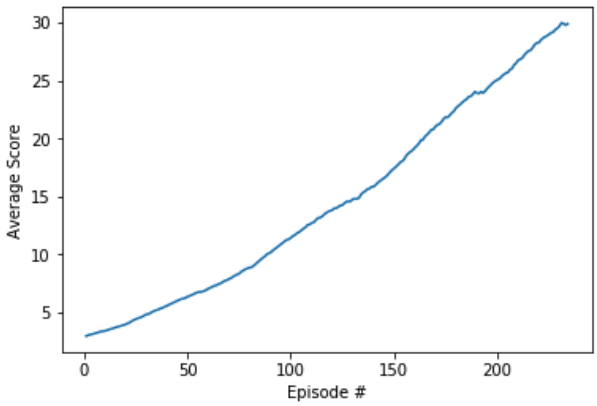
Finally I have chosen the architecture with two fully connected layers with relu activation function and 164 and 128 number of neurons which converged slightly better than 128, 128 case. The batch normalization layer is added between layers.

The reward function looked like this, I needed 334 episodes to finish trainings

```
Score is over 30.0 so objective is fullfilled. Stopping training.
```



The plot for average image is a bit easier to analyse



Future Work

As a first step I would set AWS or some other service to enable effective search of hyperparameters. In addition to algorithms already mentioned I would use repo https://stable-baselines.readthedocs.io/en/master/index.html for easy implementation of methods. I would like to try https://spinningup.openai.com/en/latest/algorithms/td3.html TD3 algorithm which should be more easier to tune.