Project 2: Continuous Control

Table of Contents

1 Author

Name: Christian Motschenbacher

Date: 02/2019

Project: Udacity Deep Reinforcement Learning Nanodegree Program: Project 2 Continuous Control (Reacher)

2 Acknowledgements

I would like to thank my wife, who has always encouraged and motivated me at the challenges during my continuous, professional, development courses. Many thanks also to all the teaching staff for the easy and understandable lectures.

3 Background and Agent's Learning Environment

The task of this project was to train 20 agents, who moves a double-jointed arm to target location and keeps the agent's hand in the goal location. The following picture show the Reacher training environment with 20 agents.

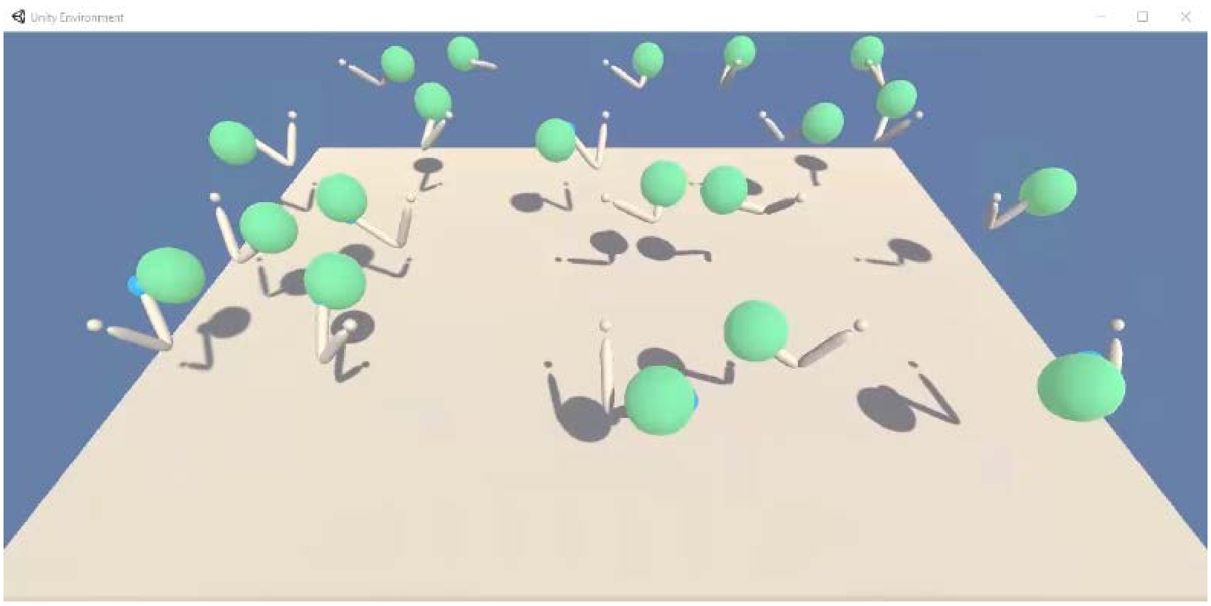


Figure 1: Example view of the Reacher Environment

A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of the agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

The task is episodic and in order to solve the environment, the agent must achieve an average score of +30 over 100 consecutive episodes.

For the solution of this task it has been used the Unity ML-Agents Reacher Environment from Unity-Technologies, the Python programming language as well the libraries NumPy, PyTorch and others.

The notebook for the training and testing "Continous_Control_solution_training_testing.ipynb" in this folder contains the chapter "Examine the State and Action Spaces", where you can find an example of the state space and action space as well more detailed information about the environment and project.

The software dependencies installation and configuration of this project are described in the md file in this repository.

4 Description of the Learning Algorithm and its implementation

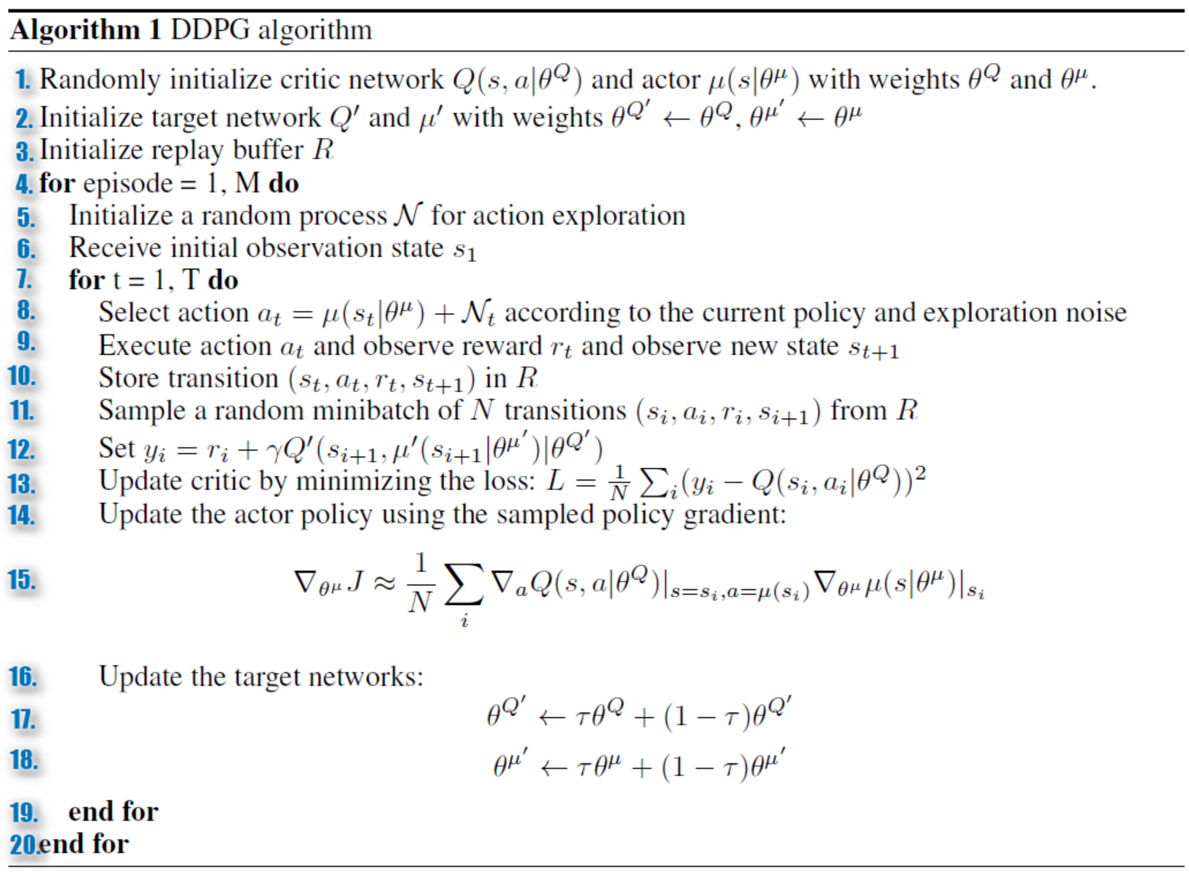
4.1 Introduction

The task for this project was to implement a reinforcement algorithm in Python and PyTorch to solve the Reacher environment, which has a continuous observation space and as well a continuous action space. I choose to implement the Deep Deterministic Policy Gradient algorithm (DDPG), which was taught in the lessons. This algorithm is an advancement of the Deep Q Network (DQN) algorithm. This means that the DQN can solve problem with high dimensional observation space as input, whereas those problems can have as output only a discrete or a low-dimensional action space. The model-free, off-policy actor-critic DDPG overcomes this disadvantage and can solve problems with a high dimensional or a continuous observation space as input and a high dimensional or a continuous action space as output. This is possible by using deep function approximators for the observation spaces and as well for the action spaces. One issue of using a non-linear function approximator such as the neural network is that reinforcement learning tends to become unstable or even to diverge. The cause of this instability is for instance the correlation, which is present in the sequence of state, actions, reward and next state. This issue can be improved via the following two features mentioned in the DQN paper and has been implemented in the DDPG.

The first feature is the implementation of a replay memory, which removes some correlations in the observation sequence (s_t,a_t,r_t,s_{t+1}) and act like a smoothing filter. The second feature is to implement a second neural network (target) in addition to the first neural network (local). The target neural network is updated only periodically to reduce the correlation. The implementation of the second neural network made it necessary to parameterize the weights θ_i so that the value function changes to $Q(s,a,\theta_i)$. The mechanism of replay memory is to store the agent`s experiences $e_t=(s_t,a_t,r_t,s_{t+1})$ at each time-step t in a data set $D_t=\{e_1,\ldots,e_t\}$, sample from those uniformly, add those to the observation sequence (s_t,a_t,r_t,s_{t+1}) and use those during learning.

The implemented DDPG algorithm contains all the element mentioned in the paper. Those include replay memory, actor and critic network with batch normalisation, Ornstein-Uhlenbeck noise for exploration. Divergent from the paper, I used as activation function the leaky_relu instead of relu, which resulted in an additional improvement, which can be seen in the result section of this report.

The following pseudocode Figure 2 show the DDPG, which is used in my project. This DDPG pseudocode is described in the following chapter and as well the additional improvement and research such as replacing the relu activation function with the leaky_relu, which I have implemented.



```
Algorithm 1 DDPG algorithm
1.  Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
2.  Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
3.  Initialize replay buffer $R$
4.  for episode = 1, M do
5.      Initialize a random process $\mathcal{N}$ for action exploration
6.      Receive initial observation state $s_1$
7.      for t = 1, T do
8.          Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
9.          Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
10.         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
11.         Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
12.         Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
13.         Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
14.         Update the actor policy using the sampled policy gradient:
15.             $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
16.         Update the target networks:
17.             $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
18.             $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
19.     end for
20. end for
```

Figure 2: Pseudocode of DDPG from DDPG paper

4.2 DDPG Pseudocode Algorithm Description (Timothy P., et al., 2016)

The following lines describe the pseudocode of the DDPG in Figure 2 and the implementation of the algorithm.

Line 1: Random initialisation of the weights θ^Q for the critic network Q and weights θ^μ for the actor network μ.

Line 2: Initialisation of the target network Q´ and μ´ by copying the weights from Q and μ.

Line 3: Initialisation of the replay buffer R.

Line 4 – 20: Episode for loop, which repeats until M has reached.

Line 5: Initialise a random process N (noise) for action exploration.

Line 6: Receiving initial observation state s_1.

Line 7 – 19: Interaction for loop, which repeats until T has reached.

Line 8: Select action according to the current policy and exploration noise.

Line 9: Perform action a_t and observe reward r_t as well new state s_{t+1}.

Line 10: Store the transition (s_t, a_t, r_t, s_{t+1}´) into the replay buffer R.

Line 11: Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from replay buffer R.

Line 12: Set target $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

Line 13: Update the critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

Line 14: Update the actor policy using the sampled policy gradient:

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

Line 15: Update the target networks:

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

This pseudocode of the DDPG has been implemented in the three following Python and Pytorch code functions and classes files.

- **ddpg_agent.py**
  This file contains the class for the agent and the class for the Replay Buffer.
- **model.py**
  This file contains the different neural network models.
- **Continous_Control_solution_training_testing.ipynb**
  This file contains the code for the training and testing of the agents.

4.3 Description of further DDPG improvement.

There are one additional improvements, which has been analysed. This improvement are DDPG model with leaky relu instead of the relu activation function. The models itself, its parameters and as well their performance reward plot are described in in the next chapters.

4.4 Parameter and model

4.4.1 Agent, Replay Buffer and DDPG

The following parameter are the same for all experiments

Environment parameter

• state_size = 33	# State space of the environment
• action_size = 4	# Action space of the agent in the environment

Training parameter

• n_episodes = 2000	# number of training episodes
• max_t = 1000	# maximum number of timesteps per episode

Agent parameter

- BUFFER_SIZE = int(1e6) # replay buffer size
- BATCH_SIZE = 512 # minibatch size
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR_ACTOR = 1e-3 # learning rate of the actor
- LR_CRITIC = 1e-3 # learning rate of the critic
- WEIGHT_DECAY = 0 # L2 weight decay regularization
- UPDATE_EVERY = 20 # when to update the network
- UPDATE_COUNT = 10 # how many times to update the network

4.4.2 Model architecture with parameter

The following sections describe the model architecture including their parameter.

4.4.2.1 Actor nn model

- Model architecture, parameter

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed  = torch.manual_seed(seed)
        self.fc1   = nn.Linear(state_size, fc1_units)
        self.bnfc1 = nn.BatchNorm1d(num_features=fc1_units)
        self.fc2   = nn.Linear(fc1_units, fc2_units)
        self.bnfc2 = nn.BatchNorm1d(num_features=fc2_units)
        self.fc3   = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.leaky_relu(self.bnfc1(self.fc1(state)))
        x = F.leaky_relu(self.bnfc2(self.fc2(x)))
        return F.tanh(self.fc3(x))
```

This model has three fully connected layer with the following parameter.

self.fc1: 1.fully connected layer with 33 inputs and 400 outputs

self.bnfc1: batch normalisation for 1. fully connected layer

self.fc2: 2.fully connected layer with 400 inputs and 300 outputs

self.bnfc2: batch normalisation for 2. fully connected layer

self.fc3: 3.fully connected layer with 300 inputs and 4 outputs

The activation function for fc1 and fc2 is leaky_relu and it is tanh activation function for fc3.

4.4.2.2 Critic nn model

- Model architecture, parameter

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=400, fc2_units=300):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed    = torch.manual_seed(seed)
        self.fcs1    = nn.Linear(state_size, fcs1_units)
        self.bnfcs1 = nn.BatchNorm1d(num_features=fcs1_units)
        self.fc2     = nn.Linear(fcs1_units+action_size, fc2_units)
        self.bnfc2   = nn.BatchNorm1d(num_features=fc2_units)
        self.fc3     = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
        xs = F.leaky_relu(self.bnfcs1(self.fcs1(state)))
        x = torch.cat((xs, action), dim=1)
        x = F.leaky_relu(self.bnfc2(self.fc2(x)))
        return self.fc3(x)
```

This model has three fully connected layer with the following parameter.

self.fcs1: 1.fully connected layer with 33 inputs and 400 outputs

self.bnfcs1: batch normalisation for 1. fully connected layer

self.fc2: 2.fully connected layer with 400 inputs and 300 outputs

self.bnfc2: batch normalisation for 2. fully connected layer

self.fc3: 3.fully connected layer with 300 inputs and 1 output

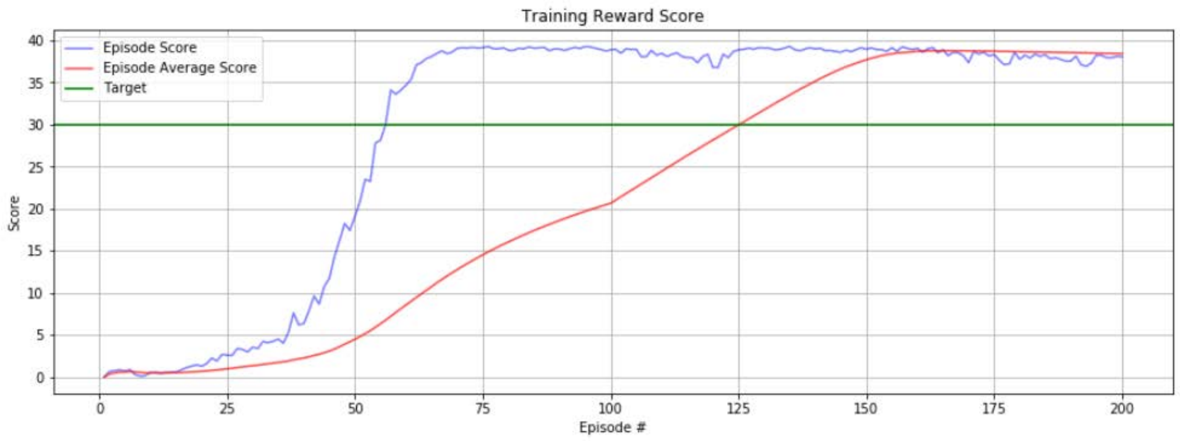
The activation function for fc1 and fc2 is leaky_relu and it is linear activation function for fc3.

5 Rewards Plot and Results of trained agent

The following reward plots show the learning reward results of the agents of different network architectures during training.

5.1 Actor NN model and Critic NN model without batch normalisation and with relu activation function

5.1.1 Reward Plot



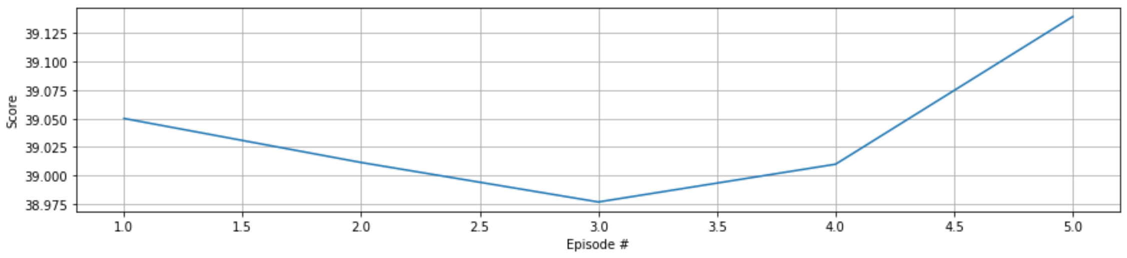
```
Environment solved in 126 episodes! Average Score: 30.02
Maximum average score of 38.78 achieved in episode 165
```

The above number from the experiment show that this DDPG exceeded an average score of 30 over 100 consecutive episodes after 126 episodes and the highest achieved average score was 38.78. After the reward achieves its maximum score at around episode 70 it stays stable at this level.

5.1.2 Result of a trained agent

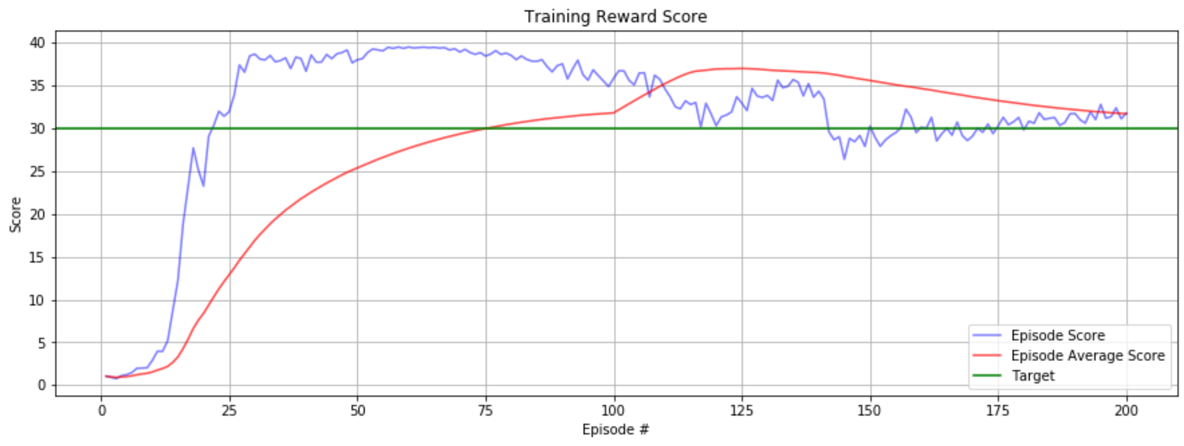
The following run chart shows the reward results of a trained DDPG agent over five episodes.

```
Score of 20 trained agents: [39.04999912716448, 39.01149912802502, 38.97699912879616, 39.00999912805855, 39.138999125175175]
```



5.2 Actor NN model and Critic NN model with batch normalisation and with relu activation function

5.2.1 Reward Plot



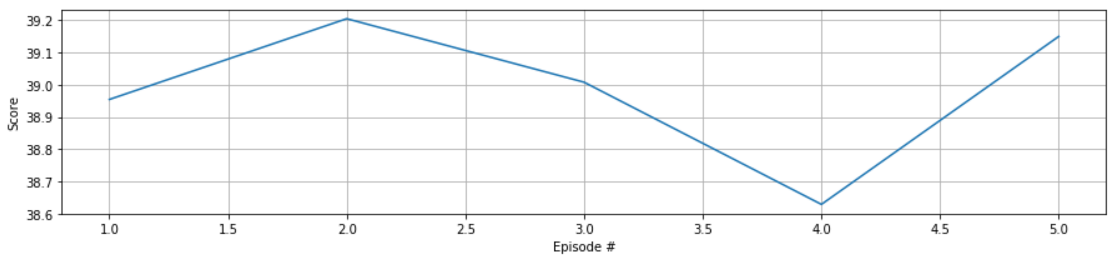
```
Environment solved in 76 episodes!  Average Score: 30.09
Maximum average score of 37.00 achieved in episode 125
```

The above number from the experiment show that this DDPG exceeded an average score of 30 over 100 consecutive episodes after 76 episodes and the highest achieved average score was 37.00. After the reward achieves its maximum score at around episode 30, stays stable until episode 75 and decrease after this point in time. The falling reward might be a result of the batch normalisation, which needs to be analysed further.

5.2.2 Result of a trained agent

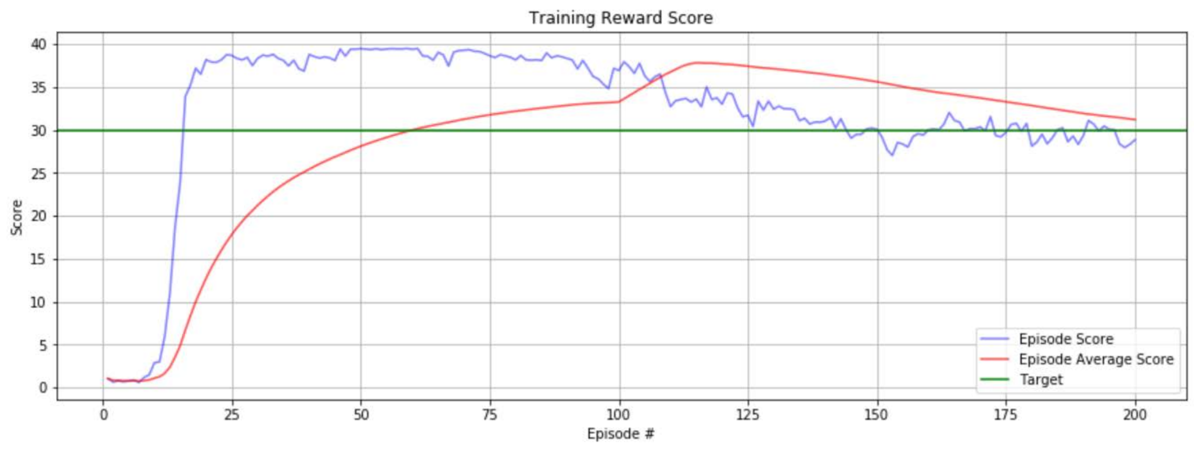
The following run chart shows the reward results of a trained DDPG agent over five episodes.

```
Score of 20 trained agents: [38.95399912931025, 39.20449912371114, 39.00749912811443, 38.628499136585745, 39.14949912494048]
```



5.3 Actor NN model and Critic NN model with batch normalisation and with leaky_relu activation function

5.3.1 Reward Plot



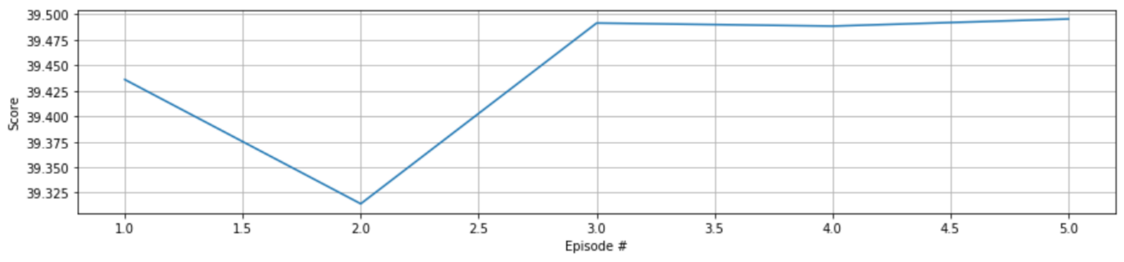
```
Environment solved in 60 episodes!  Average Score: 30.02
Maximum average score of 37.84 achieved in episode 115
```

The above number from the experiment show that this DDPG exceeded an average score of 30 over 100 consecutive episodes after 60 episodes and the highest achieved average score was 37.84. After the reward achieves its maximum score at around episode 25, stays stable until episode 80 and decrease after this point in time. The falling reward might be a result of the batch normalisation, which needs to be analysed further.

5.3.2 Result of a trained agent

The following run chart shows the reward results of a trained DDPG agent over five episodes.

```
Score of 20 trained agents: [39.435999118536714, 39.31399912126362, 39.491499117296186, 39.48849911736325, 39.49549911720678]
```



5.4 Reward plot and results summary

The data in the previous section show that the "**actor and critic with batch normalisation and leaky_relu" perform best** on this task and solve this environment after **60 episodes**. The "**actor and critic with batch normalisation and relu"** has the second best performance and solve the environment after **76 episodes**. On third place is the "**actor and critic without batch normalisation and relu"**, which solve the environment after **126 episodes**. The reward of the "**actor and critic with batch normalisation and leaky_relu"** and "**actor and critic with batch normalisation and relu"** falls after around episode 70. This might be a result of the batch normalisation, which needs to be analysed further.

6 Ideas for Future Work

There are several ideas for future work and improvements. Some possible ideas for future work are.

- Ideas and future work to improve the agent's performance
    - Implementing the prioritized experience replay for the DDPG and compare the current results.
    - Implementing and using better RL algorithm such as D4PG
- Some ideas and improvement to speed up the learning and therefore, more time to improve the agents performance are
    - Writing and using a framework, which can be used for an automatized, simplified and faster hyperparameter search.
        - This framework can include for instance and do not be limited to
            - Grid Search
            - Random Search
            - Bayesian Optimisation
        - Run multiple scripts with different parameter in parallel to reduce the hyperparameter search time.
    - Optimise the DDPG algorithm with replay buffer and code by
        - Using vectorised commands where possible instead of for loops
        - Parallelisation and synchronisation of the sampling and learning.
            - One process can sample a few episodes while the other process can perform the learning.

7 References

Further resources and references regarding this project and DDPG can be found in the following links.

- Timothy P., et al. "Continuous control with deep reinforcement learning, DDPG (Deep Deterministic Policy Gradients)." https://arxiv.org/pdf/1509.02971.pdf
- John Schulman, et al. "PPO (Proximal Policy Optimization Algorithms)." https://arxiv.org/pdf/1707.06347.pdf
- Volodymyr Mnih, et al. "Asynchronous Methods for Deep Reinforcement Learning." https://arxiv.org/pdf/1602.01783.pdf
- Gabriel Barth-Maron, et al. "D4PG (Distributed Distributional Deterministic Policy Gradients)." https://openreview.net/pdf?id=SyZipzbCb
- Andrej Karpathy. "Andrej Karpathy blog." http://karpathy.github.io/2016/05/31/rl/
- Dhruv Parthasarathy. "Write an AI to win at Pong from scratch with Reinforcement Learning." https://medium.com/@dhruvp/how-to-write-a-neural-network-to-play-pong-from-scratch-956b57d4f6e0
- Andrej Karpathy, et al. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning." https://blog.openai.com/evolution-strategies/
- Shixiang Gu, et al. "Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic." https://arxiv.org/pdf/1611.02247.pdf
- John Schulman, et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation." https://arxiv.org/pdf/1506.02438.pdf
- Shixiang Gu, et al. "Continuous Deep Q-Learning with Model-based Acceleration, NAF (normalized advantage functions)." https://arxiv.org/pdf/1603.00748.pdf
- Udacity, et al. "Deep Reinforcement Learning Nanodegree." https://github.com/udacity/deep-reinforcement-learning