Project 1: Navigation

Table of Contents

1 Author

Name: Christian Motschenbacher

Date: 01/2019

Project: Udacity Deep Reinforcement Learning Nanodegree Program: Project 1 Navigation (Banana)

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 an agent, who navigates in a large environment, collect as many yellow bananas while avoiding blue bananas. The following picture show the agent in the environment from the agent´s view.

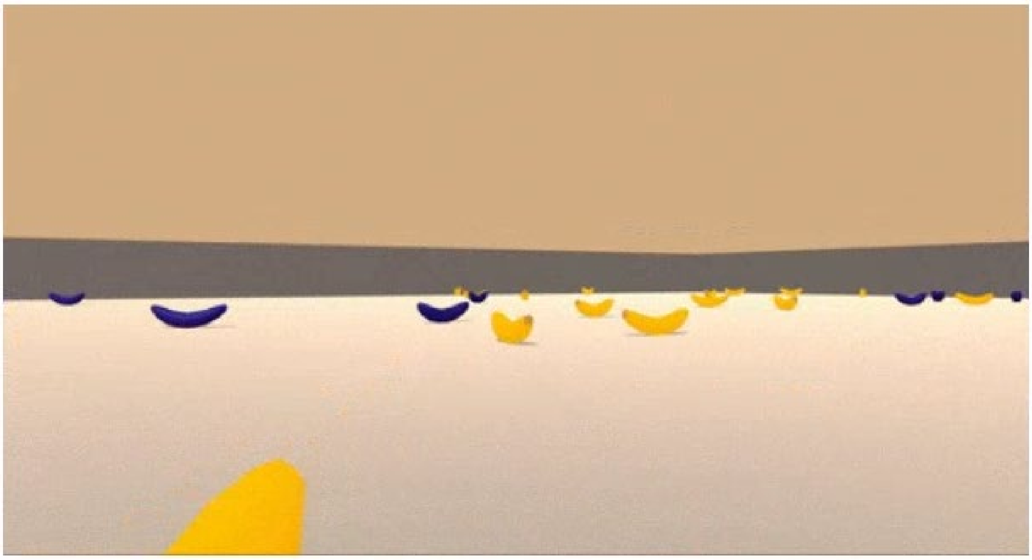


Figure 1: Example view of the Banana Environment

A reward of +1 is provided for collecting a yellow banana and a reward of -1 is provided for collecting a blue banana. Thus, the goal of the agent is to collect as many yellow bananas as possible while avoiding blue bananas.

The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. Given this information, the agent has to learn how to best select actions. Four discrete actions are available, corresponding to:

- 0 - Move forward.
- 1 - Move backward.
- 2 - Turn left.
- 3 - Turn right.

The task is episodic and in order to solve the environment, the agent must achieve an average score of +13 over 100 consecutive episodes.

In order to solve the banana environment from Unity-Technologies in this project it has been used the Python programming language as well the libraries NumPy, PyTorch and others.

The notebook for the training and testing "Navigation_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 the Deep Q-Learning algorithm (DQN) in Python and PyTorch to solve the banana environment. I choose to implement the DQN, which was taught in the lessons. This algorithm has been derived from the well-known paper “Human-level control through deep reinforcement learning”, which can be seen here (Minh et al., 2015). The implemented algorithm contains all the element as in the paper with the different that the neural network model (nnm) has no convolutional layers and no clipping of the reward implemented.

The following pseudocode in Figure 2 (from lesson) and Figure 3 (from paper) show the DQN, which is used in my project. This DQN pseudocode is described in the following chapter and as well the additional improvement and research such as Dueling model, He-initialisation, etc., which I have implemented.

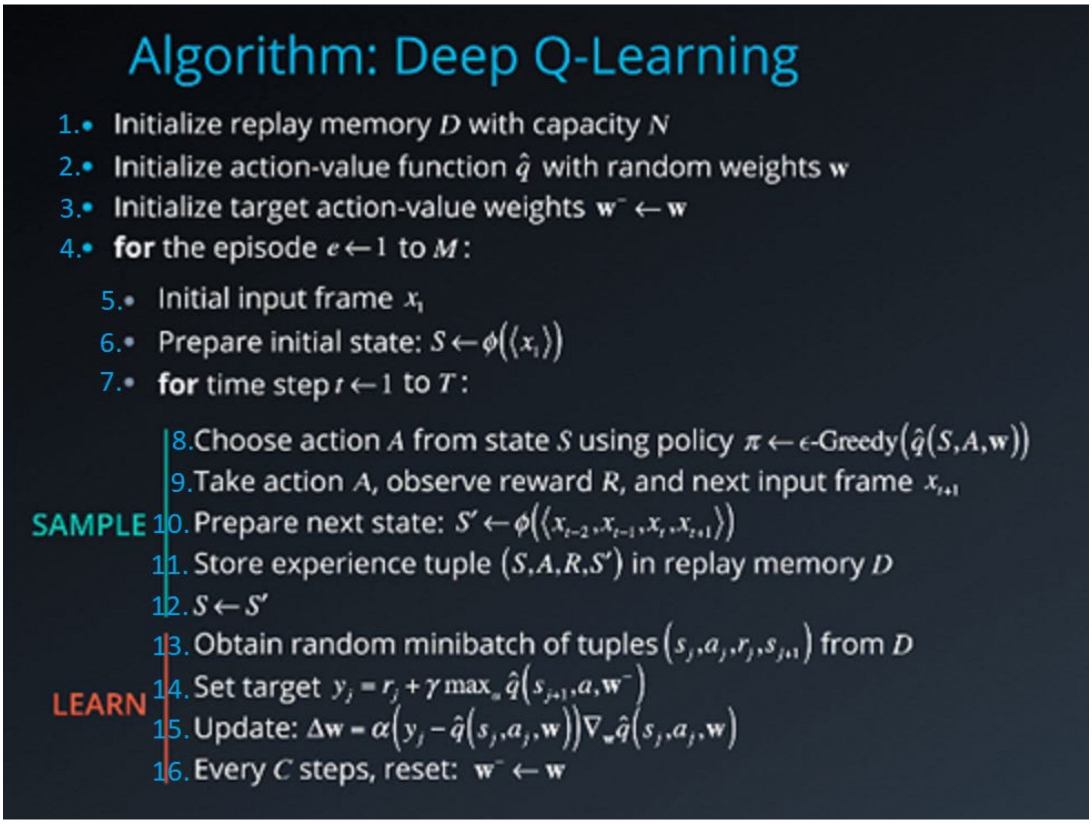


Figure 2: Pseudocode of DQN from Lesson

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode $= 1, M$ **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 3: Pseudocode of DQN from DQN paper

4.2 DQN Description (Minh et al., 2015)

DQN is an advancement of the traditional reinforcement learning (RL) or Q-Learning (action-value function) by using a deep neural network. The agent's goal is to maximise the sum Q*(s,a) of the future reward (r) discounted by (γ) through interacting/following the optimal policy (π) through actions (a) and observations of the state (s) in the environment as in the optimal action-value function

$$Q^*(s,a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ \ldots \ | s_t = s,\ a_t = a,\ \pi\right]$$

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.

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 following loss function $L_i(\theta_i)$ is updated at iteration i by the Q-learning.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim \mathrm{U}(D)}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

The target parameter θ_i^- of the target network is updated with the parameter θ_i only at every C steps otherwise the parameter is held fixed.

4.3 DQN Pseudocode Description

The lines of the pseudocode of the DQN in Figure 2 is described below as this figure is intuitive and easier to understand than Figure 3, but it is the same DQN Figure 3.

The line 1 until line 3 initialize the memory D, the action-value function $\hat{q}$ with the weights w and initialize the target-action weights w^- with w.

Line 4 show the for-loop for the episodes, which loops over the lines from line 4 until including line 16.

Line 5 and line 6 prepare the input frame x_1 (not implemented in this project as the task was to use the discrete state space and not the visual pixel as state space) and the initial state S.

Line 7 show the for-loop for the algorithm step, which loops over the lines from line 7 until including line 16.

The line 8 until line 12 sample an experience tuple (S, A, R, S´) and store it in the replay memory D.

Line 8 choose an action A from state S by using ε-Greedy.

Line 9 take action A, observe the reward R, and receive the next input frame x_{t+1} (in is project it is not a frame it is a discrete state space).

Line 10 prepare the next state S´.

Line 11 store the experience tuple (S, A, R, S´) in replay memory D.

Line 12 copy S´ to S

The line 13 until 16 take a random uniform minibatch of tuples (s_j, a_j, r_j, s_{j+1}´) from replay memory D, perform forward propagation to calculate the prediction and update the weights through backpropagation. The weights w^- get updated every step C from w.

Line 13 take a random uniform minibatch of (s_j, a_j, r_j, s_{j+1}´) from replay memory D.

Line 14 set the target y_j.

Line 15 update the weights w through backpropagation.

Line 16 update the weights w^- get updated every step C from w.

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

- **dqn_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.
- **Navigation_solution_training_testing.ipynb**
  This file contains the code for the training and testing of the agent.

4.4 Description of further DQN improvement.

There are three additional potential improvements, which has been analysed. Those improvements are DQN model with additional NN laysers, DQN model with he-initialization and Dueling DQN model. The models itself, its parameters and as well their performance reward plot are described in in the next chapters.

4.5 Parameter and model

4.5.1 Agent, Replay Buffer and DQN

The following parameter are the same for all experiments

Agent parameter

- BUFFER_SIZE = int(1e5) # replay buffer size
- BATCH_SIZE = 64 # minibatch size
- GAMMA = 0.99 # discount factor
- TAU = 1e-3 # for soft update of target parameters
- LR = 5e-4 # learning rate

- UPDATE_EVERY = 4 # how often to update the network

Training parameter

- n_episodes = 2000 maximum number of training episodes
- max_t = 1000 maximum number of timesteps per episode
- eps_start = 1.0 starting value of epsilon, for epsilon-greedy action selection
- eps_end = 0.01 minimum value of epsilon
- eps_decay = 0.995 multiplicative factor (per episode) for decreasing epsilon

Environment parameter

- state_size = 37 # State space of the environment
- action_size = 4 # Action space of the agent in the environment

4.5.2 Model architecture with parameter

The following sections describe the model architecture including their parameter.

4.5.2.1 Baseline DQN

- Model architecture, parameter

```
# DQN baseline network model
class QNetwork(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64,         64)
        self.fc3 = nn.Linear(64,         action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

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

self.fc1: 1.fully connected layer with 37 inputs and 64 outputs

self.fc2: 2.fully connected layer with 64 inputs and 64 outputs

self.fc3: 3.fully connected layer with 64 inputs and 4 outputs

The performance of this network is described in the reward plot section.

4.5.2.2 Baseline DQN with additional NN layers

As a potential improvement, it has been analysed if additional layer can result in a better performance.

- Model architecture, parameter

```
# DQN baseline network model with additional fully connected layers
class QNetwork_add_fcl(nn.Module):
    """Actor (Policy) Model with additional fully connected layer
    in comparision to the base line model QNetwork."""

    def __init__(self, state_size, action_size, seed):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork_add_fcl, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 16)
        self.fc5 = nn.Linear(16, 8)
        self.fc6 = nn.Linear(8, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        return self.fc6(x)
```

This baseline model DQN with additional NN has six fully connected layer with the following parameter.

self.fc1: 1.fully connected layer with 37 inputs and 128 outputs

self.fc2: 2.fully connected layer with 128 inputs and 64 outputs

self.fc3: 3.fully connected layer with 64 inputs and 32 outputs

self.fc4: 4.fully connected layer with 32 inputs and 16 outputs

self.fc5: 5.fully connected layer with 16 inputs and 8 outputs

self.fc6: 6.fully connected layer with 8 inputs and 4 outputs

The performance of this network is described in the reward plot section.

4.5.2.3 Baseline DQN with he-initialization

The best practise in building a NN with relu activation is to initialise the weights with he-initialisation. Therefore, as this is a potential improvement, it has been analysed if he-initialisation can result in a better performance.

- Model architecture, parameter

```
# DQN baseline network model with he initialzation
class QNetwork_he_init(nn.Module):
    """Actor (Policy) Model with he initialization."""

    def __init__(self, state_size, action_size, seed):
        """Initialize parameters and build model
        initialised with he initialisation, because
        of relu activation function.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork_he_init, self).__init__()
        # set random seed
        self.seed = torch.manual_seed(seed)
        # create three linear layers
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64,         64)
        self.fc3 = nn.Linear(64,         action_size)
        # initialize the layers with HE weights, because of the
        # relu activation
        torch.nn.init.kaiming_uniform_(self.fc1.weight,nonlinearity='relu')
        torch.nn.init.kaiming_uniform_(self.fc2.weight,nonlinearity='relu')

    def forward(self, state):
        """ Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

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

self.fc1: 1.fully connected layer with 37 inputs and 64 outputs

self.fc2: 2.fully connected layer with 64 inputs and 64 outputs

self.fc3: 3.fully connected layer with 64 inputs and 4 outputs

The performance of this network is described in the reward plot section.

4.5.2.4 Dueling DQN

Another improvement of the baseline DQN from the literature is the Dueling DQN. The following code shows the Dueling DQN architecture and their parameter. As this project do not use a visual (pixel) input from the environment the architecture do not has any convolutional neural network (CNN) layers and instead the architecture uses for the first three CNN three fully connected layers.

- Model architecture, parameter

```
# Dueling DQN network model
class QNetwork_dueling(nn.Module):
    """Dueling Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
        """
        super(QNetwork_dueling, self).__init__()

        self.seed = torch.manual_seed(seed)
        self.action_size = action_size

        self.fc1 = nn.Linear(state_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)

        self.fc1_adv = nn.Linear(32, 16)
        self.fc2_adv = nn.Linear(16, action_size)

        self.fc1_val = nn.Linear(32, 16)
        self.fc2_val = nn.Linear(16, 1)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))

        fc1_adv = F.relu(self.fc1_adv(x))
        fc1_val = F.relu(self.fc1_val(x))

        fc2_adv = self.fc2_adv(fc1_adv)
        fc2_val = self.fc2_val(fc1_val)

        return_val = fc2_val + fc2_adv - fc2_adv.mean()
        return return_val
```

This dueling DQN has three fully connected layer and it has after those additional two fully connected layers for generating the “state values V(s)” and two fully connected layers for generating the “advantage values A(s,a)”. V(s) and A(s,a) are combined to calculate Q(s,a). Those layers and their parameters are as following.

self.fc1: 1.fully connected layer with 37 inputs and 128 outputs

self.fc2: 2.fully connected layer with 128 inputs and 64 outputs

self.fc3: 3.fully connected layer with 64 inputs and 32 outputs

self.fc1_adv: 1.fully connected layer for A(s,a) with 32 inputs and 16 outputs

self.fc1_val: 1.fully connected layer for V(s) with 32 inputs and 16 outputs

self.fc2_adv: 2.fully connected layer for A(s,a) with 16 inputs and 4 outputs

self.fc2_val: 2.fully connected layer for V(s) with 16 inputs and 1 outputs

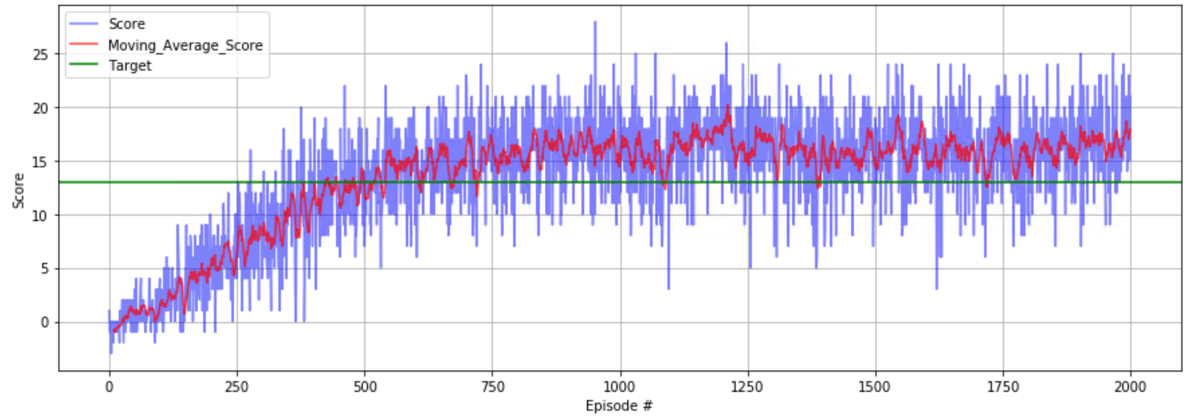
The performance of this network is described in the reward plot section.

5 Rewards Plot and Results of trained agent

The following plot show the learning reward results of the agent during training of the different network architecture.

5.1 Baseline DGN

5.1.1 Reward Plot



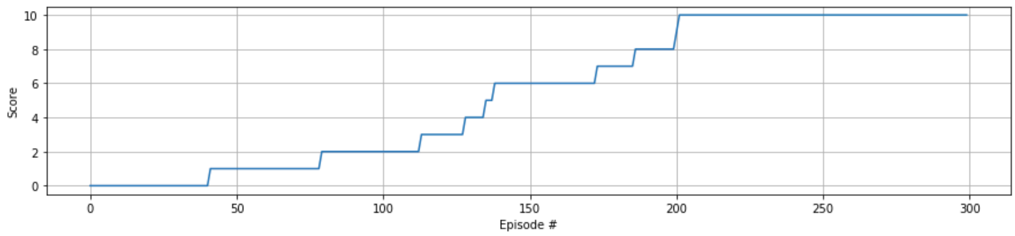

```
Environment solved in 451 episodes!	Average Score: 13.02
Maximum Score Episode 1200 Average Score: 17.14
```

The above extracted number from the experiment show that this DQN exceeded an average score of 13 over 100 consecutive episodes after **451 episodes** and the highest achieved score was **17.14**.

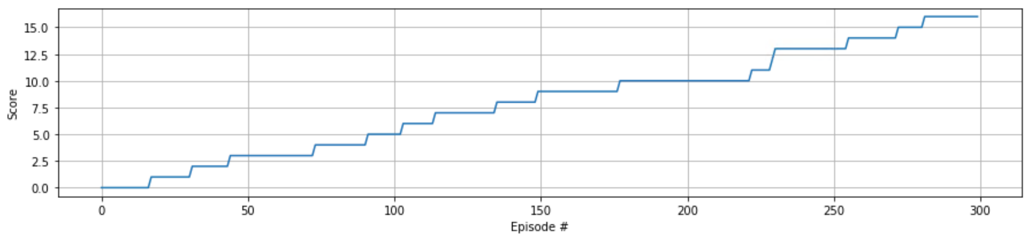
5.1.2 Result of a trained agent

The following three run charts show the results of a trained agent on this DQN.

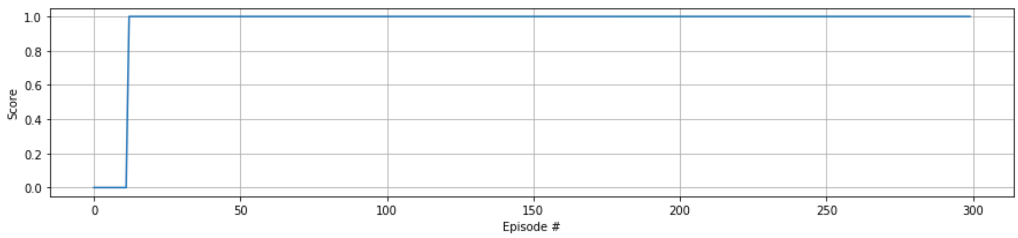
```
Score: 10.0
```




```
Score: 16.0
```



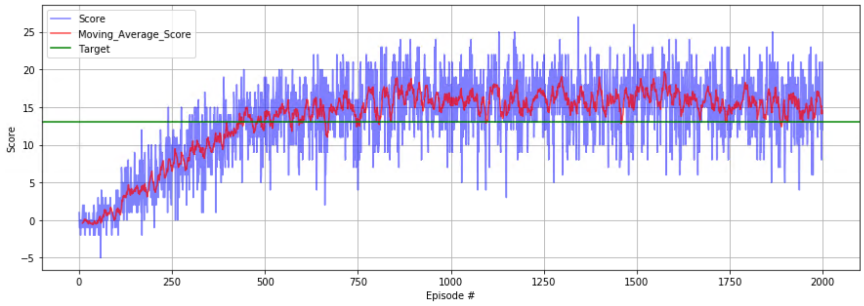

```
Score: 1.0
```



Those results are quite interesting, because of the following finding. As it can seen the reward of the third episode do not continue after the reward reaches 1. This is due to the fact that a state in such an environment can have multiple optimal policies and the agent cannot decide which action to take next and remain at this state. I observed this situation during the testing phase. This is one point, which needs to be analysed and idea for further improvement work. The same problem can be seen in the first episode where the accumulated reward score of 10 reach a plateau at the interaction about 200.

5.2 Baseline DQN with additional layers

5.2.1 Reward Plot



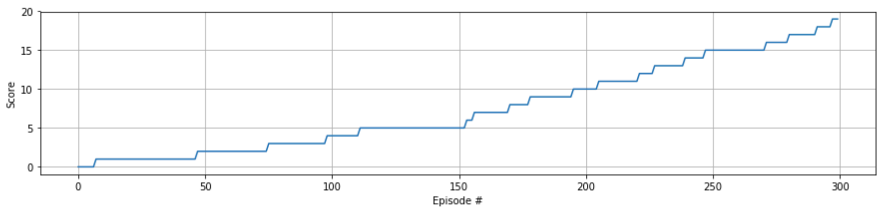

```
Environment solved in 404 episodes!  Average Score: 13.05
Maximum Score Episode 1600 Average Score: 17.04
```

The above extracted number from the experiment show that this DQN exceeded an average score of 13 over 100 consecutive episodes after **404 episodes** and the highest achieved score was **17.04**.

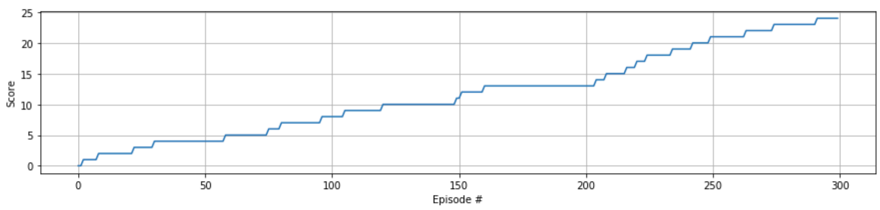
5.2.2 Result of a trained agent

The following three run charts show the results of a trained agent on this DQN.

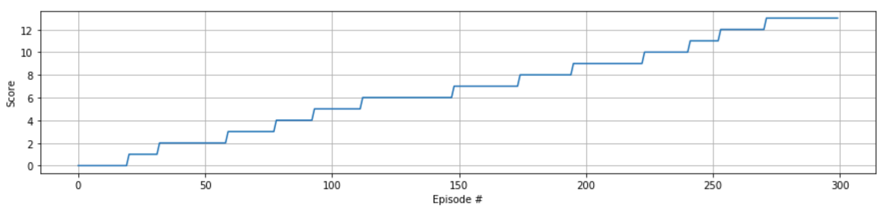
Score: 19.0



Score: 24.0



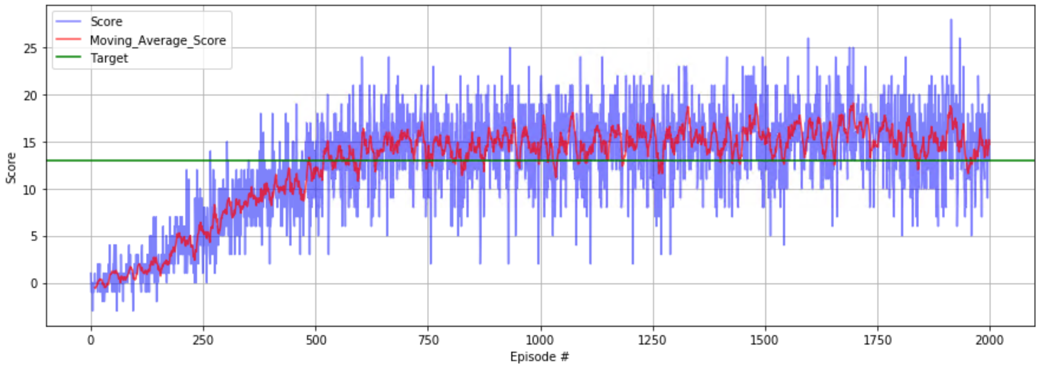
Score: 13.0



The three test episodes are quite normal and do not have any special findings.

5.3 Baseline DQN with he-initialisation for relu activation

5.3.1 Reward Plot



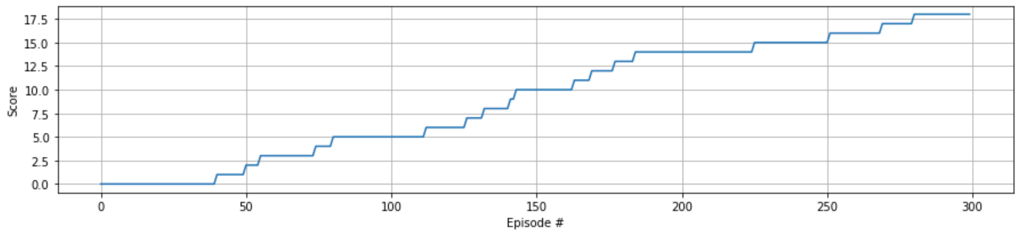

```
Environment solved in 500 episodes!  Average Score: 13.03
Maximum Score Episode 1700 Average Score: 16.11
```

The above extracted number from the experiment show that this DQN exceeded an average score of 13 over 100 consecutive episodes after **500 episodes** and the highest achieved score was **16.11**.

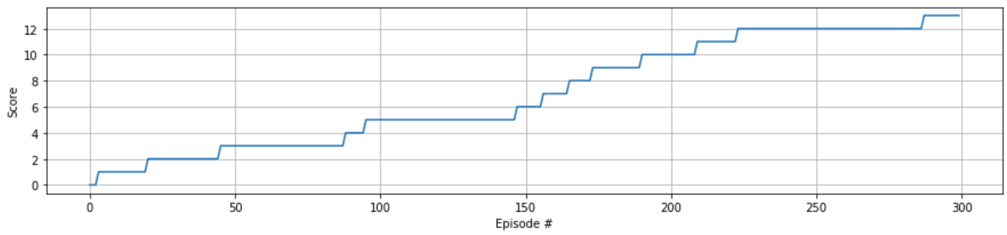
5.3.2 Result of a trained agent

The following three run charts show the results of a trained agent on this DQN.

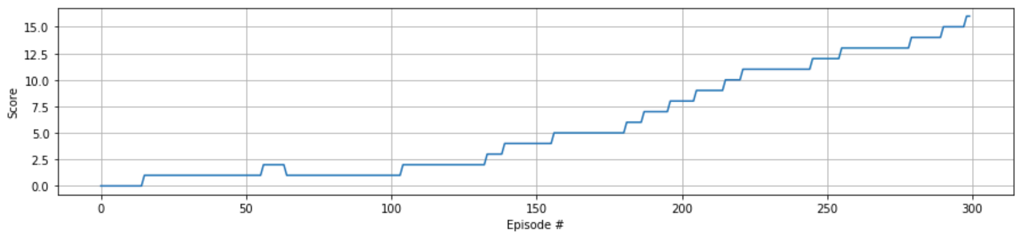
Score: 18.0



Score: 13.0



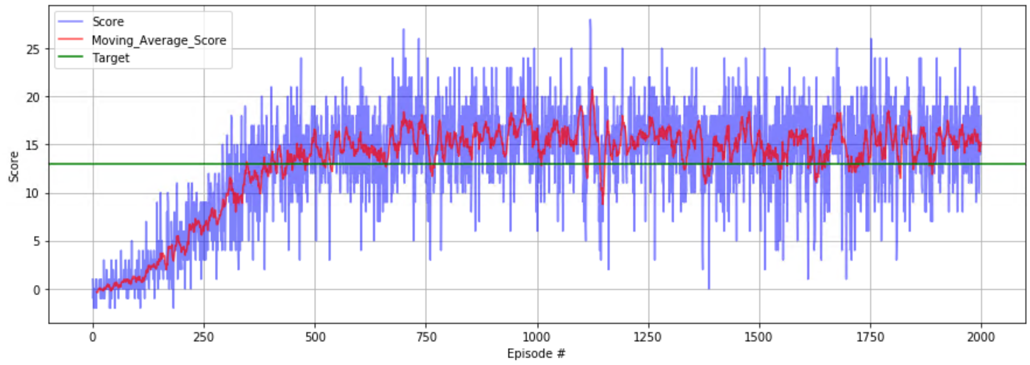
Score: 16.0



The third test episode do not increase the result award significant until it reach an interaction of about 140. It seems that the agent had difficulty and was in a state with multiple optimal policies, but the agent was able to leave this state.

5.4 Dueling DQN

5.4.1 Reward Plot



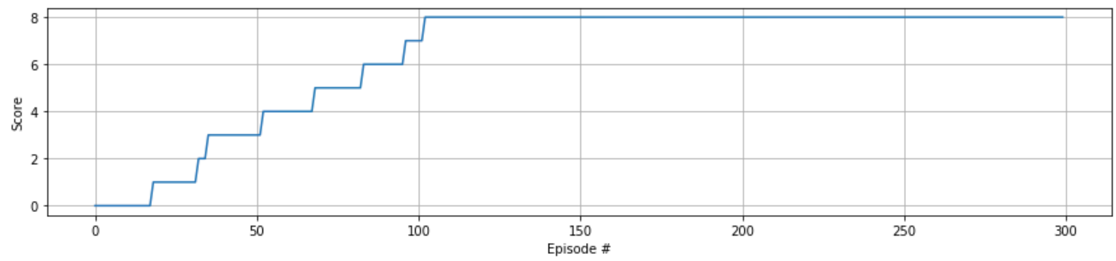

```
Environment solved in 370 episodes! Average Score: 13.14
Maximum Score Episode 1000   Average Score: 16.51
```

The above extracted number from the experiment show that this DQN exceeded an average score of 13 over 100 consecutive episodes after **370 episodes** and the highest achieved score was **16.51**.

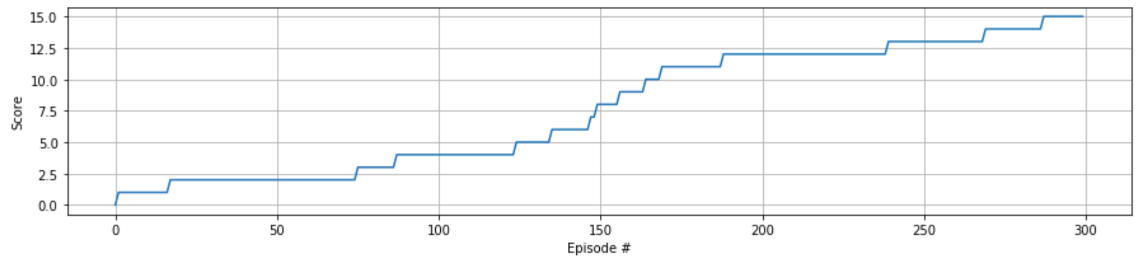
5.4.2 Result of a trained agent

The following three run charts show the results of a trained agent on this DQN.

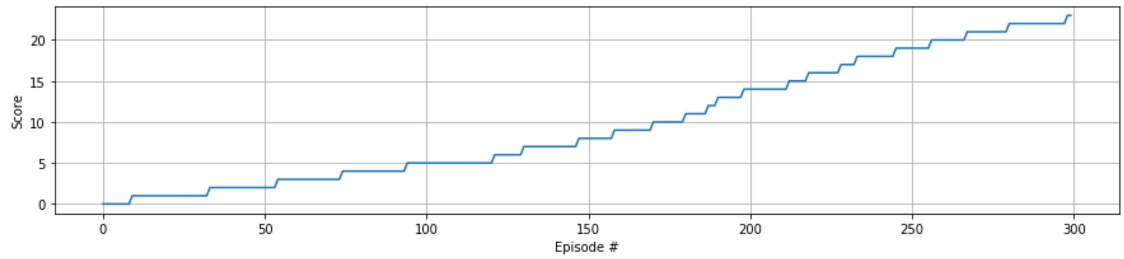
```
Score: 8.0
```




```
Score: 15.0
```




```
Score: 23.0
```



The first episode seems to have the same problem with the multiple optimal policies in one state at the interaction of about 100.

5.5 Reward plot and results summary

The data in the previous section show that the **Dueling DQN perform best** on this task and solve this environment after **370 episodes**. The **"Baseline DQN with additional layers"** has the second best performance and solve the environment after **404 episodes**. On third place is the **"Baseline DGN"**, which solve the environment after **451 episodes**. It is surprisingly that the **"Baseline DQN with he-initialisation"** is on the last place as this has the best initialisation for the relu activation function. This network solve the environment after **500 episodes**.

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 double DQN and compare the current results.
    - Implementing the prioritized experience replay for the DQN and compare the current results.
    - Implementing and analyse reward clipping of the range from -1 to + 1 https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
    - Implementing and using better RL algorithm such as A2C, A3C, DDPG, PPO, OC, C51

- 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 DGN 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 DQN can be found in the following links.

- Mnih, V. et al. Human-level control through deep reinforcement learning. Nature 518, 529–542 (2015) https://www.nature.com/articles/nature14236?rel=mas.
- Riedmiller, Martin. "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method." European Conference on Machine Learning. Springer, Berlin, Heidelberg, 2005. http://ml.informatik.uni-freiburg.de/former/_media/publications/rieecml05.pdf

- Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature518.7540 (2015): 529.http://www.davidqiu.com:8888/research/nature14236.pdf
- Hado van Hasselt, Arthur Guez, David Silver "Deep Reinforcement Learning with Double Q-learning" arXiv. https://arxiv.org/abs/1509.06461
- Tom Schaul, John Quan, Ioannis Antonoglou, David Silver "Prioritized Experience Replay" arXiv. https://arxiv.org/abs/1511.05952
- Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas "Dueling Network Architectures for Deep Reinforcement Learning" arXiv. https://arxiv.org/abs/1511.06581
- Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, Koray Kavukcuoglu "Asynchronous Methods for Deep Reinforcement Learning" arXiv. https://arxiv.org/abs/1602.01783
- Marc G. Bellemare, Will Dabney, Rémi Munos "A Distributional Perspective on Reinforcement Learning" arXiv. https://arxiv.org/abs/1707.06887
- Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, Shane Legg "Noisy Networks for Exploration" arXiv. https://arxiv.org/abs/1706.10295
- Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver "Rainbow: Combining Improvements in Deep Reinforcement Learning" arXiv. https://arxiv.org/abs/1511.06581