Udacity Deep Reinforcemente Learning Nanodegree Navigation (Banana) Project

Udacityuser4 (udacityuser4@seat.es)
March 29th, 2020

Abstract

The goal of this project is training an agent to navigate a large world and collect yellow bananas, while avoiding blue bananas. For this purpose I'll examine the DQN algorithms applied to the Unity Banana Collection Environment. The algorithms included are DQN, Double DQN, Dueling DQN and Prioritized Experience Replay. I have analyzed the performance of the agent following those algorithms and finally I propose some ideas that could be explored to improve this project.

1. DEFINITION

This project uses a Unity Environment. This environment is a flat plain with several bananas spread all over the ground. There are two kind of bananas: yellow bananas and blue bananas. The goal of this project is to collect yellow bananas and avoid blue bananas.
A reward of +1 is given for collecting a yellow banana and a reward of a -1 is given for collecting a blue banana.

In our Banana World, there are 4 available actions:

0 – Move Forward
1 – Move Backward
2 – Move Left
3 – Move Right

And the state space has 37 dimensions.
It is considered as solved when an average reward of +13 for 100 episodes is reached.

2. LEARNING ALGORITHMS

The **deep Q-network (DQN)** algorithm introduced by Mnih et al. (2015) is able to get strong performance in a variety of ATARI games, directly by learning from the pixels. And using the same algorithm for the several games it obtained a really good performance.
Rewards are clipped between +1 and -1 and that makes it easier to use the same learning rate across multiple games.

Deep Q-Learning is a kind of Reinforcement Learning algorithm that uses Neural Networks to approximate the optimal policy for achieving the maximum reward in the given environment.
Experiences (state, action, reward and next state) are added to the memory as the agent interacts with the environment to be sampled and learned from it.

The operations in **Deep Q-learning** uses the same values both to select and evaluate an action. This makes it more likely to select overestimated values in case of inaccuracy or noise.

To avoid this, in a **Double DQN** the target value is separate from the Q-learning values, to improve stability and performance. The target network with weights is used to evaluate the current greedy action. The policy is still being chosen according to the values obtained by the current weights.

In **Dueling Network Architecture** (Wang et al., 2015), the neural network architecture decouples the value and advantage function, which leads to improved performance. The learning update is done as in DQN and it is only the structure of the neural network that is modified.

Prioritized Experience Replay allows us to give priorities that are proportional to the TD Error, so we can sample from memory based on this priority. Then the agent replays experiences that it could learn more from, and converge to the optimal solution faster.

3. HYPERPARAMETERS

These are the hyperparameters that I have used in the implementation of the Network Architectures previously mentioned.

BUFFER SIZE	100.000
BATCH SIZE	32
GAMMA (discount factor)	0.99
TAU (soft update of target parameters)	0.001
LEARNING RATE	0.0005
UPDATE_RATE	Every 4 steps

Table 1: Hyperparameters

4. RESULTS

4.2 Interpretation

Every algorithm has been executed until it reaches the average reward above 13. That means that the environment has been solved.

These are the results:

Double DQN Episodes until Resolved (Average Reward>13)	**Dueling**		**Prioritized Experience Replay**	
	OFF	ON	OFF	ON
300	X		X	
677		X	X	
1107	X			X
1477		X		X

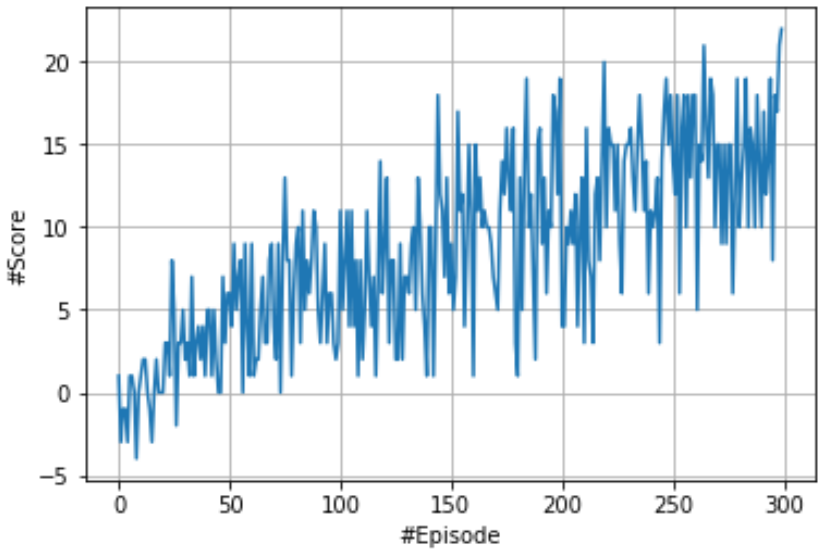
These results have surprised me a lot, as I thought I would obtain better results with the Dueling and Prioritized Experience Replay options activated. But it was simply worse.

So in this case, it seems to be the easier the algorithm, the better performance. Or maybe I should have tested much more changing the hyperparameters.

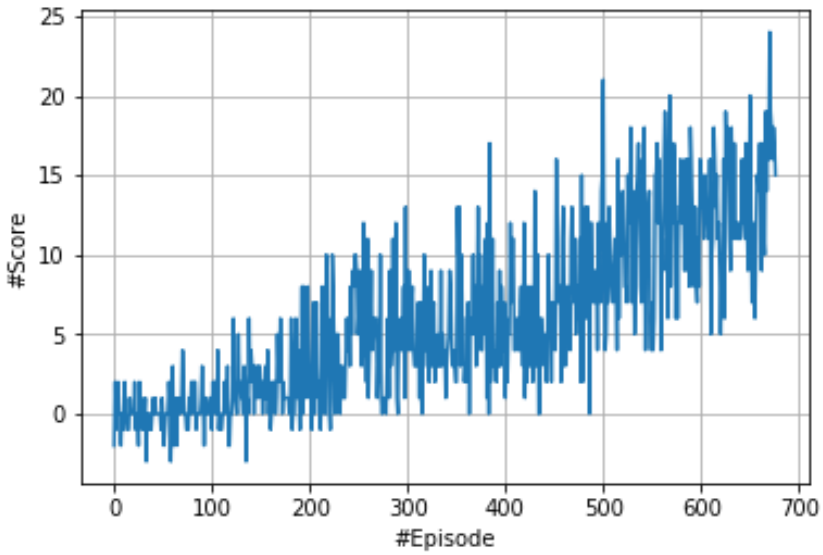
2.2 Plot of rewards

These are the plot of rewards that have been obtained in every combination of algorithms.

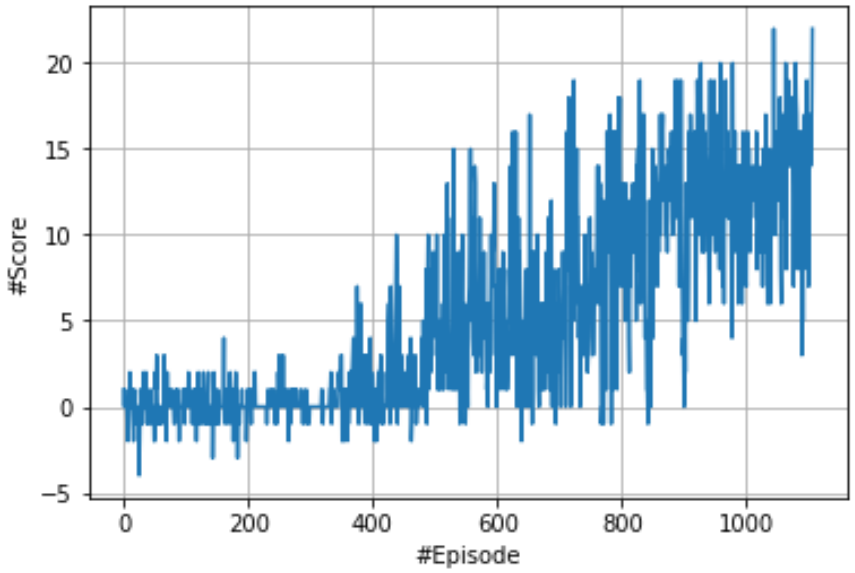
DQN
(Environment solved in 300 episodes!)



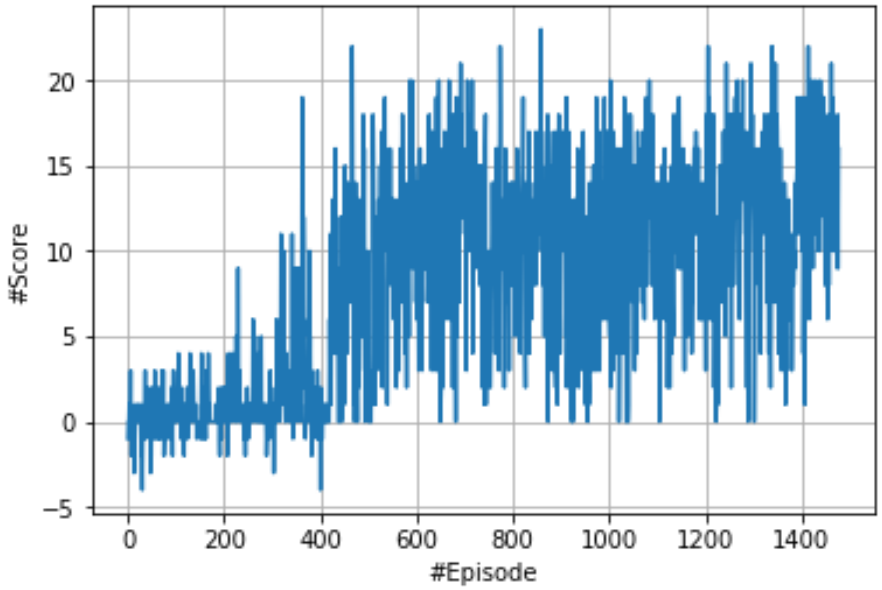
Dueling DQN and not Prioritized
(Environment solved in 667 episodes!)



Not Duel and Prioritized
(Environment solved in 1107 episodes!)



Duel and Prioritized
(Environment solved in 1477 episodes!)



2.3 Neural Network

The environment has 1 agent, 4 possible actions and the states' length is 37.
With that situation, I have designed a Q-network with 3 hidden layers of 64 neurons (after trying with one of 256 neurons), plus the final layer with dimension= 4 (dimension of the action size) and I have used the RELU activation function.

```
#NN_SIZE = 256  # Neural Network SIZE
NN_SIZE = 64  # Neural Network SIZE

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

    def __init__(self, state_size, action_size, seed, fc1_units=NN_SIZE, fc2_units=NN_SIZE, fc3=NN_SIZE, fc4=NN_SIZE):

        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            the 4 Linear Layers of the NN
        """
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1_units = nn.Linear(state_size, fc1_units)
        self.fc2_units = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, fc3)
        self.fc4 = nn.Linear(fc3,fc4)
        self.fc5 = nn.Linear(fc4,action_size)
```

For the Dueling Q-Network, I have designed a de-coupled Q-Network with 2 parallel networks of 3 hidden layers of 128 and one output layer each one. One network gives as a result a single value V that is the state value. The other network gives as a result several values (exactly the number of values is equal to the number of possible actions of the environment, in this case this number is equal to 4).
This represents the advantage function value for each action. At the end the two output layers are combined together to form the final output layer, following the mathematical lineal function: *Dueled_value = value +advantage – advantage.mean()*

```
class DuelingQNetwork(nn.Module):
    """Actor (Policy) Model."""
    def __init__(self, state_size, action_size, seed, fc1_units=NN_SIZE, fc2_units=NN_SIZE, fc3=NN_SIZE, fc4=NN_SIZE):

        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            the 4 Linear Layers of the NN
        """

        super(DuelingQNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1_units = nn.Linear(state_size, fc1_units)
        self.fc2_units = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, fc3)
        self.fc4 = nn.Linear(fc3,fc4)

        self.fc5 = nn.Linear(fc4,NN_SIZE//2)
        self.fc6 = nn.Linear(NN_SIZE//2,action_size)

        self.fc7 = nn.Linear(fc4,NN_SIZE//2)
        self.fc8 = nn.Linear(NN_SIZE//2,1)
```

```
def forward(self,state):
    x = F.relu(self.fc1_units(state))
    x = F.relu(self.fc2_units(x))
    x = F.relu(self.fc3(x))
    x = F.relu(self.fc4(x))


    #State values
    value = self.fc8(F.relu(self.fc7(x)))

    #Advantage values
    advantage = self.fc6(F.relu(self.fc5(x)))


    #calculate dueled_value
    dueled_value = value + advantage - advantage.mean()

    return dueled_value
```

5. IDEAS FOR IMPROVEMENT

There are several possibilities to evolution this project that I will list:

1. One of the possibilities could be to train the agent directly from the environment's observed raw pixels, instead of using the 37 internal states. After some initial pre-processing of the data (rescaling the image size, converting it to RGB colours, etc...) we could apply a Convolutional Neural Network to the input image, so that the agent could learn.
2. Another enhancements could be use some kind of Neural Network to get the best combinations and values for the hyperparameters.
3. Other improvement would be to activate and deactivate the replay buffer, depending of the evolution of the agent
4. Another one could be to use the Rainbow model. This model combines several improvements combining different algorithms.
5. Study the optimization of the code for the Prioritize Experience Replay, as I find it is really slow.