Navigation Project Report

Yasmeen Khaled
May 17, 2021

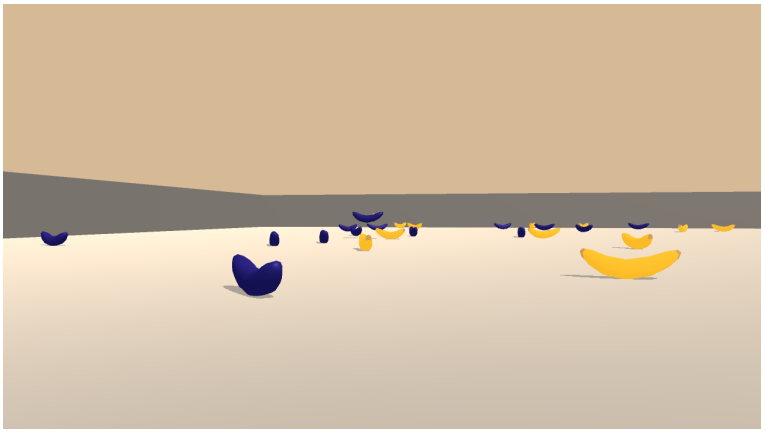

Figure 0.1: Banana Environment

1 INTRODUCTION

Deep Reinforcement Learning has witnessed a breakthrough during the last couple of years with DeepMind introducing DQN which managed to beat human experts in Atari games. The goal for any RL agent is to maximize its expected reward upon interaction with an environment. The idea behind the usage of neural networks with RL is that what we after is the true value function which is continuous on the entire state space, but this is infeasible to capture except for some very simple problems, that's why an approximate value function is the target. Neural networks acts as function approximators for this task. The environment used for this project

he Udacity version of the Banana Collector environment. The goal of the agent is to collect as many yellow bananas while avoiding blue bananas. It is an episodic task, where encountering a yellow banana results in a reward of +1, and encountering a blue banana results in a reward of -1. The state space dimension is 37 and contains the agent's velocity, along with a ray-based perception of objects around agent's forward direction. While, the action space size is 4 and it is explained as follows:

- 0 - move forward.
- 1 - move backward.
- 2 - turn left.
- 3 - turn right.

2 Learning Algorithm

Two learning algorithms were used to train 2 models for comparison:

1. Vanilla DQN
2. Dueling DQN

The project hierarchy is a model.py file which contain the two used network models, a dqn_agent.py file where the step, act and other agent related methods are defined, and in the init method the agent initializes the networks according to which model is chosen, and a Navigation notebook in which the agent is trained on both types of network types and the results are shown. The hyperparameters used for training agents were as follows:

Hyperparameter	Description	Value
BUFFER_SIZE	replay buffer size	int(1e5)
BATCH_SIZE	minibatch size	64
GAMMA	discount factor	0.995
TAU	for soft update of target parameters	1e-3
LR	Learning rate	5e-4
UPDATE_EVERY	how often to update the network	4

The dqn_agent is implemented as follows:

- **init**: the agent state_size and action_size as well the type of network model we want to train the agent on ("Vanilla" or "Dueling") is passed to the constructor and all the values are initialized. If the model is "Vanilla" a qnetwork_local and qnetwork_target are initiallized with the Vanilla QNetwork, otherwise they are initialized with DuelQNetwork. It also initialized the replay buffer and step size.
- **step**: it saves the experience in the replay buffer, and if enough samples are existing in the memory, the agents samples from the buffer and learns every update_every times.

- **act**: returns the action picked based on an epsilon greedy policy.
- **learn**: it calculates the loss between between Q_target and Q_expected, optimizes the weights, do a backward pass and updates the target network.
- **soft_update**: the learn method updates the target network using this method which updates the target network based on the values of the local network.

The dqn_agent.py also contains the ReplayBuffer class which has the following methods:

- **add**: which adds an experience to the replay buffer memory which is inititialized as a deque with maxlen of BUFFER_SIZE.
- **sample**: which randomly samples a batch from the replay buffer memory.

2.1 Vanilla DQN

The first algorithm used is the Vanilla DQN following DeepMind research paper, the algorithm goes as follows:

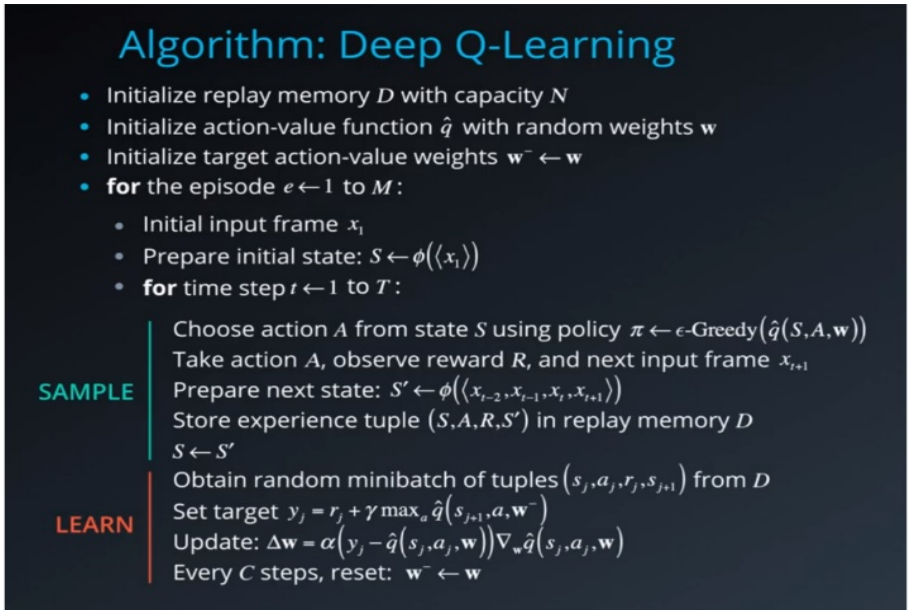

Figure 2.1: Vanilla DQN Algorithm

The network architecture was as follows:

```
QNetwork(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=4, bias=True)
)
```

The weights are initialized using xavier_uniform_ weight initialization, and fc1 is passed to RELU activator then the output is passed to fc2 then another RELU activator in the forward pass.

2.2 Dueling DQN

The second model uses the Dueling DQN architecture which is explained in the following research paper. The Dueling DQN separates the results of the conv layers(in case of using the real pixels) into two streams, one that estimates the state value function and one that estimates the advantage of taking each action in that state. The intuition is that the values of most states do not vary that much across actions, so it makes sense to try and estimate them directly. But the difference that actions make in each state is still needed to be captured.

```
DuelQNetwork(
  (fc1): Linear(in_features=37, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=64, bias=True)
  (Value): Linear(in_features=64, out_features=1, bias=True)
  (Advantage): Linear(in_features=64, out_features=4, bias=True)
```

) The weights are initialized using xavier_uniform_ weight initialization,and fc1 is passed to RELU activator then the output is passed to fc2 then another RELU activator in the forward pass. The output is then passed to both Value and Advantage.

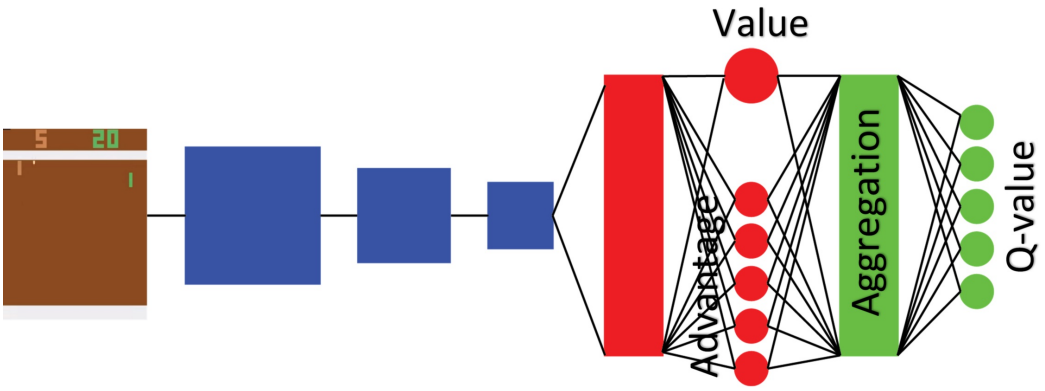

Figure 2.2: Dueling DQN Architecture

To aggregate the value and the advantage in order to get the estimated Q-Value , we add the Value to the Advantage and the subtract the mean of the advantages over the the action values.

3 Training and Results

3.1 Vanilla DQN

```
Episode 100     Average Score: 0.31
Episode 200     Average Score: 4.02
Episode 300     Average Score: 7.85
Episode 400     Average Score: 10.09
Episode 500     Average Score: 12.65
Episode 600     Average Score: 13.74
Episode 620     Average Score: 14.02
Environment solved in 520 episodes!     Average Score: 14.02

CPU times: user 5min 24s, sys: 26.9 s, total: 5min 51s
Wall time: 8min 38s
```

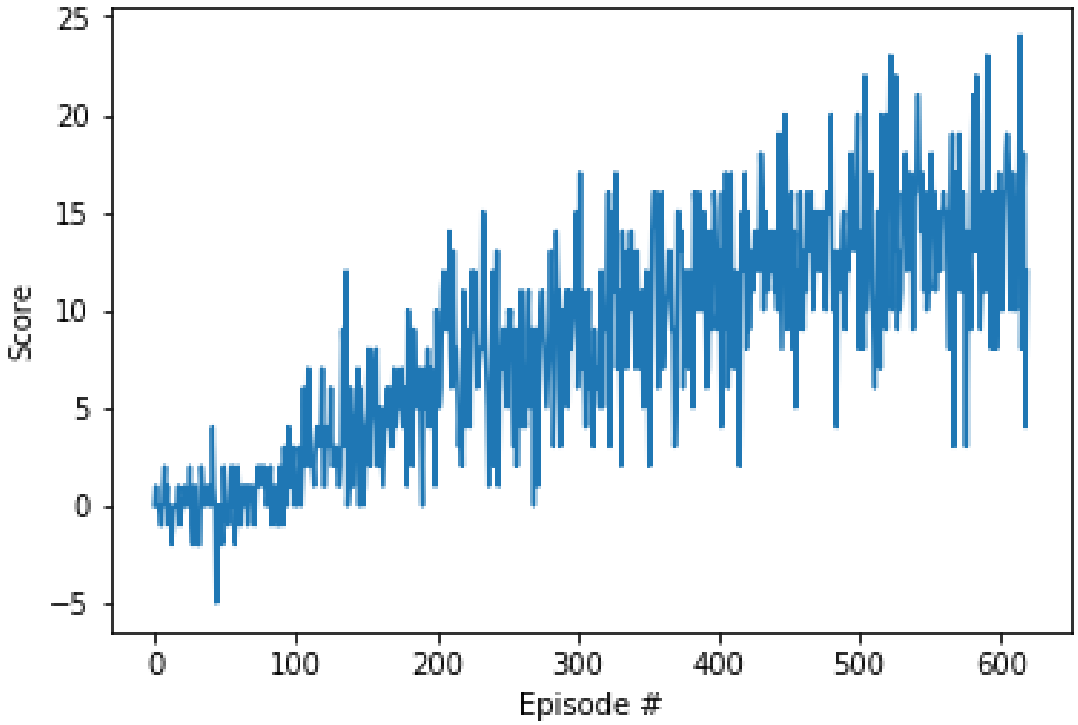

Figure 3.1: Vanilla DQN Training

My Vanilla DQN agent managed to solve the task in 520 episodes to be able to get a reward of at least +13 over 100 consecutive episodes.

```
Episode 100     Average Score: 0.80
Episode 200     Average Score: 4.59
Episode 300     Average Score: 8.10
Episode 400     Average Score: 9.82
Episode 500     Average Score: 11.90
Episode 600     Average Score: 13.17
Episode 676     Average Score: 13.87
Environment solved in 576 episodes!     Average Score: 13.87

CPU times: user 6min 55s, sys: 36.1 s, total: 7min 31s
Wall time: 10min 31s
```

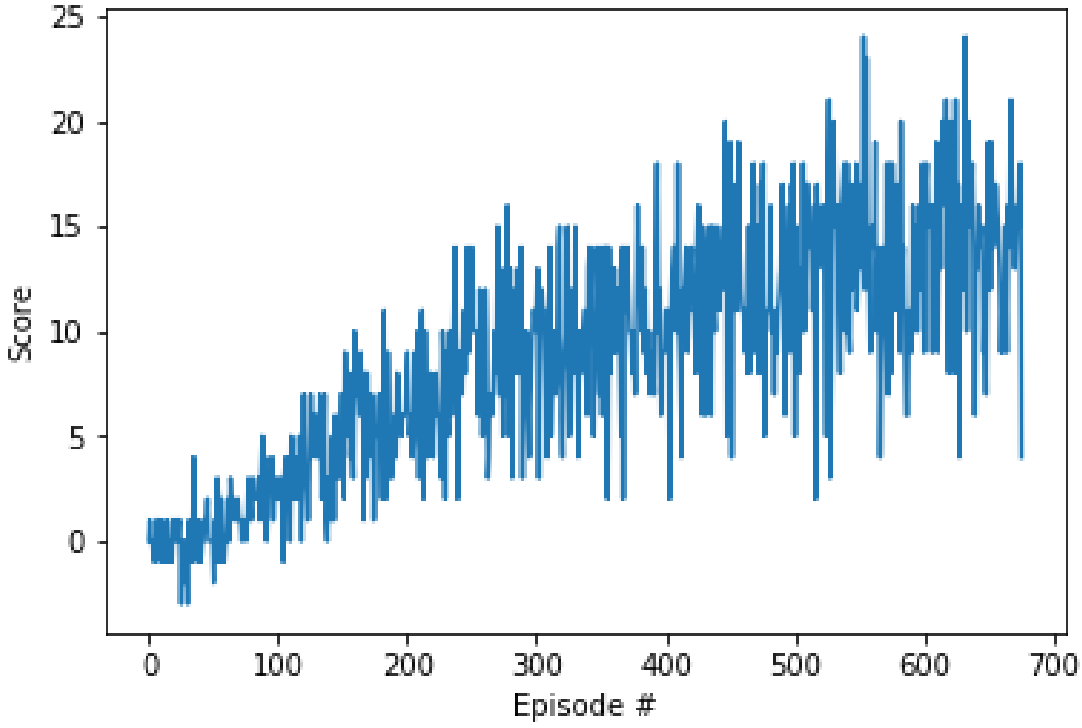

Figure 3.2: Dueling DQN Training

My Dueling DQN agent managed to solve the task in 576 episodes to be able to get a reward of at least +13 over 100 consecutive episodes.

3.3 Comparison

The two trained models were then loaded and they were both run for 5 episodes and the average of the runs was calculated:

- Vanilla DQN :

```
Score : 18.0
Score : 6.0
Score : 8.0
Score : 3.0
Score : 14.0
Average Score for 5 runs: 9.8
```

- Dueling DQN

```
Score : 2.0
Score : 4.0
Score : 14.0
Score : 13.0
Score : 9.0
Average Score for 5 runs: 8.4
```

4 Future Work

To further improve the performance of the model, the current methods are proposed to do in the future:

- **Prioritized Experience Replay**: instead of using random sampling of replay buffer, we can uses prioritized experience replay which is based on the idea that the agent can learn better from some experiences than the other.
- **Hyper-parameter search**: one can try to train multiple agents with different parameters in order to reach the set that significantly improve the performance.
- **Rainbow DQN**: one can try to implement Rainbow DQN which combines different DQN ideas in one to reach optimal performance.