"NAVIGATION AGENT"

by

Ruben Kapp

A project submitted in partial fulfillment
of the requirements for the degree of

Deep Reinforcement Learning
Nanodegree

UDACITY

2020

UDACITY

ABSTRACT

Navigation Agent

by Ruben Kapp

A report about the project how to use reinforcement learning to train an agent to navigate and collect bananas in a square virtual environment. The agent learns how to navigate by receiving +1 reward for collecting yellow bananas and -1 reward for collecting blue bananas. A double Q-Agent with *experience reply* and *fixed q-targets* was implemented and successfully trained to reach the target score. Methods and problems deep Q-Learning are described and results shown. Finally, some methods for improvements will be given.

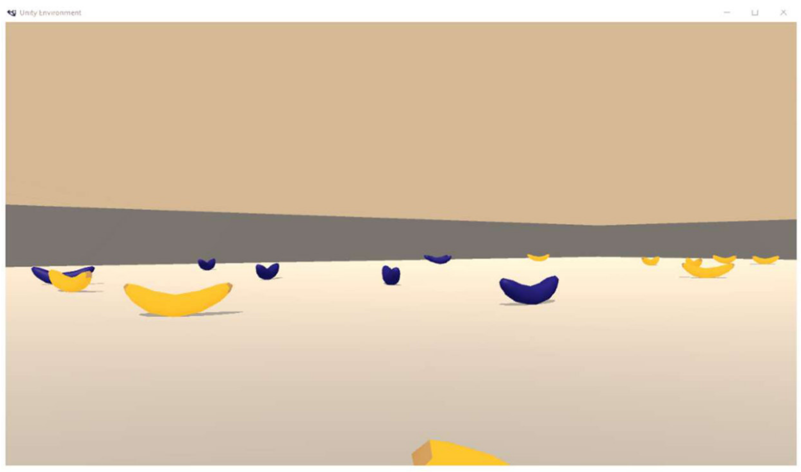

TABLE OF CONTENTS

Content

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to UDACITY and the whole AI community. In addition, special thanks to https://stackoverflow.com/ and all sources which have been very helpful during the early programming phase and understanding of this undertaking.

NAVIGATION PROBLEM AND CHALLENGES

Navigation Problem:

We are using a Deep Q Learning algorithm to train or agent to collect only yellow bananas and avoid the blue ones in square virtual environment.

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 – move left
- 3 – move right

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

Q-Learning challenges:

Within the Deep Q-Learing algorithm the optimal action-value function is represented by a neural network and not by a table. Using a neural network brings the side effects that it is notoriously unstable when using reinforcement learning. Many of these problems arise as the action-value representation in a neural network is global and not local, hence a weight change induced by a update for some state space might negatively influence the action-value representation and therefore destroy the learning effort done so far [Riedmiller, 2005].

Another problem is called *moving targets*. Deep Q Networks take as input the state of the environment and output a Q value for each possible action. The maximum Q value determines, which action the agent will perform. The training of the agents uses as loss the TD Error (Temporal Difference), which is the difference between the maximum possible value for the next state and the current prediction of the Q-value (as the Bellman equation suggests). As a result, we manage to approximate the Q-tables using a Neural Network. The TD Error is the Q-Target and it is calculated as the immediate reward plus the discounted max Q-value for the next state. When we train our agent, we update the weights accordingly to the TD Error. But the same weights apply to both the target and the predicted value. We move the output closer to the target, but we also move the target. So, we end up chasing the target and we get a highly oscillated training process [Karagiannakos, 2018].

A third problem with Deep Q Networks used for RL is the tendency to overestimate the value and the action-value functions, as shown by Thrun and Schwartz, 1993. This problem is known as the *maximization bias*, it occurs during the training if for some reason a q value was overestimated and therefore chosen as the go-to action. Next turn this overrated value will be used as a target value. Now how to evaluate if the selected action was the actual best action?

In this report we will apply two methods to deal with this instability and overestimation problems, namely *experience reply* and *fixed q-targets* via Double Q-Agent.

METHODS

The article by Riedmiller 2005 introduced the training of the acute value function represented by a multilayer perceptron, to achieve robust learning they introduced *Neural Fitted Q Iteration* (NFQ), a memory-based method to train Q-value functions based on multi-layer perceptrons. By storing and reusing all transition experiences, the neural learning process can be made more data efficient and reliable. Also, by rolling over the stored data via replay pool more efficient batch and supervised learning techniques can be applied. By using the stored data as a replay pool, the behavior distribution is average over many of its previous states, smoothing learning and avoid oscillations. This has the benefit that each step of the experience may be used in many weight updates. The whole technic is known as *experience reply*.

Now to deal with *moving target* and *maximization bias* we use a second Deep Q-Network. One as the main Deep Q Network and second network called *target network* to update only periodically. Thus, the weights of the target are fixed and only updated periodically. Hence, we decouple the prediction and target and get better convergence, this technic is called *fixed q-targets*. Furthermore, we solve the problem with overestimation of q values as we use one for the prediction of the next action and the other (the target network) to evaluate the chosen action. This concept was introduced by DeepMined [Hasselt, Guez & Silve, 2016] and is called *Double Deep Q Network*.

Summary Experience Replay

When the agent interacts with the environment, the sequence of experience tuples can be highly correlated. The naive Q-learning algorithm that learns from each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation. By instead keeping track of a replay buffer and using experience replay to sample from the buffer at random, we can prevent action values from oscillating or diverging catastrophically. In addition, we can roll over rare experiences several times and reach better generalization.

The replay buffer contains a collection of experience tuples (S, A, R, S'). The tuples are gradually added to the buffer as we are interacting with the environment.

Summary Fixed Q-Targets (Double DQN)

In Q-Learning, we update target and the predicted value with the same network, and this can potentially lead to harmful correlations. To avoid this, we can update the weight **w** in the network $\hat{q}$ to better approximate the action value corresponding to state **S** and action **A** with the following update rule:

$$\Delta w = \alpha \cdot \overbrace{(\underbrace{R + \gamma \max_a \hat{q}(S', a, w^-)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{old value}})}^{\text{TD error}} \nabla_w \hat{q}(S, A, w)$$

where **w**⁻ are the weights of a separate target network that are not changed during the learning step, and *(S, A, R, S')* is an experience tuple, with R corresponding to state reward and S' to the next state. The discount factor is represented by y.

Q-Network architecture

As Q-Network we use a three-layer neural network with 37 neurons as input level (fc1), as the environments is represented via 37 features. The second hidden layer (fc2) has 150 hidden neurons and the output level (fc3) has 4 output neurons as the action values.

```
QNetwork(
        (fc1): Linear(in_features=37, out_features=150, bias=True)
        (fc2): Linear(in_features=150, out_features=150, bias=True)
        (fc3): Linear(in_features=150, out_features=4, bias=True)
        )
```

Hyperparameters

Hyperparameter of the Double Q-Network with replay buffer.

Replay buffer size:
- BUFFER_SIZE = int(1e5)

Minibatch size:
- BATCH_SIZE = 64

Discount factor for update rule:
- GAMMA = 0.99

As optimization algorithm we use the pytorch implementation of the adam optizer [ADAM].

Learning rate for ADAM optimizer:
- LR = 1e-4

How often to update the main-network:
- UPDATE_EVERY = 4

How often to update the target-network:
- UPDATE_TARGET_EVERY = 110

Furthermore, we implemented epsilon greedy policy with a decreasing epsilon. Epsilon greedy will lead to a better exploration at the beginning gathering more experience and later on to a better exploitation.

Start with epsilon:
- EPS_START = 1.0

End with epsilon:
- EPS_END = 0.01

Decay epsilon with factor:
- EPS_DECAY = 0.995

RESULTS

The goal was to reach an average score of 13 over the last 100 episodes. Using the given model and hyperparameter or agent reached the target score after 550 training episodes (see Figure 1).

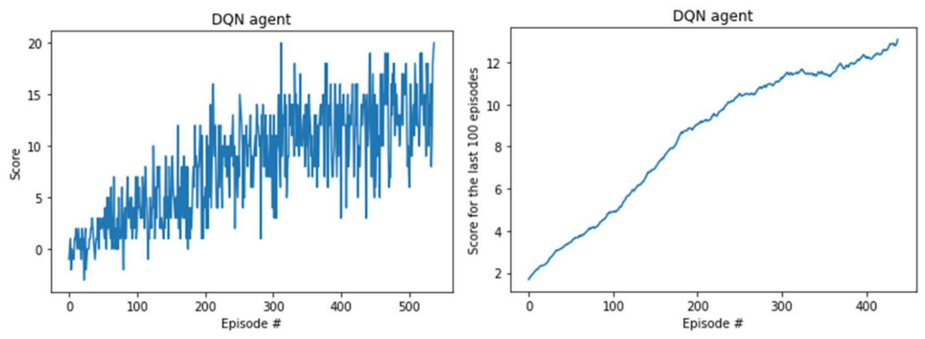

Figure 1, Left: Agent score each episode. *Right:* Average score over last 100 episodes

To see if we could reach even better results, we trained the agent for 1400 episodes. The Agent did improve further and reached a to score of around 15 after 1200 episodes (see Figure 2).

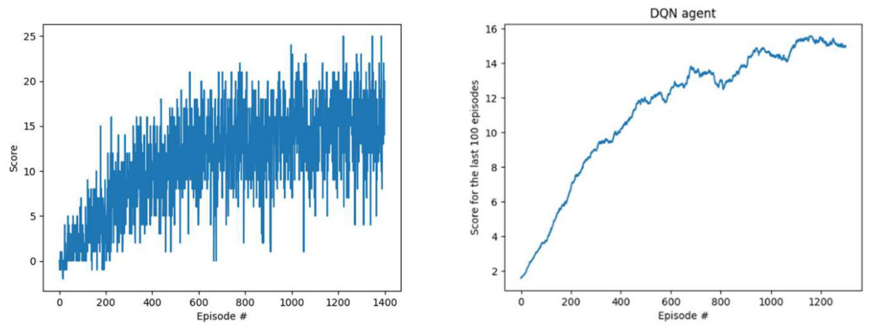

Figure 1, Left: Agent score each episode. *Right:* Average score over last 100 episodes. Trained over 1400 episodes leading to an average max score of around 15.

IMPROVEMENTS

Prioritized Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. Prioritizing experience leads to replay important transitions more frequently, and therefore learn more efficiently. Deep Q-Learning with prioritized experience replay achieves a new state-of-the-art, outperforming DQN with uniform replay on 41 out of 49 games [Schaul, Quan, Antonoglou & Silver, 2016].

Dueling DQN

Currently, in order to determine which states are (or are not) valuable, we have to estimate the corresponding action values for each action. However, by replacing the traditional Deep Q-Network architecture with a dueling architecture, we can assess the value of each state, without having to learn the effect of each action. A dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. The results show this architecture leads to better policy evaluation in the presence of many similar-valued actions [Wang, Schaul, Hessel, van Hasselt, Lanctot & de Freitas, 2016].

SOURCES & LITERATURE

[Riedmiller, 2005]
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

[Volodymyr, et al., 2015]
Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature518.7540 (2015): 529. http://www.davidqiu.com:8888/research/nature14236.pdf

[Thrun & Schwartz,1993]
Sebastian Thrun, Anton Schwartz, "Issues in Using Function Approximation for Reinforcement Learning", Proceedings of the Fourth Connectionist Models Summer School, 1993, https://www.ri.cmu.edu/pub_files/pub1/thrun_sebastian_1993_1/thrun_sebastian_1993_1.pdf

[Hasselt, Guez & Silve, 2016]
Hado van Hasselt, Arthur Guez, David Silver, Deep "Reinforcement Learning with Double Q-learning", AAAI 2016, https://arxiv.org/abs/1509.06461

[Schaul, Quan, Antonoglou & Silver, 2016]
Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, "Prioritized Experience Replay", 2016, https://arxiv.org/abs/1511.05952

[Wang, Schaul, Hessel, van Hasselt, Lanctot & de Freitas, 2016]
Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas, „Dueling Network Architectures for Deep Reinforcement Learning", 2016, https://arxiv.org/abs/1511.06581

[Karagiannakos, 2018]
Sergios Karagiannakos, 2018 "Q-targets, Double DQN and Dueling DQN (Taking Deep Q Networks a step further)"
https://theaisummer.com/Taking_Deep_Q_Networks_a_step_further/

[Simonini, 2020]
Thomas Simonini
https://cugtyt.github.io/blog/rl-notes/201807201658.html

[ADAM]
ADAM torch.optim, https://pytorch.org/docs/stable/optim.html