Project 1 - Navigation

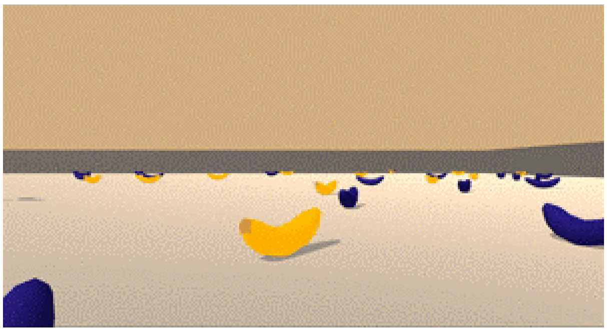
Deep Reinforcement Learning Nanodegree

M. Lippl

March 19, 2021

1 Introduction

The challenge of this project is to develop and train a reinforcement learning agent to navigate a room with randomly distributed bananas and pick an many of the yellow (or "good") ones as possible, while avoiding the dark blue ("bad") ones. Upon collecting the former, the agent receives a score of +1 whereas the collection of the latter yields a score of -1.



One episode consists of 300 consecutive steps, and at each step the agent is controlled through 4 discrete actions, namely forward ('0'), backward ('1'), left ('2'), and right ('3'). The scenario is considered solved if the agent reaches an averaged score of +13 over 100 consecutive episodes. In formulæ, the challenge is considered solved at the i^{th} episode if we have $\text{score}_i > +13$ where

$$\text{score}_i = \frac{1}{100} \sum_{j=0}^{99} s_{i-j}$$

and s_k is the score of the k^{th} episode.

2 Methodology

The learning consists of two nested loops, the outer iterating over the number of episodes which is theoretically limitless, the inner loop over the number of steps which are limited to 300 for this project.

- Deep Q-Networks [1]
- Double DQNs [1]

2.1 Q-Learning

Before we discuss Deep Reinforcement Learning, we take a quick glance over a reinforcement learning technique which is sometimes called tabular learning. For more details on this and the fundamentals of reinforcement learning, cf. [2].

The action-value-function Q with respect to a policy π is intuitively defined as the expected reward when being in the state s and taking the action a.

2.2 Monte-Carlo Learning

Monte-Carlo Learning works by choosing randomly a policy π, starting with a state-action-pair (s, a), and following the policy until the episode is finished. At the end, the total reward, denoted by G_t, is summed up with regard to an discounting factor $0 < \gamma < 1$, i.e.

$$G_t = \sum_{i=0}^{N} \gamma^i R_i$$

where R_i is the reward after the i^{th} step and the episode is terminated after the N^{th} step. With a learning rate set to $0 < \alpha < 1$, the action-value function at (s, a) is then updated as

$$Q_\pi(s, a) \longleftarrow Q_\pi(s, a) + \alpha\big(G_t - Q_\pi(s, a)\big).$$

2.3 Temporal-Difference Learning (TD-Learning)

In contrast to the previous method, in Temporal-Difference Learning (TD-Learning) is using a bootstraping *ansatz* which means that instead of simulating a whole episode, the expected value of total rewards is approximated by using the the immediate reward after (s, a), $r(s, a)$, and the value of the Q-function at the next state-action pair (s', a') where $s' = \pi(s, a)$ and a' is chosen according to the policy π. This sum is called the TD-target. Consequently, the update equation for the Q-value at (s, a) is given by

$$Q_\pi(s, a) \longleftarrow Q_\pi(s, a) + \alpha\Big(\underbrace{r(s, a) + Q_\pi(s', a')}_{\text{TD-target}} - Q_\pi(s, a)\Big).$$

A particular special case is the strategy called *SARSA-max* or *Q-Learning* in which the update equation is defined as follows:

$$Q_\pi(s, a) \longleftarrow Q_\pi(s, a) + \alpha\Big(r(s, a) + \max_{a'} Q_\pi(s', a') - Q_\pi(s, a)\Big)$$

Here, the action of the "next step" (s', a') is selected such that the Q-value is maximized. This update equation is also the starting point for the application using artificial neural networks which we will discuss next.

2.4 Deep Q-Learning

The basic idea of Deep Q-Learning involves modeling the Q-table from Q-Learning by a neural network. Let (s, a, r, s', a'), where s' and a' are the next steps, respectively. Let θ be the collection of weights of the network. Since DQL incorporates two different networks, we designate the main ("local") on by θ, whereas, θ^- denotes the weights of the target network. The approximates local Q-function is given by $Q(s, a; \theta)$, the target Q-function by $Q(s, a; \theta^-)$. If we define the loss function by

$$\mathcal{L}(\theta) = \frac{1}{2}\Big(r(s,a) + Q(s', a', \theta^-) - Q(s, a, \theta)\Big)^2$$

and let the network learn using the backpropagation algorithm, this results in the update equation

$$\theta \longleftarrow \theta + \alpha\Big(r(s,a) + \max_{a'} Q(s', a', \theta^-) - Q(s, a, \theta)\Big) \cdot \nabla_\theta Q(s, a, \theta).$$

Unfortunately, updating the ANN as new experiences are made by the agent leads to multiple problems. For example, there are inherent correlations between subsequent states among leads to poor learning results using a naïve implementation of Q-Learning using ANNs. The solution to this problem is the use of fixed Q-targets. On the other hand, incoming experiences are only used once and are immediately discarded. To solve the latter problem, the use of an experience buffer in which experience tuples (s, a, r, s', a') are inserted and after randomly sampled batches.

1. Fixed Q-targets
2. Experience Replay
3. ϵ-greedy Policy

The ϵ-greedy policy introduces a parameter $0 < \epsilon < 1$ to deal with the exploration-exploitation-problem. At each turn, with a probability less than 1, the agent selects a completely random action in order to generate new experiences. In practice, this parameter is dealt with dynamically during training, for example starting at $\epsilon = 1$ and then letting the parameter decrease exponentially.

2.5 Double Deep Q-Learning

Double DQL distinguished itself from DQL by changing the next action a' from a simple maximizer of the target Q-function to the maximizer of the local Q-function, i.e.

$$a' = \arg\max_a Q(s', a, \theta)$$

Hence the update step for the network needs to be modified to

$$\theta \longleftarrow \theta + \alpha\Bigg(r(s,a) + Q(s', \arg\max_a Q(s', a, \theta), \theta^-) - Q(s, a, \theta)\Bigg) \cdot \nabla_\theta Q(s, a, \theta)$$

It has been shown that Double DQL leads to higher convergence rates compared to (simple) DQL [1].

2.6 Soft Updates

Another method to speed up convergence is the use of soft-updates. Here, in contrast to updates where the whole network is copied to the target network after a pre-defined interval of steps, the target update is modified by a interpolation, i.e.

$$\theta^- \longleftarrow \tau \cdot \theta + (1 - \tau) \cdot \theta^-$$

where $0 < \tau < 1$ is a small number. It has been shown that soft-updates lead to better convergence, hence we have

3 Implementation and Results

3.1 Structure of the Project

As the previous projects before, our project has the following structure:

```
udacity-rl-p1/
 |__checkpoints_torch/
 |__Navigation.ipynb
 |__agent_torch.py
 |__network_torch.py
 |__train.py
 |__play.py
```

The Jupyter-notebook is self-contained and contains the results of the last numerical experiment. Since the bulk of the development has been done in the bash shell on a unix system, the main training loop and hyperparameters are also contained in the separate script called `train.py`. The file `play.py` contains a script which enables the replay of the Agent in the environment. The agent itself is defined in the `Agent`-class and is contained in `agent_torch.py`.[1] The file `network_torch.py` contains the definition of the ANN.

3.2 Architecture

The architecture of the Q-network is comparitively simple and is sketched in figure 1.

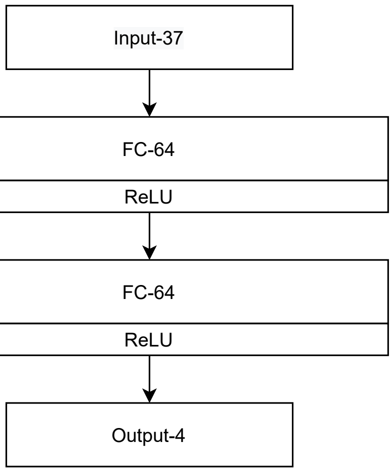


Figure 1: The network architecture used in the implementation

Notice that the discrete action variable is not part of the input layer but parametrizes the four output neurons. This is in contrast to the subsequent projects (e.g. Continuous Control) where (when using the so-called actor-critic method) *a* is part of the critic's input layer.

[1] Note that the suffix `_torch` stems from the fact that there have been additional efforts to implement the algorithms using TensorFlow 2.x instead of PyTorch.

3.3 Hyperparameters

The hyperparameters were as follows:

Hyperparameter	Value	Description/Comment
`BUFFER_SIZE`	10,000	Total size of replay buffer
`BATCH_SIZE`	64	Size of training batches
`GAMMA`	0.99	Discounting factor
`TAU`	0.001	Soft updates
`learn_rate`	$5 \cdot 10^{-4}$	Learning rate of Adam-optimizer
`UPDATE_EVERY`	4	Update cycle during training
`seed`	2	Manual seed for neural networks
`epsilon`	0.02	Final value during training

3.4 Results

The code and corresponding results of our experiments can be found in the GitHub-repository `https://github.com/asiopueo/udacity-rl-p1/Navigation.ipynb`. The eventually successful implementation was using Double DQL.

As indicated in the following graph and logged in the Jupyter-notebook, the game has been solved after 552 episodes.

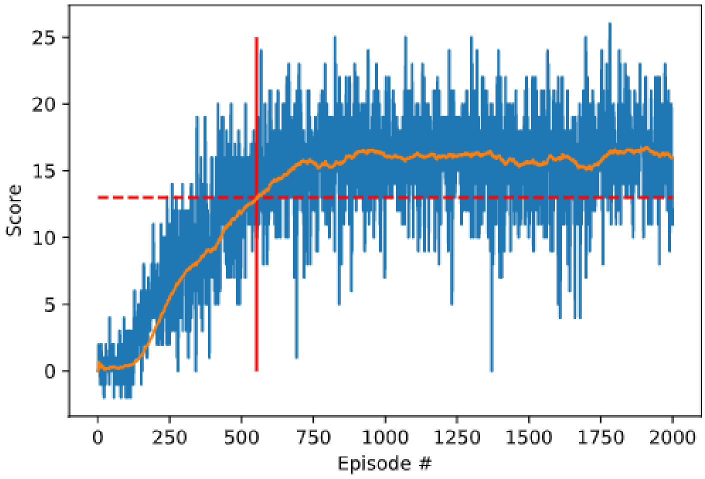


Figure 2: Results of the experiment

4 Future Considerations

There are a variety of avenues we may pursue from here. Among them are:

1. Prioritized Experience Replay [3]
2. Dueling DQN [4]
3. Distributional DQN [5]
4. Noisy DQN [6]

Another consideration from a more technical point of view is the implementation of the algorithms using *TensorFlow* instead of *PyTorch*. The author has already made attempts in this direction by converting the agent and its corresponding network to Tensorflow as can be seen in the files `agent_tf.py` and `network_tf.py`. Due to time constraints, the code is unfortunately still buggy.

References

[1] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[4] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2016.

[5] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," 2017.

[6] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, "Noisy networks for exploration," 2019.