Project Navigation

Luís Melo dos Santos

May 31, 2019

This is the first project in a series of projects linked to the *Deep Reinforcement Learning for Enterprise Nanodegree Program* at *Udacity*.

In this project, we aim to train an agent in the `Banana Collector` *Unity* environment.[1]

In this environment the agent navigates a large squared world and collects bananas. 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 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 backwards;

[1] An environment tailored by Udacity specifically for this nanodegree, similar to the original `Banana Collector` environment publicly available in the *Unity* github repository.

- 2 – turn left;
- 3 – turn right.

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

1 Learning Algorithm

We started by adapting the algorithm used in class to solve the `LunarLander-v2` *OpenAI* environment. The algorithm is inspired in the Deep Q-Networks model (DQN-model) as introduced in the Nature paper [1] in 2015.

The DQN-model is based on a classical 1-step temporal-difference method – TD(0) – also known as *Q-learning* [2] where a $Q(S, A)$ state-action value function – *Q-matrix* – gets updated by ΔQ at each step with new information $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ acquired by the agent as it interacts with the environment,

$$\Delta Q(S_t, A_t) = \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{1}$$

S_t is the current state that the agent occupies in the environment, A_t is the action that the agent takes, R_{t+1} is the reward that the agent receives from the environment, S_{t+1} is the state where the agent ends up after taking the action and A_{t+1} is defined by the greedy policy,

$$A_{t+1} = \operatorname*{argmax}_a Q(S_{t+1}, a). \tag{2}$$

as Q-learning has an off-policy TD control.

The remaining two parameters in the *Q-update* equation (1) are,

- γ – the discount rate of future rewards;
- α – the learning rate.

γ dictates how eager our agent is to defer rewards and α controls the speed at which the new information propagates through the Q-matrix.

As the State-space $\mathcal{S}$ of the problem is large[2], we need to apply a function approximation method in order to make the problem tractable. In the DQN-model, *Deep Neural Networks* are used as the method to approximate the Q-matrix, $\tilde{Q}(S_t, A_t, \theta)$, where θ is the set of weights that fully define a neural network N.[3] N is constrained by an input layer with $|\mathcal{S}|$-units and output layer with $|\mathcal{A}|$-units.

We adapt the Q-update equation (1) in this context to an update to the weights θ at each learning step,

$$\Delta\theta = \alpha\Big[R_{t+1} + \gamma \max_a \tilde{Q}(S_{t+1}, a, \theta) - \tilde{Q}(S_t, A_t, \theta)\Big] \cdot \nabla_\theta \tilde{Q}(S_t, A_t, \theta) \tag{4}$$

[2] $\mathcal{S}$ is a 37-dimensional continuous space.

[3] In the DQN-model, $\tilde{Q}(S_t, A_t, \theta)$ is structured as a vector function N defined by weights θ with inputs from the State-space $\mathcal{S}$ and an output vector with same dimensions as the

If we define a loss function L that would get optimised by equation (4),

$$L(\theta) \sim \left[R_{t+1} + \gamma \max_a \tilde{Q}(S_{t+1}, a, \theta) - \tilde{Q}(S_t, A_t, \theta)\right]^2, \tag{5}$$

we can then apply the usual gradient decent techniques and deep learning tools to train our neural network.

With enough data collected by the agent from the environment and after sufficient training we would expect the neural network $\tilde{Q}(S, A, \theta)$ to be a good approximation of the *optimal* state-value function q_* of the environment.

There are two further aspects of the DQN-model worth mentioning due to the stability and improvement in performance that they bring to the training process. First, the *Experience Replay* – as the agent interacts with the environment the set of experiences $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ get stored in memory for a period of time in order to potentially be reused for further learning.[4] This makes the learning process more efficient because experiences are expensive to collect. As much information as possible should be extracted from experiences before discarding them. Also, if we focus our agent to only learn from what is currently experiencing we might introduce a significant bias and a potential amplification of noise due to a feedback loop between our exploration policy – still significantly greedy – and our state-action value updates ΔQ at particular regions of $\mathcal{S}$.

A further improvement in stability in the learning process can be achieved by *Fixed Q-Targets*. Equation (4) cannot technically be obtained from the loss function differentiation $\nabla_\theta L$ if we do not fix the first $\tilde{Q}$-function as a fixed target in the loss function L,

$$L(\theta) \sim \left[R_{t+1} + \gamma \max_a \tilde{Q}(S_{t+1}, a, \theta^-) - \tilde{Q}(S_t, A_t, \theta)\right]^2, \tag{6}$$

where θ^- is fixed.

Effectively this means that in the learning process we keep two sets of weights (θ, θ^-). We train θ keeping θ^- constant and after a few training cycles we update θ^-. We could copy directly from trained θ,

$$\theta^- \longleftarrow \theta, \tag{7}$$

but in this project we perform a soft update instead at each learning step,

$$\theta^- = \theta^- + \tau(\theta - \theta^-), \tag{8}$$

Action-space $\mathcal{A}$,

$$N(\theta): \quad \mathcal{S} \longrightarrow \mathbb{R}^{|\mathcal{A}|}$$

$$S_t \rightharpoonup \tilde{Q}(S_t, \ldots, \theta) = \begin{bmatrix} \tilde{Q}(S_t, 0, \theta) \\ \vdots \\ \tilde{Q}(S_t, 3, \theta) \end{bmatrix}$$

[4] In the original DQN-model [1] the memory is uniformly randomly accessed.

where τ is another learning rate parameter controlling the rate at which new information propagates through the target neural network.

In the next section, we will detail how we implemented this learning algorithm in practice.

2 Implementation

Our implementation of the learning algorithm, as mention previously, follows closely the code architecture introduced in class. The definition of the agent is split between two files,

- `dqn_agent.py`
- `model.py`

In `dqn_agent.py`, we define the *Agent()* class. One of the alterations to the class definition we introduce was to expand the inputs and include a dictionary with all the hyperparameters to improve flexibility in analysing and tuning the model. In this class the methods *step*, *act*, *learn* and *soft_update* get defined. In *step()* we sequence how the agent stores each experience in memory and how often it starts the learning process. In *act()*, we define the *behaviour policy* of the agent – in this case an ϵ-greedy policy. It is in the *learn()* method that the loss function L, as defined in (6), gets calculated and optimised. In this implementation we use an Adam optimization.[5] It is also worth mentioning that we perform batch learning – for each training step we sample the agent's memory for a set of experiences and use their information in the optimisation step. At each learning step we also perform a soft update of the target neural network parameters θ^-, according to equation (8), using the *soft_update()* method.

A second class is defined in `dqn_agent.py` – the *ReplayBuffer()* class – where the agent's memory structure gets defined. In this case a *queue* of fixed maximum size is initialised and *add()* and *sample()* methods get defined. In the DQN-model, memory sampling is done randomly where each experience has an equal probability[6] to be picked for the learning process.

In `model.py`, the neural network gets defined. We use *PyTorch* to define a fully connected deep neural network with two hidden layers.[7] Both input layer units and output layer units are fixed in the DQN-model to $|\mathcal{S}|$ and $|\mathcal{A}|$ respectively, but we kept the number of units for both hidden layers variable in order to be able to test and tune them.

We created a third file `training_dqn_agent.py` in order to systematise our agent's training. We loop the agent through episodes keeping track of its performance in order to establish when he has successfully solved the environment. We also define a maximum amount of episodes `n_episodes` after which if our agent is still not successful we stop the training process.

[5] We kept Adam optimisation from the original implementation and it was something we have not tested in terms of performance against other optimisation schemes.

[6] with replacement.

[7] We have not tested the performance of our neural network architecture against other types

3 Analysis

The analysis of our DQN-model implementation is structured in four notebooks,

0. `Navigation_dqn_agent_baseline.ipynb`
1. `Baseline DQN Model Analysis.ipynb`
2. `Hyperparameters Tuning.ipynb`
3. `Double DQN Model Analysis.ipynb`

Each notebook improves on the previous one and furthers the understanding and performance of the model.

3.0 Navigation Deep Q-Network Agent

This notebook is an initial proof of concept. We first initialise and explore the *Unity* environment provided. We watch an untrained agent explore the environment. We train the agent until achieving success. And finally watch the trained agent navigate the environment.

3.1 Baseline DQN-Model Analysis

In this notebook we analyse the learning performance of the initial model in more detail. We keep track of the *number of episodes* required to train a successful agent as our main key performance indicator as suggested by the project's rubric but we also keep an eye on the amount of time needed to solve the environment as a secondary *kpi*.

Name	Value	Description
`n_episodes`	2000	maximum number of episodes to learn from
`max_t`	1000	maximum time steps per episode
`eps_start`	1.0	initial ϵ value
`eps_end`	0.01	final ϵ value
`eps_decay`	0.995	decay rate for ϵ per episode
`fc1_units`	64	number of units of first hidden layer
`fc2_units`	64	number of units of second hidden layer
`buffer_size`	10^5	size of the memory
`batch_size`	64	size of the number of examples fed at each learning step
`gamma`	0.99	discounting rate of rewards
`tau`	10^{-3}	soft update learning rate of target neural network
`lr`	10^{-4}	learning rate for the local neural network parameters
`update_every`	4	how often between action steps we have a learning step

Table 1: Hyperparameters values used in the baseline model.

On table 1 we present the hyperparameters values we used initially in the model.

We repeated the learning process 50 times in order to determine the distribution of number of episodes and time required in training successful agents with this selection of hyperparameter values.

3.2 Hyperparameters Tuning

We proceed in the next notebook exploring how the learning performance of the model would react to changes in the hyperparameters. We shocked each hyperparameter point-wise sequentially either in absolute or relative terms depending on the nature of the hyperparameter to have a better understanding to which hyperparameters the model is more sensitive. We used our understanding of the performance distribution determined in the last notebook to infer statistical significant results.

After a few iterations in the most relevant directions in the hyperparameter space we settle on the final values presented in table 2,

Name	Base	Tuned
`n_episodes`	2000	2000
`max_t`	1000	1000
`eps_start`	1.0	0.5
`eps_end`	0.01	0.01
`eps_decay`	0.995	0.92
`fc1_units`	64	32
`fc2_units`	64	64
`buffer_size`	10^5	2×10^4
`batch_size`	64	64
`gamma`	0.99	0.99
`tau`	10^{-3}	10^{-3}
`lr`	10^{-4}	10^{-4}
`update_every`	4	3

Table 2: Comparison of hyperparameters values used in the final tuned model versus baseline model.

3.3 Double DQN-Model Analysis

Once we exhausted all the opportunities of optimising the DQN-model we turned our attention to improve aspects of the model and explore some of the recent advances published in the literature. One of such advances comes from an adaptation of the classical *Double Q-learning* algorithm to the DQN-model in [3].

It has been discussed and explored at length in the literature [4] how Function Approximation in the context of Q-learning can be problematic as the algorithm

can amplify the initial approximation error through particular regions of the Q-matrix.

This stems from using the same Q-matrix approximation twice when defining the expected target state-value function $q_\star$ when updating the Q-matrix in equation (1) as we can see in the breakdown below,

$$\max_a Q(S_{t+1}, a, \theta) = Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a, \theta), \theta) \tag{9}$$

The TD control uses the approximation of Q-matrix $Q(S_{t+1}, \ldots, \theta)$ to make the decision of which next action A_{t+1} to take and the state-action value used in the update uses the same Q-matrix approximation.

We can see how this could lead to overestimation in the case when all the actions for a particular state have the same value in the environment. As Q-learning adopts a greedy policy in the TD control, it would pick the action with the highest positive error in the Q-matrix approximation for that specific state $A_{t+1} = \operatorname{argmax}_a Q(S_{t+1}, a, \theta)$ and propagate the noise further to the neighbouring states when updating the Q-matrix using the same approximation $Q(S_{t+1}, A_{t+1}, \theta)$. Q-learning thus tends to overestimate $q_\star$.

This feedback loop can be interrupted or largely attenuated if we carry two different versions of Q-matrix approximation $(Q(S, A, \theta_1), Q(S, A, \theta_2))$ that get updated in different cycles. This is the premise of the classical Double Q-learning algorithm. Where the Q-matrix update rule changes to,

$$\Delta Q(S_t, A_t, \theta_1) = \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a, \theta_1), \theta_2) - Q(S_t, A_t, \theta_1)\Big] \tag{10}$$

This improvement to the classical algorithm can easily be incorporated in the DQN-model framework as it already carries two approximations of the Q-matrix around in its calculations defined by the neural network weights (θ, θ_-). We can easily change equation (4) to incorporate the Double Q update (10),

$$\Delta\theta = \alpha\Big[R_{t+1} + \gamma\tilde{Q}(S_{t+1}, \operatorname*{argmax}_a \tilde{Q}(S_{t+1}, a, \theta), \theta_-) - \tilde{Q}(S_t, A_t, \theta)\Big] \cdot \nabla_\theta \tilde{Q}(S_t, A_t, \theta) \tag{11}$$

We have introduced this change in file `double_dqn_agent.py` and indeed we have experienced an improvement in the learning performance as it will be detailed in the next section where we present our results.

4 Results

In this section we will present a summary of our results so far. We will focus on our main kpi – *number of episodes* experienced by the agent until it solved the environment. This will be the measure we will use to compare the performance of our learning algorithms.

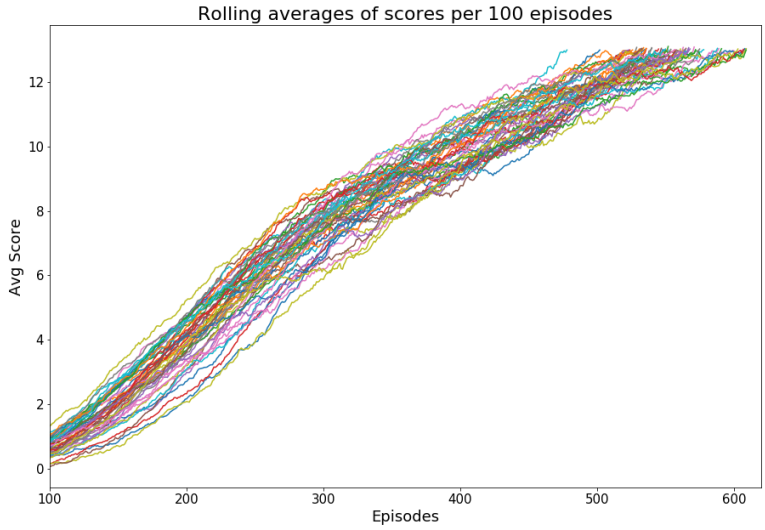

Figure 1: Rolling average scores of 50 different agents learning paths using our initial DQN-model learning algorithm.

4.1 Initial DQN-model

Our first implementation of the DQN-model using table 1 hyperparameter values was analysed in notebook `Baseline DQN Model Analysis.ipynb`.

It took an agent using our initial DQN-model an average of *460 episodes* to achieve success. The learning paths of our agents kept a close distance to each other – no larger than 4 scores at all times – as we can see in figure 1. The standard deviation of the number of episodes required to achieve success was only *30* episodes.

4.2 Tuned DQN-model

In notebook `Hyperparameters Tuning.ipynb` we analysed the dependency of the model with relation to the different hyperparameters and searched for the combination of values that would optimally improved the performance of the model. After tuning the hyperparameters to the values in table 2 we can see in figure 2 how the learning paths became steeper as the learning ability of the agents vastly increased.

For our final tuned DQN-model, it took an agent on average *270 episodes* to achieve success. We experienced with this model a much wider spectrum of learning experiences from different agents. The range of learning paths are not anymore as tightly bound. We have now situations when some agents already

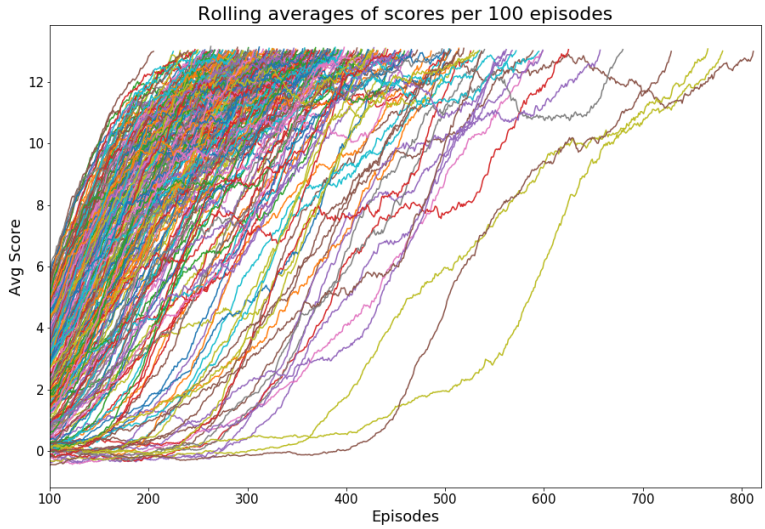

Figure 2: Rolling average scores of 250 different agents learning paths using our final tuned DQN-model learning algorithm.

solved the environment whilst others are still averaging null scores. The dispersion of the number of episodes distribution has significantly increased to *100 epsiodes* standard deviation.

4.3 Double DQN-model

We next implemented a Double DQN-model and analysed it in the `Double DQN Model Analysis.ipynb` notebook. We kept the same final tuned hyperparameters as in DQN-model, in table 2.

We can see in figure 3 that the agents' learning ability has not increased using the Double DQN-model. In fact, we still have *270 episodes* as the average number of episodes needed by the agents to learn how to solve the environment. But, we were able to reduce the dispersion of the distribution – standard deviation reduced by one-fourth to *75 episodes* and even normalised kurtosis became negative[8] suggesting a significant reduction in the likelihood of tail events.[9]

[8] Normalised kurtosis for the different models:

- Tuned DQN-model: 3.20
- Double DQN-model: −0.10

[9] A tail event in this context would be an agent taking a significant number of episodes – 2000+ – to learn the environment or failing to do so all together.

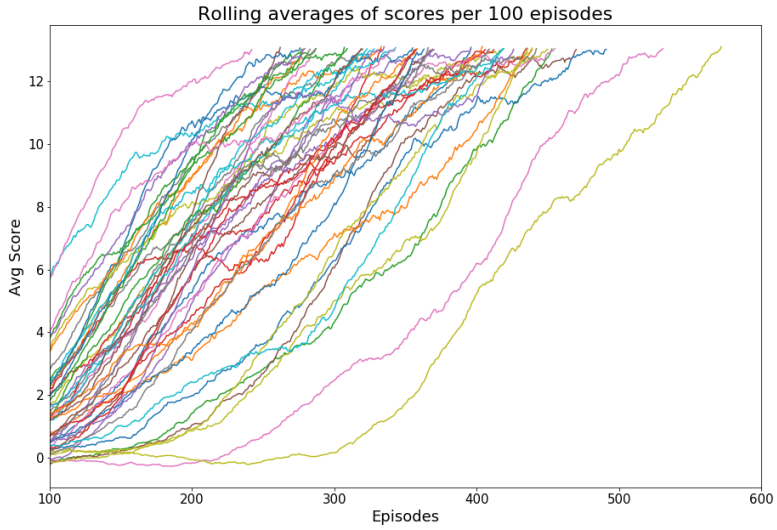

Figure 3: Rolling average scores of 50 different agents learning paths using our final tuned DQN-model learning algorithm.

5 Ideas for Future Work

Next in the improvements stack we will be tuning the hyperparameters of Double DQN-model. In the current implementation of the model we used the tuned parameters of the DQN-model. It will be interesting to find out if we can improve further the performance of the Double DQN-model with different hyperparameter values.

In terms of exploration there are mainly two different directions we can take this project forward,

- Testing different *Deep learning* methods and architectures:
  - try different neural networks – increase the number of deep layers.
  - try different optimisation schemes – we have only used Adam in our project.
  - try to use pixels as input data – as suggested in the project's rubric, use convolution neural networks and learn directly from the raw pixel data.
- Improve certain aspects of the *Reinforcement Learning Algorithm*
  - we have been working on introducing *Prioritised Experience Replay* in the model based on [5] and expect to publish the notebook with the results soon.

- we would also like to delve and introduce in our model some of the other improvements and techniques described and analysed in the literature [6, 7, 8, 9, 10],
  - Dueling DQN-model
  - multi-step bootstrap targets (A3C)
  - Distributional DQN
  - Noisy DQN

Our idea is to keep this project going and continue posting more notebooks with further improvements and more results of our exploration as we go along.

Watch this space... ☺

References

[1] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2015.

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

[3] H. van Hasselt *et al.*, "Deep reinforcement learning with double q-learning," *arxiv.org*, December 2015.

[4] *Issues in Using Function Approximation for Reinforcement Learning*, Lawrence Erlbaum, 1993.

[5] T. Schaul *et al.*, "Prioritized experience replay," *arxiv.org*, 2016.

[6] Z. Wang *et al.*, "Dueling network architectures for deep reinforcement learning," *arxiv.org*, no. 1511.06581, 2015.

[7] V. Mnih *et al.*, "Asynchronous methods for deep reinforcement learning," *arxiv.org*, no. 1602.01783, 2016.

[8] M. G. Bellemare *et al.*, "A distributional perspective on reinforcement learning," *arxiv.org*, no. 1707.06887, 2017.

[9] M. Fortunato *et al.*, "Noisy networks for exploration," *arxiv.org*, no. 1706.10295, 2017.

[10] M. Hessel *et al.*, "Rainbow: Combining improvements in deep reinforcement learning," *arxiv.org*, no. 1710.02298, 2017.