Reinforcement learning project 1: Navigation

Reza Doobary

June 23, 2019

Contents

---

1 Introduction

In this short report we provide model implementation details and analysis on the reinforcement learning project entitled 'Navigation' as part of the Udacity course on deep reinforcement learning.

Whilst full task details of the agent, the environment and success criteria have been outlined in the accompanying README.md, we repeat these for completeness. The state space of the agent is a 37 dimensional real float space and contains the agent's velocity, along with ray-based perception of objects around the agent's forward direction. The space of actions of that the agent can take is 4 dimensional real integer space and have physical interpretations, namely to move:

1. forwards
2. backwards
3. left
4. right

The task is deemed solved if the agent gets an average score of +13 over 100 consecutive episodes.

NOTE : Some basic introductory concepts have been added in this report for completeness. Please go directly to section 3, for implementation and hyperparameter details. In particular, the reader may wish to go straight to section 3.3.6 for a very quick report with numerical results of the exercise.

2 General concepts

2.1 Basic concepts

When training the reinforcement learning (RL) agent to perform a task optimally, we are interested in finding the optimal decision making process. For now, we are modelling the problem as a Markov decision problem. This is referred to as the **policy** which is usually denoted by π. The agent itself has **states** and must move from state to state, whilst in each transition executing an **action**. In doing so, the environment provides feedback by way of a **reward**. In principle, we can always find a representation of the state and action is some vector space, thus we can define $\mathcal{S} \sim \mathbb{R}^{n_\mathcal{S}}$ and $\mathcal{A} \sim \mathbb{R}^{n_\mathcal{A}}$, where $n_\mathcal{S}$ and $n_\mathcal{A}$ are the dimension of the state (observational) space and the dimension of the action space respectively. A simple deterministic policy is given by

$$\pi : \mathcal{S} \to \mathcal{A}. \tag{1}$$

Of course, we can invent a policy which maps the tuple $(s,a) \in \mathcal{S} \times \mathcal{A}$ onto a $[0,1]$ thus yielding a conditional probabilistic policy $\pi(a|s)$.

The rewards are often accumulated to produce returns defined as

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k}, \tag{2}$$

where in principle T could be infinity, but we consider episodic tasks, thus having a well defined temporal endpoint for each episode. Here, γ is a discount factor which is tells the agent how strongly or weakly it should value immediate or long term rewards.

2.1.1 State-value and Action-value functions

The state-value function is defined to be:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right]. \tag{3}$$

The action-value function is defined to be:

$$q_\pi(s,a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right]. \tag{4}$$

The relationship between these two expressions is given by

$$\max_a q_\pi(s,a) = v_\pi(s). \tag{5}$$

The optimal policy is the one which satisfies

$$q_*(s,a) = \max_\pi q_\pi(s,a), \tag{6}$$

and indeed it is the fundamental goal of all reinforcement learning algorithms to find the optimal function $q_*(s,a)$.

2.1.2 A first strategy for training : ϵ-greedy policy

A bare bones reinforcement learning problem would involve initialising $q_\pi(s,a)$ with zero and providing uniform random distribution of possible actions given any state. Then at the end of an action taking place, an update rule (for example Q-learning or temporal difference) is employed to update the value of a state action pair. A very basic on-policy method would involve re-employing the newly updated $q_\pi(s,a)$ method to perform the next action which maximises the function. Essentially :

$$a = \max_a q_\pi(s,a). \tag{7}$$

With each new iteration, the policy π changes, and hopefully the agent's action-value function q_π converges to the optimal policy

However, much like any other optimisation problem, we do fear finding a local optimal in the 'surface' of policies. In this context, we would worry about locking ourselves on a specific policy by finding a policy that persists to give us incremental positive feedback whilst ignoring other potential choices which could provide larger returns or even large 'one time' low probability pay-offs. This is the content of the explore vs. exploit paradigm. As the agent works in the environment it must exploit previous found policies to given definite incremental returns, but must also spend some small amount of time exploring the state-action space for other pay-offs. This is done by introducing a small parameter ϵ which measures the probability that the agent acts in a random and uniform manner. In our algorithms, we employ such a paradigm.

2.2 Models for action-value functions

In this report, we are interested in modelling the action-value function $q_\pi(s,a)$ in order to find the optimal policy. In principle, this is allowed to be anything from a linear function, all the way to highly non-linear function. Let us be clear about what we want; the input of the function would be a state and the output would be a vector of state-value functions corresponding to the action taken. In the current context of the banana eating agent: the state input is such that $\mathcal{S} \sim \mathbb{R}^{37}$, which the action space is $\mathcal{A} \sim \mathbb{R}^4$. The $q_\pi(s,a)$ function is thus doing:

$$q_\pi(s) \to \left(v_\pi(s_1), v_\pi(s_2), v_\pi(s_3), v_\pi(s_3)\right), \tag{8}$$

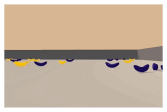
that is, the output of this function is the set of value functions whose index corresponds to the action to reach that particular state, e.g. $v_\pi(s_1)$ is the value of using action 0 (going forward) to reach state s_1. We can then reuse s_1 and perform the action on it, thus iterating the procedure and performing the algorithm. This is the function we wish to model with a neural network.

3 Implementation and Analysis

We now discuss the exact implementation details of our algorithm used in the aforementioned RL problem.

3.1 The architecture of the model

The model chosen has three hidden layers, together with an input layer and an output layer. There is no preprocessing of the states before being put into the input layer and the output layer is simply the value associated to the chosen state dictated by the action taken (given by the index of the output). The structure of the model is thus:



$$\Longrightarrow \begin{array}{c}\text{37 dim'l}\\ \text{input layer}\end{array} \Longrightarrow \begin{array}{c}\text{64 dim'l}\\ \text{hidden layer 1}\end{array} \Longrightarrow \begin{array}{c}\text{128 dim'l}\\ \text{hidden layer 2}\end{array} \Longrightarrow \begin{array}{c}\text{64 dim'l}\\ \text{hidden layer 3}\end{array} \Longrightarrow \begin{array}{c}\text{4 dim'l}\\ \text{output layer}\end{array} \tag{9}$$

The input is a 37 dimensional real vector : $\mathbb{R}^{37}$ whilst the output is a vector in $\mathbb{R}^4$ which is the value of the states found had the agent taken the action associated to the vector index. A basic non-exploration algorithm would define the next action to be the one which leads to the state of highest value, e.g. in this particular case if we get $(12, 13, 14, 10)$, then we would say going left is the choice which gives the overall optimal policy (although this may once again be updated).

We take the mean square error (MSE) to be the loss function we use to minimise the difference between the expected output and the received output. Depending on various algorithm improvements that are employed, the precise MSE inputs may change and these are to be discussed shortly.

3.2 ϵ-greedy policy and γ

As discussed in the introduction an ϵ-greedy policy is used to decrease any concentration risk upon immediate high returns with long term low quality policies. The prescription that is used here is rather simple, the a starting point of $\epsilon = 1$ is used, and then with each episode used ϵ decreased by a factor of 0.8 up until it reaches a minimum value of 0.01 where it is held fixed.

A second parameter that is used is γ, which is used to manage the immediate reward against those which take longer to occur. This parameter is set to 0.99 through the study.

3.3 Implementations and results

In the study presented here we looked at three main improvements over the very basic implementation of the deep neural network.

- Experience Replay (this is treated as a benchmark model)
- Double DQN
- Duelling DQN

We now provide basic implementation details of each in turn, together with the obtained results. At the end on the section we provide a comparative plot of all methods used.

There are some consistent numbers that we have used throughout every piece of the study. Namely for the model, the optimisation used is the Adam optimiser which has a learning rate set at 0.0005. This can be changed in `agent.py`.

For the result analysis we average over 5 training simulations and taken the floor of the average itself. For example, if the problem is solved in 300, 301, 300, 302 and 304 episodes, we would take the number of episodes required to solve the problem as floor$(301.4) = 301$

3.3.1 Experience Replay : Benchmark model

The very basic implementation would process each and every state input and provide an immediate update to the state-action model. However, this may concentrate on certain winning strategies whilst missing out on other ones. As a result a better option is to have a memory of a certain number of input and process these as a whole, effectively making the actual neural network training problem a batch-wise supervised learning problem, since a dataset is actually obtained.

In the code, this is implemented by a ReplayBuffer[1] which is built on top a deque (i.e. a reverse queue) which stores the state, action, reward, the next state and whether the given iteration which a terminal one or not. The parameters of the ReplayBuffer are given as:

- BUFFER SIZE : 10000 - The maximal size of the deque.
- BATCH SIZE : 64 - When a call to randomly sample from the buffer is made, a sample of 64 data points are made.

[1]Note that this implementation came from a previous Udacity exercise where it was written by the exercise author by assumption

- UPDATE EVERY : 4 - Every 4 time steps, given that the the data set is big enough (i.e. more than 64 data points in the ReplayBuffer), an attempt to randomly sample from the buffer is made and an update on the agents policy is computed. This is also when the parameters of the 'target' network are updated with the 'local' ones, to be discussed shortly.

These variables can be changed in `agent.py`.

The implementation itself that was employed can be found in [4]. Essentially, we would handle two identical neural networks; a target one $\hat{Q}$ and a local one Q. From the supervised learning perspective, the target variable at each time step is:

$$Y_t = R_{t+1} + \gamma \max_a \hat{Q}(s_{t+1}, a)\left(1 - \mathbb{I}_{\text{done}}\right), \tag{10}$$

and represents the expected return. The current realised return is $Q(s_t, a_t)$, thus we wish to minimise

$$\text{MSE}\left(Y_t - Q(s_t, a_t)\right), \tag{11}$$

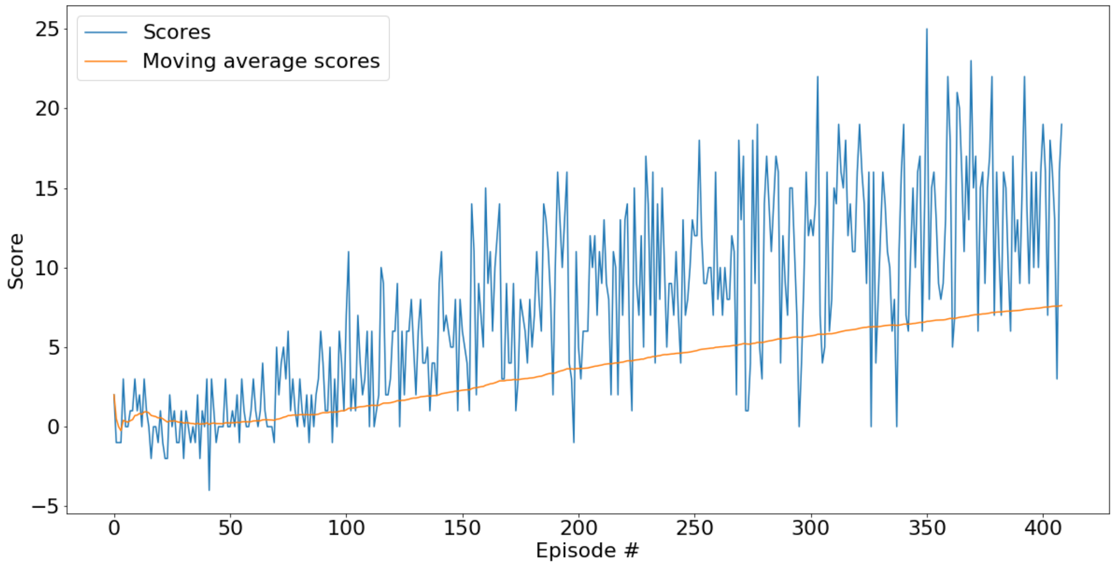
and we do this in a batch-wise manner. The main point is that every so often, the $\hat{Q}$ parameters are updated with the Q ones. For us the most relevant part of the implementation is located in `agent.py` and is in the `learn` method of the `Agent` class.

```
Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

# Compute Q targets for current states
Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

# Get expected Q values from local model
Q_expected = self.qnetwork_local(states).gather(1, actions)

# Compute loss
loss = F.mse_loss(Q_expected, Q_targets)
# Minimize the loss
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()

# update target network
self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

where `Q_targets` is given the target model as defined in the [4] to be $\hat{Q}$, which is essentially the model with the weights fixed. `Q_local` is Q in the paper, and is the model that is updated when learning. In [4] a 'hard' $\hat{Q} = Q$ is implemented, here, a 'soft' update is implemented where the model parameters are unloaded into the target model via $\text{W\_target} = \tau\text{W\_local} + (1 - \tau)\text{W\_target}$ where τ is defined to be 0.001 can be changed in `agent.py`.

3.3.1.1 Results

The plot of the scores against episode until success as defined by gaining a average return of 13 over 100 episodes. The problem was on average effectively solved at 262 episodes, since then up until 362 episodes an average return of 13 was obtained. The minimum number of episodes required of the 5 attempts was 225.



The blue line represents the exact return, whilst the orange one is the continuous average over all past episodes.

This represents our benchmark model.

3.3.2 Double DQN

The first improvement over the benchmark model is the so-called double DQN. This was studied in the Atari games context in [6]. There it was found that for certain Atari games, the general DQN has the capacity to overestimate action values. The problem comes from the (temporal difference) TD estimator for the action-value function, which is given by

$$Y_t = R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, w) = R_{t+1} + \gamma \hat{Q}(S_{t+1}, \text{argmax}_a \hat{Q}(S_{t+1}, a, w), w). \qquad (12)$$

Note that this is written a little differently to equation 10, to emphasise the dependence on the weights of the deep neural network. It becomes clear that early on in the training, some concentration risk may occur on some action-value pairs of the function which persist throughout the training and leads to an overestimation of those specific pairs.

The solution to this problem is essentially to introduce an identical model with a set of different weights. Then, we find the optimal action on one parameters space w whilst evaluating the estimator on a different parameter space w'. As it turns out, the current algorithm already has two models, the target and local ones (or $\hat{Q}$ and Q respectively) and the parameters of these two models can be employed to suit the needs of the double DQN. This is done by finding the optimal action using the local (Q) model and evaluating the TD estimator on the target $(\hat{Q})$ model.

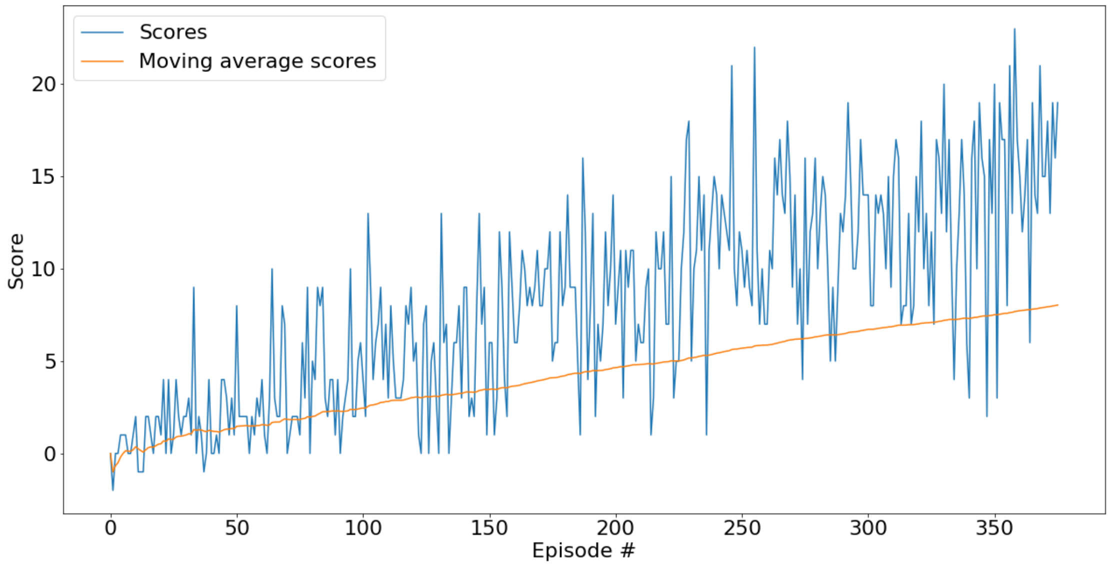
For us the most relevant part of the implementation is located in `agent.py` and is in the `learn` method of the `Agent` class.

```
if self.isDDQN:
        # Get optimal action from local model and feed forward next_states on target network
        best_local_actions = self.qnetwork_local(states).max(1)[1].unsqueeze(1)
        double_dqn_targets = self.qnetwork_target(next_states)
        # Get value of the target dqn vialocal optimal action
        Q_targets_next = torch.gather(double_dqn_targets, 1, best_local_actions)
 else:
        # Get max predicted Q values (for next states) from target model (without ddqn)
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

 # Compute Q targets for current states
 Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

 # Get expected Q values from local model
 Q_expected = self.qnetwork_local(states).gather(1, actions)

 # Compute loss
 loss = F.mse_loss(Q_expected, Q_targets)
 # Minimize the loss
 self.optimizer.zero_grad()
 loss.backward()
 self.optimizer.step()

 # update target network
 self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

It should be noted that this piece of code reduces to the benchmark model (Experience replay) explained earlier.

3.3.2.1 Results

The plot of the scores against episode until success as defined by gaining a average return of 13 over 100 episodes. The problem was on average effectively solved at 258 episodes, since then up until 358 episodes an average return of 13 was obtained. The minimum number of episodes required of the 5 attempts was 224.



The blue line represents the exact return, whilst the orange one is the continuous average over all past episodes.

3.3.3 Duelling DQN

The second improvement is to consider the fact that most the values of the states do not vary very much across actions, thus it makes sense to directly estimate the value function for each state. However, it is crucial that we understand the difference betweem specific actions since this is how we deduce the optimal policy. As a result, two sub-architectures are constructed. The first one is identical to the current architecture and produces the so-called 'advantage function', whilst the second one will have its final layer being the value function (thus being only one-dimensional). The final layer simply adds the output of these two layers to produce the estimation of the action-value function. This was studied in [7]. The structure of the model now takes the form:

$$\begin{tikzcd}
 & & & & \text{4 dim'l Adv. function} \arrow[dr, Rightarrow] & \\
{} \arrow[r, Rightarrow] & \text{37 dim'l input layer} \arrow[r, Rightarrow] & \text{64 dim'l hidden layer 1} \arrow[r, Rightarrow] & \text{128 dim'l hidden layer 2} \arrow[r, Rightarrow] & \text{64 dim'l hidden layer 3} \arrow[u, Rightarrow] \arrow[d, Rightarrow] & \text{4 dim'l output layer} \\
 & & & & \text{1 dim'l value function} \arrow[ur, Rightarrow] &
\end{tikzcd} \tag{13}$$

In [7], it is expressed that there is the problem of 'identifiability', which defines the fact that given the action-value function Q, one cannot uniquely recover the advantage function and value function. This is circumvented by taking the mean of the advantage function away from itself[2].

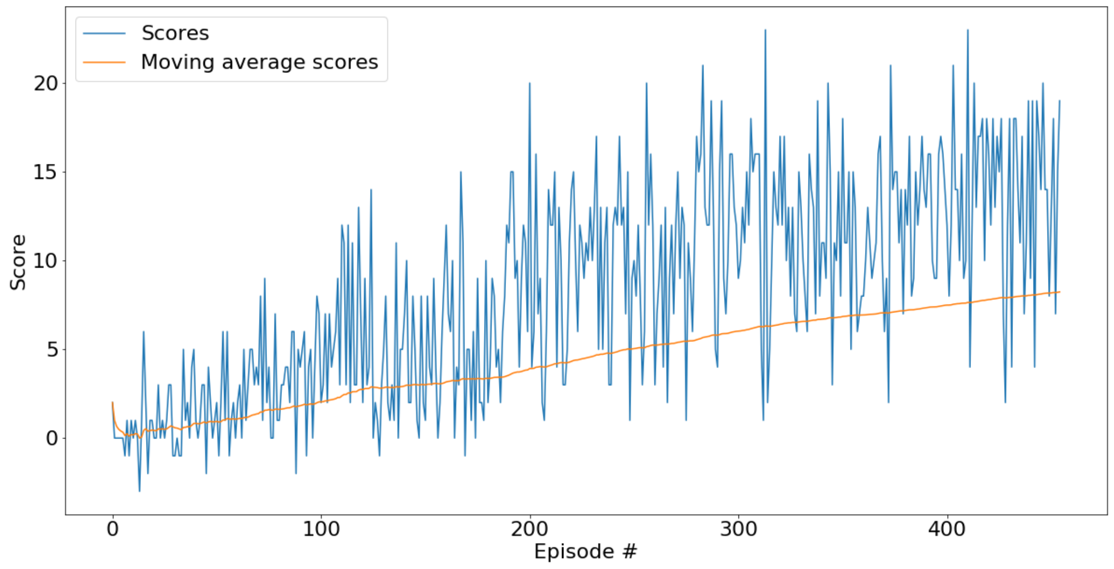
Thus for us the most relevant part of the implementation is located in `model.py` and is in the `forward` method of the `QNetwork` class.

```python
def forward(self, input):
    """Build a network that maps state -> action values.

    Params
    ======
        input (Tensor[torch.Variable]): Input tensor in PyTorch model
    """
    for linear in self.hidden_layers:
        input = F.relu(linear(input))
        input = self.dropout(input)

    if self.with_dueling:
        advantage_function = self.output(input)
        output = self.state_value(input) + (advantage_function-torch.mean(advantage_function))
    else:
        output = self.output(input)
    return output
```

3.3.3.1 Results

The plot of the scores against episode until success as defined by gaining a average return of 13 over 100 episodes. The problem was on average effectively solved at 249 episodes, since then up until 349 episodes an average return of 13 was obtained. The minimum number of episodes required of the 5 attempts was 219.

[2]Eqution 9 in [7]



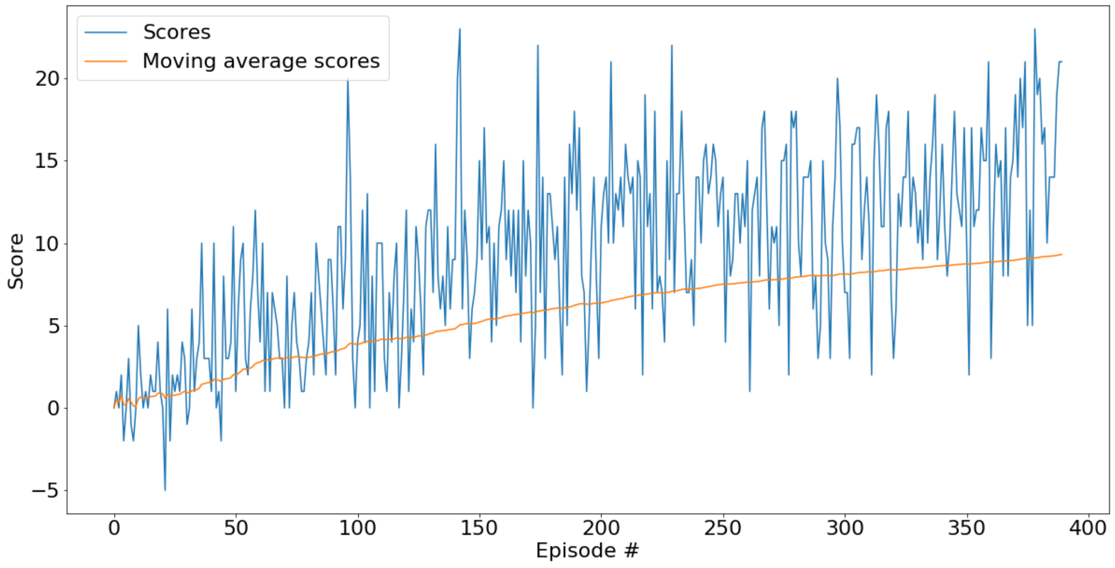
The blue line represents the exact return, whilst the orange one is the continuous average over all past episodes.

3.3.4 Combination

Given that the training has been run for a double DQN as well as with duelling DQN, it is a natural curiosity as to what happens if we combine these methodologies. The task is performed in this section.

3.3.4.1 Results

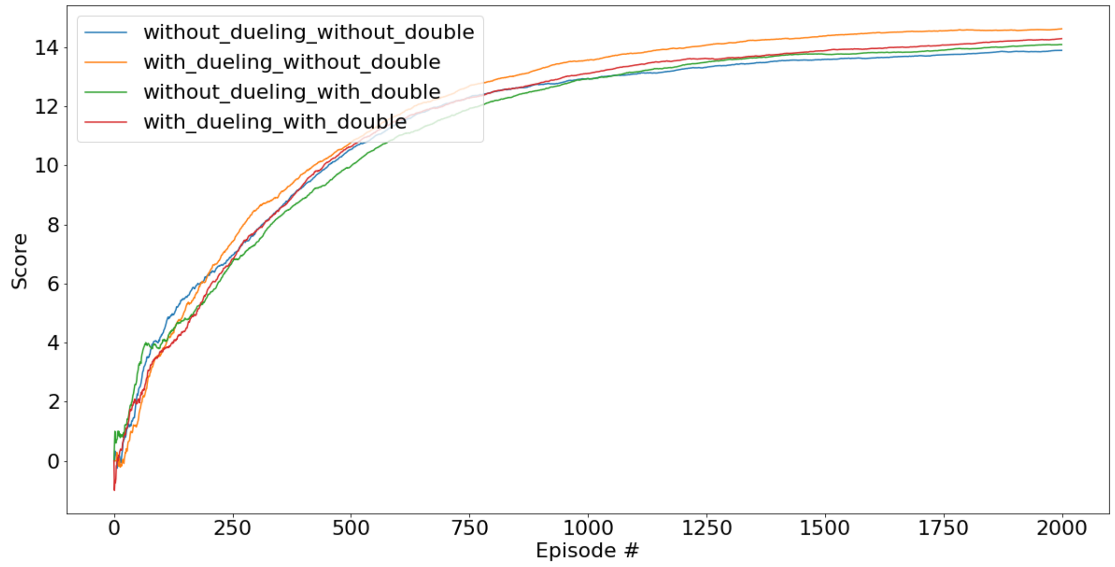
The plot of the scores against episode until success as defined by gaining a average return of 13 over 100 episodes. The problem was on average effectively solved at 266 episodes, since then up until 366 episodes an average return of 13 was obtained. The minimum number of episodes required of the 5 attempts was 240.



The blue line represents the exact return, whilst the orange one is the continuous average over all past episodes.

3.3.5 Direct comparison for 2000 episodes

The final thing that can be tried is to directly compare all of the aforementioned improvement methodologies for 2000 episodes, which is well past the number of episodes required for the problem to be solved.



3.3.6 Quick final analysis

An ‘at a glance’ report is provided of the results here: The model architecture used has three hidden layers with structure 64 – 128 – 64 and the activation function for each layer is ‘relu’.

We employ ϵ-greedy policies, with the initialised $\epsilon = 1$, being reduced by a factor of 0.95 until 0.01 where its held stationary. The discount factor for all algorithms is 0.99.

The metric for successful training is the smallest number of episodes before getting a return of +13 over the last 100 episodes. We do the exercise 5 times and average over the number of episodes. We also include the smallest number of required episodes in our analysis. This is given in the table below.

Double DQN	Duelling DQN	Average number of episodes	Smallest number of episodes
✗	✗	262	225
✓	✗	258	224
✗	✓	249	219
✓	✓	266	240

Thus the best performing improvement methodology as a result of this study is with Duelling DQN since it produces both the lowest average amount of episodes to solve the environment, but also the lowest minimum number of episodes.

From the comparative study in section 3.3.5, we see the the double DQN performs better for about 50 episodes, whilst being finally caught up by the no methodology improvement algorithm. Ultimately, the duelling DQN maintains its stable long term performance over the other algorithms considered.

We conclude that the duelling DQN is the best performing improvement algorithm!

4 Conclusion and further work

Our main conclusion is that of the algorithms studied, the duelling DQN performed the best.

There are number of further algorithm improvements we could have studied. They are outlined as follows:

1. **Prioritized experience replay**
   The Prioritized experience replay improvment which was studied in [5], is an extension of the basic experience replay set up. The idea here is the assign probabilities to the difference between the TD estimator and the realised return by simply normalising these values. Then given the dataset used in experience replay, instead of sampling randomly one performs a sampling in accordance with the derived probability. The intuition here is that very unlikely events are those where the TD estimation and the realised returns are quite different, which triggers us/the agent to organically be more weary of these unlikely events than those typically expected.

2. **Distribution Approach**
   A distributional approach to the returns could be investigated and was first studied in [1]. This is an almost completely different approach to solving the problem of optimising the action-value function. Here, instead of finding the expected return given a state-action pair, we are instead interested in the distribution of returns.

3. **NoisyNets**
   Finally, we may also consider small adjustments to the neural network in order to improve the explorative quality of the agent. This can be done by introducing a small normally distributed $\epsilon \sim \mathcal{N}(0,1)$ value to the weights of the neural network. This was studied in [2]. This essentially comes down to the change $y = a + xb \to a + \sigma\epsilon + x(b + \sigma\epsilon)$, where σ is the standard deviation of the adjustment.

We can also consider the (sub) combinations and how the impact the performance. This was studied in [3], and comes under the name ‘Rainbow’.

References

[1] Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017.

[2] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017.

[3] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[5] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *International Conference on Learning Representations*, Puerto Rico, 2016.

[6] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[7] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.