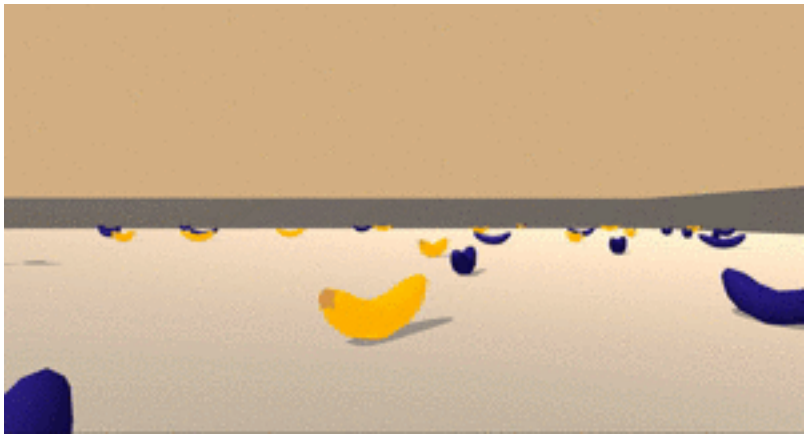
Project 1 Report: Navigation

TAN H. CAO*

July 14, 2020

1 Project Goal

The goal of this project is to train an agent to **navigate** and collect bananas in a large square world.



Reinforcement Learning framework:

(i) Reward: The agent will receive a reward of **+1** for collecting a yellow banana and a reward of **-1** for collecting a blue banana. The agent aims to collect as many yellow bananas as possible while avoiding blue bananas.

(ii) State: 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.

(iii) Actions: The agent can select one of the four discrete actions corresponding to:

- $\boxed{0}$ - move forward
- $\boxed{1}$ - move backward
- $\boxed{2}$ - turn left
- $\boxed{3}$ - turn right

(iv) Goal: The agent aims to learn how to best selection actions to maximize the accumulated reward.

(v) Task: The task is **episodic**, and to solve the environment, the agent get an average score of +13 over 100 consecutive episodes.

*Department of Applied Mathematics and Statistics, SUNY (State University of New York) Korea, Yeonsu-Gu, Incheon, Korea (tan.cao@stonybrook.edu).

2 Learning Algorithms

2.1 Introduction

Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps; for example, maximize the points won in a game over many moves. They can start from a blank slate, and under the right conditions they achieve superhuman performance. Like a child incentivized by spankings and candy, these algorithms are penalized when they make the wrong decisions and rewarded when they make the right ones – this is reinforcement.

In order to solve reinforcement learning (RL) problems, one possible approach is constructing a **Q-table** showing all possible *action* values to derive an optimal policy. There are two main types of methods that we studied in the course to solve RL problems with discrete state spaces: **Monte Carlo** (MC) and **Temporal Difference** (TD) methods. These two methods are used to construct the approximate values for the action value functions $Q(s, a)$ after running sufficiently large numbers of episodes. But there is a difference between them: the former one needs to complete an entire episode before updating the Q-table, while the latter updates it after every time step. There are actually three different kinds of TD control methods: SARSA, SARSA max, and expected SARSA. SARSA and expected SARSA are classified as **on-policy** TD methods, while SARSA max is considered as an **off-policy** TD method. Usually on-policy TD control methods have a better online performance than off-policy TD control methods.

It seems that, however, MC and TD methods are not good candidates for solving RL problems with large or continuous state spaces. One possible approach to reduce a continuous space to a discrete space is using some discretization techniques that are mentioned in the course: tile coding and coarse coding. Discretizing the state space can work efficiently for some particular environments. Once the state space is discretized, one can use existing algorithms (MC or TD) with little or no modification after that. Nevertheless, there are some limitations of this method. First, when the space is complicated, the number of discrete states is very large. Second, the values of nearby positions are expected to be similar or smoothly changing but the discretization method does not exploit this characteristic.

Another possible approach to problems with large state spaces is using neural networks. A long history about the integration of reinforcement learning and neural networks can be found in [1] and [2]. Deep learning, or deep neural networks, has been prevailing in reinforcement learning recently in games, robotics, natural language processing, etc; see [2] and the references therein. Deep learning and reinforcement learning, being selected as one of the MIT technology Review 10 Breakthrough Technologies in 2013 and 2017 respectively, will play their crucial role in achieving artificial general intelligence. David Silver, the major contributor of **AlphaGo**, even made a formula:

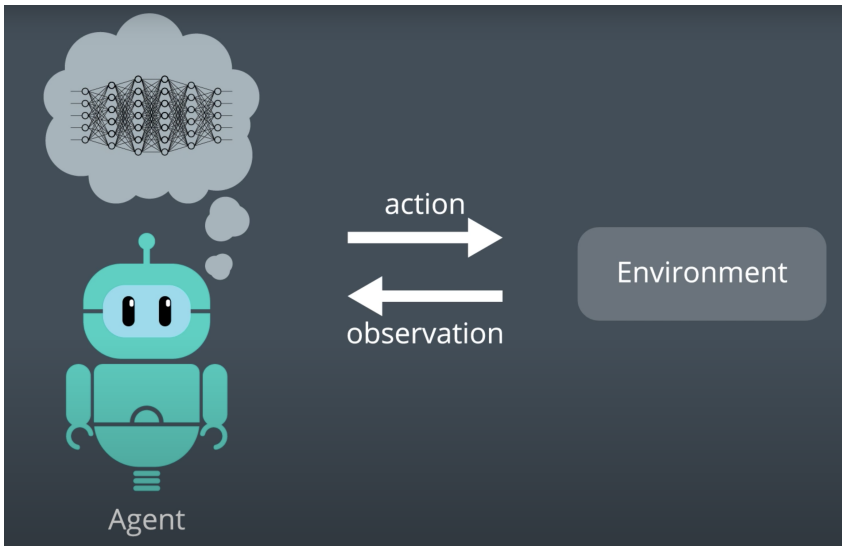
$$\text{artificial intelligence} = \text{reinforcement learning} + \text{deep learning}.$$

Deep reinforcement learning is actually a combination of reinforcement learning and deep learning. The idea is building up a function approximator $Q(s, a, w)$ with parameter w that enables us to obtain a good approximation of the value function $Q(s, a)$.

2.2 Deep Q-Learning

In deep Q-learning, nonlinear approximators are used to calculate the action values based directly on observation from the environment. We represent function approximators as a deep neural network. Then we will use deep learning to find optimal parameters w for these function approximators. Deep Q-learning algorithm represents the optimal value function $Q*(\cdot, \cdot)$ as a neural network (instead of a table). Unfortunately, reinforcement is notoriously unstable when neural networks are used to represent the action values. For instance, there are situations where the network weights can oscillate or diverge

because of high correlation between actions and states which may result in a very unstable and ineffective policy.



These instabilities can be addressed by using two techniques that slightly modified the base Q-learning algorithm: **experience replay** and **fixed Q-Target**.

- Experience replay can break the high correlation effects between consecutive experienced tuples by sampling them randomly out of order and consequently helps us learn a more robust policy.
- Fixed Q-Target is used to deal with another kind of correlation that Q-learning is susceptible to Q-learning update:

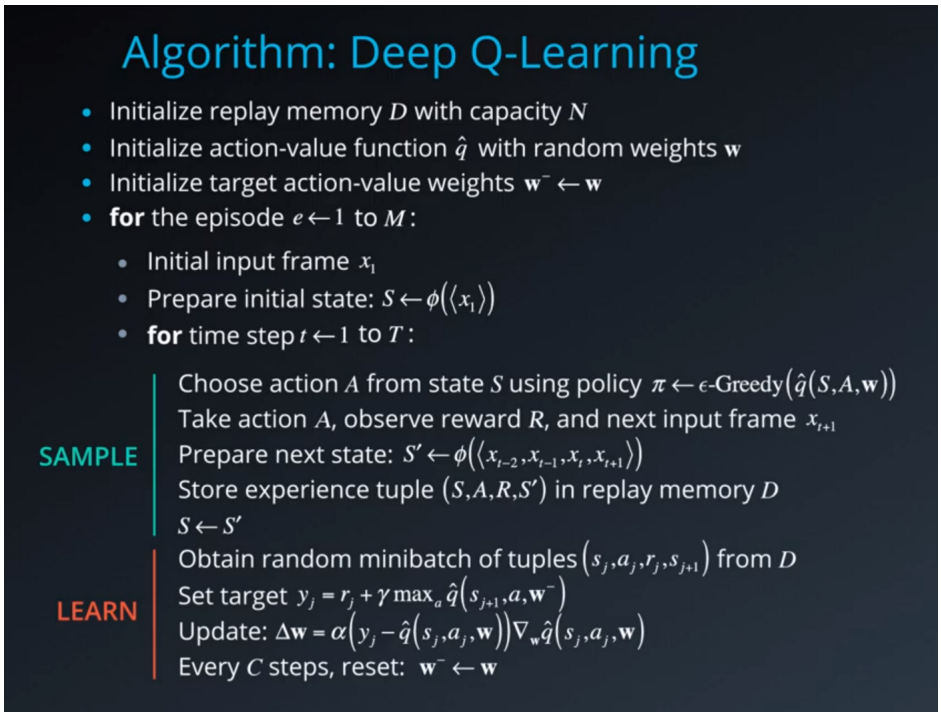
$$\Delta w = \alpha \left(\underbrace{R + \gamma \max_a \hat{q}(S', a, w)}_{\text{TD target}} - \underbrace{\hat{q}(S, A, w)}_{\text{current value}} \right) \nabla_w \hat{q}(S, A, w)$$

  where the TD target and the current value share the same parameter. This updating process is like chasing a moving target; e.g., training a donkey to walk straight by sitting on it and dangling a carrot in front. This, as a consequence, will make our learning algorithm oscillate or diverge. The idea of the fixed Q-Target is decoupling the parameter w in TD target and current value as

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

  where w^- is a copy of w that we don't change during the learning step. This improvement will make the learning algorithm more stable and less likely to diverge or fall into oscillations.

Here is the algorithm of Deep Q-Learning:



3 Code Implementation of Deep Q-Learning

To install the necessary packages and set up the environment, please refer to our Readme.md.

3.1 Basic Deep Q-Learning

In this section, we present the code implementing the above algorithm of the deep Q-learning. Our code here is a slight modification from the code of "Lunar Lander" from the course " Deep Reinforcement Learning Nanodegree".

The code consists of the following files:

(i) **model.py**: The *PyTorch QNetwork* is implemented in this Python file. This is a regular fully connected Deep Neural Network using the PyTorch Framework. This network will be trained to approximate the action value functions $Q(s, a)$ based on the environment observed states. The *architecture* of this neural network is given as follows:

- `Input nodes (37)`
- `Fully connected layer (64 nodes, Relu activation) --> Fully connected layer (64 nodes, Relu activation)`
- `Output noces (4)`

(ii) **dqn_agent**: this Python file includes the DQN agent and replay buffer implementation.

- DQN hyperparameters

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
```

- The DQN agent class has the following methods
  - `Constructor()`:
    - initializes two neural network: `qnetwork_local` and `qnetwork_target`
    - implements Adam optimization algorithm using `optim.Adam` to obtain the optimal parameters of `qnetwork_local`

```
self.qnetwork_local = QNetwork(state_size, action_size, seed, hidden_layer_sizes).to(device)
self.qnetwork_target = QNetwork(state_size, action_size, seed, hidden_layer_sizes).to(device)
self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)
```

    - initializes `memory buffer`.
  - `step()`:
    - stores a tuple (state, action, reward, next state, done) in the replay/buffer memory
    - every four steps and if there are enough samples in the memory (e.g. there are more than 64 = `batch size`), the agent will update the weights/parameters of the target network `qnetwork_target` using the weights/parameters from the local network `qnetwork_local`. This step will be explained later.
  - `act()`: returns actions for a given state as per current policy.
    - the current state will be passed to the neural network `qnetwork_local`
    - this network will be trained to output the approximation $\hat{q}(S, A, w)$

```
state = torch.from_numpy(state).float().unsqueeze(0).to(device)
self.qnetwork_local.eval()
with torch.no_grad():
    action_values = self.qnetwork_local(state)
self.qnetwork_local.train()
```

    - the agent will then select an action based on the **epsilon-greedy policy**.
  - `learn()`: updates parameters using given batch of experience tuples. For all tuples (states, actions, rewards, next states, dones) in the experiences, the agent will do the following:
    - gets the maximum predicted Q-values for the next states from the target model `qnetwork_target`

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

      This code means we
      + pass `next_states` to the target neural network `qnetwork_target`
      + detach the resulting to return a tensor copy detached from the graph (no gradient)
      + get the max value indexes in the second array: `max(1)[0]`
      + finally get a new Tensor of size one inserted at the given position: `unsqueeze(1)`.
      The code will give us $\max_{a \in \mathcal{A}} \hat{q}(\text{next\_state}, a, w)$, where w is the initial weight chosen randomly.
    - computes the value of `Q_targets` for current states:

$$\begin{aligned} Q_{\text{target}} &= R + \gamma Q_{\text{target\_next}}(1 - \text{dones}) \\ &= R + \gamma \max_{a \in \mathcal{A}} \hat{q}(\text{next\_state}, a, w)(1 - \text{dones}) \end{aligned}$$

    - gets the expected Q values from the local model `qnetwork_local` (being trained) by passing the `states` to the network `qnetwork_local`

```
Q_expected = self.qnetwork_local(states).gather(1, actions)
```

* computes the gap between `Q_expected` and `Q_targets`

```
loss = F.mse_loss(Q_expected, Q_targets)
```

* minimizes the loss function using the **gradient descent method**

```
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

* updates the target network using the `soft_update()` function

```
self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)
```

– `soft_update()`: updates the weights/parameters of the target neural network `qnetwork_target` using the ones of the local neural network `qnetwork_local`. Note that this update follows the rule below:

$$w_{\text{target}} = \tau w_{\text{local}} + (1-\tau) w_{\text{target}^-}$$

$$w_{\text{target}^-} = w_{\text{target}}$$

where w_{local} is the optimal weight of the local neural network `qnetwork_local` after training, and w_{target^-} is the weight of the current target neural network before the update. In our model, the hyperparameter $\tau = 0.001$ which implies that w_{target} is close to

- The `ReplayBuffer` class has the following methods:
  - `add()`: adds a new experience to memory
  - `sample()`: randomly samples a batch of experience steps for learning.

(iii) **Navigation.ipynb**: this Jupyter notebook has the reference project starter code and the code that we have implemented to train the agent.

3.2 Double DQN

It is well known that the recent DQN algorithm suffers from overestimations of action values. Consider the TD target in the Q-learning update Δw given by

$$\text{TD Target} = R + \gamma \max_a \hat{q}(S'a, w) = R + \gamma \hat{q}\left(S', \arg\max_a \hat{q}(S', a, w), w\right)$$

where the quantity $\arg\max_a \hat{q}(S', a, w)$ may make a mistake for the following reasons:

- Q-values are still evolving.
- The agents may not gather enough information to select the best action.
- The accuracy of Q-values depends on what actions have been tried and what neighboring states have been explored.

All of these may result in overestimating Q-values. To make our estimates more robust, we decouple the parameter w in the TD target by using w to select the best action $A = \arg\max_a \hat{q}(S', a, w)$ and using w^- to evaluate that action $\hat{q}(S', A, w^-)$. This will prevent the algorithm from propagating high rewards by chance. We refer the readers to [3] for more discussions in detail. The paper has five contributions.

(i) Explain why Q-learning can be overoptimistic in large-scale problems.

(ii) These overestimations are more common and severe om practice than previously acknowledged.

(iii) Double Q-learning can be used at scale to successfully reduce this overoptimism, resulting in more stable and reliable learning.

(iv) The authors proposed the double DQN using the existing architecture and DNN of the DQN algorithm without requiring additional networks or parameters.

(v) Double DQN finds better policies, obtaining new state-of-the-art results on the Atari 2600 domain.

In order to implement the Double DQN algorithm, we modified the code in **dqn_agent.py** by adding

```
if self.use_DDQN:
    best_action_next = self.qnetwork_local(next_states).detach().max(1)[1].unsqueeze(1)
    Q_targets_next = self.qnetwork_target(next_states).detach().gather(1, best_action_next)
```

And to use the Double DQN, we have to set `use_DDQN = True` in the function `__init__` in the class `Agent`.

3.3 Results

We first presented the results with the following data:

(i) $\#_{\text{episodes}} = 2000$

(ii) $t_{\max} = 1000$

(iii) $\epsilon_{\text{start}} = 1$

where the basic DQN (Deep Q-Network) is implemented. We changed the values of ϵ_{end} and ϵ_{decay} and recorded our result in the following table.

Agent	Model (FC1, FC2 units)	Eps Decay	Eps End	Result (# episodes)
DQN	64, 64	0.90	0.02	186
DQN	64, 64	0.90	0.02	185
DQN	64, 64	0.95	0.03	186
DQN	**64, 64**	0.95	**0.03**	**143**
DQN	64, 64	0.98	0.02	253
DQN	64, 64	0.98	0.02	200
DQN	64, 64	0.99	0.01	351
DQN	64, 64	0.99	0.01	318
DQN	32, 32	0.90	0.02	217
DQN	32, 32	0.90	0.02	225
DQN	32, 32	0.95	0.03	184
DQN	32, 32	0.95	0.03	174
DQN	32, 32	0.98	0.02	213
DQN	32, 32	0.98	0.02	229
DQN	**128, 32**	**0.90**	**0.02**	**142**
DQN	128, 32	0.95	0.03	157
DQN	**128, 32**	**0.95**	**0.03**	**141**
DQN	128, 32	0.98	0.02	229

The best performing agents with the model architecture (128, 32) were able to solve the environment in 141 – 229 episodes and in 207 episodes in average.

Using the same model architecture (128, 32) while changing $t_{\max} = 300$ and $\epsilon_{\text{start}} = 0.1$ we get the following results

Agent	Model (FC1, FC2 units)	Eps Decay	Eps End	Result (# episodes)
DQN	128, 32	0.90	0.01	154
DQN	128, 32	0.90	0.01	182
DQN	128, 32	0.95	0.01	151
DQN	128, 32	0.95	0.01	111
DQN	128, 32	0.98	0.01	133
DQN	128, 32	0.98	0.01	130
DQN	128, 32	**0.99**	0.01	**99**
DQN	128, 32	0.99	0.01	101
DQN	128, 32	0.99	0.01	150
DQN	128, 32	0.99	0.01	126
DQN	128, 32	0.99	0.01	168
DQN	128, 32	0.99	0.01	136
DQN	128, 32	0.99	0.01	145
DQN	128, 32	0.99	0.01	168
DQN	128, 32	0.99	0.01	128
DQN	128, 32	0.99	0.01	177

The agents can solve the environment in 141 episodes in average, which is better than the previous case. This somehow shows that allowing the agents to be greedy to high action value functions at the beginning would achieve a better result.

When we implemented DDQN (Double DQN) in our code, we get the following result with the same data above.

Agent	Model (FC1, FC2 units)	Eps Decay	Eps End	Result (# episodes)
DDQN	128, 32	0.90	0.01	142
DDQN	128, 32	0.90	0.01	142
DDQN	128, 32	0.95	0.03	160
DDQN	128, 32	0.95	0.03	160
DDQN	128, 32	0.98	0.02	123
DDQN	128, 32	0.98	0.02	98
DDQN	128, 32	0.99	0.01	119
DDQN	128, 32	0.99	0.01	140
DDQN	128, 32	0.99	0.01	121
DDQN	128, 32	0.99	0.01	83
DDQN	128, 32	0.99	0.01	215
DDQN	128, 32	0.99	0.01	161
DDQN	128, 32	0.99	0.01	107
DDQN	128, 32	0.99	0.01	157
DDQN	128, 32	0.99	0.01	141
DDQN	128, 32	0.99	0.01	128

In this case, the agents can solve the environments in 137 episodes in average.

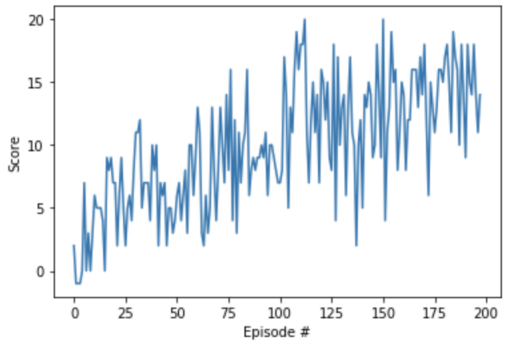
It looks like the agents with $\epsilon_{\text{decay}} = 0.99$ and $\epsilon_{\text{end}} = 0.01$ give the better results than the others. Also, the performance of the agents with $\epsilon_{\text{decay}} = 0.98$ and $\epsilon_{\text{end}} = 0.01$ is sometimes very good; for example, they can achieve the goal within 98 episodes. This gives us an idea of combining these two cases with $\epsilon_{\text{decay}} = 0.987$ and $\epsilon_{\text{end}} = 0.01$ to get better results.

Agent	Model (FC1, FC2 units)	Eps Decay	Eps End	Result (# episodes)
DDQN	128, 32	0.987	0.01	112
DDQN	128, 32	0.987	0.01	139
DDQN	128, 32	0.987	0.01	145
DDQN	128, 32	0.987	0.01	119
DDQN	128, 32	0.987	0.01	127
DDQN	128, 32	0.987	0.01	100
DDQN	128, 32	0.987	0.01	122
DDQN	128, 32	0.987	0.01	127
DDQN	128, 32	0.987	0.01	155
DDQN	128, 32	0.987	0.01	159

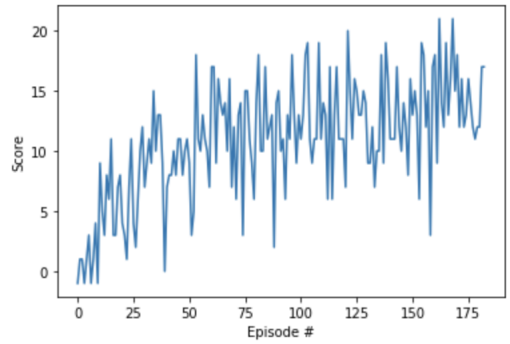
Overall, the agents can solve the environments in 125 episodes in average, which seems better than the previous cases.

The pictures below represent the two best agents with $\epsilon_{\text{decay}} = 0.98$, $\epsilon_{\text{end}} = 0.02$ and $\epsilon_{\text{decay}} = 0.99$, $\epsilon_{\text{end}} = 0.01$ respectively.

```
Episode 100     Average Score: 6.73 ,Time to train from the beginning:  1.0 mins
Episode 198     Average Score: 13.00
Environment solved in 98 episodes!      Average Score: 13.00 ,Total time to train:  2.0 mins
```




```
Episode 100     Average Score: 8.92 ,Time to train from the beginning:  1.0 mins
Episode 183     Average Score: 13.01
Environment solved in 83 episodes!      Average Score: 13.01 ,Total time to train:  2.0 mins
```



We observe that during the training the agents seem to get stuck in some state for a while, for example when they hit the wall. To some extent, they get confused on which direction they should follow and hence may make mistakes. The ϵ-greedy policy barely helps them overcome this problem, and this seems to be an issue in *Value-based* method: the agents are unlikely to get out of the stuck states.

4 Future Work

For the future work, we would like to work on the following:

- Training the agents directly from the environment's observed raw pixels instead of using the environment's internal states (37 dimensions), as discussed in this course.
- Training the agents by implementing the **Dueling DQN** algorithm and **Prioritized experience replay** algorithm as discussed in [4] and [5].
- We are particularly interested in training the agents by combining several extensions to the DQN algorithm: rainbow as discussed in [6].

We tried to implement the DDQN and Dueling DQN and obtained the following result. This kind of combination is expected to give better results, but we were not successful so far. We will think of how to select good hyperparameters in order to achieve the better performance in the future.

Agent	Model (FC1, FC2 units)	Eps Decay	Eps End	Result (# episodes)
DDQN + Dueling	128, 32	0.90	0.01	182
DDQN + Dueling	128, 32	0.95	0.03	167
DDQN + Dueling	128, 32	0.98	0.02	124
DDQN + Dueling	128, 32	0.99	0.01	140

5 Acknowledgments

We are gratefully indebted to Nilson Chapagain for his useful remarks and discussions on how to select good hyperparameters so that the agents could achieve their highest performance.

References

[1] Richard S. Sutton and Andrew G.Barto, Reinforcement Learning.

[2] Yuxi Li, Deep Reinforcement Learning: An overview.

[3] Hado V. Hasselt, Arthur Guez, and David Silver, Deep Reinforcement Learning with Double Q-Learning.

[4] Tom. S, John. Q, Ioannis. A, and David. S, Prioritized Experience Replay.

[5] Ziyu. W, Tom. S, Matteo. H, Hado H., Marc. L, and Nando F., Dueling Network Architectures for Deep Reinforcement Learning.

[6] Matteo. H, Joseph. M, Hado. H, Tom. S, Georg. O, Will. D, Dan. H, Bilal. P, Mohammad. A, and David. S, Rainbow: Combining Improvements in Deep Reinforcement Learning.