Navigation Udacity Deep RL project report

Overview:

This report illustrates the conducted work to pass the navigation udacity deep reinforcement learning project. We will present the implementation, results, and future work.

Link to the project repository: [github link]

1- Implementation details:

The structure of the project is as follows:

- **Experiments:** Directory that contains the configs.yaml files used to launch experiments. It summarizes roughly all the hyperparameters that serve to train the DQN agent.
- **Results:** Directory containing the checkpoints from the different trained agents and their corresponding average reward plots for every 100 episodes.
- **.Isort.cfg:** config file that specifies some code setting (line_length, packages,...).
- **.pre-commit-config.yaml:** config file to specify the hooks used when running `pre-commit`.
- **Core.py:** Serves as a wrapper to encapsulate the banana unity environment.
- **Dqn_agent.py:** Implementation of dqn inspired from the DQN course.
- **Environment.yml:** contains the dependencies and packages to be able to run the code.
- **Train.py:** Train the agent.
- **Evaluate.py:** Evaluate the agent
- **Model.py:** Contains two architectures (One for Vanilla DQN and other for dueling DQN) that serves to map the states and the actions and constructs the policy.
- **Utils.py:** Contains utility functions.

2- Learning algorithm:

In this project we used a DQN algorithm to train our agent.
Following is the pseudocode for Deep Q-learning with experience replay:

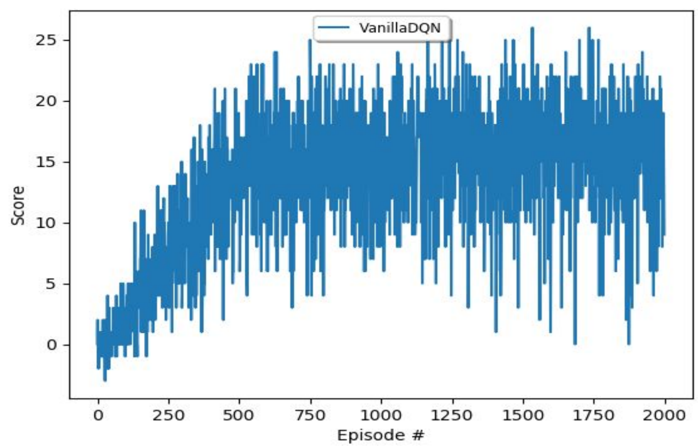
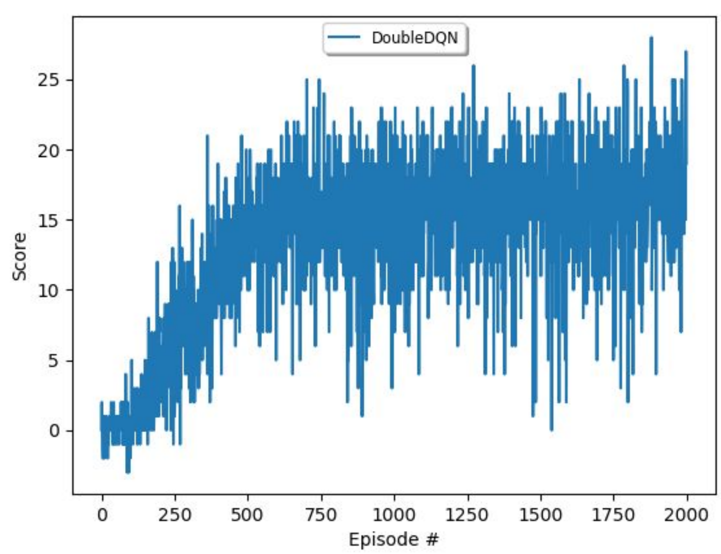
Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \mathrm{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

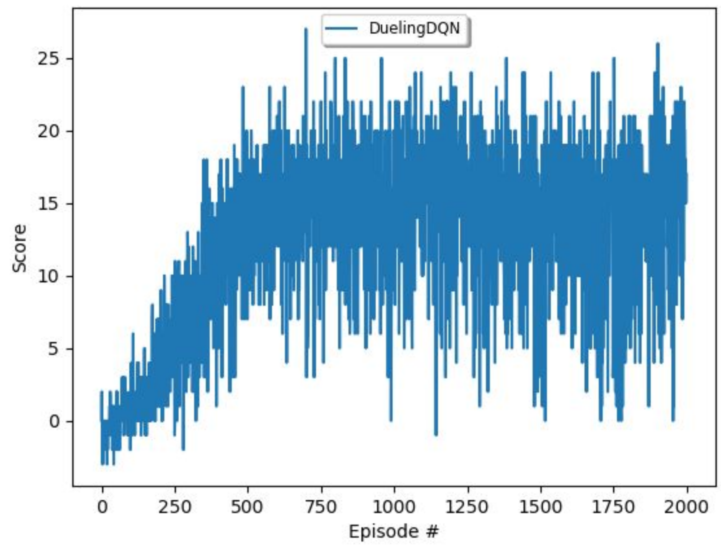
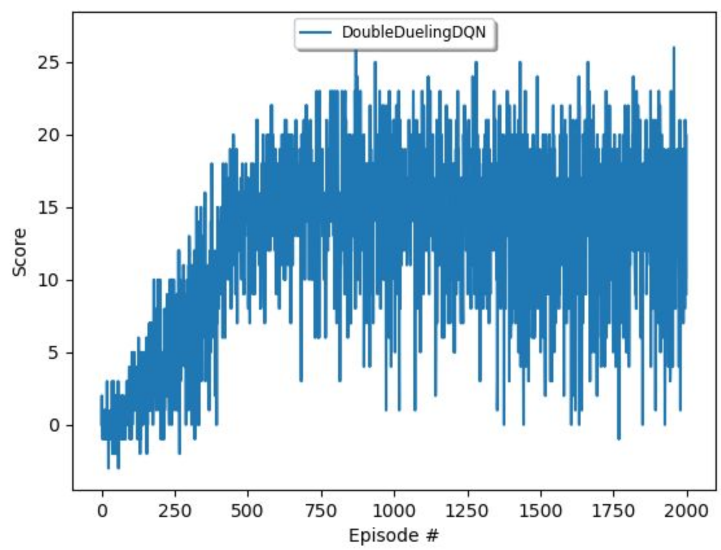
[Source = Towards data science blog]

→ For ***Vanilla DQN***, we used a basic architecture consisting of ***3 fully connected layers*** with ***Relu*** as activation function.
→ For ***Dueling DQN***, one ***fully connected layer*** with ***Relu*** activation, and then two separate streams:
The **value stream** is a FCL with relu activation and the same from the **advantage stream.**

The used ***hyperparameters*** can be found under experiments directory in git repository.

3- Results:

Experiment	Plot	Episodes before solved
Vanilla DQN		=~ 600
Double DQN		=~ 500

Dueling DQN		=~ 500
Dueling Double DQN		=~ 500

→ The convergence is faster using variants of DQN than the Vanilla DQN.
→ We were able to reach an average score of **17,32** for **100** episodes using double dqn.
→ Using a **lr = 0.001** gave us poor results. We tuned the learning rate and we found that **lr = 0.0005** was efficient.

4- Future work:

- Implement the 7 variants of dqn together and experiment with them.
- Investigate the possibility of using gradient based methods to train the agent from the next course.
- Try different and more sophisticated architecture.