Solving the LunarLanderContinuous-v2

Ron Ben Shimol (ID. 205610751), Ori Cohen (ID. 207375783)

1 Introduction

In this project we'll solve the LunarLanderContinuous-v2 environment as demonstrated in openai-gym.
"Gym is a toolkit for developing and comparing reinforcement learning algorithms" – gym documentation, gym environments have a shared interface, therefore one may find some gym-compatible challenges which are not part of the actual library.
Gym allows us to easily train and test our algorithm using simple commands like: env.reset(), env.render(), env.step() etc.

LunarLander challenge requires you to "navigate a lander to its landing pad".
It consists of two versions: Discrete and Continuous. The Continuous version, which is considered harder, is the version we're going to focus on. We'll try to:

1. **Maximize our agent's score** (points calculation defined in documentation) For detailed info see: https://gym.openai.com/envs/LunarLanderContinuous-v2/
2. Solve the problem as **fast** as we can, while fast refers to the number of episodes - "games" required in order to train the model.

Providing good solutions for an environment like LunarLander can help us achieve challenges of great importance, such as stabilizing a drone or robot.

1.1 Related Works

Several methods had been used in the past in order to solve this environment.
Q-learning, tries to find the optimal policy by learning the optimal Q-values for each state-action pair, using value iteration. It fails to achieve sufficient results even for the discrete action space.
Deep Q-network manages to solve the environment, however from our experience it takes a very large number of episodes before reaching the desired reward. DQN is basically a Q-learning algorithm, replacing the Q-table with a deep NN as the action-value function.
Although being able to solve the problem, further work should be done in order to achieve better results for complex environments, where it is not required to have the value of each action at every timestep. Gym Examples are Atari games.
Therefore, we will implement a Dueling DDQN algorithm, introduced in this article https://arxiv.org/pdf/1511.06581.pdf

2 Solution

2.1 General approach

Our approach was affected with some trial and error experiments with different algorithms.
First we tried the PPO algorithm, a relatively new algorithm introduced in 2017 with the help of some fellows at OpenAI. It is considered best in class, fast and accurate. We've read some papers, tried to implement it and finally came across a dead end, with poor results (avg score up to 100 in 700+ episodes) and hard to implement algorithm.
We believe that PPO could still probably provide the best results for this environment, however we wanted to explore other options as well.

Another approach was to go with the basics, DQN, nice and easy, but unfortunately DQN struggles with continuous space, due to the nature of the action selection process, calculating the Q-Learning "maximizer" actions is impossible in a continuous environment since we have infinite possible actions. There's a nice explanation here.
Eventually, we realized that since the problem is quite easy in discrete environments, we could come up with some "conversion" process, a simple mechanism to map the continuous to finite amount of actions, in other words discrete.
We mapped the available actions to different 72 actions - each action is a pair, indicating power at each engine.
Then we could solve the problem as discrete and hopefully get good results.
Eventually we chose Dueling DDQN, our approach was affected by this graph: https://www.researchgate.net/figure/DQN-DDQN-and-Duel-DDQN-performance-Results-were-normalized-by-subtracting-the-a-random_fig1_309738626
comparing the results of different algorithms.

We used few methodologies in order to improve our algorithm:

- **Experience Replay -** in order to "prevent action values from oscillating or diverging catastrophically", basically we want to "learn" from a few recent steps and not just the most recent one, so we'll store them, then sample some of them randomly and update the network accordingly. The rational is to prevent overfitting since there's correlations between the steps (detailed explanation)
- **Mean Square Error -** probably the most common **loss functions**, easy and reliable and even built-in. formula:

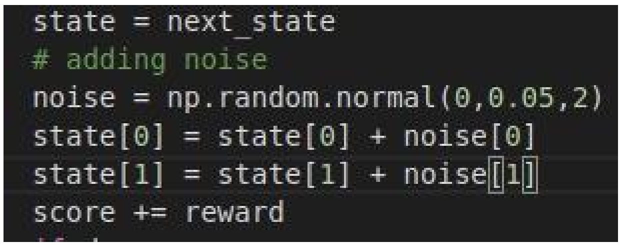
$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

- **Target Network** - instead of updating our main network each step, we'll update it slowly and gradually. We'll use a static (not changing) copy of the network, called Target network to calculate steps. Periodically we'll synchronize the target network with our main network. The motivation is to keep the training process more stable. Otherwise step B is directly affected by step A since the network N would act differently because it was just updated. Those constant updates proved to be harmful and therefore researchers came up with this solution. (detailed explanation, and here)

2.2 Design

The code was written in Python using Pytorch.
It took about **10-15 minutes** to train over notebook CPU until solving[1] the environment, with episodes ranging between 500-600. A remarkable result compared to our PPO implementation which took significantly longer (700+ episodes, 30~ minutes) to produce inferior results. We used few[2] Linear layers in our NN (2 to 5), with different activation functions (Sigmoid, ReLU, LeakyReLU etc.) and finally after some trial and error we fixed the parameters. We also used **Experience Replay** and **Target Network** (and soft updates) as mentioned above.

As requested, we treated our observation with uncertainty, creating noise in the following way:



```
state = next_state
# adding noise
noise = np.random.normal(0,0.05,2)
state[0] = state[0] + noise[0]
state[1] = state[1] + noise[1]
score += reward
```

Architecture
Double-DQN creates Q-value using two function estimators - one that estimates the advantage function, and another that estimates the value function. Advantage function estimates the benefits of taking a given action.

The architecture of a Dueling-DQN is similiar to DDQN, however in the Dueling version, the Deep NN is splitted to two streams - one for the state-value estimator, the other state-dependent action advantages[3] estimator.

[1] "Getting an average reward of 200 over 100 consecutive trials"
[2] To date, simple is better, very deep networks didn't prove, in our tests, to be more effective
[3] Advantage is defined as the subtraction between Q-value by V-value under policy π

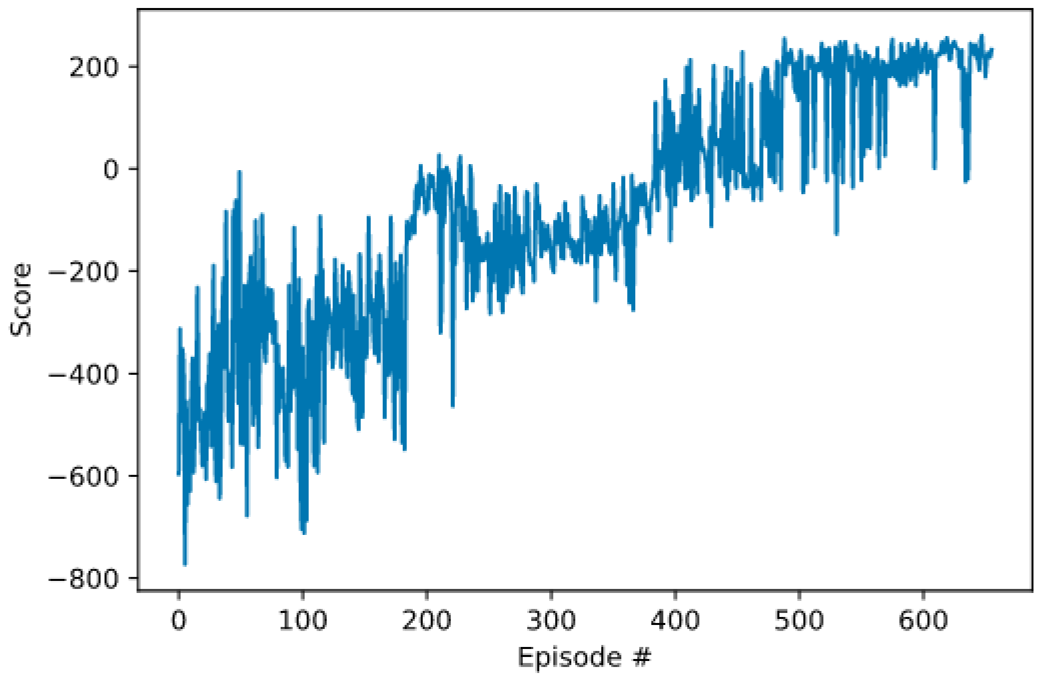
We split the network into two separate streams, one for estimating the state-value and the other for estimating state-dependent action advantages. After the two streams, the last module of the network combines the state-value and advantage outputs.

3 Experimental results

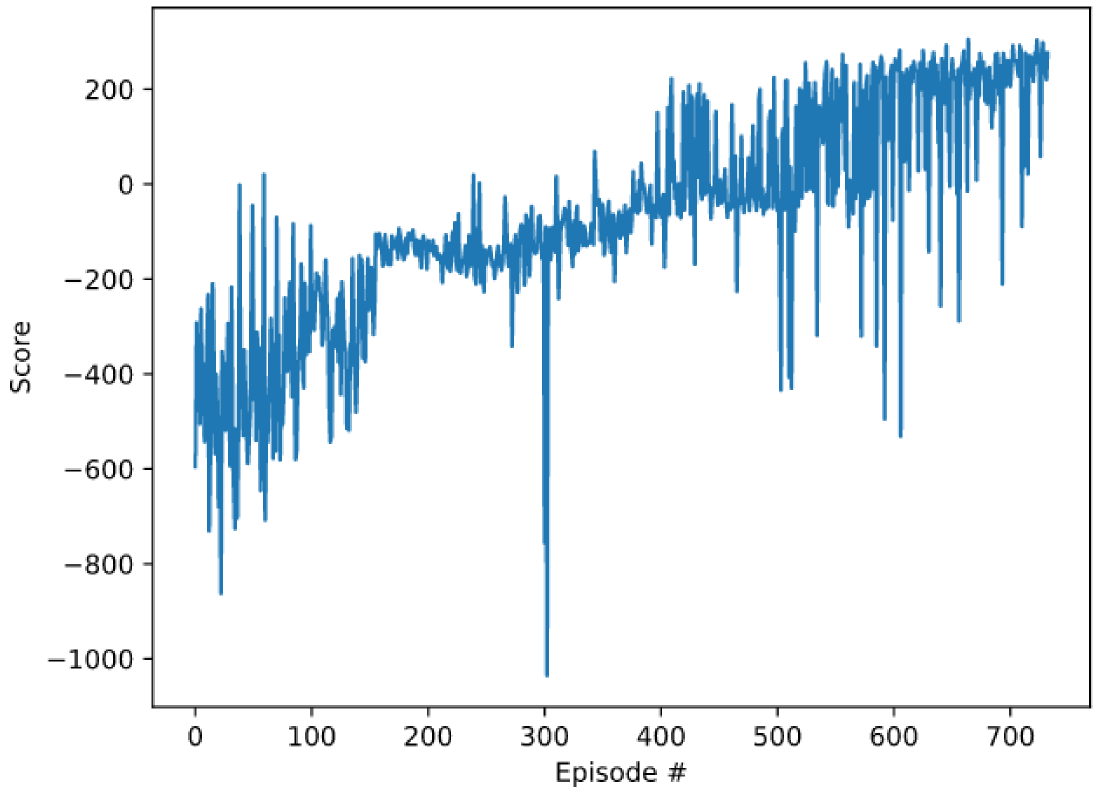
We'll compare models slight variations and their respective results:

Experiment No.	Input Size	fc1	fc2	fc3 value	fc4 value	fc3 advantage	fc4 advantage	Output size	Episodes until solved	Time To Solve (Min)
1	8	64 - relu	64 - relu	32 - relu	1	32 - relu	72	72		10
2	8	128 - relu	128 - relu	128 - relu	1	128 - relu	72	72	733	30

Experiment No. 1:



Experiment No. 2:



Unfortunately, in a noisy environment, we were unable to converge in under 1000 episodes.

We hypothesis that tuning hyperparameters - such as the discount factor, who's in charge of discounting future reward - can help with the convergence. Giving less weight to future rewards in a noisy environment is essential.

In addition, priority replay memory can be useful as well in order to learn from high-impact frames.

4 Discussion

We reached some interesting conclusions:

- More neurons in hidden layers provided worse results and significant increment in training time, see Experiment No. 2 above.
- Deeper networks (5+ layers) didn't provide better results
- Mapping continuous action space to discrete is not an easy task - we've tried mapping to 132, but the results were bad. Then, we tried 46 options - results were a bit better but still insufficient. The real improvement was when we decided to give the main engine much less options - we figured it doesn't need to be as "gentle" as the secondary engine. It proved to be right, and at 72 discrete options we observed a great improvement.

5 Future Work

- We believe that a successful implementation of the PPO algorithm could produce good results. It will be fascinating to compare those two algorithms.
- We didn't explore complex image oriented networks CNN and the likes, which may have potential of providing even better results, when receiving the observations from gym as a rgb-array.
- Better transformation from continuous to discrete action space may be achieved by forming a different NN - output layer size would be 2 - one for each action. Activation for the last layer would be tanh, normalizing the results between -1 and 1, treating the problem as more of a regression than classification.

6 Code

https://github.com/CohenOri/LunarLander-DuelingDQN