Project 2: Lunar Lander Solution with DQN

Feng Xiao

Oct, 28th, 2017

This report discusses method and procedures to solve Lunar Lander problem. Open AI Gym Lunar Lander was used as environment for our implementation, Keras was used as python deep learning library to implement neural networks in our algorithm. The implementation details are described in this paper and final results meet criteria in the instruction of this project.

Introduction

Lunar Lander Environment

This problem contains an 8-dimensional continuous state space and a discrete action space. There are four available actions: do nothing, fire left engine, fire right engine and fire main engine. Eight states includes: (x, y, vx, vy, θ, vθ, left-leg, right-leg). Where x and y are the 2D coordinates of the lander position, vθ, vx and vy are the velocity of lunar lander, θ is the angle of the lander, left-leg and right-leg are binary values to indicate if left or right leg of lander touches the ground.

Our goal is to land onto the landing pad at coordinates (0,0). Coordinates consist of the first two numbers in the state vector. The total reward for successful landing ranges from 100 - 140 points depending on lander placement on the pad. The lander will be punished if lander moves away from landing pad. An episode is defined as finished if the lander crashes or comes to ground. Each leg ground contact is worth positive 10 points. Firing main engine incurs a negative .3 penalty. Fuel is infinite in this environment. The problem is considered as solved when the score reaches 200 points or higher.

Reinforcement Learning and Deep Q Network:

Reinforcement Learning implements Markov Decision Process, in which current state is the only state that matters. In Reinforcement Learning, an agent learns optimized policy based only on the reward output. An action is the interaction between agent and environment and it is performed by agent. Each action paired with current state and next state will result in specific reward. The agent can learn to choose the policy with maximum future rewards.

Q-Learning is one of the most utilized algorithm in reinforcement learning. It results in optimal policy by learning action-Qvalue function including immediate reward, future ward for combinations of actions and states. Initially, all the value of Q are set to zero and such Q value table is to be updated after every actions taken. The rule of updating Q value works as following equation:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a \mathrm{Q}(s_{t+1}, \mathrm{a})) \quad (1)$$

Equation (1) is used to update Q value for each state-action pair for the exploration process. Finally, the Q value should reach optimal and policy can be generated by picking according action at each state.

Neural Networks

Implementing Q-tables to specific environment could be problematic because of its large size of states. Neural networks can represent Q-function and generate Q-values with one forwarding pass, which simplify Q learning process a lot. Deep neural network shows great advantage in analyzing large pixel data as input and mapping all of information into analyzable patterns. Such convolutional neural network condenses the input pixel data into few neurons and forward these information to next neural layer. Finally, the output layer is to be forwarded and generated simplified output data. Through convolutional neural network, data analysis becomes simplified and easy to access with high efficiency. The Deep Q Learning method is implemented through a high-level neural network API called Keras. Keras simplified environment building process and saves a lot of lines of codes.

Implementation Details:

In this experiment, several algorithms are taken into consideration, Deep Q Learning (DQN) is used for this implementation for these reasons: 1. The large amount of input data we need to take and the great ability of DQN to process pixel data. 2. The applied neural network simplified both of input and output data. 3. Ability to detect nonlinear relationship dependencies.

In Keras module, we built our convolutional neural network as below: for our DQN algorithm, four fully-connected layers of neural networks were used in this experiment: the first and last layer are for input layer and output layer respectively, the two middle layers are implemented with 40 neurons as dense layer (fully connected layer) and activation method as ReLU. The loss function is defined as mean square error and optimizer picked is Adam.

Moreover, for Q learning process, we used greedy-epsilon-policy in order to explore as many possibilities as possible, therefore, more information and possible state-action pairs can be explored. Finally, we can achieve a balancing point for exploration and exploitation by tuning epsilon.

After finishing setting up environment, optimization of hyperparameters becomes the core task of this implementation. After several thousands of runs, the output score still shows great divergence in each episode, which means there were great fluctuations in output data. In order to solve the problem of divergence, we imported an array of previous experiences called memory and it can add the information of state, action, reward and next state to the memory, therefore, the past experience can be used in this DQN to improve the performance of current decision making. However, due to the limit size of memory and limited time to run this program, divergence in near episodes got improved but not minimized in this experiment.

Experiment1:

One full run until get score of 200 or more was implemented, and the maximum episodes is 6000. The training data and trial data were recorded.

Experiment2:

Different combinations of hyperparameters are being tested to show the score evolution in episodes as output. Because of the extremely long time run for each set of parameters, the number of episode to compare for each set of parameters is fixed as 500, which should be able to distinguish differences in performance between two different set of parameters.

Results:

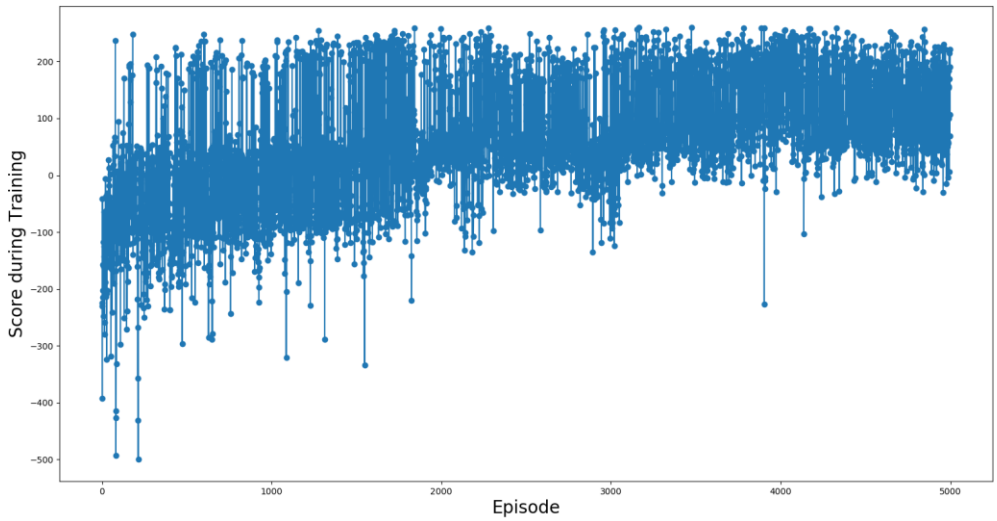

Figure 1. Training sets after 5K Episodes

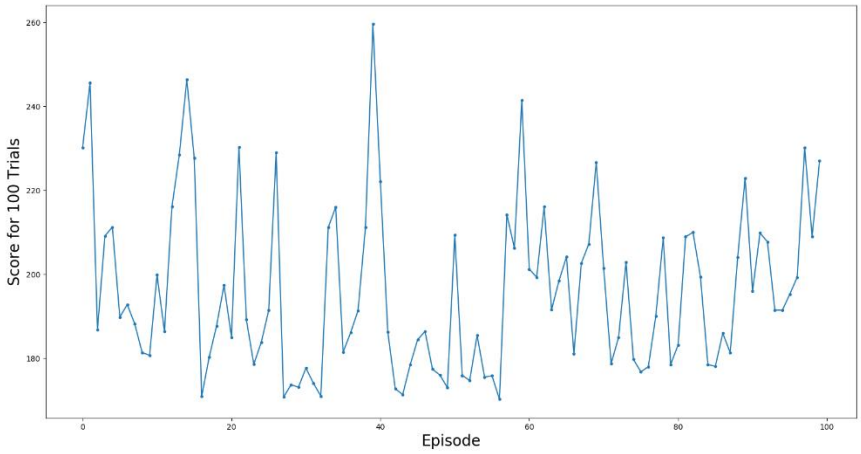

Figure 2. 100 consecutive trials after training.

Figure 1 and 2 shows the data from experiment 1. The epsilon was set to 0.1, whereas gamma is gradually increasing from 0.1 to 0.95 to improve the weight of future rewards instead of first few explorations in episodes. We can see the score over 200 at first few attempts, and the episodes over 200 points become more and more with the increment of learning process. The final reward after each episode was negative at first, and then gradually improves as DQN runs during training set. Finally, it reaches 202.6 of the average of 100 trials, which is above the average of 200 in the requirement.

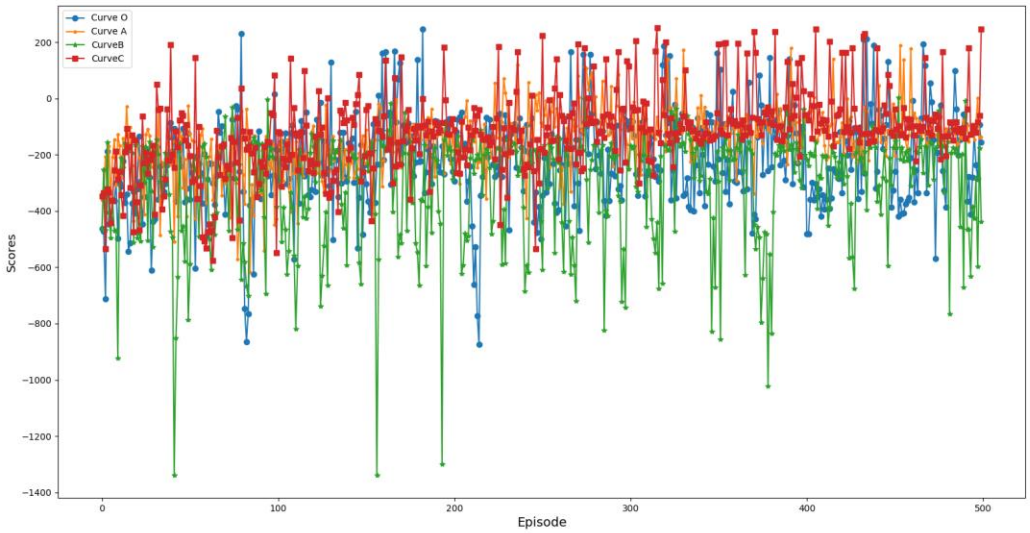

Figure 3. Different hyperparameter setup

Figure 3 shows the data from experiment 2 with different parameter sets. Curve A (yellow) uses epsilon-full-greedy at first (ε = 1.0), and then epsilon is gradually decayed through episodes; curve B (green) uses a larger learning rate = 0.1; curve C (red) uses gamma is fixed at 0.9. Curve O (blue) is the origin data grabbed from experiment 1 with first 500 episodes.

For curve A and curve O, the epsilon-full-greedy and fixed epsilon algorithm is to be compared. The average of epsilon-full-greedy is slightly better than a fixed epsilon at 0.1. This is because at first few attempts, exploration is a good choice when we set epsilon to 1, which also indicate that exploration in first few attempts generates a slight better result than regular method.

For curve B and curve O, the effect of learning rate is to be evaluated. If the learning rate is tuned to large value, the curve becomes more unstable and the fluctuation between episodes are larger, which results in the worst average reward. High learning rate results in huge growth in loss function, leading to inaccurate results. Therefore, picking appropriate learning rate is crucial in this example.

For curve C and curve O, the effect of gradually increasing gamma versus fixed gamma at 0.9 is evaluated. This one shows best performance with least fluctuation. The running results show that a larger gamma rate is increasing the time of running for each episode significantly, and the reward improved significantly through training sets of episodes because the learning process is evaluated with a larger gamma. To gradually increase gamma, future rewards are evaluated with higher weights then the result could be more accurate.

Moreover, another hyperparameter to be analyzed is the maximum number of episode. The number of episodes decides the maximum exploration and exploitation to this environment, and the number of episodes should improve the performance of our agent based on our knowledge in reinforcement learning, which matches our experiment result. Our approach did not run over 10K episodes due to limited time.

Conclusion

In this paper, DQN was implemented to solve lunar lander-v2 problem. First, 6000 episodes were run and achieved an average reward of 200 or higher per 100 episodes. Moreover, different hyperparameter sets are to be evaluated and analyzed to obtain information of performance under special hyperparameters. Four hyperparameters gamma, epsilon, learning rate and number of episodes were analyzed in this approach.

Reference:

1. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.

2. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

3. Keras: Python Deep Learning library. https://keras.io/

4. OpenAI Gym: https://github.com/openai/gym