Value-based Approximation for solving Lunar-Lander Problem

0. Introduction

This report aims to describe the detailed approach with respect to how to leverage neural network as a value-based approximator to play one of the Atari games- Lunar Lander- with human-level performance. The report will detail on various aspects including base network architecture design, tricks to train the network, fine-tuning hyperparameters and their effect as well as Cross-Validation and performance evaluation. The code can be found in the following link:

https://github.gatech.edu/tyao31/Project2.git

1. Several routes for problem solving

There are several model-free RL algorithms that could be leveraged to solve Lunar-Lander Problem,including value-based generalization method such as DQN, linear function approximator and other machine learning model, policy-based approach such as policy gradient and value-and-policy-based approach such as Actor Critic algorithm. For this report, a method that is similar to Double DQN is used to solve the problem.

2. Base Network Design

To solve Lunar Lander problem, neural network is used as the Q-Function approxiamator for its expressive power. The neural network used in the project consists of 3 hidden layers with 128, 64 and 10 neurons in each hidden layer respectively and takes as input a 8-dimensional vector which represents the state space for the agent and outputs 4 values- each value corresponds to a Q value for state-action pair. The non-linearity leveraged in the neural network is RELU, before which a Batch Normalization layer is used to facilitate the gradient flow.

3. Training Strategy

To make the training process more stable and effective, several novel ideas from the influential papers- Playing Atari with Deep Reinforcement Learning and Deep Reinforcement Learning with Double Q-Learning are exploited in this projectspecifically, experience replay is used- during the training phase, training data- a five value tuple that consists of current state, action, next reward, next state and a flag that indicates if the episode is done- from previous episodes are stored in a database and retrieved randomly to update the parameters of neural network using a ADAM optimizer. Using the method helps reusing the previous data and decorrelates noise and make the training examples more uncorrelated, thus make the training process more stable and easier to converge as mentioned by the author of Playing Atari with Deep Reinforcement Learning. Second, instead of updating the target network periodically as mentioned in the same paper, soft updating(Polyak Averaging) is utilized in this project since it is more stable to use soft update according to multiple trials using both methods. Additionally, the cost function chosen for this project is L1 smooth loss or huber loss function for its robust to the outliers, therefore the gradient flow will be smooth when performing model update. Unlike traditional machine learning and deep learning problem, for this project regularization is not used. Last but not least, the idea from Double DQN is exploited here for action selection resulting the best Q-value using current model rather than target model to further reduce variance.

4. Exploration Strategy

During the training process, for each step the training data needs to be drawn and saved in the replay buffer. In order to find the optimal policy, the agent needs to solve the exploitation-exploration dilemma. In this project epsilon-greedy strategy is leveraged for the agent to explore the state space. Since the state is continuous with 8 dimensions which can be fairly large, the epsilon value is scheduled in a way that it will gradually decay from 0.95 to 0.05 with a slow cooling rate which is similar to Simulated Annealing that has been covered in Randomized Optimization section in Machine Learning course. The intuition is that with a slow cooling rate, the agent will explore the state space more often, leading to higher probability to match the true probability distribution and the global optimal policy. The decaying formula is shown below which should look like similar to the Boltzmann distribution.

$$\varepsilon = \varepsilon_{start} + (\varepsilon_{start} - \varepsilon_{\mathrm{end}}) * \mathrm{e}^{cumulative\_steps/window\_size}$$

In this project, ε_{start} =0.95, $\varepsilon_{\mathrm{end}}$ =0.05, $window\_size$ = 260000.

In this project, the decaying rate is set to be slow such that even with 3500 episodes the epsilon value is around 0.30, however it is also worth noting that running the first 3500 episodes for the agent is fairly fast since the exploration is so aggressive initially. As a result, the whole training process doesn't take too long with 6000 episodes- around 3-4 hours in my laptop. Fewer episodes definitely are possible given higher cooling rate and other combinations of hyperparameters including learning rate, minibatch size and gamma. However in this project a flat decaying rate and a large number of episodes is chosen for the agent to achieve sufficient exploration of the state space such that we have more confidence that it will find a optima policy.

5. Hyperparameters Fine-tuning and Debugging

It is an important aspect to fine-tuning the hyperparameters in order to make the network converge to the true Q-function. There are several hyperparameters in the algorithm that will affect the training efficiency- learning rate, discounted reward rate, epsilon decaying rate and minibatch size. First thing to do is to find the value range of hyperparameters in conjunction with early stopping. For each hyperparameter, some extreme values are first experimented to find the effective range. Learning rate for instance, while set too big, 500 episodes will finish very fast- rendering the agent's behavior in the environment shows that it always use the same action in the entire episode, alluding that the learning rate is too big and the gradient might be exploding, as a result the agent takes the same action using the argmax method in the program. While the learning rate too small, after 500 episodes the agent doesn't learn anything as well which can also be identified in the rendered environment. As for gamma, since small gamma will lead the agent to be myopic, the final reward when landing in the ground may be quite small and the resulting gradient flow cannot affect the parameters update sufficiently. In conclusion, a large gamma value should be better for this problem.Experimenting with various gamma value ranging from 0.5 to 0.99, it is shown that small gamma will be likely to make the agent to find a suboptimal policy- flying around in the sky without landing for a long time. With a large gamma value, the agent learns to landing to maximize the reward. With respect to how to choose epsilon decaying rate is already discussed in the previous section thus won't be covered in this section. Our last hyperparameter for fine-tuning is minibatch size, it is intuitive that with a larger minibatch size, it is more beneficial to decorrelate the sequence and facilitate the gradient flow, however with a minibatch size being too large it will be more computationally expensive. Hence, in this project, 256 mini batches and 512 mini batches are experimented, and both results in similar performance. Once the value range of hyperparameters determined- 5e-5 to 5e-4 for learning rate, 0.9 to 0.99 for gamma, 260000 steps for epsilon decay and 256 for mini batch size- a grid search is performed to find the best combination of hyperparameters that will lead to the best total reward for the agent playing Lunar Lander game. One last idea used in the training process is that every 50 episodes when the total trained episodes exceeds 3000, the agent will run 10 episodes and the average total reward is evaluated with model parameters being saved once the average total reward exceeds 200 points. The reason for doing this is that the agent's performance is very sensitive to the update of model parameters, even with sufficient training, the total reward for each episode may still fluctuate. Hence, periodically performing cross-validation helps to find model parameters that generalize well for the agent.

6. Training/Test time performance

In the last section, several figures illustrates what's happening during training process and the average total reward over 100 consecutive trials under testing time. The first figure illustrates the cost for every 200 steps over the entire training process- around 1250000 steps and 6000 episodes. As shown in the figure, although the loss is noisy, the cost first arises then decreases gradually. Second figure illustrates the total reward for every episodes in the training process. As can be seen, after sufficient exploration- with about 4500 episodes, epsilon decreases to 0.1- the performance is getting better and better and stabilize at around 200 points. Figure 3 illustrates that for the last 500 episodes, the total rewards exceed 200 points for most episodes in training time.

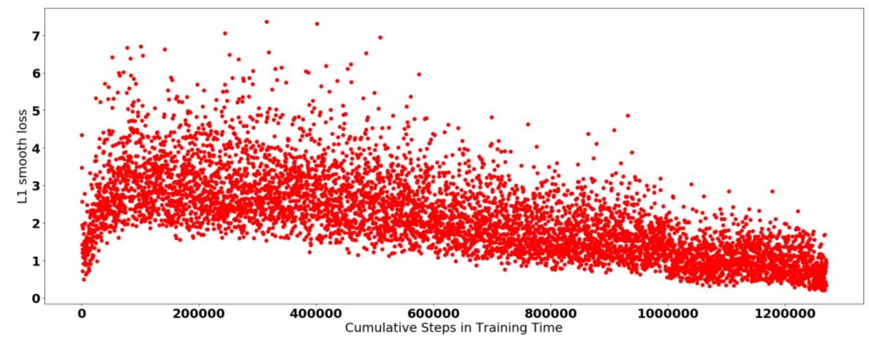
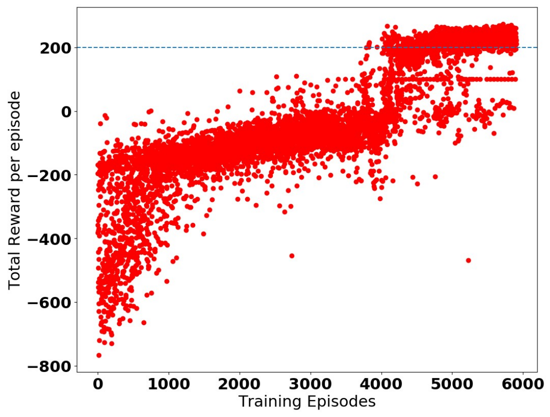


Fig.1 L1 smooth loss in Training Process with 6000 episodes and 1250000 steps



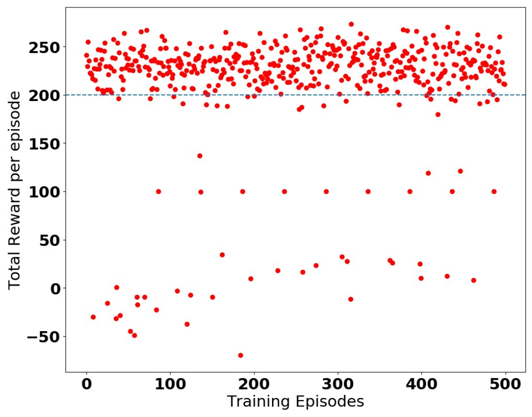


Fig.2 Total Reward for each episodes in training time *Fig.3 Total Reward for last 500 episodes in training time*

Last figure shows that in the validation time, with the best model saved at training time which achieves 240 points averagely for 10 episodes, the average score achieves 230.426 over consecutive 100 episodes.

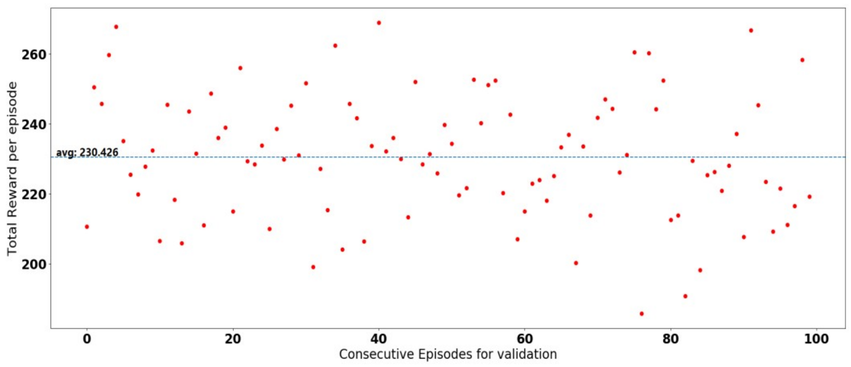


Fig.4 Total Reward for 100 consecutive episodes in testing time