Prasad Vaidya

Project 2:- Lunar Lander

CS 7642

Latest Github (git #) commit: 8f2bb2bbf09af7c0a9e229a070c26ad97c775da5

March 19, 2021

Introduction

This paper attempts to create & train a reinforcement learning (RL) agent for lunar landing (a game provided by open AI gym). In the game, each state is represented by 8 dimensions representing lander's location (horizontal & vertical co-ordinates), its speed(horizontal,vertical & angular), it's angle and if the first or second leg has contact with the surface. At each state, the agent can take 4 actions (do nothing, fire left, right or main engine). Depending on the action taken, the environment will give reward and present next state. And than agent will take another action accordingly and this will continue till the episode ends. With these environment dynamics in perspective, the goal of the paper was to train the agent to land at the launching pad located at co-ordinates (0,0). In order to train this agent, Q-learning estimated by function approximation was utilized.

Q-learning & Function Approximation

Q-learning is a model free RL algorithm to learn state-action value (Q(s,a)) function which is used to develop a policy that dictates what action to take in which state. Q(s,a) can be learnt recursively based on the Bellman Optimality Equation as shown below:

$$Q(s,a) = r(s,a) + \gamma \times \max_a \times Q(s',a) \tag{1}$$

In the above equation, s is the current state, a is the action taken, s' is next state and γ is the discount factor. The Bellman equation along with temporal difference learning enables finding Q(s,a) in iterative manner known as Q-learning. Essentially, Q(s,a) is randomly initialized for all states and actions and than at each step/iteration the Q(s,a) for all pairs, is updated using values from previous step by using the following equation till convergence:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \times \max_a \times Q(S_{t+1},A_t) - Q(S_t,A_t)] \tag{2}$$

In the above equation, t is the time-step to indicate current state vs next state. The term is $R_{t+1} + \gamma \times \max_a \times Q(S_{t+1},A_t)$ in equation 2 is the known as the target value and the difference between target value and current value ($Q(S_t,A_t)$) in combination with learning rate (α) dictates by how much to update the current value for Q. Although, this method works, the issue with this method is it requires enumeration of all Q-values for each state and action pair. The current problem of lunar lander at hand has infinite state space and so it's not feasible to enumerate all the Q-values. Hence, for such cases, function approximation is used for Q-learning. Function approximation is basically a mapping from state to state-action value using some function with a goal to minimize a loss function based difference between actual Q(s,a) and predicted $\hat{Q}(s,a)$. For the current project, deep neural network (DNN) was used as function approximator for Q-learning hence the name Deep Q Network (DQN).

Deep Q Network (DQN) Architecture & Training

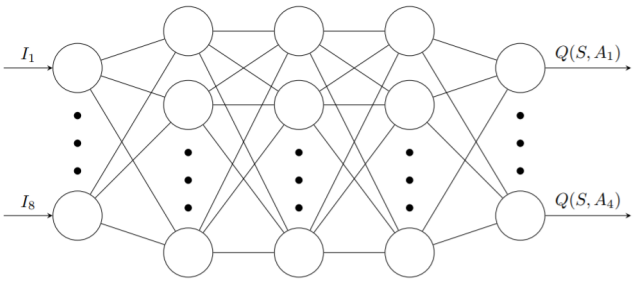

Figure 1 A

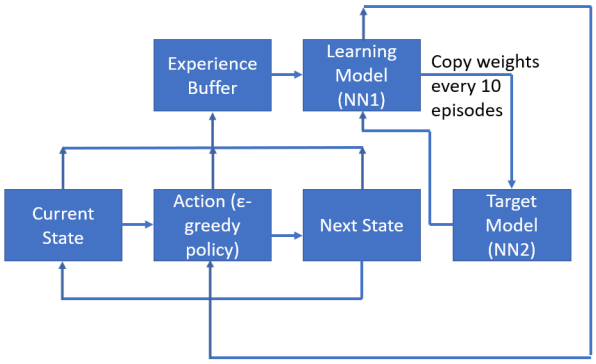

Figure 1B

The DQN was built using two neural networks (NN) one for learning (NN1) and other to create target (NN2) as shown in Figure 1B. Both the NN's consisted of an input layer (containing 8 neurons - one for each dimension of the state), 3 hidden layers (each containing 24 neurons) and one output layer (containing 4 neurons - one for each action). For all the dense layers, 'ReLU' activation was used except for output, where 'None' or linear activation was used. The linear activation was used to predict Q(s,a). Mean squared error (MSE) was used as a loss function (as explained above) to train the DQN and Adam optimizer was used to optimize the loss function.

Before the training, the weights for learning NN were initialized randomly and were transferred to target NN. The environment was reset to initial position and than action was taken. The decision of what action to take at

each step was driven by ϵ-greedy policy wherein the value of ϵ decayed with number of episodes based on ϵ decay rate with minimum ϵ of 0.01. Based on this policy, once the action was taken, the environment presented next state and rewards. This interaction was added as tuple (consisting of current state, action taken, reward, next state, done) to episode replay buffer.After every 2 steps, a sample of M random observations from replay buffer were taken to train the learning NN (to mimic Episode replay [3] training). For training, the current value for $Q(S_t, A_t)$ was predicted by learning NN, while the Q part of target value ($Q(S_{t+1}, A_t)$) was predicted by target NN. However, no fitting was done for the Target NN. The target NN was updated by copying the weights from learning NN after every 10 episodes. The training & updating process continued till reward goal or 5000 episodes was reached. Once this methodology was finalized, than various hyper-parameters were evaluated and optimized to determine best training set.

Hyper Parameters Optimization & Evaluation of Trained Agent

To determine the effect of hyper-parameters while minimizing effect of environment stochasticity, each experiment was run **5 times for 5000 episodes** for each set of hyper-parameters. The experiment was run in such a way that agent kept training till it achieved an average episodic reward (sum of all rewards in episode till episode ends) of at least **210 in 100 previous episodes**. If the score of 210 was achieved in less than 5000 episodes, than no further training was done and the remaining episodes were run as an evaluation with greedy policy to see how the agent generalized.

Two types of hyper-parameters were evaluated. The first type of hyper-parameters were specific to NN training while the second type of hyper-parameters were related to RL. Three experiments were performed for hyper-parameters specific to NN training. The first experiment evaluated the effect of maximum buffer length of episode replay buffer, the second experiment evaluated the effect of batch size (number of samples randomly selected from replay buffer for training NN) and the third experiment evaluated effect of learning rate (α) for Adam optimizer. All the 3 experiments evaluated their effect against reward/episode. Similarly, for RL hyper-parameters two other experiments were conducted. The first experiment evaluated the effect of ϵ-decay rate while the second experiment evaluated effect of discount factor (γ). Both experiments evaluated their effect against reward/episode. The hyper-parameters tested for experiments can be found in table shown below. From the table below, it can be observed that for each experiment, only the hyper-parameter in question was varied while all the other hyper-parameters were held constant. (For instance, in max buffer length experiment, only max buffer length was varied between 1000,10000 & 100000, while the other hyper-parameters were constant for each value of max buffer length).

Experiment Name	Max Buffer Length	Batch Size	Learning Rate (α)	ϵ-decay rate	Discount Factor (γ)
Maximum Buffer Length	1000,10000,100000	64	0.001	0.997	0.99
Batch Size	10000	16,32,64,128	0.001	0.997	0.99
Learning Rate	10000	64	0.01,0.001,0.0001	0.997	0.99
Epsilon Decay	10000	64	0.001	0.99,0.997,0.999	0.99
Discount Factor	10000	64	0.001	0.997	0.9,0.99,0.999

After hyper-parameters analysis, the best set of hyper-parameters were selected. These hyper-parameters were used to build a new agent with stringent criterion (average reward of at least **250 in previous 100 episodes**). This agent was evaluated by running a greedy policy for 1000 episodes. This evaluation is henceforth, called as **"No Uncertainty Evaluation"** as this evaluation did not involve addition of any uncertainty. To further check the robustness of the best agent, two kinds of uncertainties were added to the system [2]. The first uncertainty (henceforth called as "**Location Uncertainty**") impacted the state returned by the environment. Instead of using the state as returned by the environment, the state was modified by adding a perturbation (random number generated using the state's x co-ordinate as mean and 0.05 std. deviation) to state's x co-ordinate. The second uncertainty impacted action (henceforth called "**Engine Failure Uncertainty**") wherein all the engines fire only with probability of 0.8, while with probability of 0.2 even if the policy recommends firing the engine, the engine will fail and the action it will take is "do nothing". These uncertainties were used only for evaluating the agent and not for training.

Results & Discussions

The first set of experiments involved evaluating the effect of hyper-parameters on training the agent. It was observed that due to environment stochasticity, there were some differences in if or when the agent achieved the average reward of 210 in previous 100 episodes. Hence, each experiment with each set of hyper-parameters was run for 5 times for 5000 episodes and the charts are based on average of the 5 runs. **Within each chart, the transparent and noisy data represents the episodic reward while the solid curve with same color represents the moving average of episodic rewards over the last 100 episodes**.

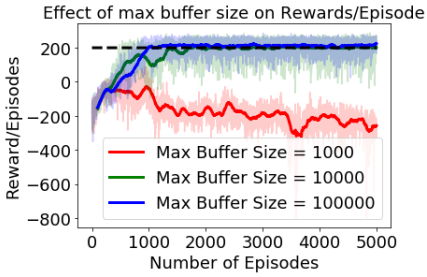

Figure 2A

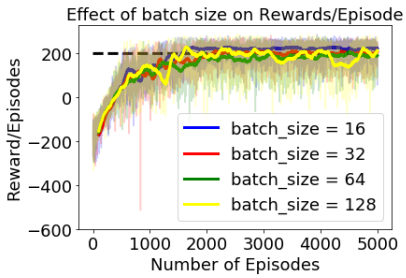

Figure 2B

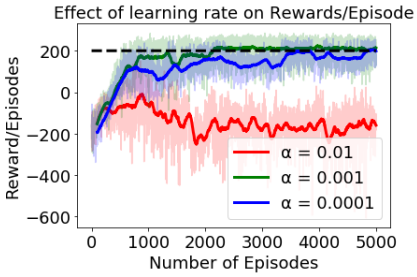

Figure 2C

Figure 2 shows the effect of NN specific hyper-parameters while training the agent. Specifically, Figure 2A shows the effect of maximum length of replay buffer used for Experience Replay. While in literature, the default value for experience replay buffer is usually set to 10^6 [6], this paper evaluated the effect of different buffer sizes (Also smaller buffer size were considered as this problem seems to be relatively on smaller scale compared to other problems in literature). For the buffer size of 1000 (4 to 10 previous episodes), the agent never learnt in any of the 5 runs while for the two other buffer sizes lengths 10,000 (40 - 100 previous episodes) and 100,000 (400 - 1000 previous episodes), the agent learnt in all 5 runs. Figure 2A shows the average of 5 runs, where we can see the same thing, that for buffer length of 1000, the agent never reached avg. reward 0+ while for both other lengths, it achieved the goal. This observation is probably result of trade-off between data freshness and data correlation. For a smaller buffer (1000 size), while the data used for learning will be most recent, it is probably highly correlated (batch size of 64, so 6.4% of data is used for training). Hence, due to this correlation the agent probably never learnt correct function. On the other hand, [6] showed that extremely large buffer size also fails to achieve convergence and that may be due the fact that large buffer size will retain lot of old data and so the agent is training on lot of outdated data. Although, the issue with large buffer length was not encountered for the buffer lengths chosen as both 10K and 100K converged in all 5 runs, keeping the literature [6] in mind, the **medium buffer length of 10,000** was deemed optimal and used in further experiments. Figure 2B shows the effect of batch size (number of samples selected from episode replay buffer) in which we can see that all the batch sizes seem to do well. All the batch sizes reached the goal of 210 in all 5 runs except batch size of 128 (which reached the reward in 4 out of 5 runs). For the specific case of 128, there might be some correlation between the data as it takes 1.3% data for training and so may not have reached the goal. However, this observation implies that batch sizes tested didn't have significant impact on training. The **batch size of 16** was found to be one which had consistently highest reward after training and so it was used to train the best model. Figure 2C shows the effect of different learning rate for Adam optimizer to achieve the goal. For learning rate experiment, $\alpha = 0.01$ never reached average reward ≥ 0 in any of the 5 runs. On the other hand, $\alpha = 0.001$ achieved goal in all the 5 runs and $\alpha = 0.0001$ achieved goal in 4 out of 5 runs. If the learning rate is high, than it can cause weights changes to be large which might lead optimizer to overshoot minimum and not learn. And so $\alpha = 0.01$ may be high. On the other hand, if the learning rate is low, than it may take longer to learn and $\alpha = 0.0001$ might be low as we can see from Fig 2C, it took 3500+ to train and was little sub-optimal. The $\alpha = 0.001$ seems optimum as it converged to 210 value in all the 5 runs in around 1000 to 2000 episodes and performed optimally after training. Hence, the α **= 0.001** was deemed optimal to train the best model. While this experiment, evaluated the effect of α for Adam optimizer, the optimizer itself (Adam vs RMSprop vs SGD vs SGD + momentum) can be chosen as hyper-parameter and this could be another experiment for hyper-parameters optimization for NN. However, Adam optimizer has been very widely used for NN and so it was used for this study.

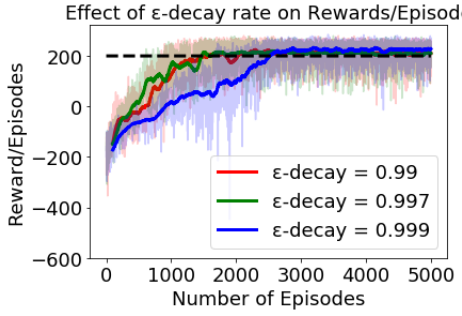

Figure 3A

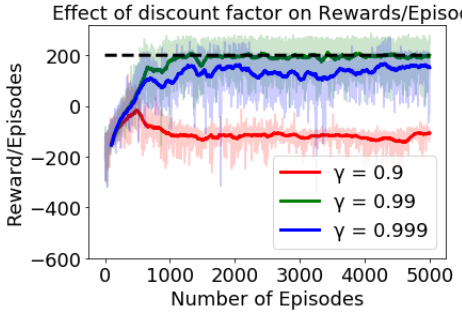

Figure 3B

Apart from hyper-parameter optimization for NN, hyper-parameters specific to RL were also optimized (Figure 3). Specifically, to check the effect of exploration vs exploitation, ϵ-greedy policy with different ϵ-decay rates were used (Figure 3A). The three ϵ-decay rates were chosen such that almost complete exploitation will start at different times (when ϵ decay rate was 0.99, then almost complete exploitation will start at 500 episodes $\epsilon = 0.99^{500} = 0.0065$, for ϵ rate = 0.997, almost complete exploitation will start at 1500 and for ϵ rate = 0.999, almost complete exploitation will start at 4500 episodes. Also, if the agent did not reach the goal till complete exploitation, given min ϵ was 0.01 agent still kept on training till it reached goal or 5000 episodes). All the three decay rates achieved the goal in all 5 runs. Figure 3A, shows the average of 5 runs for the various ϵ decay rate and it can be seen that all the 3 ϵ decay rates did well and performed optimally after training as well. However, the ϵ decay rate = 0.999 took around 3000 episodes to reach the reward indicating that because the decay rate was slow, there was lot of exploration and so it took time for the agent to learn. Based on this observation, for the range of the ϵ-decay rate that was tested, that range doesn't have significant impact on training the agent. May be, if we lower the ϵ-decay rate below 0.99, at certain value, the exploration will be really minimal and the agent may never learn. Anyway, **ϵ-decay rate of 0.997** was found to be one which had consistently highest reward after training and so it was used to train the best model. On the other hand, Figure 3B shows that the discount rate (γ) seems to have significant impact on training the agent. For $\gamma = 0.9$, the agent never learnt in the 5 trials. However, for the value of $\gamma = 0.99$, the agent achieved goal in all the 5 runs, while for $\gamma = 0.999$, agent achieved goal in 4 out of 5 runs. This may be indicative that the low γ made it hard for the agent to train because of less planning horizon. On the other hand, $\gamma = 0.999$ although did reach the goal, it performed sub-optimally during evaluation. This indicates that the agent didnot generalize well. And this may probably be due the reason that when γ is high and close to 1, there might be instabilities. The medium γ **of 0.99** converged in all episodes and performed optimally in evaluation too as it doesn't suffer from both the issues mentioned above and hence was chosen as the best hyper-parameter.

Figure 4 A

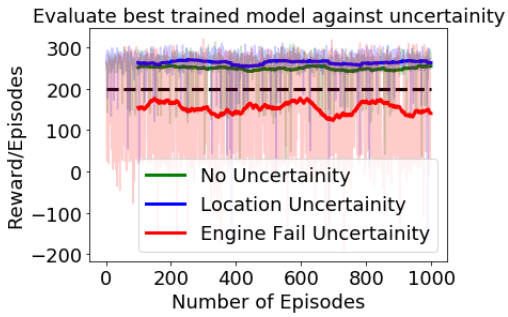

Figure 4B

The best hyper-parameters based on above experiments were : max buffer length of episode replay = 10000, batch size of 16, $\alpha = 0.001$, ϵ-decay rate = 0.997 and $\gamma = 0.99$. These hyper-parameters were used to train the agent on more stringent criterion of 250 average reward in last 100 episodes. Figure 4A denotes that with these hyper-parameters, the agent learned successfully to achieve the new goal, however, it took it more than 2000 episodes to achieve that. This observation makes sense since we have higher goal compared to previous studies. Once, the training was complete, the agent was than evaluated for next 1000 episodes under a) No Uncertainty (Using the same state as given by environment and taking same action as recommended by the policy) b) Location Uncertainty and c) Engine Failure Uncertainty. Figure 4B shows the agent had learned really well and performed extremely well under both no uncertainty as well location uncertainty. This observation confirms that the agent generalized the system well. In all the 1000 episodes, the average reward per 100 episodes was 240+ for both types of uncertainties. However, the agent performed little sub-optimally for engine failure uncertainty. Its was hard for agent to achieve optimality when the action recommended by policy was not correctly executed 20% of the times due to engine anomaly. The same kind of trend was observed by [2]. So one of the alternatives could be to train agent using this uncertainty. However, [2] showed that such model performed poorly. And so although, sub-optimal, given that our average reward is 150+ under this uncertainty which is hard to train, it shows that this agent is really robust.

Alternative Training methods

This paper uses DQN with two NN and experience replay to train the RL agent for lunar landing. However, there are several alternatives for training RL agent. Broadly the alternatives can be classified into three classes a) Generalization function b) type of policy c) Modifications to NN. For lunar lander, given that it has infinite state space, visiting every state multiple times is not feasible and so some sort of generalization function should be applied to solve it. Although, DQN was applied to in this paper, random forest regressor, K-nearest regressor, as well as

linear models could also be applied to generalize Q-learning. However, it was hard to find lot of literature on efficacy of these other methods and hence DQN was selected as function approximator . On the other hand, apart from Q-learning, SARSA policy could be used for training the RL agent. However, as shown by [2], the average reward of native SARSA for lunar lander was 170+ as opposed to DQN which was 200+, hence SARSA was not implemented for this paper.

Apart the basic DQN, training for NN as well as NN architecture itself could be changed to improve the learning. For instance, current paper uses uniform experience replay to train NN which can be replaced by prioritized experience replay[4]. While in uniform replay, the sample for training is drawn randomly, for prioritized experience replay, the samples are weighted based on importance and the important ones are used for training more frequently. Another alternative could be, instead of double DQN, dueling DQN [5] architecture consisting of two NNs (one for value function and another for state related action advantage) can be used. While all these NN use experience replay to avoid correlation between the data for NN training stability, the correlated data might also be useful for training in some settings. Hence, an addition to the DQN can be a recurrent NN/LSTM as demonstrated by [1].

Conclusions

This paper uses DQN for function approximation to learn the state space and predict Q(S,A) to train the agent to land the lunar lander within the launch pad. The paper demonstrated that the training is highly dependent on the hyper-parameter optimization. Specifically, the maximum episode replay buffer length and learning rate for Adam optimizer had a huge effect on training the DQN. While for both medium and low learning rate, agent learned the goal, it took more time to learn for low learning rate and the agent never learnt when the learning rate was high. For the case of replay buffer, the agent never learnt when the buffer size was extremely low (around 10 previous episodes), while the agent was successful in learning when the buffer size was around 100 to 1000 previous episodes. Similarly, even the hyper-parameters specific to RL also impacted the agent's training ability. When the discount factor (γ) was low the agent did not learn at all, while when γ was high, it was close to optimal but still did not reach the goal. Only for medium γ the agent actually achieved the goal. On the other hand, for ϵ-decay optimization, all the three ϵ decay rate that were tested converged, however, the lowest ϵ decay rate took more time to converge due to more exploration. Based on these experience, the best hyper-parameters were chosen and the model was trained from scratch. The best model was extremely robust in evaluation where the average episodic reward for last 100 episodes for 1000 episodes was 250+. To further evaluate the agent, two uncertainties were added. The agent again showed it robustness for location Uncertainty case. However, the best model did fall little short for Engine Fail Uncertainty case which hard to train. So based on these observations, it can be concluded that DQN architecture selected above with the optimized hyper-parameters was successful in training the agent to learn and to land the lunar lander at launch pad.

References

[1] Clare Chen, Vincent Ying, and Dillon Laird. Deep q-learning with recurrent neural networks. *Stanford Cs229 Course Report*, 4:3, 2016.

[2] Soham Gadgil, Yunfeng Xin, and Chengzhe Xu. Solving the lunar lander problem under uncertainty using reinforcement learning. *arXiv preprint arXiv:2011.11850*, 2020.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[5] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[6] Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.