CS7642 - Learning to Land our Lunar Lander

7de54c13c25711801831823acca181b259279581 (Git Hash)

Preprint, compiled October 26, 2020

Sangeet Moy Das (sangeet.das@gatech.edu)

College of Computing, Georgia Tech

Abstract

The objective of this project report is to discuss the fundamentals of policy gradient algorithm and how to use them to develop a reinforcement learning algorithm to learn how to land a rocket on an landing pad by controlling the different rocket engine actions. For this we will be using OpenAI Gym's lunar lander v2.0, which is an environment that takes in one of four discrete actions at each time step, and returns a state in an 8-dimensional continuous state along with reward for it's action. An average score of 200 for the last 100 episodes is required for getting a sub-optimal solution.

1 Introduction

Advancements in AI are skyrocketing at the moment with researchers developing ground breakig ideas every year and we are still nascent in this amazing field. Reinforcement Learning has been responsible for some of the coolest examples of AI, such as OpenAI-5 which was developed to beat the top pro human players in the e-sport game named Dota 2. For an AI to be able to generilize and learn complex strategies in a game as sophisticated as Dota 2 is a huge achievement and will push research even closer to achieving artificial general intelligence (AGI). Lunar Lander is one such concept which belongs to the reinforcement learning field and is provided for training and experimentation through the OpenAI library. The aim of this specific game is to land the spaceship on it's landing pad by navigating it through the space environment. However, the issue with this kind of problems is the absence of labeled data, that we are so used to. The spaceship appears from above and has to land at zero speed on the landing pad (at coordinate (x,y) = (0, 0)). We have to achieve this outcome by firing the engines of the spaceship, assuming we have infinite fuel. If the spaceship is landed(either inside or outside the landing pad), the corresponding reward is 100 points. The environment is considered solved if you get a total reward of at least 200. Therefore, the agent acts upon observations formed from the environment, and depending on the result of its action it receives the corresponding reward. There are four possible actions: do nothing, fire left orientation engine, fire right orientation engine and fire main engine. Each time the agent chooses an action it gains a negative or positive reward measured in points provided by the environment. So, it implies that the reward increases, as the agent chooses right actions, which efficiently lead it in the landing pad. Finally, the episode ends when the agent lands, crashes or flies out of the space environment. Each one of the states generated by the environment are represented by a 1×8 vector.

$$\left(x, y, \hat{x}, \hat{y}, \theta, \hat{\theta}, leg_L, leg_R\right)$$

The state variables x and y are the horizontal and vertical position, $\hat{x}$ and $\hat{y}$ are the horizontal and vertical speed, and θ and $\hat{\theta}$ are the angle and angular speed of the lander. Finally, leg_L and leg_R are binary values to indicate whether the left leg or right leg of the lunar lander is touching the ground.

2 Cost Function

Function used to evaluate a solution(i.e. a set of weights) is referred to as the objective function. We may seek to maximize or minimize the objective function, meaning that we are searching for a solution that has the highest or lowest score respectively. Typically, with neural networks, we seek to minimize the error. As such, the objective function is often referred to as a cost function or a loss function and the value calculated by the loss function is referred to as simply "loss."

2.1 Sigmoid function

In mathematical definition way of saying the sigmoid function take any range real number and returns the output value which falls in the range of 0 to 1. Based on the convention we can expect the output value in the range of -1 to 1. The sigmoid function produces the curve which will be in the Shape "S." These curves used in the statistics too. With the cumulative distribution function (The output will range from 0 to 1)

$$\Pr(Y_i = 0) = \frac{e^{-\beta \cdot \mathbf{X}_i}}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$
$$\Pr(Y_i = 1) = 1 - \Pr(Y_i = 0) = \frac{1}{1 + e^{-\beta \cdot \mathbf{X}_i}}$$

2.2 Softmax function

Softmax function calculates the probability distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs. The main advantage of using Softmax is the output probabilities range. The range will 0 to 1, and the sum of all the probabilities will be equal to one. If the softmax function used for multi-classification model it returns the probabilities of each class and the target class will have the high probability. The formula computes the exponential (e-power) of the given input value and the sum of exponential values of all the values in the inputs. Then the ratio of the exponential of the input value and the sum of exponential values is the output of the softmax function.

$$P(C_i|\boldsymbol{x}) = \text{softmax}(z_i) = \frac{e^{z_i}}{e^{z_0} + e^{z_1}},\ i \in \{0, 1\}.$$

3 POLICY GRADIENT

The goal of reinforcement learning is to find an optimal behavior strategy for the agent to obtain optimal rewards. The policy gradient methods target at modeling and optimizing the policy directly. The policy is usually modeled with a parameterized function respect to θ, $\pi_\theta(a|s)$. The value of the reward (objective) function depends on this policy and then various algorithms can be applied to optimize θ for the best reward. The goal of our policy gradient method is to learn neural network weights that can output action probabilities that maximize reward for each state. We consider the result successful if the average reward of a trained neural network is >=200.

Algorithm 1 Vanilla Policy Gradient Algorithm

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Estimate policy gradient as

$$\hat{g}_k = \frac{1}{|\mathcal{D}_k|} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{A}_t.$$

  Compute policy update, either using standard gradient ascent,

$$\theta_{k+1} = \theta_k + \alpha_k \hat{g}_k,$$

  or via another gradient ascent algorithm like Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2,$$

  typically via some gradient descent algorithm.
end for

Algorithm 1: Vanilla Policy Gradient Algorithm

3.1 Policy Gradient Theorem

Computing the gradient $\nabla_\theta J(\theta)$ is tricky because it depends on both the action selection (directly determined by π_θ) and the stationary distribution of states following the target selection behavior (indirectly determined by π_θ). Given that the environment is generally unknown, it is difficult to estimate the effect on the state distribution by a policy update. The **Policy Gradient Theorem** provides a reformation of the derivative of the objective function to not involve the derivative of the state distribution $d^\pi(.)$ and simplify the gradient computation $\nabla_\theta J(\theta)$ a lot.

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s, a) \pi_\theta(a|s) \\ &\propto \sum_{s \in \mathcal{S}} d^\pi(s) \sum_{a \in \mathcal{A}} Q^\pi(s, a) \nabla_\theta \pi_\theta(a|s)\end{aligned}$$

3.2 Approach

In the context of this report for CS7642, a Policy Gradient algorithm has been created with the following feature:

1. A Neural Network based approach
   (a) 8 Input nodes, one for each component of the state space
   (b) 4 Output nodes representing each possible action
   (c) 1 hidden layer with variable number of nodes
2. A stochastic action selector(selects actions in line with the action probabilities prescribed by the neural network's output)
3. A model saving features that saves the best model if the algorithm is unable to reach and average reward of 200 within the set number of episodes. We can then use this to compare different hyperparameters when the most sets of hyperparameters don't meet our prescribed goal.

Our devised Policy Gradient algorithm does the following:

1. Randomly initializes the weights for the neural network
2. For each episode, take the following steps repeatedly until the environment returns DONE:
   (a) Use feedforward to produce a vector of action probabilities
      i. Take the current state as input and use the current neural network weights to calculate the output activation function
      ii. Use softmax function to normalize the outputs into probabilities that sums upto 1
   (b) Use a stochastic action selection method to select an action given the probability vector
   (c) Step into the environment using the selected action
3. Once the DONE flag has been returned:
   (a) Calculate the discounted reward for each step of the episode
      i. Working backwards through each step, add the discounted(by a discount factor – a hyperparameter) rewards of later steps to each step.
      ii. Normalize the discounted rewards by subtracting the mean and dividing by the standard deviation
   (b) Multiply the normalized discounted reward at each step in the episode by the action probabilities (weighted toward the action taken – [Y – action probabilities])
   (c) Use the neural network backpropagation algorithm to calculate the change in each weight vector due to the error (this step is what pushes the method toward reward maximization)
   (d) Store the weight updates calculated above for batch updating (batch size is a hyperparameter of the algorithm). Update the weights every 'batch size' episodes.
   (e) Calculate the total reward of the episode (the sum of the rewards of each individual action) and add the reward to a running array.

4. After 100 episodes, start calculating the average reward for the most recent 100 episodes. When this average reward exceeds 200 the algorithm has been successful.

4 EXPERIMENT

Using the Policy Gradient method described above a set of hyperparameters were created to test:

1. **Number of Neurons in the Hidden Layer:** Changing the neurons in the hidden layer increases the potential complexity that the neural network is capable of modeling. For the scope of this project I tried with 10, 16 and 128 number of hidden layers of neuron.
2. **Batch Size(the number of episodes to run before updating the model's weights):** By averaging the weight changes before applying them, the model stands to reduce some of the natural volatility that may arise. I tried frequent updates (batch size 1 – update the weights every episode), moderately frequent updates (batch size 10 – update the weights every 10 episodes) and infrequent updates (batch size 25 – update the weights every 25 episodes).
3. **Learning rate (the proportion of the gradient to use to update the model's weight):** Rather than updating the model's weight by the full amount suggested by the backpropagation algorithm, we multiply the update by a learning rate (less than 1) to scale back the power of these updates and decrease volatility. I tried learning rates that were very high (0.7 – 70% of the update from the backprop algorithm is added to the weights) and very low (0.0001).
4. **Gamma (the discount factor for the reward):** Discounting the rewards from late steps and applying them to earlier steps is a critical component of the policy gradient algorithm. The discount factor determines how much of the later reward is applied to each earlier step. A high discount factor (like 0.99) means that almost all of the reward from the last step is applied to the step before it (and so on). A low discount factor (like 0.1) means that only a small amount of the reward from the last step is applied to the step before it.

The hyperparameters above have 120 unique combinations. I ran all of them through the Policy Gradient algorithm, with a maximum number of episodes of 3,000.

4.1 Save the best model from training

- For each combination, during each episode it calculates the running average reward for the prior 100 episodes. When that reward is maximized the model should be saved and the model should be used to calculate the average reward over a new 100-trial run. However, with my batch sizes (1, 10, 25) there may be anywhere from 4-100 different models used across any given 100-episode set. I selected and saved the model used in the middle of the 100 episodes.
- After running all 120 combinations I found only one (H = 10, batch size = 10, learning rate = 0.01, gamma = 0.99) that had an average reward per trial (with the best trained weights within the training timeframe) that exceeded 200.
- I then created a new set of hyperparameters to test. I tested 9 new combinations with H set to 10 and batch size set to 10. I tested 4 learning rates (0.1, 0.01, 0.001, 0.0001) and 4 discount factors (0.99, 0.95, 0.90, 0.1). These hyperparameters were tested with 3,000 maximum episodes.

5 RESULTS

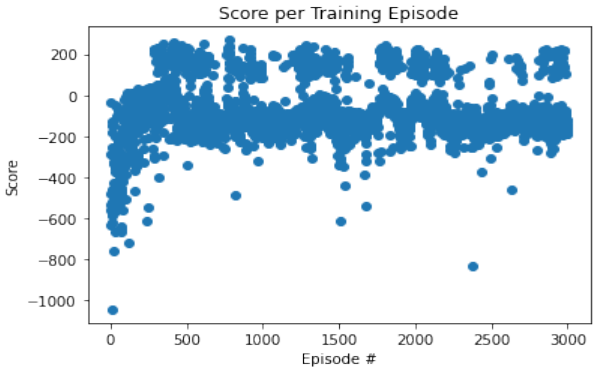

Figure 1: This figure shows the score per training episode for the best set of hyperparameters (hidden layer size = 16, batch size = 10, learning rate = 0.01, discount factor = 0.95). In early episodes the score hovers between -1000 and 0 with a few outliers. As the training continues the average score improves. By episode 500 the scores consistently bounce around between -200 and 200. The agent achieves a 100-episode average that exceeds 200 (average reward = 203.7) on episode 1541. The model generated on episode 1541 is used to create Figure 2 below.

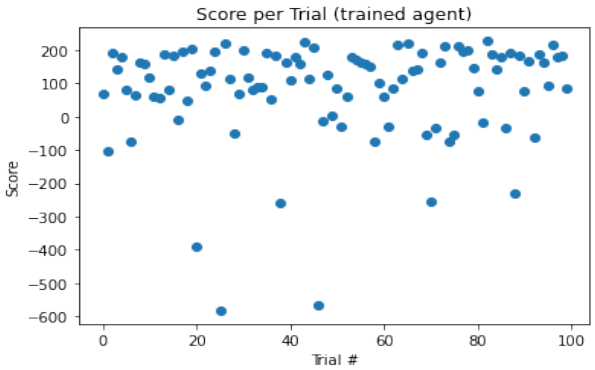

Figure 2: This figure shows the score per trial after training has been completed. Most of the scores fall between 100 and 200 while a few fall as low as -600. The average reward for these 100 trials 181.32

5.1 Hyperparameters

Hyperparameters discount rate γ and learning rate have been studied to assess their influence on the convergence speed and learning outcomes. The experiments are conducted with the maximum episode of 3000 with a maximum step of each episode as 500. Discount Factor γ in general determines how future

reward impacts the decision of action at the current state. A larger discount factor treats rewards in distance future closer to the immediate reward. A smaller discount factor favors the immediate reward and tends to maximize short-term gain. Figure 3, 4, 5 illustrates such effects.

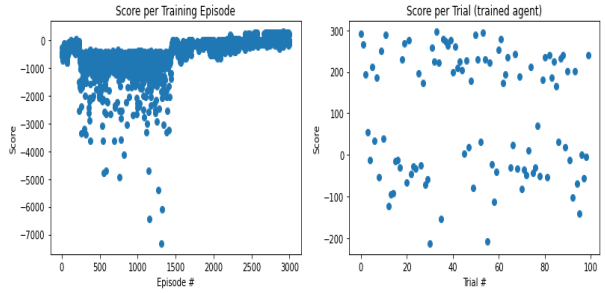

Figure 3: Hyperparameter tuning with 16 hidden layers with a batch size of 10 and learning rate of 0.01 with a discount factor of 0.99

Figure 4: Hyperparameter tuning with 16 hidden layers with a batch size of 1 and learning rate of 0.001 with a discount factor of 0.95

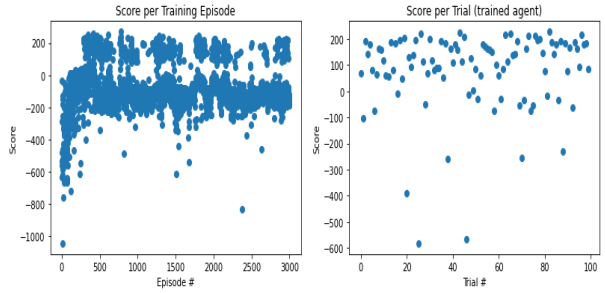

Figure 5: Hyperparameter tuning with 16 hidden layers with a batch size of 10 and learning rate of 0.01 with a discount factor of 0.99

6 ANALYSIS

Nothing about solving this problem was easy or worked right out of the box. It was challenging to correctly implement the Policy Gradient algorithm. Further, it was challenging to effectively tune the hyperparameters.

6.1 What worked best?

- **Debugging:** One good thing about not using any Deep-RL algorithm is that it makes debugging a lot easier, since the the code mostly used Numpy, making it easy to debug any component of the algorithm, which makes it easier to understand the entire flow of the implementation.
- **Code Reproduction:** Give the code is functionalized it was easier to scale it to run on any set of hyperparameters.
- **Visualizing Output:** I used matplotlib to plot the graphs once a set of hyperparameter has ran.

6.2 What caused trouble?

- **Hyperparameter tuning:** The biggest challenge in this project was hyperparameter tuning. Many initial experiments failed to solve the Lunar lander. Computation and learning were sufficient slow to converge that scores did not imporve for over 20 hours. Reducing learning rate to 0.001 improved performance significantly, allowing convergence to 7.5hrs. However, it was not until suitable batch size, annealing size and update steps used that Lunar Lander was solved within an hour, demonstrating that it is critical to tune hyperparameter.
- **Development effort:** It took me over 40 hours of development time to correctly implement the algorithm. I repeatedly incorrectly implemented key components and had to debug each line until I understood what was supposed to be happening and what was actually happening. Once implemented, identifying a set of hyperparameters to test was another struggle. I manually played with each of the four parameters until I got a sense of what the bounds should be, then implemented the experiment above (where I tested 120 different hyperparameter combinations) to find a suitable set.
- **Number of Episodes:** Most of my peers were using DQN and were able to solve this problem in well under 1,500 episodes. With Policy Gradient I was able to come close in under 2,000 episodes and successfully solved the problem with different hyperparameters in 3,000-4,000 episodes.
- **Number of Steps per Episode:** I frequently ran into the same problem that peers reported: the lander hovers and never lands eventually timing out after 1,000 steps.

6.3 What I would have tried if I had more time?

I would try to implement a few other algorithms:

- **REINFORCE** (Monte-Carlo Policy Gradient), which is a member of a class of algorithms called Policy Gradient methods.
- **Actor-Critic methods** are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function.
- **Deep Q-Network algorithm** which is a reinforcement learning algorithm that combines Q-Learning with deep neural networks to let RL work for complex, high-dimensional environments, like video games, or robotics.

- **Double Q Learning** corrects the stock DQN algorithm's tendency to sometimes overestimate the values tied to specific actions.

These algorithms are widely written about and are claimed to have better performance than plain Policy Gradient (these are all policy gradient methods). Unfortunately, I was not able to handle the added complexity for this project.

7 CONCLUSION

The author of this report has walked through the theoretical foundations of Policy Gradient algorithms which has been further used to land our Lunar Lander in a landing pad using OpenAI Gym. The author successfully trained the agent and met the solving condition. In this report, considerations related to number of hidden layers in our network, batch size, learning rate and our gamma discount factor. Last but not least, the trained agent has been evaluated along with the analysis on hyperparameter selections and their implications on learning. If given with more time and bandwidth, the author plans to explore and implement other advanced reinforcement learning algorithms like DQN and other variants of Policy gradient algorithms in order to solve the lunar lander problem.

8 REFERENCES

[1] Sutton R., McAllester D. - Policy Gradient Methods for Reinforcement Learning with Function Approximation

[2] Sutton R. and Barto A. - Reinforcement Learning: An Introduction, MIT Press, 2020

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., & Bellemare, M. et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533. doi: 10.1038/nature14236

[4] Xinli Yu (2019) - Deep Q-Learning on Lunar Lander Game

[5] Solving Lunar Lander with Double Dueling Deep Q-Network and PyTorch url: https://drawar.github.io/blog/2019/05/12/lunar-lander-dqn.html

[6] Rupesh Kumar Srivastava, Pranav Shyam, Filipe Mutz, Wojciech Jaśkowski, Jürgen Schmidhuber (2019) Training Agents using Upside-Down Reinforcement Learning