Project 2: Landing the Lunar Lander with Reinforcement Learning

Myles Lefkovitz
gth836x
GitHub Hash: c0226f23c7d6c02968c8a9f66497dd6098574389

1. INTRODUCTION

In this project, I develop a reinforcement learning algorithm to learn to land a rocket on a landing pad by controlling rocket engine actions. OpenAI Gym's Lunar Lander is an environment that takes in one of 4 discrete actions at each time step returns a state in an 8-dimensional continuous state space along with a reward.

2. POLICY GRADIENT

Policy Gradient methods are methods to learn a 'parameterized policy' that can select actions without a separate value function (Sutton 2018). "PG is [an] end-to-end [algorithm]: there's an explicit policy and a principled approach that directly optimizes the expected reward." (Karpaty 2016)

The goal of our policy gradient method is to learn neural network weights that 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.

I've constructed a Policy Gradient algorithm with the following features:

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

The Policy Gradient method does the following:

1. Randomly initialize the neural network (randomly initialize the weights)
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 NN weights to calculate the output activation function
        ii. Use softmax (a function described in formula 1 below) to normalize the outputs into probabilities that sum to 1.

$$softmax(a) = \frac{e^{a_j}}{\sum_{k=1}^{K} e^{a_k}}$$

Formula 1. Softmax

    b. Use a stochastic action selection method to select an action given the vector of probabilities (say, if action 1 has a .621 probability in the action probabilities vector, then this action should be selected with a.621 probability).
    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.

3. EXPERIMENTS

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

- **Number of neurons in the hidden layer: [10, 64]**
  Changing the neurons in the hidden layer increases the potential complexity that the neural network is capable of modeling. I tried both a relatively small number of neurons (10) and a relatively large number (64).

- **Batch size** (the number of episodes to run before updating the model's weights): **[1, 10, 25]**
  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).

- **Learning rate** (the proportion of the gradient to use to update the model's weight): **[0.7, 0.1, 0.01, 0.001, 0.0001]**
  Rather than updating the model's weight by the full amount suggest 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 (1x10^-4).

- **Gamma** (the discount factor for reward): **[0.99, 0.9, 0.7, 0.1]**
  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 2,000.

Saving the best model:

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 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 3 learning rates (0.03, 0.01, 0.006) and 3 discount factors (0.995, 0.99, 0.95). These hyperparameters were tested with 4,000 maximum episodes.

4. RESULTS

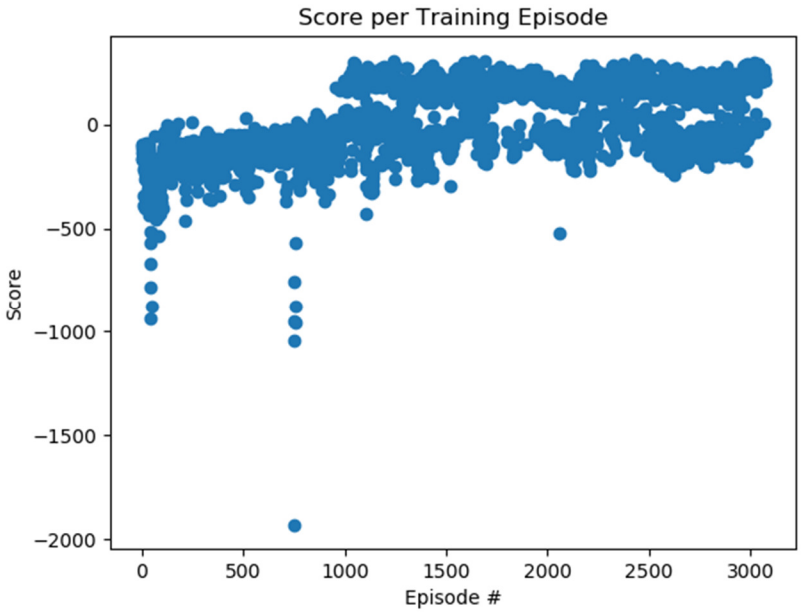

Figure 1. Score per Training Episode

Figure 1 above shows the score per training episode for the best set of hyperparameters (hidden layer size = 10, batch size = 10, learning rate = 0.03, discount factor = 0.95). In early episodes the score hovers between -1,000 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 0. The agent achieves a 100-episode average that exceeds 200 (average reward = 202.2) on episode 2971. The model generated on episode 2920 is used to create Figure 2 below.

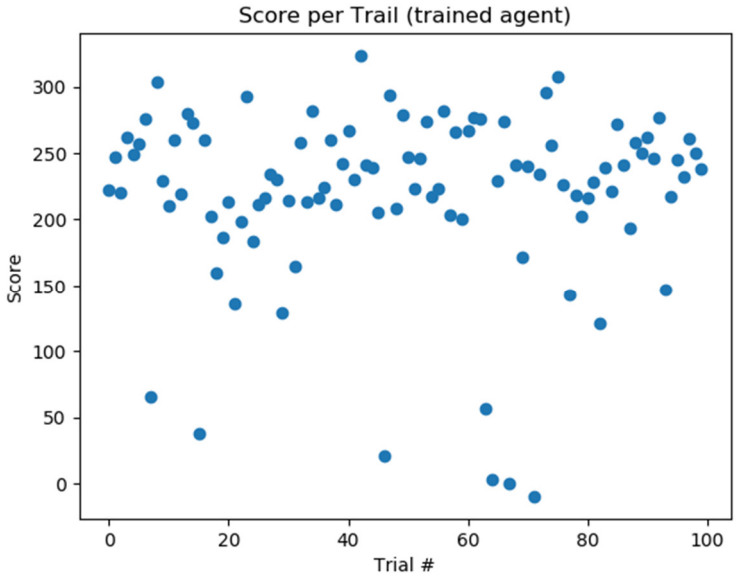

Figure 2. Trained Agent Score per Trial

Figure 2 above shows the score per trial after training has been completed. Most of the scores fall between 200 and 300 while a few fall as low as 0. The average reward for these 100 trials 219.77.

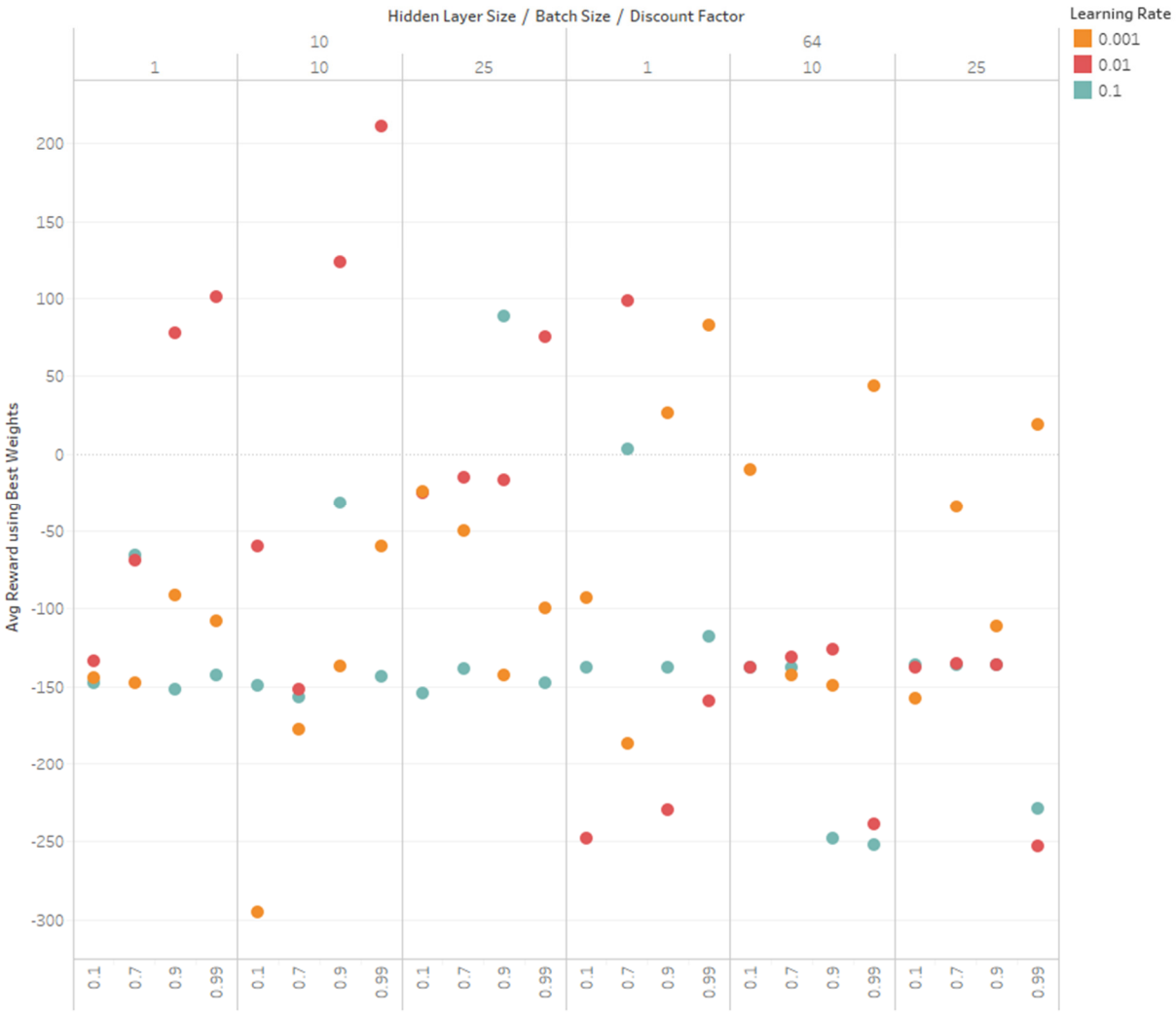

Figure 3. Hyperparameter Comparison Results

Figure 3 above shows the average score per trial after training has been completed for 72 hyperparameter combinations (all combinations with learning rates 0.7 and 0.0001 have been excluded). The horizontal axis is a combination of 3 dimensions: hidden layer size on top, batch size underneath, and discount factor across the bottom. The vertical axis is the average reward per trial after training has been completed. Color indicates learning rate.

Most of the scores fall below 0 while a few are around 100. Only one combination (hidden layer size = 10, batch size = 10, discount factor = 0.99, learning rate = 0.01) is above 200.

5. ANALYSIS

What worked best

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.

Debugging:

Since all of my code was hand written (without using any package implementations other than Numpy) I was able to painstakingly debug each component of my algorithm. This worked well because I could see exactly what was happening at each line of code.

Code Simplicity:

Once I was able to correct all of the issues with my implementation, the hyperparameter testing was seamless. I simply turned the code into a function and called that function with each hyperparameter as an input.

Graph Creation:

I used matplotlib and pyplot to create each graph automatically.

What didn't work

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.

What I would try if I had more time

I would try to implement a few other algorithms:

- REINFORCE
- Actor-Critic methods
- Proximal Policy Optimization (PPO).

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.

REFERENCES

1. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
2. Karpathy, A. (2016). Deep reinforcement learning: Pong from pixels. *url: http://karpathy. github. io/2016/05/31/rl*.