Application of Q-Learning, Deep Q-Network, REINFORCE to CartPole

Quentin Truong
UCLA
Los Angeles, CA, 90095
quentintruong@g.ucla.edu

Abstract

I implemented Q-Learning, Deep Q-Network, and Policy Gradient through the REINFORCE algorithm using Keras to solve the CartPole game from the OpenAI Gym environment. Each algorithm was able to find a solution with sufficient iterations; however, DQN and REINFORCE were unable to consistently stabilize on a solution. Both were prone to cycling on and off the solution, which may be preventable with a different choice of architecture and hyperparameters.

1. Introduction

CartPole is a canonical game environment from the OpenAI Gym where the goal is to balance a pendulum upright on a cart. The game has two inputs (corresponding to pushing the cart left or right) and four outputs (corresponding to horizontal position, horizontal velocity, angular position, and angular velocity). The game is solved by balancing the pendulum upright for an average of 195 frames or more within a constrained horizontal region for 100 consecutive games.

Reinforcement learning is a natural approach for building a CartPole solver because the CartPole environment assigns rewards based on the action taken in a particular state. Supervised learning was not considered because there is no labeled data to learn from. I chose to implement two value functions and one policy function, namely, Q-Learning, Deep Q-Learning, and Policy Gradient with REINFORCE.

1.1. Q-Learning

First, I start with Q-Learning. Here, we model our game as a Markov Decision Process [1]. To form the finite set of states for our MDP, I discretize Cartpole's continuous output into bins, treating the bounds and number of buckets as a hyperparameter. CartPole's actions are discrete and serve as our finite set of actions. The state transition probabilities and reward function are supplied by the CartPole environment. The discount rate is treated as a hyperparameter.

The agent navigates the game using a policy implicitly defined by the Q-table. We determine the agent's Q-table by iterating over the Bellman equation [1].

1.2. Deep Q-Network

Second, I try a Deep Q-Network to solve CartPole. One advantage of a DQN over Q-Learning is that the state space may be very large, allowing for a much wider range of games to be played. For example, if we were to try Q-Learning on an Atari game using a preprocessed grayscale stack of four images as our state, there would be 128 ^ (4 * 84 * 84 * 1) states (128 values possible for each pixel in the 4 84-pixel square cropped grayscale images). Each state then has 18 possible actions (due to the Atari controller). This is well over the total number of atoms in the universe and would not fit in memory, and, even if it could, would be an inefficient use of the data due to correlation between states [2]. While the CartPole state space is not large enough to require a DQN, I implemented one as an exercise, because ultimately, CartPole is largely for practice and validation of one's implementation before moving onto more interesting games. Since DQN's tend to be unstable, I also implemented experience replay and target network in an effort to stabilize the learning [3].

1.3. Policy Gradient with REINFORCE

Third, I implemented Policy Gradient using the REINFORCE algorithm, which differs from Q-Learning and DQN in that it is a policy-function, meaning that it directly optimizes a policy, rather than implicitly from the action-values of a Q-table. One notable advantage is that PG can learn stochastic policies, which is useful in dealing with aliased states [4]. Also, unlike a DQN, PG works in continuous action spaces. While CartPole does not require either of these, I implemented it as an exercise and experiment, similar to as with the DQN.

2. Results and Discussion

I tested the Q-Learning, Deep Q-Network, and REINFORCE algorithms on the CartPole-v0 environment

and measured the average time per iteration, average number of episodes per iteration, and plotted the distribution of when the solution was found.

Refer to table on the next page for experimental results. For the full results, see the excel sheet. Distributions plots were excluded from the report due to space constraints but are included in the zip with the code.

2.1. Q-Learning

First, for Q-Learning, I tested the impact of varying the number of buckets used for discretization, the epsilon update rule for the exploitation-exploration tradeoff, the learning rate, and the discount rate.

Foremost, the number of buckets had the most significant impact of any hyperparameter with respect to learning and convergence. In particular, setting the number of buckets for the horizontal position and horizontal velocity to one each seemed to both stabilize and accelerate convergence. This can be understood by the observation that the cart doesn't move very far within the 195 frames if the pendulum is upright (and so isn't relevant for winning the game); thus, we can reduce the dimensionality of the state space in half by effectively ignoring the first two outputs.

The next most important parameter is the epsilon update rule. Inspired by an update rule that many others were using, I initially started exploring using a bounded logarithmic decay and saw minor improvements by adjusting the parameters [5], [6]. Ultimately, I grew suspicious that the logarithmic decay was really necessary, leading me to approximate it using a linear equation, which performed just as well as the others.

Similarly for the learning rate update rule, I initially used a bounded logarithmic decay but found that it was well substituted by a linear expression.

The discount rate seemed to have little effect unless below 0.90, at which point the DQN is unstable. With a discount of 0.90, the reward from 28 frames ahead have an impact of roughly 5% of what they'd have if the discount were 1.0; perhaps the DQN requires a discount greater than 0.90 because the effect of pushing the cart persists for more than 28 frames.

Ultimately, Q-Learning was able to consistently find a solution within 200 episodes using a linear epsilon update rule, linear learning rate, halving the model input dimensionality, and using a discount rate above 0.90.

2.2. Deep Q-Network

Second, I tested the Deep Q-Network in a similar manner to with Q-Learning, with the exception of performing fewer tests and iterations because training the DQN is much slower. I tested the impact of changing the optimizer and its learning rate and the model architecture.

I found that the optimizer had a significant impact on stability; in particular, low learning rates around 0.001 are essential. Learning rate decay was not found to be helpful; however, this may simply be due to a poor choice in other hyperparameters. I was unable to test further due to the lack of computational power.

More hidden layers and units seemed to help the network stabilize on a solution. This may be because the solution can be redundantly encoded with more neurons.

The target network and experience replay were found to be essential to learning, as the model is very unstable and is unable to achieve the optimal policy without both.

Ultimately, I was not able to find a set of parameters would consistently allow the DQN to stabilize on a solution in under 1000 iterations. I believe that it would be possible given more computational power and time.

2.3. Policy Gradient with REINFORCE

Third, I tested the Policy Gradient with REINFORCE similarly to as in previous sections. I tested the impact of varying the discount, learning rate, and model architecture.

A discount rate around 0.90 empirically seemed to work well, although it did not completely stabilize optimization. The discount rate may prevent stabilization because it affects the emphasis placed on later rewards and having an incorrect discount rate may cause the algorithm to undervalue rewards from more relevant frames.

A learning rate of 0.007 using the Adam optimizer seems to be most stable, perhaps because of the topology of the solution space. Both lower and higher learning rates were tested but resulted in unstable convergence and/or no learning at all.

Similar to with the DQN, having more neurons seemed to help with stability, possibly by redundantly encoding the solution.

Ultimately, PG was not able to consistently find converge on a solution in under 500 iterations. This may have been solved with a better choice of hyperparameters.

2.4. Conclusion

Ultimately, Q-Learning was able to learn a solution to CartPole more consistently, with fewer iterations, and less computational time than either the DQN or PG. Q-Learning is able to learn with fewer iterations largely due to the dimensionality reduction enabled by ignoring the first two outputs when discretizing and learns with less computational time because the Q-Table requires fewer iterations than the neural net to approximate the optimal policy.

References

[1] Silver, David, 2015. [online] http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf [Accessed 16 Mar. 2019].

[2] Mnih, Volodymyr, et al. Playing Atari Games with Deep Reinforcement Learning, 2013. [online] https://arxiv.org/pdf/1312.5602v1.pdf [Accessed 16 Mar. 2019].

[3] Seno, Takuma. Welcome to Deep Reinforcement Learning Part 1: DQN, 2017. [online] https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b [Accessed 16 Mar. 2019].

[4] Silver, David, 2015. [online] http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf [Accessed 16 Mar. 2019].

[5] n1try, 2018. [online] https://gist.github.com/n1try/2a6722407117e4d668921fce53845432 [Accessed 16 Mar. 2019].

[6] icoxfog417, 2016. [online]. https://github.com/icoxfog417/cartpole-q-learning/blob/master/cartpole.py [Accessed 16 Mar. 2019].

Table - QL, DQN, and PG Performance (For complete data, see excel in zip with the code)

QL – Epsilon Update Rule	Avg seconds per iteration	Avg episodes per iteration
max(0.01, min(1, 1.0 - np.log10((episode + 1) / 10)))	0.979908538	194.4166667
max(0.01, min(1, 1.0 - np.log10((episode + 1) / 20)))	1.037151484	229.9666667
max(0.01, 1 - 0.007 * episode)	0.966020505	217.3166667
QL - Buckets	Avg seconds per iteration	Avg episodes per iteration
(1, 1, 5, 2)	4.604125428	1000
(1, 1, 6, 3)	1.343216999	253.8
(2, 2, 6, 3)	1.900543404	332.5666667
(3, 3, 6, 3)	3.865438668	591.4
QL - Learning Rate	Avg seconds per iteration	Avg episodes per iteration
max(0.01, 1 - 0.013 * episode)	1.063338931	191.0666667
max(0.01, 1 - 0.010 * episode)	1.027786628	191.3333333
max(0.01, 1 - 0.007 * episode)	1.143879167	210.2
max(0.01, 1 - 0.005 * episode)	1.090440893	248.4
QL - Discount	Avg seconds per iteration	Avg episodes per iteration
1	1.189791234	189.3
99	1.075836718	189.8833333
95	1.03534133	190.6166667
90	0.933877317	195.0666667
DQN - Optimizer	Avg seconds per iteration	Avg episodes per iteration
adamlr0001	134.7002476	969.7333333
adamlr001	512.2180434	447.5333333
adamlr001decay999999	347.6028351	1000
nadam	2048.399644	621.4666667
PG - Discount	Avg seconds per iteration	Avg episodes per iteration
99	33.00578433	323.55
95	62.76174185	319.775
90	119.5940577	337.15
85	161.9942066	343.025
PG - Learning Rate	Avg seconds per iteration	Avg episodes per iteration
11	256.6009196	335.9
7	358.8206514	311.975
3	159.9628298	373
PG - Model	Avg seconds per iteration	Avg episodes per iteration
32	31.1447825	401.4
128	34.6781903	325.3
32,32	43.1360358	334.2
128,128	60.36620789	433.7

Methods - QL, DQN, and PG architecture and training details
QL – Epsilon Update Rule – 60 Iterations
Number of buckets for each state value: (1, 1, 6, 3)
Min/max of each state value range: ([1,-1], [1, -1], [0.5, -0.5], [1, -1])
Learning rate update rule: max(0.01, min(1.0, 1.0 - np.log10((episode + 1) / 25)))
Discount rate: 1.0

QL – Buckets – 30 Iterations
Min/max of each state value range: ([1,-1], [1, -1], [0.5, -0.5], [1, -1])
Epsilon update rule: max(0.1, min(1, 1.0 - np.log10((episode + 1) / 25)))
Learning rate update rule: max(0.01, min(1.0, 1.0 - np.log10((episode + 1) / 25)))
Discount rate: 1.0

QL – Learning Rate – 30 Iterations
Number of buckets for each state value: (1, 1, 6, 3)
Min/max of each state value range: ([1,-1], [1, -1], [0.5, -0.5], [1, -1])
Epsilon update rule: 1 - 0.007 * episode,
Discount rate: 1.0

QL – Discount – 60 Iterations
Number of buckets for each state value: (1, 1, 6, 3)
Min/max of each state value range: ([1,-1], [1, -1], [0.5, -0.5], [1, -1])
Epsilon update rule: max(0.1, min(1, 1.0 - np.log10((episode + 1) / 25)))
Learning rate update rule: max(0.01, min(1.0, 1.0 - np.log10((episode + 1) / 25)))

DQN – Optimizer – 15 Iterations
Model: 4 Inputs – 16 Affine – Relu – 2 Affine – MSE
Target network update frequency: 5 frames
Experience replay: size 50000
Discount: 0.85
Epsilon update rule: 1 / (episode / 10 + 1)
Batch size: 64

PG – Discount – 40 Iterations
Model: 4 Inputs – 128 Affine – Relu – 2 Affine – Softmax
Optimizer: Adam w/ learning rate 0.007

PG – Learning Rate – 40 Iterations
Model: 4 Inputs – 128 Affine – Relu – 2 Affine – Softmax
Discount: 0.95
Optimizer: Adam

PG – Model – 40 Iterations
Discount: 0.95
Optimizer: Adam w/ learning rate 0.007