MONTE-CARLO POLICY-GRADIENT REINFORCEMENT LEARNING

Irie Railton
S3292037
i.r.railton@umail.leidenuniv.nl

Pablo Ubilla Pavez
S3430839
p.ubilla.pavez@umail.leidenuniv.nl

1 INTRODUCTION

In the following report we will study the implementation of the Monte-Carlo Policy-Gradient Control method. The main motivation for this reinforcement learning method is that it is well suited to continuous or large state spaces as it is not tabular, and instead approximates the policy based on a vector of parameters where the number of parameters is much smaller (or at least countable if we are in a continuous case) than the number of states. Another advantage over other reinforcement learning strategies is that the approximate policy can approach an exploitative policy, instead of always having a chance of selecting a random action as in an ϵ-greedy action selection approach that has a fixed exploration value.

We will be working with the *CartPole* environment, a widely used environment in this field of reinforcement learning. The basic idea is that there is a pendulum attached to a cart that can be moved left or right for each step (hence, the set of actions $\mathcal{A}$ has a cardinality of 2). The states $s \in \mathcal{S}$ will be vectors in $\mathbb{R}^4$, representing: (Cart Position, Cart Velocity, Pole Angle, Pole Angular Velocity). These states can be rendered to obtain a visualization such as Figure 1. In principle, all of these measures are continuous, hence we could consider that $\mathcal{S}$ is uncountable. The basic idea is to maintain the pole in balance, so for each step that the pole does not fall we receive a reward of 1. When the pole falls or when we reach an accumulated reward of 50 the episode ends.

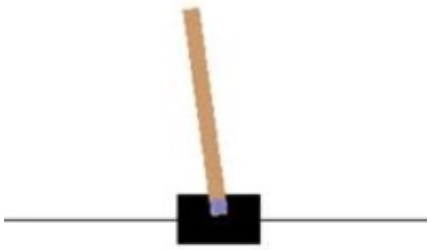

Figure 1: Cartpole Environment visualization.

We can observe that this problem can not be approached with a tabular method, since we have a continuous action space. For this report we will study how we can solve this problem using the Monte-Carlo Policy Gradient Control and how the effectiveness of this method varies through its parameters.

2 METHOD

The basic idea of the method is to find a policy $\pi(s, \theta)$, where θ is a vector of parameters used to generalize this policy for each $s \in \mathcal{S}$. This policy could be derived from various functions, but in this particular implementation we will work with a Sequential Neural Network.

The network we will implement have 4 dense layers, including the output layer. The first three have ReLU activation functions, hence $\sigma_R(x) = \max\{0, x\}$. The output layer has a softmax activation function, hence $\sigma_s(x)_i = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$, for each i in the K output neurons. The activation functions were chosen mainly because they are widely used in neural networks problems and are generally good options. However, it should be noted that this arbitrary and we could have experimented different activation functions.

If we consider x to be the input vector, and $z^(i)$ to be the output of the i-th layer, the overall network would be:

$$z^{(1)} = \sigma_R(W^{(1)}x + w_0^{(1)})$$
$$z^{(i)} = \sigma_R(W^{(i)}z^{(i-1)} + w_0^{(i)}) \quad i = 2, 3..., n-1$$
$$z^{out} = \sigma_S(W^{(n)}z^{(n-1)} + w_0^{(i)})$$

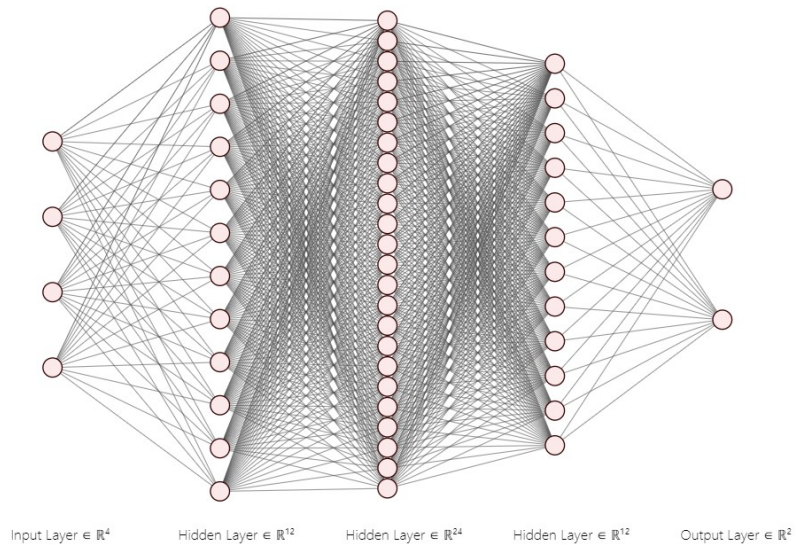

Figure 2: Sequential Neural Network architecture selected for policy approximation.

Here we see $W^{(i)}$ as the weight matrix for layer i, and n to be the number of layers. For this particular case we will work with $n = 4$. Regarding the weights we consider that: $W^{(1)} \in \mathbb{R}^{12\times4}$, $W^{(2)} \in \mathbb{R}^{24\times12}$, $W^{(3)} \in \mathbb{R}^{12\times24}$ and $W^{(4)} \in \mathbb{R}^{2\times12}$. W_0^i it's the bias for layer i and it should match the dimension of z^{i-1} or x for the first layer. Just like the activation functions, the hidden layers dimensions were chosen arbitrarily, taking into account the computing time available and the complexity of the problem. The input size will be determined by the dimension of x, in this case $x \in R^{4\times1}$ We may consider $\theta = W^{(1)}, W^{(2)}, W^{(3)}, W^{(4)}, W_0^{(1)}, W_0^{(2)}, W_0^{(3)}, W_0^{(4)}$, as we are going to optimize this method by varying all these different weights. If we bring back this network to the reinforcement learning notation, we can consider $x = s$ (the input is a state), and $z^{(out)} = \pi(s, \theta)$ (the output is a probability vector associated with the actions, as a function of the state and the parameters).

For optimizing this policy we should use a loss function. In general we can consider it to be $\nabla J(\theta)$ as the true value function for the policy. Since we can not derive this value, we will need an approximation, which Sutton and Barto do very elegantly in their book, ending up with:

$$\nabla J(\theta) \propto \mathbb{E}_\pi \left[G_t \frac{\nabla \pi \left(A_t \mid S_t, \boldsymbol{\theta} \right)}{\pi \left(A_t \mid S_t, \boldsymbol{\theta} \right)} \right]$$

Where G_t can be derived from an episode run as $G_t = \gamma G_{t+1} + R_{t+1}$, with $G_{T-1} = R_T$. We will consider this γ value as a discount factor.

Finally, the main idea of the algorithm will be to sample an episode using the policy $\pi(\theta)$, and then for each step t we can do an update to the parameters as following:

$$\theta = \theta + \alpha \nabla \gamma^t G_t \ln\left(\pi(a_t|s_t, \theta)\right)$$

We can repeat this process (sample more episodes) to keep upgrading the parameters with the hope of improving the performance in the environment.

We will experiment with our algorithm with varying values of α and γ on the *CartPole* environment for 200 episodes. We will repeat each experiment 5 times and average the results over the five experiments. The values we will experiment with are:

$\alpha = [10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}]$ and $\gamma = [0.9, 0.95, 0.99, 1]$

It should be noted that for the α values we are changing the order of magnitude, since this parameter is present once in each step. Regarding γ we do not vary the magnitude order, since it is iteratively multiplied in the method, meaning that small changes can lead to a significant discount in the further rewards from the step we are evaluating.

For each value of α we will try all values of γ so that there are 16 different configurations in total. We will then compare and discuss the results of each experiment.

3 RESULTS

In figure 3 we observe the results regarding average reward for the different combinations. This shows that the best result is obtained with $\gamma = 0.99$ and $\alpha = 0.001$. In general, a value of $\alpha = 0.001$ leads to good results regardless of the γ value. However, it should be noted that these results may have differed slightly if we had been able to run more repetitions. In regards of these results we will study the effect over α with a γ fixed to 0.99, and the effect over γ with a α fixed to 0.001.

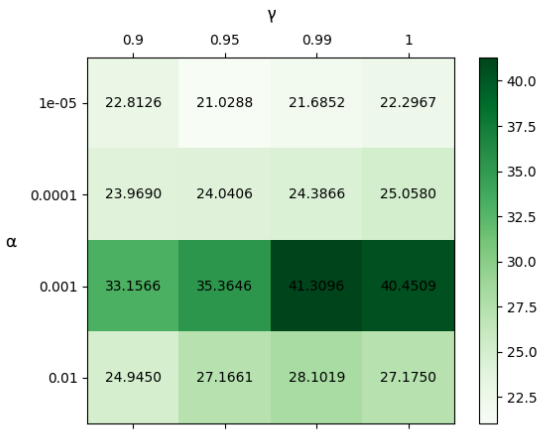

Figure 3: Average reward over 200 episodes and 5 repetitions for each value of α and γ plotted on a colormap with increasing color intensity denoting higher reward.

From figure 4 we can clearly see that a learning rate of $\alpha = 0.001$ performs better than the other values of α over the 200 episodes. The $\alpha = 0.01$, while eventually reaching almost the same value as $\alpha = 0.001$, performs far worse overall and is very unstable. This shows that a learning rate that is too high can cause the model to somewhat over fit by learning too much from every episode, rather than learning a small amount and correcting itself from worse episodes. However, a learning rate that is too small will result in not enough being learned from each episode, as seen in the $\alpha = 0.0001$ and $\alpha = 0.00001$ curves. The smallest learning rate appears to learn almost nothing, but running the algorithm for more episodes and over more episodes would be needed to decide if this were true.

Figure 5 shows the comparison of varying values of γ. From this graph it is clear that a value close to 1 is best, and straying too far from this can have an adverse effect on the results of the algorithm. This is shown by the learning curves for $\gamma = 0.9$ and $\gamma = 1$ reaching higher reward, having greater average reward, and being more stable than the learning curves for the lower values. While 1 does seem to be a good value for γ, discounting the reward by 0.01 as in the $\gamma = 0.9$ learning curve shows that some discount is good and should be explored when applying this algorithm to new problems.

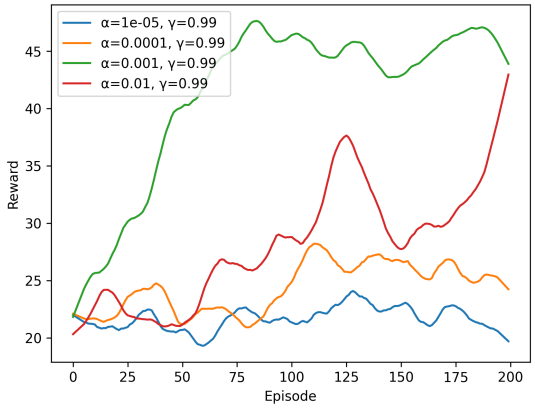

Figure 4: Average reward over 5 repetitions for 200 episodes with different values of α and a fixed $\gamma = 0.99$.

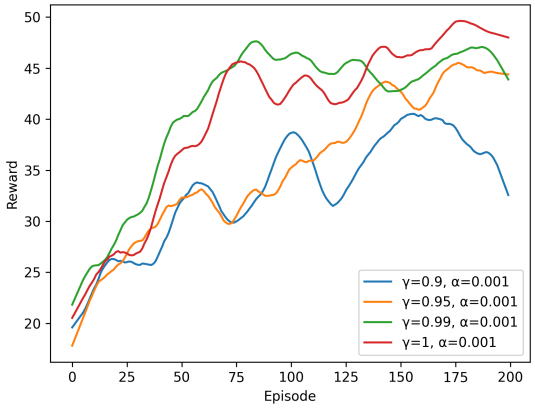

Figure 5: Average reward over 5 repetitions for 200 episodes with different values of γ and a fixed $\alpha = 0.001$.

4 Conclusion

In this experiment we observe the promising power of reinforcement learning regarding approximate solutions. We were able to obtain good average cumulative rewards after no more than 200 episodes in the environment (the average reward got close to 50, being this the maximum value possible). But we could also observe the importance of parameter selection, where some of the configurations lead to poor results in the limited settings studied.

It should also be noted that more computing time could lead to a better understanding regarding the learning rate and how it reaches an optimal solution. One of the biggest problems regarding the use of gradient descent is the convergence to a local minimum (where we would like a global minimum), it is not clear if a smaller learning rate could lead to better results in the long run.

We also need to mention that the network parameters and architecture were not optimized, and this could have also lead to some more interesting results and conclusions. The number of layers and neurons could improve or decrease the performance, as well as the activation functions used. There are numerous settings that can be tried which creates a lot of different solutions for the same problem.

Following work could regard implementing the actor-critic method, which includes an estimation of the state values with an additional approximate function. This may upgrade the learning and rewards obtained, but the inclusion of a second function (that could also be a neural network), should also increase computing time, being an interesting trade-off to study.