CartPole Report

Patrick Mederitsch

January 2025

1 Introduction

The Cart-Pole problem consists of a box on which a pole is connected through a joint. The pole is not actuated. The goal is to balance the pole by moving the cart left or right. Each episode in the environment starts with the cart in the middle and the pole in the vertical position, with some noise added. For every time step in which the pole has not tilted so much that the pole's angle leaves the specified range of $[-24°, 24°]$, the environment provides a reward of 1 for every time step. If the pole's angle leaves the specified range, the episode ends. After 500 timesteps an episode ends automatically. The Cart-Pole problem is considered to be solved if the agent can balance the pole for 500 timesteps. For an illustration of the environment, see Figure 1.

2 Methods

For solving the Cart-Pole problem, I used the vanilla policy gradient (VPG) and further added two modifications for comparison. First, I describe the VPG and then the modifications. For my implementation of the VPG, I relied on OpenAI's Spinning Up introduction to policy optimization. Why do I use VPG? The policy of **State-Value** methods selects an action a_t based on the values of all the next attainable states $v(S_{t+1})$ given the current state S_t, where the value of a state is the expected return from that state onward:

$$\begin{aligned} v(s) &:= \mathbb{E}[G_t | S_t = s] \\ &= \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\Big|\; S_t = s\Big], \quad \text{for all } s \in \mathcal{S}, \end{aligned}$$

with $\gamma \in (0, 1]$ being the so called discount factor. A greedy action choice, given the value function of the current policy, denoted by v_π, can be found by comparing which of the possible actions leads to the state with the highest estimated value:

$$\pi(s_t) = \underset{a \in \mathcal{A}(s_t)}{\operatorname{argmax}}\, \mathbb{E}\Big[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\Big].$$

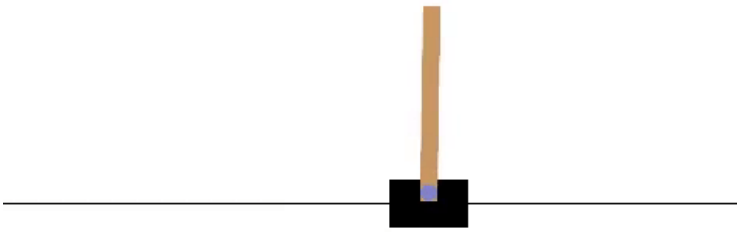

Figure 1: The black box represents the cart and the brown stick is the pole. The observation space is continuous, consisting of four features, cart position, cart velocity, pole angle and pole angular velocity. The action space consists of 2 discrete actions, pushing the cart left or right. More information about the Cart-Pole environment can be found in the Gymnasium Documentation.

Using **State-Value** methods would mean that we have to first learn a value function (expected return in each state) before being able to choose a reasonable action. Since we are in a continuous setting, with infinitely many possible states, this is quite impractical. Thus, motivating the use of VPG or **Policy-Gradient** methods in general.

2.1 Vanilla Policy Gradient

Instead of learning a value function to determine which action to take, we want to learn the policy directly. **Policy-Gradient** methods do exactly that. We have a parameterized policy with $\theta \in \mathbb{R}^d$ being the parameter vector. We can write the policy as follows:

$$\pi(a|s, \theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}.$$

For shorthand notation we write $\pi_\theta(a|s)$ instead. Now the question is how to find a suitable θ such that the policy chooses actions which maximize the expected return. We define a performance measure for our policy, which we want to maximize through gradient ascent, hence the name **Policy-Gradient**. We denote the performance measure by $J(\theta)$ with respect to the policy parameter. We choose the expected return as our performance measure since this is the quantity we want to maximize. We write:

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\big[R(\tau)\big],$$

where $R(\tau)$ gives the finite-horizon undiscounted return. We use the following update rule to optimize our policy via gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_t}.$$

where

$$\begin{aligned}
\nabla_\theta J(\pi_\theta)|_{\theta_t} &= \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\big[R(\tau)\big] \\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) \\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) \\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\big[\nabla_\theta \log P(\tau|\theta) R(\tau)\big] \\
&= \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\Big].
\end{aligned}$$

The expectation can be approximated through a sample mean, with $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$. Each τ_i is a trajectory sampled from the environment by applying the agent's current policy. The approximation of $\nabla_\theta J(\pi_\theta)$ can be written as:

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

If we represent π_τ as a differentiable function, then we can compute $\hat{g}$ and thus update our policy. This particular update rule is called the vanilla policy gradient (VPG). We can make further adjustments to this update rule to get update rules with (theoretically) more desirable properties, such as reduced variance.

2.2 Reward to Go (RTG)

Note that in the expression of the gradient of the performance measure in VPG, we weigh action choices at each time step with the return of the entire trajectory. Intuitively, this does not make much sense, since actions chosen at a later time should be weighted based on the consequences that happen afterwards and not be influenced by previous choices. Therefore we want to calculate the rewards as we go and come up with the following expression:

$$\nabla_\theta J_{rtg}(\pi_\theta)|_{\theta_t} = \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1})\Big].$$

This update rule captures this intuition. Thus, our new update rules can be written as:

$$\hat{g}_{rtg} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}).$$

2.3 Policy Gradient with Baseline

Before introducing baselines into our policy gradient update rule, we first establish the so called *Expected Grad-Log-Prob* (EGLP) lemma.

EGLP-Lemma: Consider a parameterized probability distribution P_θ over a random variable x, then:

$$\mathbb{E}_{x \sim P_\theta}[\nabla_\theta \log P_\theta(x)] = 0.$$

Proof:

$$\int_x P_\theta(x) = 1.$$

$$\nabla_\theta \int_x P_\theta(x) = \nabla_\theta 1 = 0.$$

$$\begin{aligned} 0 &= \nabla_\theta \int_x P_\theta(x) \\ &= \int_x \nabla_\theta P_\theta(x) \\ &= \int_x P_\theta(x) \nabla_\theta \log P_\theta(x) \\ &= \mathbb{E}_{x \sim P_\theta}[\nabla_\theta \log P_\theta(x)]. \end{aligned}$$

□

It follows from the EGLP-Lemma that for a function $b(s_t)$, which only depends on the state s_t we have that:

$$\mathbb{E}_{x \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) b(s_t)] = 0.$$

This allows us to write $\nabla_\theta J_{rtg}(\pi_\theta)$ in the following form:

$$\nabla_\theta J_{rtg}(\pi_\theta)|_{\theta_t} = \mathbb{E}_{\tau \sim \pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \Big(\sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) - b(s_t) \Big) \Big].$$

Any function b is called a baseline. Adding a baseline reduces the variance of our gradient computation. For an explanation why this is the case I refer to

Fundamentals of Policy Gradients. Intuitively we can set $b(s_t) = V^\pi(s_t)$, where $V^\pi(s_t)$ (representing the value function according to the current policy) gives value of the state s_t, or the expected return from that state towards the end of the episode. Why is this choice intuitive? If the value function is correct and the agent receives the rewards it expected, then there should be no update in the policy. As already discussed above we cannot simply obtain the exact value function for each state, since there are infinitely many states. We can approximated it, using a parameterized function approximator (in our case a neural network) $V_\phi(s_t)$, where:

$$\phi_{k+1} = \underset{\phi}{\operatorname{argmin}}\, \mathbb{E}_{\tau \sim \pi_{\theta_k}} \left[\left(V_{\phi_k}(s_t) - \sum_{t'=t}^{T} R(s_{t'}, a_{t'}, s_{t'+1}) \right)^2 \right].$$

The expectation can approximated with a sample mean.

For more detailed explaination and derivations of the used formulas, we again refer to OpenAI's Spinning Up, which also explains the additional properties of the reward-to-go and value-function-baseline method. The code implementation of the described algorithms can be found on my GitHub page.

3 Results

I trained three agents for 80 episodes each with the same batch size of 50.000. The batch size corresponds to the number of trajectories sampled from the environment, before making an update to the policy, according to the above derived update rules.

The parameterized policy is represented by a *multilayer perceptron* (MLP) neural network with four input features, a hidden layer consisting of 32 nodes, and an output layer with two nodes, representing the likelihood of both actions given the current state. The hidden nodes use a Tanh-activation function. For computing the gradients, I use the Adam optimizer with the default settings. The learning rate is set to 0.01 and was kept constant throughout the training. The agent using a learned value function as baseline, used the same MLP architecture for implementing the parametrized value function, as the policy network. The only exception is, that the value network only has one node in the output layer.

As can be seen from Figure 2, all three methods solved the problem by reaching an average trajectory length of 500. Figure 3 shows the "loss" curves, which are misleading since this is not an actual loss, but the mean logarithmic probabilities of an action taken, given the current state. The losses for each method are different in the sense that the log probabilities are weighted differently, see the different update rules above. This quantity is what we actually want to maximize, since we want actions, which yield rewards, to have increased probability of occuring.

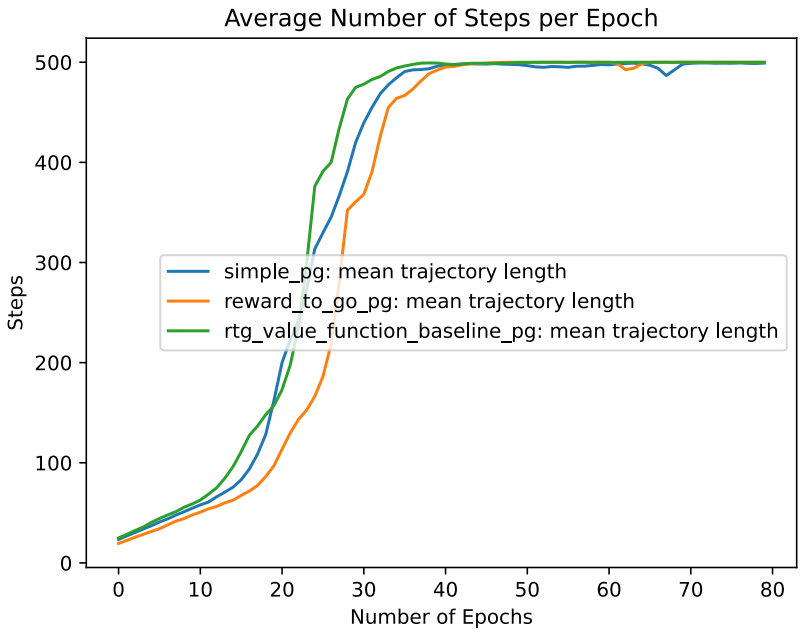

Figure 2: The agent using the baseline value function seems to learn the fastest to reach an average trajectory length of 500 and also the most stable. In comparison, the VPG agent and RTG agent seem to experience slight instabilities between episode 60 and 70. Both then remain stable at the same value as the agent using the baseline value function.

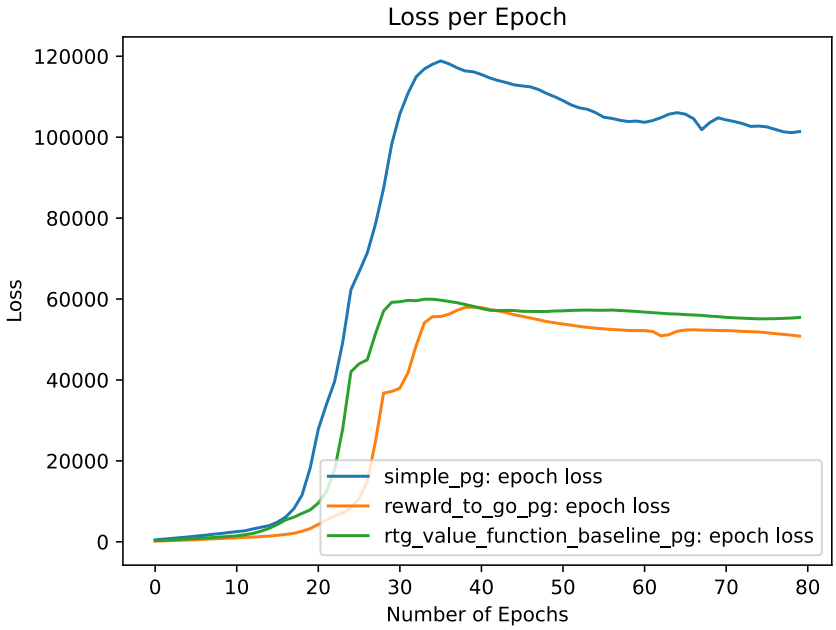

Figure 3: The "loss" go up initially. The reason why in the second half of the training the "loss" decreases again might be that in some states it does not matter too much which action is chosen. For example, as the policy improves, there are more situations experienced by the agent, where the pole is almost upright. In this case choosing action "push left" of "push right" might not have any significant impact in the future success of balancing the pole, and can be recovered in later time steps. Therefore it could be that the policy becomes more uniform in those states, which decreases the loss.