Reinforcement Learning Lab: Actor Critic architecture

Olivier Pietquin
INF581

March 7, 2018
Updated March 8, 2018.

Introduction

In this lab, we will use neural networks to learn a policy for controlling an environment with continuous state space. To address this issue we will directly search in policy spaces and we will use gradually more complex techniques going from basic policy gradient to more complex actor-critic ones.

More specifically, we will first search in a family of parameterized policies $\pi_\theta(s, a)$ using a policy gradient method. This method performs gradient ascent in the policy space so that the total return is maximized. We will restrict our work to episodic tasks, *i.e.* tasks that have a starting state and last for a finite and fixed number of steps H, called horizon.

More formally, we define an optimization criterion that we want to maximize:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{H} r(s_t, a_t)\right],$$

where $\mathbb{E}_{\pi_\theta}$ means $a \sim \pi_\theta(s, .)$ and H is the horizon of the episode. In other words, we want to maximize the value of the starting state: $V^{\pi_\theta}(s)$. The policy gradient theorem tells us that:

$$\nabla_\theta J(\theta) = \nabla_\theta V^{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)\right],$$

With

$$Q^{\pi_\theta}(s, a) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{H} r(s_t, a_t) | s = s_1, a = a_1\right].$$

Policy Gradient theorem is extremely powerful because it says one doesn't need to know the dynamics of the system to compute the gradient if one can compute the Q-function of the current policy. By applying the policy and observing the one-step transitions is enough. Using a stochastic gradient ascent and replacing $Q^{\pi_\theta}(s_t, a_t)$ by a Monte Carlo estimate $R_t = \sum_{t'=t}^{H} r(s_{t'}, a_{t'})$ over one single trajectory, we end up with a special case of the REINFORCE algorithm (see Algorithm 1).

For the purpose of focusing on the algorithms, we will use standard environments provided by OpenAI Gym suite. OpenAI Gym provides controllable environments (`https://gym.openai.com/envs/`) for research in reinforcement learning. Especially, we will try to solve the CartPole-v0 environment (`https://gym.openai.com/envs/CartPole-v0/`) which offers a continuous state space and discrete action space. The Cart Pole task consists in maintaining a pole in a vertical position by moving a cart on which the pole is attached with a joint. No friction is considered. The task is supposed to be solved if the pole stays up-right (within 15 degrees) for 195 steps in average over 100 episodes while keeping the cart position within reasonable bounds. The state is given by $\{x, \frac{\partial x}{\partial t}, \omega, \frac{\partial \omega}{\partial t}\}$ where x is the position of the cart and ω is the angle between the pole and vertical position. There are only two possible actions: {LEFT,RIGHT}.

Algorithm 1 REINFORCE with PG theorem Algorithm

Initialize θ^0 as random
Initialize step-size α_0
n = 0
while no convergence **do**
  Generate rollout $h_n = \{s_1^n, a_1^n, r_1^n, \ldots, s_H^n, a_H^n, r_H^n\} \sim \pi_{\theta^n}$
  $PG_\theta = 0$
  for $t = 1$ to H **do**
    $R_t = \sum_{t'=t}^{H} \gamma^{(t'-t)} r_{t'}^n$
    $PG_\theta \mathrel{+}= \nabla_\theta \log \pi_{\theta^n}(s_t, a_t) R_t$
  end for
  n++
  $\theta^n \leftarrow \theta^{n-1} + \alpha_n PG_\theta$
  update α_n (if step-size scheduling)
end while
return θ^n

INSTRUCTIONS:

- **For each exercise, plot a learning curve showing the average total return achieved by your algorithm as a function of the number of training steps (episodes).**
- **Assemble these plots together in a report (*maximum* two pages) – additionally including details for steps marked by $\star$. For each plot, denote which exercise/task/algorithm it refers to, the parameters used, and include a small caption/description (a few sentences describing the meaning of the plot and your observations/interpretation).**
- **Adjust hyper-parameters (α, γ, and decay of learning rate) as you see suitable. A suggestion is to begin with: $\alpha = 0.01, \gamma = 0.99$. Absolute performance is not a criteria for grading, but the performance curve should have a shape expected from each respective exercise. Use a sufficient number of episodes in the plot to give an idea of how the algorithm performs.**
- **Write a `README.txt` file to accompany your code with instructions on how to run it to obtain each of the plots given in your report.**
- **Submit your report (as a .pdf file) and your code to Moodle in a single .zip file.**

Note: Exercises 1–3 are worth 5 points each. Exercises 4–5 are bonus exercises worth 1 point each. The lab is marked out of 15 points.

Warm Up Hands on Cart Pole

First, open the `test_envs.py` file (lab materials) and run it to get used to basic functions on Cart Pole. Identify the action selection command in the code. You can notice it's not different from previously. Some more functions are available for multidimensional continuous states.

The most important command is still the `env.step(action)` one. It applies the selected action to the environment and returns an observation (next state), a reward, a flag that is set to *True* if the episode has terminated and some info.

- Try to use a different policy (for instance, a constant action) to understand the role of that command.

Although the Cart Pole has easy dynamics that can be computed analytically, we will solve this problem with Policy Gradient based Reinforcement learning.

Exercise 1. Implement and use a neural policy

A neural policy will be a neural network that outputs a probability of selecting an action in a given state. This can be roughly seen as a classification problem that classifies states (inputs) into actions (labels). Implementing a classifier with neural networks usually means estimating probabilities of belonging to a class for an input by computing a softmax probability over classes. We will thus implement a softmax policy network. Let's build a single hidden layer neural net to do so.

- Open the `PolicyNetwork_inc.py` file and read it through. The `PolicyNetwork` class implements a randomly initialised network with one hidden layer of 10 tanh units.
- ⋆ Write a loop that runs 10 episodes using that policy and computes the total reward of each episode.
- As a measure of performance, count the average return (sum of rewards) over 100 episodes on a sliding window.

Exercise 2. Implement REINFORCE

Now that we can run a neural-based policy and measure its performance, we can use REINFORCE to learn the weights of the network. Let's implement a training method.

- Implement a method in the `PolicyNetwork` class to run an episode with the current policy and store transitions.
- Implement a method that computes the discounted return starting from each state s_t in an episode $R_t = \sum_{t'=t}^{H} \gamma^{(t'-t)} r(s_{t'}, a_{t'})$
- Use that value to train the network using REINFORCE: $\nabla_\theta \log \pi_\theta(s_t, a_t) R_t$.

Notes (Update 8 March): The loss we are interested in is $J(\theta)$; and, from the policy gradient theorem: $\log \pi_\theta(s, a) R_t$ (see above), which you need to pass to the optimiser. A suggestion is to use the Adam optimizer:

```
tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

which is more stable than SGD. Recall that in standard classification you would maximize the log likelihood of the right *class*. Here, you want to maximise the log likelihood of the right *action* multiplied by the return. You can define the loss in TensorFlow using:

```
tf.nn.sparse_softmax_cross_entropy_with_logits
```

and refer the MNIST/TensorFlow tutorial for a reminder on how to setup a loss.

Exercise 3. Implementing a baseline

Because we are trying to optimize for action selection, adding or subtracting a state-only-dependent variable from the total return (or the Q-value estimate in general) doesn't change the solution of the gradient ascent. In practice, we can replace $R_t = \sum_{t'=t}^{H} \gamma^{t'} r(s_{t'}, a_{t'})$ by $R_t = \sum_{t'=t}^{H} \gamma^{t'} r(s_{t'}, a_{t'}) - b(s_t)$ where $b(s)$ is called a baseline. This would reduce the variance of the Monte Carlo estimate and thus make the learning more stable. A good candidate for the baseline is the average return. Indeed, subtracting the average return makes the gradient positive for returns that are bigger than average and negative for returns that are smaller than average. The policy gradient thus becomes $\nabla_\theta \pi_\theta(s_t, a_t)(R_t - b(s_t))$

- First, we will compute the average total reward starting from initial state as our baseline. To do so, we will use the average return computed on a sliding window as in Exercise 2.
- Second, we will use our network to predict the average return for each state. To do so, we will add a head on the top of the hidden layer of the policy network (in parallel to the softmax) which will consist of a set of weights (linear layer) which will be learnt by minimising a Mean Squared Error Loss. For each state s_t in a trajectory we will use the partial return R_t as a target for a MSE loss ($\|y - R_t\|^2$).

Note that $(R_t - b(s_t))$ is an estimate of the so-called Advantage Function $(A(s,a) = Q(s,a) - V(s))$ which quantifies how better is action a_t in state s_t compared to the action selected by the current policy.

Exercise 4 – Bonus. Adding a critic

In the previous exercise, we subtracted a baseline, that was computed as the average return for each state, to the actual return to make the learning more stable. But, by definition, the average return for a state is actually the value function for that state under the current policy $V^\pi(s)$. Therefore, this value can be learnt by temporal differences.

- We now want the extra head of our network to learn a value function instead of an average return. To do so, instead of learning the baseline using MSE, we now minimize the TD error: $r_t + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)$.
- Then use the estimated $V^\pi(s)$ as a baseline for REINFORCE. The policy gradient becomes $\nabla_\theta \pi_\theta(s_t, a_t)(R_t - V(s_t))$.

Exercise 5 – Bonus. Advantage actor critic (A2C)

Remember that the total return is a Monte Carlo estimate of the Q-function. Yet, we could also estimate the Q-function thanks to the value function we computed in the previous exercise. Indeed, the Q-function can be estimated as

$$\hat{Q}^\pi(s,a) = r(s,a) + \gamma V^\pi(s')$$

where s' is the state visited after s.

From this, the policy gradient with baseline becomes $\nabla_\theta \pi_\theta(s_t, a_t)\big(r(s_t, a_t) + \gamma V^\pi(s_{t+1}) - V^\pi(s_t)\big)$. Notice that we voluntarily omit the parameters in the value function although it is also estimated through the network. But the parameters of the value head are not learnt through policy gradient (the output of the value head is used only, it is fixed from the optimizer point of view).

- Implement this one-step version of the Advantage Actor Critic (A2C) algorithm and thus use the TD-error to learn both the critic and the actor (policy network).

Note that the TD error is now used as an estimate of the advantage function. The MC return is replaced by the bootstrapped estimate.