Introduction to Reinforcement Learning. Part 5: Policy Gradient Algorithms

Markel Sanz Ausin Nov 25, 2020 · 7 min read ★

In this part we will slightly change topics to focus on another family of Reinforcement Learning algorithms: **Policy Gradient Algorithms [1]**. We will mathematically derive the policy gradient and build an algorithm that uses it.

Among the different ways to classify the Reinforcement Learning algorithms we have mentioned so far, we still haven't described one of the families yet. These algorithms can be grouped into **Value-Based algorithms** and **Policy-Based algorithms**.

The algorithms that only use a value or action-value function and do not implement an explicit policy are in the value-based family. These algorithms do not tell you which action you should take explicitly. Instead, they show you how much reward you will collect from each state or state-action pair. Therefore, you must choose what action to take after seeing those values. An example of this could be to always take the action with the highest Q-value. Another one could be to follow an **ε-greedy policy**. Among the value-based algorithms we can find Q-Learning, DQN and the other algorithms seem up to this point.

The algorithms that explicitly implement a **policy function** which decides which action to take in each step form the **policy-based** algorithm family.

This function does not tell us how much reward the agent will receive from each state. It learns neither a value function **V(s)** nor an action-value function **Q(s, a)**. These algorithms define only a policy function π**(a|s)**, which estimates the probability of taking each action from each state.

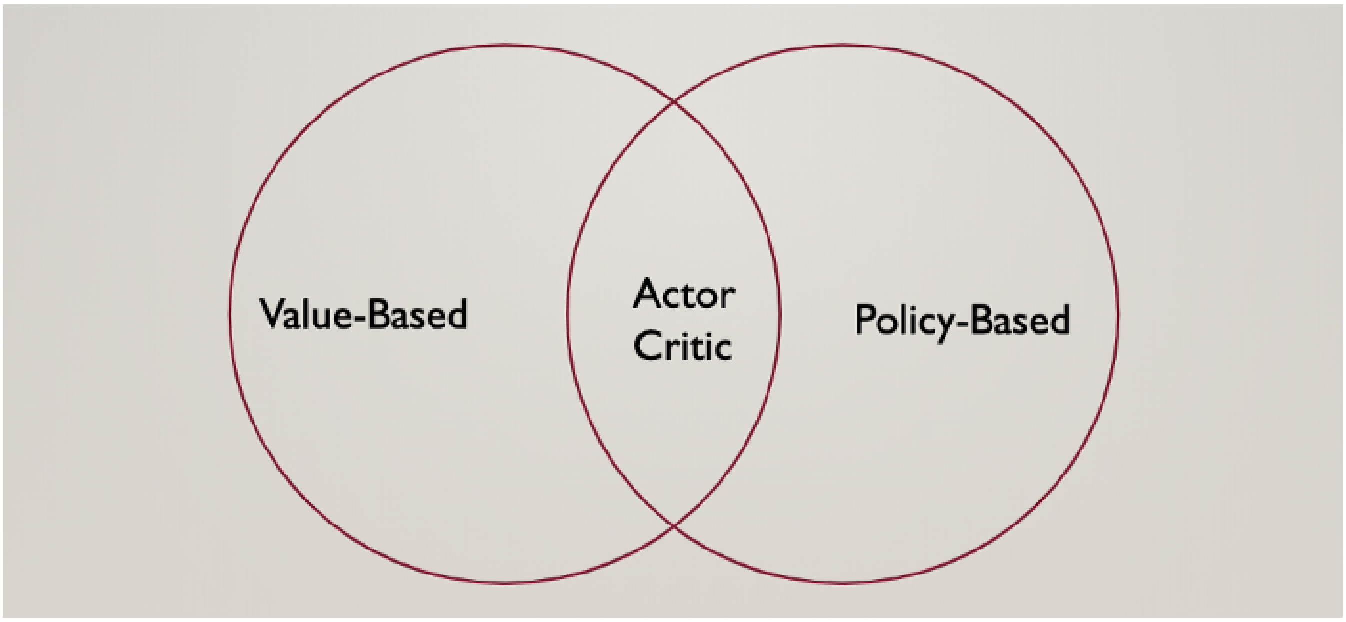
Comparison of value-based and policy-based algorithms [1]

- **Advantages of value-based algorithms:**
  1. They can be trained off-policy more easily.
  2. Significantly better sample efficiency.
  3. Lower variance.

- **Advantages of policy-based algorithms:**
  1. They can represent continuous actions, so they are more adequate for stochastic environments or for environments with continuous or high-dimensional actions.
  2. They optimize directly the function that we wish to optimize: the policy.
  3. Faster convergence.

Actor-Critic Algorithms

There exists a third group of algorithms. It tries to combine the best of both worlds: **actor-critic algorithms**. To achieve that, they use both a value function and a policy function. We will talk more about this family of algorithms in the future, but here's a figure of how the different families fit together.



Deriving the policy gradient [2]

> ***If you do not wish to read the mathematical derivation of the policy gradient, you may skip this section.***

We will now focus on **policy-based** algorithms for the case of **stochastic policies**. As we mentioned before, these algorithms learn a probabilistic function, which determines the probability of taking each action from each state: **π(a|s)**. In the future, we will describe the policy gradient for deterministic policies. Usually, **stochastic policies use the symbol π, and the sampling process is denoted a ~ π(a|s). The deterministic policies use the symbol μ, and the process of choosing an action is denoted a = μ(s)**.

The **optimal policy** will be the one that achieves the **highest possible return in a finite trajectory τ**. The **expectation of the return**, which is what we want to maximize, is defined as:

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[R(\tau)] = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[\sum_{t=0}^{T} r(s_t, a_t)] = \int \pi_\theta(\tau)R(\tau)d\tau$$

Definition of the expectation of the return.

We will employ **gradient ascent** to move the parameters of our function in **the direction what increases the expectation of the return**, which will make the received rewards go up. In order to be able to execute the gradient ascent algorithm, we will need to calculate the gradient of this function. We will **move the parameters θ of our policy π in the direction indicated by the gradient of the return:**

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta)$$

Gradient ascent to improve our policy parameters.

To calculate the **gradient of the return, ∇ J(π)**, we will begin by calculating **the gradient of the policy function ∇ π(τ)**. For that, we will use two tricks that will make the math much easier to understand. The first one is consists of **multiplying the gradient by the constant π(τ)/π(τ). The gradient of our policy function with respect to the parameters θ is then:**

Multiplying the policy gradient by a constant to make the calculus simpler.

The second trick is to remember that **the derivative of ln(x) is equal to 1/x**. Therefore, **the derivative of ln(f(x)) will be (1/f(x)) * f´(x)**. To simplify the notation, we will generalize the logarithm as ¨log¨ from now on. The gradient of our policy function is:

$$\nabla_\theta \pi_\theta(\tau) = \pi_\theta(\tau)\frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau)$$

Gradient of the policy

Now that we have calculated the gradient of the policy, let's use it to calculate the **gradient of the return**, which is what we want to maximize. This is the final policy gradient that we will use.

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \int \pi_\theta(\tau)R(\tau)d\tau = \int \nabla_\theta \pi_\theta(\tau)R(\tau)d\tau = \int \pi_\theta(\tau)\nabla_\theta \log \pi_\theta(\tau)R(\tau)d\tau = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[\nabla_\theta \log \pi_\theta(\tau)R(\tau)] = \underset{\tau\sim\pi_\theta}{\mathbb{E}}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)]$$

The gradient of the return.

> **This is the simplest form of the final policy gradient for policy-based algorithms.**

We will move the parameters of our policy function in the direction that increases R(τ).

This version of the policy gradient has **high variance**, and it is not completely adequate for complicated environments. It will be appropriate for now, but in the next few parts we will see how it can be improved. We will implement an algorithm that uses this gradient to improve the policy.

link), or keep reading to see the code without running it.

The implementation of the code will follow the same structure as in the previous parts. The first thing we will change is the architecture of the neural network, which will act as a policy. This time, our neural network must be capable of generating probabilities for the actions, so we will use a **categorical distribution**, because the actions are discrete in this environment. In the future, we will use other distributions that may be better suited to handle discrete actions. We will pass the output of the last layer in the neural network to this distribution. The distribution will provide us with the probabilities of choosing each action. We will also need to calculate the logarithms of those probabilities.

```python
# Create Neural Network for Policy Gradient-based Agent
class Network(tf.keras.Model):
  def __init__(self):
    super(Network, self).__init__()
    self.dense1 = tf.keras.layers.Dense(32, activation='relu')
    self.out = tf.keras.layers.Dense(num_actions)
    self.dist = tfp.distributions.Categorical

  def call(self, x):
    x = self.dense1(x)
    logits = self.out(x)
    action = self.dist(logits=logits).sample()
    probs = tf.nn.softmax(logits, axis=-1)
    log_probs = tf.nn.log_softmax(logits, axis=-1)
    return logits, action, probs, log_probs

net = Network()
optimizer = tf.keras.optimizers.Adam(learning_rate=2e-2)
```

part5-1.py hosted with ❤ by GitHub view raw

Now that we have defined the neural network that will act as our policy, let´s define the function that we will use to train it, using the policy gradient derived above.

We will build a function that gets the **states**, the **actions**, and the **returns** as inputs, and use them to train the algorithm in each iteration. We will pass the states to the neural network, and it will generate the probabilities and their logarithms for each state

multiplying the return and the log probability of each action, which is exactly what we defined in the previous section. Then, we will minimize the loss function to train our neural network.

```python
@tf.function
def train_step(batch_states, batch_actions, batch_returns):
  with tf.GradientTape() as tape:
    logits, actions, probs, log_probs = net(batch_states)
    action_masks = tf.one_hot(batch_actions, num_actions)
    masked_log_probs = tf.reduce_sum(action_masks * log_probs, axis=-1)
    loss = -tf.reduce_mean(batch_returns * masked_log_probs)
  net_gradients = tape.gradient(loss, net.trainable_variables)
  optimizer.apply_gradients(zip(net_gradients, net.trainable_variables))
  return loss
```

part5-2.py hosted with ❤ by **GitHub** **view raw**

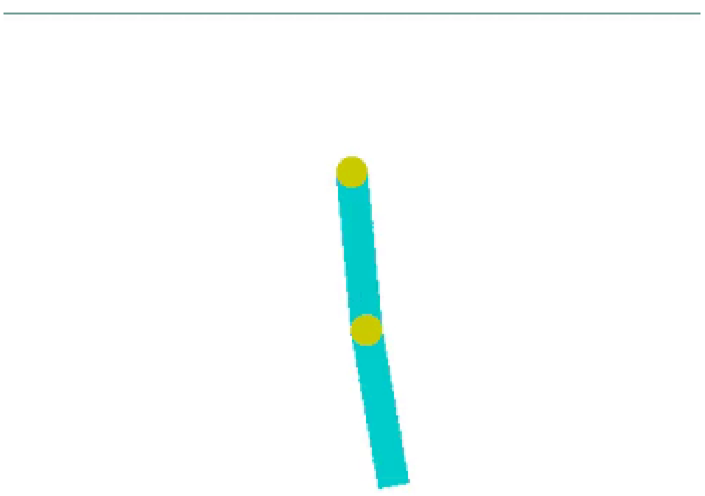
Finally, we will run the main loop, where we will collect the states and actions, as well as accumulating the rewards of the whole episode to create the return, and pass them to the function that trains the neural network.

```python
batch_states, batch_actions, batch_returns = [], [], []
for episode in range(num_episodes):
  # Start a new episode and reset the environment.
  state = env.reset()
  done, ep_rew = False, []
  while not done:
    state_in = np.expand_dims(state, 0)
    # Sample action from policy and take that action in the env.
    logits, action, probs, log_probs = net(state_in)
    next_state, reward, done, info = env.step(action[0].numpy())
    batch_states.append(state)
    batch_actions.append(action[0])
    ep_rew.append(reward)
    state = next_state

  # Create episode returns for policy gradient step.
  episode_ret = sum(ep_rew)
  episode_len = len(ep_rew)
  batch_returns += [episode_ret] * episode_len

  # Keep collecting experience with the current policy.
```

```
                             np.array(batch_returns, dtype=np.float32))
        batch_states, batch_actions, batch_returns = [], [], []
```

part5-3.py hosted with ❤ by GitHub view raw

Result of training an agent using the policy gradient.



Acrobot.

This time, we will use the Acrobot environment, where the goal is to move the arm until it is positioned above the horizontal line. The faster it achieves this goal, the higher the reward will be. The result of training this algorithm can be seen in the image on the left.

You can run the TensorFlow code yourself **in this link** (or a PyTorch version **in this link**).

References

[1] David Silver´s RL course at UCL

[2] SpinUp OpenAI

Entire series of Introduction to Reinforcement Learning:

1. Part 1: Multi-Armed Bandit Problem
2. Part 2: Q-Learning
3. Part 3: Q-Learning with Neural Networks, algorithm DQN
4. Part 4: Double DQN and Dueling DQN
5. **Part 5: Policy gradient algorithms**

My GitHub repository with common Deep Reinforcement Learning algorithms (in development): **https://github.com/markelsanz14/independent-rl-agents**

Reinforcement Learning Policy Gradient AI Neural Networks TensorFlow