

Deriving Policy Gradients and Implementing REINFORCE



Chris Yoon · [Follow](#)

4 min read · Dec 30, 2018

...

Listen

Share

More

Policy gradient methods are ubiquitous in model free reinforcement learning algorithms — they appear frequently in reinforcement learning algorithms, especially so in recent publications. The policy gradient method is also the “actor” part of Actor-Critic methods (check out my post on [Actor Critic Methods](#)), so understanding it is foundational to studying reinforcement learning!

Here, we are going to derive the policy gradient step-by-step, and implement the REINFORCE algorithm, also known as Monte Carlo Policy Gradients.

This post assumes some familiarity in reinforcement learning!

Preliminaries

In essence, policy gradient methods update the probability distribution of actions so that actions with higher expected reward have a higher probability value for an observed state. We will assume discrete (finite) action space and a stochastic (non-deterministic) policy for this post.

Some Definitions

1. Reinforcement Learning Objective

The objective function for policy gradients is defined as:

$$J(\theta) = \mathbb{E} \left[\sum_{t=0}^{T-1} r_{t+1} \right]$$

In other words, the objective is to learn a policy that maximizes the cumulative future reward to be received starting from any given time t until the terminal time T .

Note that r_{t+1} is the reward received by performing action a_t at state s_t ; $r_{t+1} = R(s_t, a_t)$ where R is the reward function.

Since this is a maximization problem, we optimize the policy by taking the gradient ascent with the partial derivative of the objective with respect to the policy parameter θ .

$$\theta \leftarrow \theta + \frac{\partial}{\partial \theta} J(\theta)$$

The policy function is parameterized by a neural network (since we live in the world of deep learning).

2. Expectation

Frequently appearing in literature is the expectation notation — it is used because we want to optimize long term future (predicted) rewards, which has a degree of uncertainty.

The expectation, also known as the expected value or the mean, is computed by the summation of the product of every x value and its probability.

$$\mathbb{E}[f(x)] = \sum_x P(x)f(x)$$

Where $P(x)$ represents the probability of the occurrence of random variable x , and $f(x)$ is a function denoting the value of x .

Deriving the Policy Gradient

Let us start with the defined objective function $J(\theta)$. We can expand the expectation as:

$$\begin{aligned} J(\theta) &= \mathbb{E}\left[\sum_{t=0}^{T-1} r_{t+1} | \pi_\theta\right] \\ &= \sum_{t=i}^{T-1} P(s_t, a_t | \tau) r_{t+1} \end{aligned}$$

where i is an arbitrary starting point in a trajectory, $P(s_t, a_t | \tau)$ is the probability of the occurrence of s_t, a_t given the trajectory τ .

Differentiate both sides with respect to policy parameter θ :

$$\text{Using } \frac{d}{dx} \log f(x) = \frac{f'(x)}{f(x)},$$

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_{t=i}^{T-1} \nabla_\theta P(s_t, a_t | \tau) r_{t+1} \\ &= \sum_{t=i}^{T-1} P(s_t, a_t | \tau) \frac{\nabla_\theta P(s_t, a_t | \tau)}{P(s_t, a_t | \tau)} r_{t+1} \\ &= \sum_{t=i}^{T-1} P(s_t, a_t | \tau) \nabla_\theta \log P(s_t, a_t | \tau) r_{t+1} \\ &= \mathbb{E}\left[\sum_{t=i}^{T-1} \nabla_\theta \log P(s_t, a_t | \tau) r_{t+1}\right] \end{aligned}$$

However, during learning, we take random samples of episodes instead of computing the expectation, so we can replace the expectation with

$$\nabla_\theta J(\theta) \sim \sum_{t=i}^{T-1} \nabla_\theta \log P(s_t, a_t | \tau) r_{t+1}$$

From here, let us take a more careful look into $\nabla_\theta \log P(s_t, a_t | \tau)$. First, by definition,

$$\begin{aligned} P(s_t, a_t | \tau) &= P(s_0, a_0, s_1, a_1, \dots, s_{t-1}, a_{t-1}, s_t, a_t | \pi_\theta) \\ &= P(s_0) \pi_\theta(a_1 | s_0) P(s_1 | s_0, a_0) \pi_\theta(a_2 | s_1) P(s_2 | s_1, a_1) \pi_\theta(a_3 | s_2) \\ &\quad \dots P(s_{t-1} | s_{t-2}, a_{t-2}) \pi_\theta(a_{t-1} | s_{t-2}) P(s_t | s_{t-1}, a_{t-1}) \pi_\theta(a_t | s_{t-1}) \end{aligned}$$

If we *log* both sides,

$$\begin{aligned}
\log P(s_t, a_t | \tau) &= \log(P(s_0) \pi_\theta(a_1 | s_0) P(s_1 | s_0, a_0) \pi_\theta(a_2 | s_1) P(s_2 | s_1, a_1) \pi_\theta(a_3 | s_2) \dots \\
&\quad P(s_{t-1} | s_{t-2}, a_{t-2}) \pi_\theta(a_{t-1} | s_{t-2}) P(s_t) \log \pi_\theta(a_t | s_{t-1})) \\
&= \log P(s_0) + \log \pi_\theta(a_1 | s_0) + \log P(s_1 | s_0, a_0) + \log \pi_\theta(a_2 | s_1) \\
&\quad + \log P(s_2 | s_1, a_1) + \log \pi_\theta(a_3 | s_2) + \dots + \log P(s_{t-1} | s_{t-2}, a_{t-2}) \\
&\quad + \log \pi_\theta(a_{t-1} | s_{t-2}) + \log P(s_t | s_{t-1}, a_{t-1}) + \log \pi_\theta(a_t | s_{t-1})
\end{aligned}$$

Then, differentiating $\log P(s_t, a_t | \tau)$ with respect to θ yields:

$$\begin{aligned}
\nabla_\theta \log P(s_t, a_t | \tau) &= \nabla_\theta \log P(s_0) + \nabla_\theta \log \pi_\theta(a_1 | s_0) + \nabla_\theta \log P(s_1 | s_0, a_0) \\
&\quad + \nabla_\theta \log \pi_\theta(a_2 | s_1) + \nabla_\theta \log P(s_2 | s_1, a_1) + \nabla_\theta \log \pi_\theta(a_3 | s_2) + \\
&\quad \dots + \nabla_\theta \log P(s_{t-1} | s_{t-2}, a_{t-2}) + \nabla_\theta \log \pi_\theta(a_{t-1} | s_{t-2}) + \\
&\quad \nabla_\theta \log P(s_t | s_{t-1}, a_{t-1}) + \nabla_\theta \log \pi_\theta(a_t | s_{t-1})
\end{aligned}$$

However, note that the $P(s_t | s_{t-1}, a_{t-1})$ is not dependent on the policy parameter θ , and is solely dependant on the environment on which the reinforcement learning is acting on; it is assumed that the state transition is unknown to the agent in model free reinforcement learning. Thus, the gradient of it with respect to θ will be 0. How convenient! So:

$$\begin{aligned}
\nabla_\theta \log P(s_t, a_t | \tau) &= 0 + \nabla_\theta \log \pi_\theta(a_1 | s_0) + 0 + \nabla_\theta \log \pi_\theta(a_2 | s_1) + 0 + \nabla_\theta \log \pi_\theta(a_3 | s_2) + \\
&\quad \dots + 0 + \nabla_\theta \log \pi_\theta(a_{t-1} | s_{t-2}) + 0 \\
&= \nabla_\theta \log \pi_\theta(a_1 | s_0) + \nabla_\theta \log \pi_\theta(a_2 | s_1) + \nabla_\theta \log \pi_\theta(a_3 | s_2) + \\
&\quad \dots + \nabla_\theta \log \pi_\theta(a_{t-1} | s_{t-2}) + \log \pi_\theta(a_t | s_{t-1}) \\
&= \sum_{t'=0}^t \nabla_\theta \log \pi_\theta(a_{t'} | s_{t'})
\end{aligned}$$

Plugging this into our $\nabla_\theta J(\theta)$ yields:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_{t=0}^{T-1} r_{t+1} \nabla_\theta P(s_t, a_t | \tau) \\
&= \sum_{t=0}^{T-1} r_{t+1} \left(\sum_{t'=0}^t \nabla_\theta \log \pi_\theta(a_{t'} | s_{t'}) \right)
\end{aligned}$$

Lets expand that!

$$\begin{aligned}
\nabla_{\theta} J(\theta) &= \sum_{t=0}^{T-1} r_{t+1} \left(\sum_{t'=0}^t \nabla_{\theta} \log \pi_{\theta}(a_{t'} | s_{t'}) \right) \\
&= r_1 \left(\sum_{t'=0}^0 \nabla_{\theta} \log \pi_{\theta}(a_{t'} | s_{t'}) \right) + r_2 \left(\sum_{t'=0}^1 \nabla_{\theta} \log \pi_{\theta}(a_{t'} | s_{t'}) \right) \\
&\quad + r_3 \left(\sum_{t'=0}^2 \nabla_{\theta} \log \pi_{\theta}(a_{t'} | s_{t'}) \right) + \dots + r_{T-1} \left(\sum_{t'=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_{t'} | s_{t'}) \right) \\
&= r_1 \nabla_{\theta} \log \pi_{\theta}(a_0 | s_0) + r_2 (\nabla_{\theta} \log \pi_{\theta}(a_0 | s_0) + \nabla_{\theta} \log \pi_{\theta}(a_1 | s_1)) \\
&\quad + r_3 (\nabla_{\theta} \log \pi_{\theta}(a_0 | s_0) + \nabla_{\theta} \log \pi_{\theta}(a_1 | s_1) + \nabla_{\theta} \log \pi_{\theta}(a_2 | s_2)) \\
&\quad + \dots + r_T (\nabla_{\theta} \log \pi_{\theta}(a_0 | s_0) + \nabla_{\theta} \log \pi_{\theta}(a_1 | s_1) + \dots + \nabla_{\theta} \log \pi_{\theta}(a_{T-1} | s_{T-1})) \\
&= \nabla_{\theta} \log \pi_{\theta}(a_0 | s_0) (r_1 + r_2 + \dots + r_T) + \nabla_{\theta} \log \pi_{\theta}(a_1 | s_1) (r_2 + r_3 + \dots + r_T) \\
&\quad + \nabla_{\theta} \log \pi_{\theta}(a_2 | s_2) (r_3 + r_4 + \dots + r_T) + \dots + \nabla_{\theta} \log \pi_{\theta}(a_{T-1} | s_{T-1}) r_T \\
&= \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left(\sum_{t'=t+1}^T r_{t'} \right)
\end{aligned}$$

Simplifying the term $\sum_{t'=t+1}^T r_{t'}$ to G_t , we can derive the policy gradient

$$\sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t$$

Incorporating the discount factor $\gamma \in [0, 1]$ into our objective (in order to weight immediate rewards more than future rewards):

$$J(\theta) = \mathbb{E}[\gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \dots + \gamma^{T-1} r_T | \pi_{\theta}]$$

We can perform a similar derivation to obtain

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \left(\sum_{t'=t+1}^T \gamma^{t'-t-1} r_{t'} \right)$$

and simplifying $\sum_{t'=t+1}^T \gamma^{t'-t-1} r_{t'}$ to G_t ,

$$\nabla_{\theta} J(\theta) = \sum_{t=0}^{T-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) G_t$$

Please let me know if there are errors in the derivation!

Implementing the REINFORCE algorithm

REINFORCE is a Monte-Carlo variant of policy gradients (Monte-Carlo: taking random samples). The agent collects a trajectory τ of one episode using its current policy, and uses it to update the policy parameter. Since one full trajectory must be completed to construct a sample space, REINFORCE is updated in an off-policy way.

Here is the pseudo code for REINFORCE :

```
function REINFORCE
    Initialise  $\theta$  arbitrarily
    for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
        for  $t = 1$  to  $T - 1$  do
             $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$ 
        end for
    end for
    return  $\theta$ 
end function
```

Pseudo code from UToronto lecture slides

So, the flow of the algorithm is:

1. Perform a trajectory roll-out using the current policy
2. Store log probabilities (of policy) and reward values at each step
3. Calculate discounted cumulative future reward at each step
4. Compute policy gradient and update policy parameter
5. Repeat 1–4

We are now going to solve the `CartPole-v0` environment using REINFORCE with normalized rewards*!

Let's first set up the **policy network**:

```
1 import sys
2 import torch
3 import gym
4 import numpy as np
5 import torch.nn as nn
6 import torch.optim as optim
7 import torch.nn.functional as F
8 from torch.autograd import Variable
9 import matplotlib.pyplot as plt
10
11 # Constants
12 GAMMA = 0.9
13
14 class PolicyNetwork(nn.Module):
15     def __init__(self, num_inputs, num_actions, hidden_size, learning_rate=3e-4):
16         super(PolicyNetwork, self).__init__()
17
18         self.num_actions = num_actions
19         self.linear1 = nn.Linear(num_inputs, hidden_size)
20         self.linear2 = nn.Linear(hidden_size, num_actions)
21         self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)
22
23     def forward(self, state):
24         x = F.relu(self.linear1(state))
25         x = F.softmax(self.linear2(x), dim=1)
26         return x
27
28     def get_action(self, state):
29         state = torch.from_numpy(state).float().unsqueeze(0)
30         probs = self.forward(Variable(state))
31         highest_prob_action = np.random.choice(self.num_actions, p=np.squeeze(probs.detach().numpy()))
32         log_prob = torch.log(probs.squeeze(0)[highest_prob_action])
33         return highest_prob_action, log_prob
```

reinforce model.py hosted with ❤ by GitHub

[view raw](#)

The update function:

```

1  def update_policy(policy_network, rewards, log_probs):
2      discounted_rewards = []
3
4      for t in range(len(rewards)):
5          Gt = 0
6          pw = 0
7          for r in rewards[t:]:
8              Gt = Gt + GAMMA**pw * r
9              pw = pw + 1
10         discounted_rewards.append(Gt)
11
12     discounted_rewards = torch.tensor(discounted_rewards)
13     discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewards.std())
14
15     policy_gradient = []
16     for log_prob, Gt in zip(log_probs, discounted_rewards):
17         policy_gradient.append(-log_prob * Gt)
18
19     policy_network.optimizer.zero_grad()
20     policy_gradient = torch.stack(policy_gradient).sum()
21     policy_gradient.backward()
22     policy_network.optimizer.step()

```

[reinforce_update.py](#) hosted with ❤ by GitHub

[view raw](#)

*Notice that the discounted reward is normalized (i.e. subtract by mean and divide by the standard deviation of all rewards in the episode). This provides stability in training, and is explained further in Andrej Karpathy's post:

"In practice it can also be important to normalize these. For example, suppose we compute [discounted cumulative reward] for all of the 20,000 actions in the batch of 100 Pong game rollouts above. One good idea is to "standardize" these returns (e.g. subtract mean, divide by standard deviation) before we plug them into backprop. This way we're always encouraging and discouraging roughly half of the performed actions. Mathematically you can also interpret these tricks as a way of controlling the variance of the policy gradient estimator. A more in-depth exploration can be found [here](#)."

Finally, the main loop:

```

1 def main():
2     env = gym.make('CartPole-v0')
3     policy_net = PolicyNetwork(env.observation_space.shape[0], env.action_space.n, 128)
4
5     max_episode_num = 5000
6     max_steps = 10000
7     numsteps = []
8     avg_numsteps = []
9     all_rewards = []
10
11    for episode in range(max_episode_num):
12        state = env.reset()
13        log_probs = []
14        rewards = []
15
16        for steps in range(max_steps):
17            env.render()
18            action, log_prob = policy_net.get_action(state)
19            new_state, reward, done, _ = env.step(action)
20            log_probs.append(log_prob)
21            rewards.append(reward)

```

[Open in app ↗](#)



```

26         avg_numsteps.append(np.mean(numsteps[-10:]))
27         all_rewards.append(np.sum(rewards))
28         if episode % 1 == 0:
29             sys.stdout.write("episode: {}, total reward: {}, average_reward: {}")
30             break
31
32         state = new_state
33
34     plt.plot(numsteps)
35     plt.plot(avg_numsteps)
36     plt.xlabel('Episode')
37     plt.show()

```

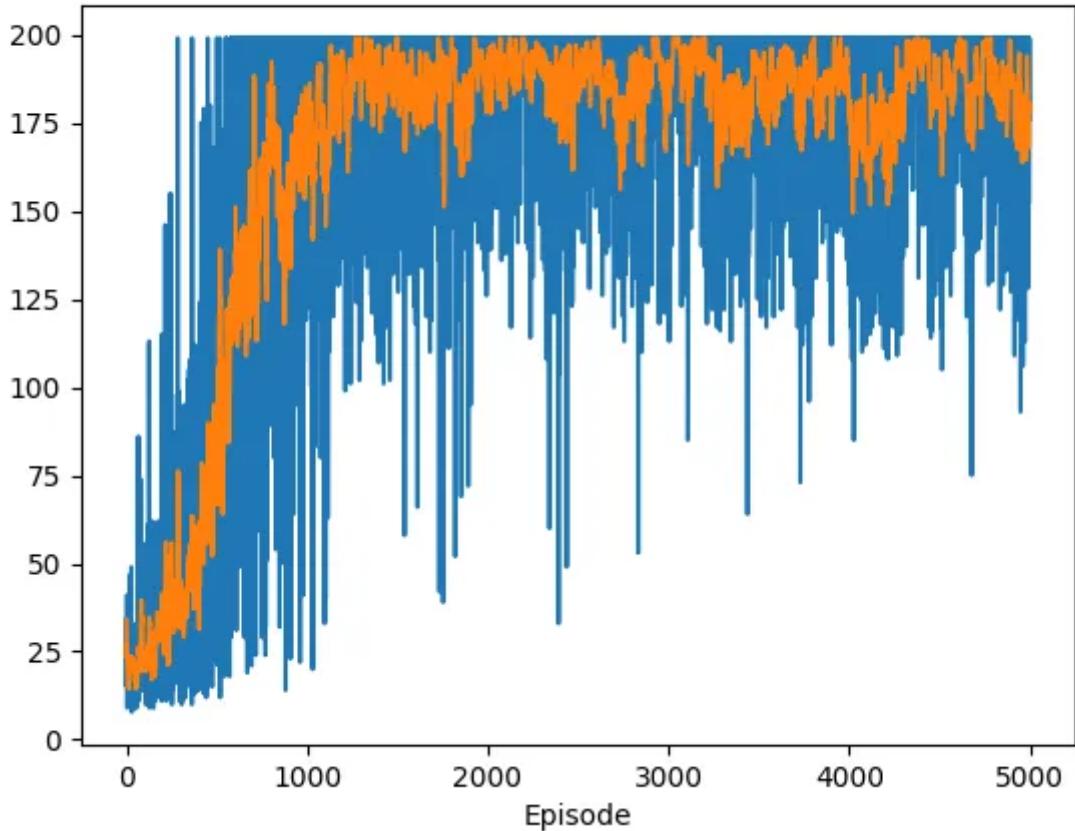
[reinforce.py](#) hosted with ❤ by GitHub

[view raw](#)

Results:

Running the main loop, we observe how the policy is learned over 5000 training episodes. Here, we will use the length of the episode as a performance index; longer episodes mean that the agent balanced the inverted pendulum for a longer time, which is what we want to see.

With the y-axis representing the number of steps the agent balances the pole before letting it fall, we see that, over time, the agent learns to balance the pole for a longer duration.



Length of episode (Blue) and average length for 10 most recent episodes (orange)

Find the full implementation and write-up on
<https://github.com/thechrisyoon08/Reinforcement-Learning!>

References:

Andrej Kaparthy's post: <http://karpathy.github.io/2016/05/31/r1/>

Official PyTorch implementation in <https://github.com/pytorch/examples>

Lecture slides from University of Toronto:
<http://www.cs.toronto.edu/~tingwuwang/REINFORCE.pdf>



Follow

Written by Chris Yoon

651 Followers

Student in NYC. <https://www.linkedin.com/in/chris-yoon-75847418b/>

More from Chris Yoon

Parameters:

θ^Q : Q network

θ^μ : Deterministic policy function

$\theta^{Q'}$: target Q network

$\theta^{\mu'}$: target policy network



Chris Yoon in Towards Data Science

Deep Deterministic Policy Gradients Explained

Reinforcement Learning in Continuous Action Spaces

6 min read · Mar 20, 2019

 1.1K 9 +

...

hm 1 Double Q-learning

ialize Q^A, Q^B, s

eat

Choose a , based on $Q^A(s, \cdot)$ and $Q^B(s, \cdot)$, observe r, s'

Choose (e.g. random) either UPDATE(A) or UPDATE(B)

if UPDATE(A) **then**

 Define $a^* = \arg \max_a Q^A(s', a)$

$Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a)(r + \gamma Q^B(s', a^*) - Q^A(s, a))$

lse if UPDATE(B) **then**

 Define $b^* = \arg \max_a Q^B(s', a)$

$Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a)(r + \gamma Q^A(s', b^*) - Q^B(s, a))$

nd if

$s \leftarrow s'$

il end

 Chris Yoon in Towards Data Science

Double Deep Q Networks

Tackling maximization bias in Deep Q-learning

5 min read · Jul 17, 2019

 451 4 +

...

↳ $[R_t \mid s_t = s]$

↳ $a \sim \pi(s) [Q^\pi(s)]$



Chris Yoon in Towards Data Science

Dueling Deep Q Networks

Dueling Network Architectures for Deep Reinforcement Learning

4 min read · Oct 19, 2019

👏 341

🗨 3



...

Derive π_{k+1} from V_{π_k}

$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \dots \xrightarrow{I} \pi_k \xrightarrow{E} V_{\pi_k}$

$$V_{k+1}(s) \doteq E_{\pi}[r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s]$$

apply iteratively until

$$|V_k(s) - V_{k+1}(s)| < \epsilon$$



Chris Yoon in Towards Data Science

Vanilla Deep Q Networks

Deep Q Learning Explained

6 min read · Jul 15, 2019

👏 380

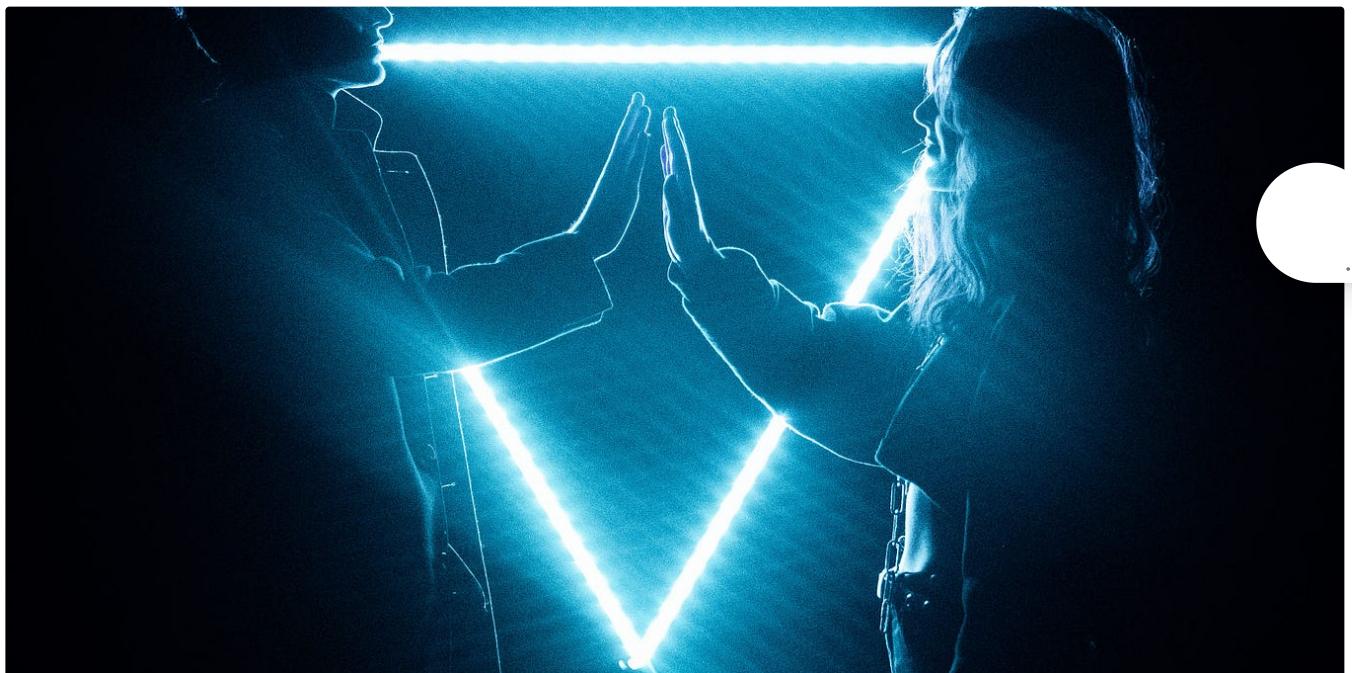
🗨 2



...

See all from Chris Yoon

Recommended from Medium



Wouter van Heeswijk, PhD in Towards Data Science

Proximal Policy Optimization (PPO) Explained

The journey from REINFORCE to the go-to algorithm in continuous control

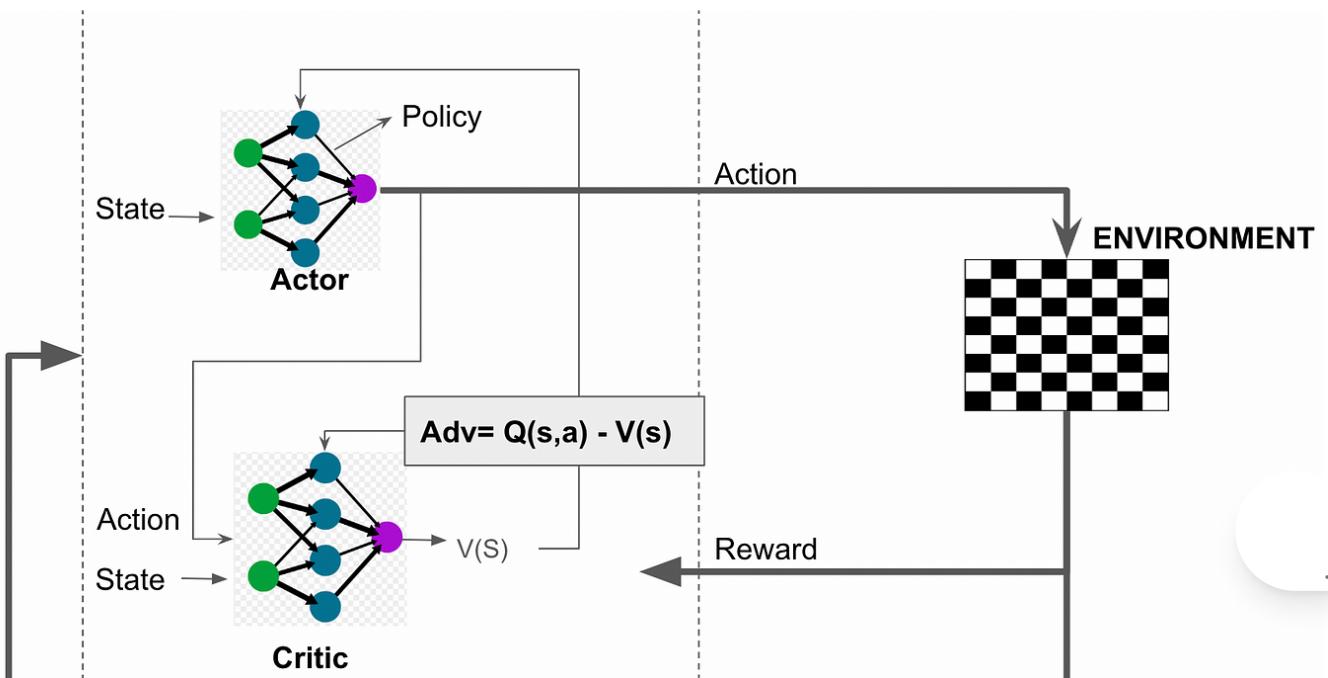
★ · 13 min read · Nov 29, 2022

191

3



...



 Renu Khandelwal

Unlocking the Secrets of Actor-Critic Reinforcement Learning: A Beginner's Guide

Understanding Actor-Critic Mechanisms, Different Flavors of Actor-Critic Algorithms, and a Simple Implementation in PyTorch

★ · 6 min read · Feb 21

 59  1



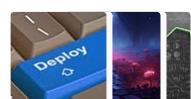
...

Lists



Practical Guides to Machine Learning

10 stories · 344 saves



Predictive Modeling w/ Python

20 stories · 313 saves



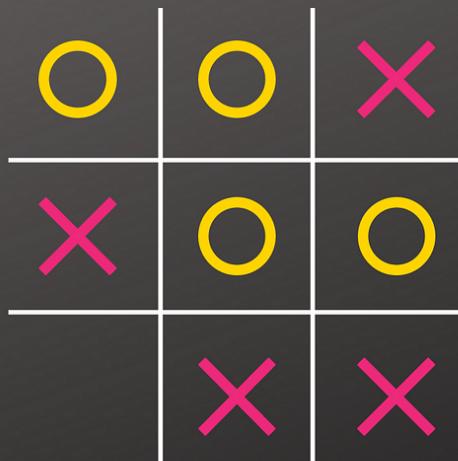
Natural Language Processing

551 stories · 174 saves



General Coding Knowledge

20 stories · 251 saves



*tic
tac
toe*

REINFORCEMENT LEARNING



Waleed Mousa in Artificial Intelligence in Plain English

Building a Tic-Tac-Toe Game with Reinforcement Learning in Python: A Step-by-Step Tutorial

Welcome to this step-by-step tutorial on how to build a Tic-Tac-Toe game using reinforcement learning in Python. In this tutorial, we will...

9 min read · Mar 13

39

1

+

...

cal Programmatic Reinforcement ia Learning to Compose Program

¹ En-Pei Hu *¹ Pu-Jen Cheng¹ Hung-Yi Lee¹



Ming-Hao Hsu

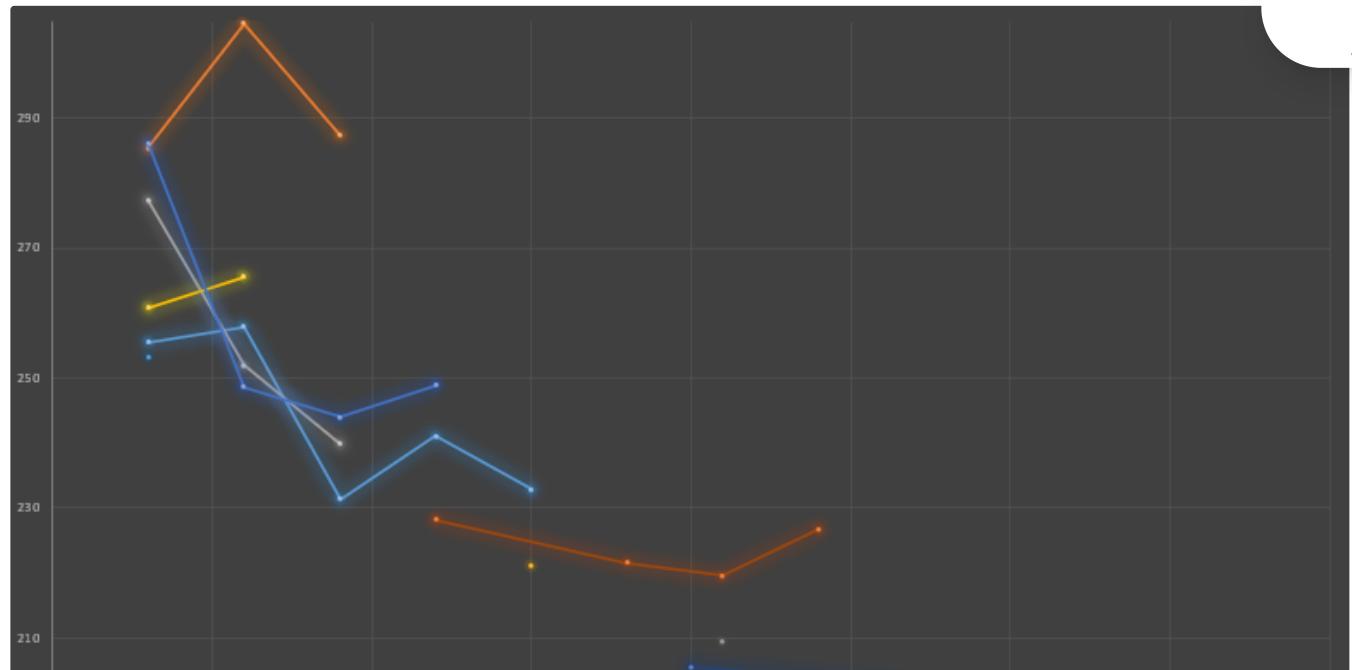
[RL] Hierarchical Programmatic Reinforcement Learning via Learning to Compose Programs (ICML23)

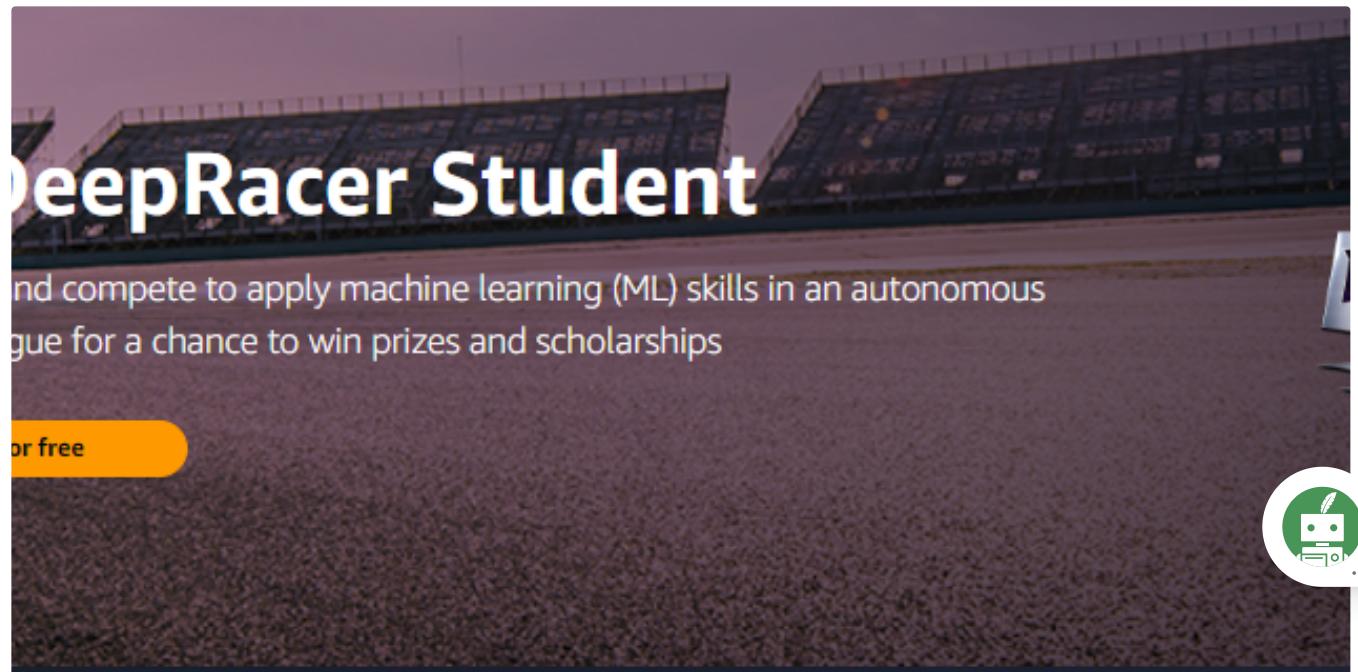
Paper Link: [Hierarchical Programmatic Reinforcement Learning via Learning to Compose Programs](#)

4 min read · Jul 21



...





 Ansh Tanwar

Dominate the Student League of AWS DeepRacer

PART-I

4 min read · Mar 24

 158

 3



...

[See more recommendations](#)