Deriving Policy Gradients and Implementing REINFORCE

Chris Yoon · Follow

4 min read · Dec 29, 2018

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:

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

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 `theta`.

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

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:

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

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:

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

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

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,

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

If we *log* both sides,

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

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

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

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:

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

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

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

Lets expand that!

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

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):

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

We can perform a similar derivation to obtain

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

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

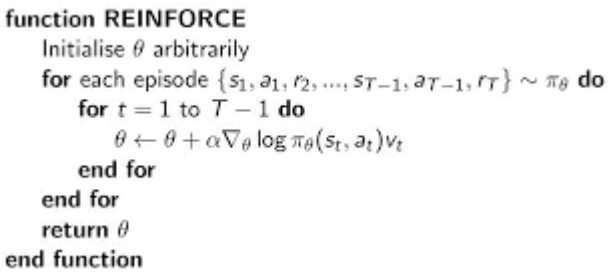
$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta log\pi_\theta(a_t|s_t) G_t$$

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 :



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:**

```python
import sys
import torch
import gym
import numpy as np
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.autograd import Variable
import matplotlib.pyplot as plt

# Constants
GAMMA = 0.9

class PolicyNetwork(nn.Module):
    def __init__(self, num_inputs, num_actions, hidden_size, learning_rate=3e-4):
        super(PolicyNetwork, self).__init__()

        self.num_actions = num_actions
        self.linear1 = nn.Linear(num_inputs, hidden_size)
        self.linear2 = nn.Linear(hidden_size, num_actions)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, state):
        x = F.relu(self.linear1(state))
        x = F.softmax(self.linear2(x), dim=1)
        return x

    def get_action(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0)
        probs = self.forward(Variable(state))
        highest_prob_action = np.random.choice(self.num_actions, p=np.squeeze(probs.detach().numpy
        log_prob = torch.log(probs.squeeze(0)[highest_prob_action])
        return highest_prob_action, log_prob
```

reinforce_model.py hosted with ❤ by GitHub view raw

The update function:

```python
def update_policy(policy_network, rewards, log_probs):
    discounted_rewards = []

    for t in range(len(rewards)):
        Gt = 0
        pw = 0
        for r in rewards[t:]:
            Gt = Gt + GAMMA**pw * r
            pw = pw + 1
        discounted_rewards.append(Gt)

    discounted_rewards = torch.tensor(discounted_rewards)
    discounted_rewards = (discounted_rewards - discounted_rewards.mean()) / (discounted_rewards.st

    policy_gradient = []
    for log_prob, Gt in zip(log_probs, discounted_rewards):
        policy_gradient.append(-log_prob * Gt)

    policy_network.optimizer.zero_grad()
    policy_gradient = torch.stack(policy_gradient).sum()
    policy_gradient.backward()
    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 Kaparthy's post:

> *"In practice it can 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:

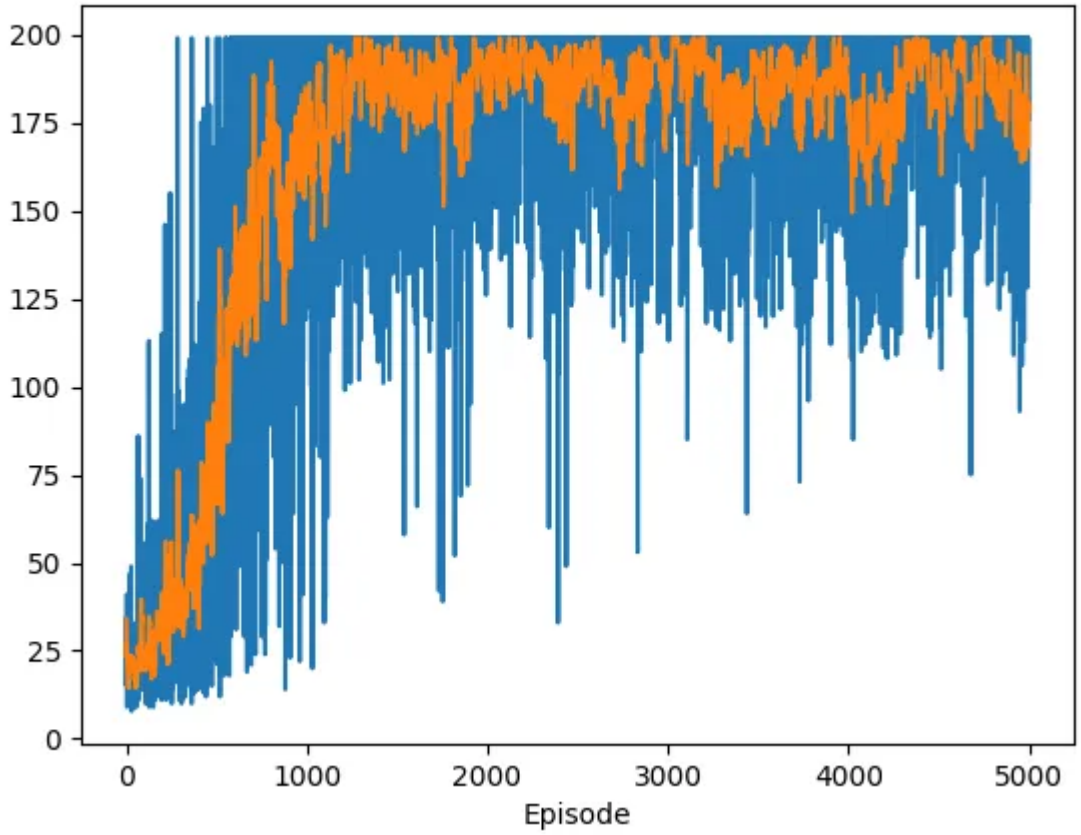
```python
def main():
    env = gym.make('CartPole-v0')
    policy_net = PolicyNetwork(env.observation_space.shape[0], env.action_space.n, 128)

    max_episode_num = 5000
    max_steps = 10000
    numsteps = []
    avg_numsteps = []
    all_rewards = []

    for episode in range(max_episode_num):
        state = env.reset()
        log_probs = []
        rewards = []

        for steps in range(max_steps):
            env.render()
            action, log_prob = policy_net.get_action(state)
            new_state, reward, done, _ = env.step(action)
            log_probs.append(log_prob)
            rewards.append(reward)

            if done:
                update_policy(policy_net, rewards, log_probs)
                numsteps.append(steps)
                avg_numsteps.append(np.mean(numsteps[-10:]))
                all_rewards.append(np.sum(rewards))
                if episode % 1 == 0:
                    sys.stdout.write("episode: {}, total reward: {}, average_reward: {}, length: {
                break

            state = new_state

    plt.plot(numsteps)
    plt.plot(avg_numsteps)
    plt.xlabel('Episode')
    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/rl/

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

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

Machine Learning | Reinforcement Learning | Deep Learning | AI | Programming

Follow

Written by Chris Yoon

651 Followers

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

More from Chris Yoon

$$\sum_{t=0}^{T-1}$$

$$\mathbb{E}_\tau\left[\sum \nabla_\theta \log$$

Chris Yoon in Towards Data Science

Understanding Actor Critic Methods

Preliminaries

6 min read · Feb 5, 2019

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)
f UPDATE(A) **then**
Define $a^* = \arg\max_a Q^A(s',a)$
$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
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
$\leftarrow s'$
il end

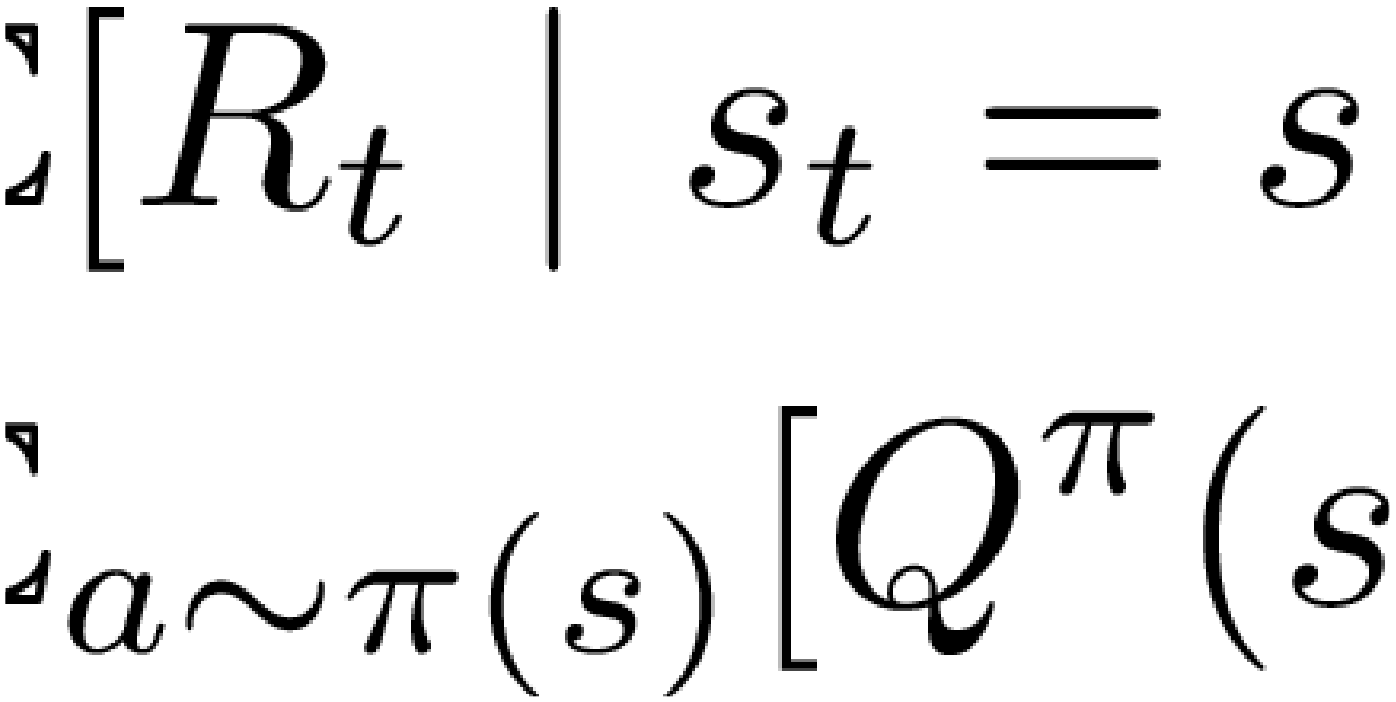
Chris Yoon in Towards Data Science

Double Deep Q Networks

Tackling maximization bias in Deep Q-learning

5 min read · Jul 17, 2019

450 4

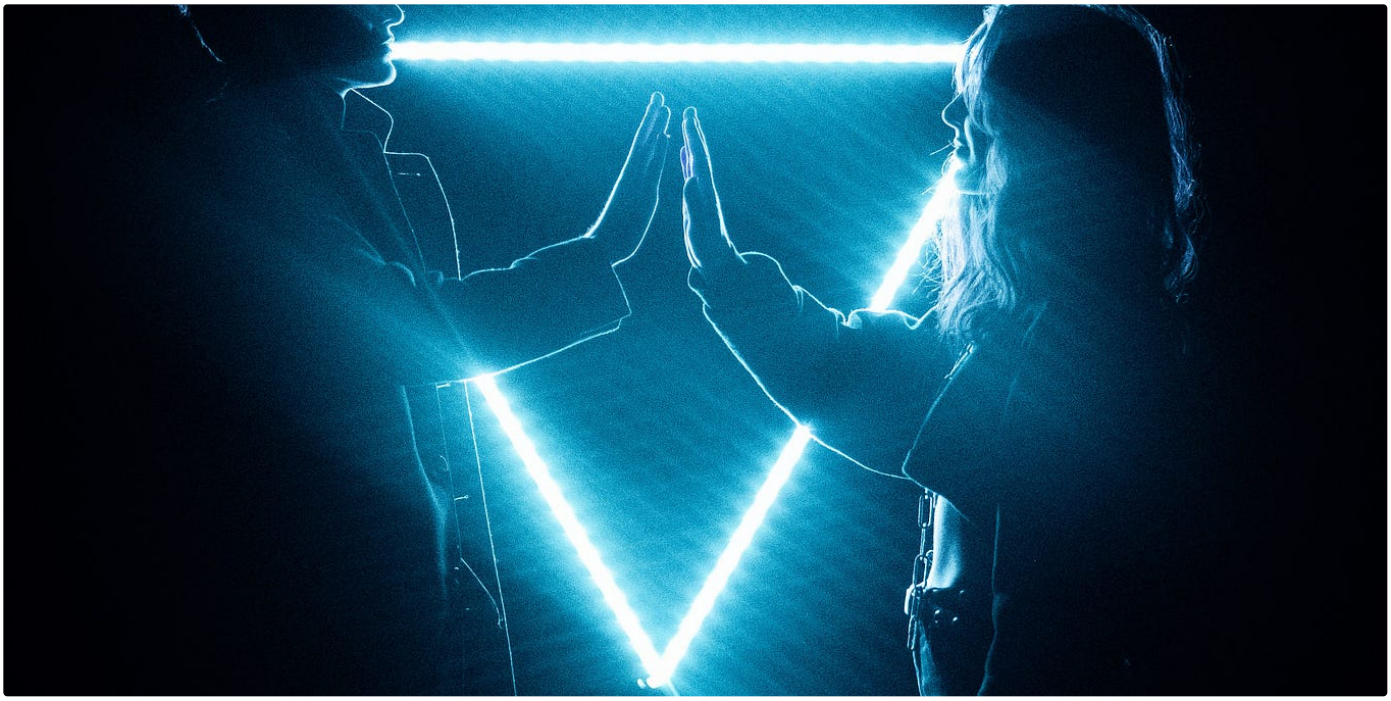


Chris Yoon in Towards Data Science

Dueling Deep Q Networks

Dueling Network Architectures for Deep Reinforcement Learning

4 min read · Oct 19, 2019

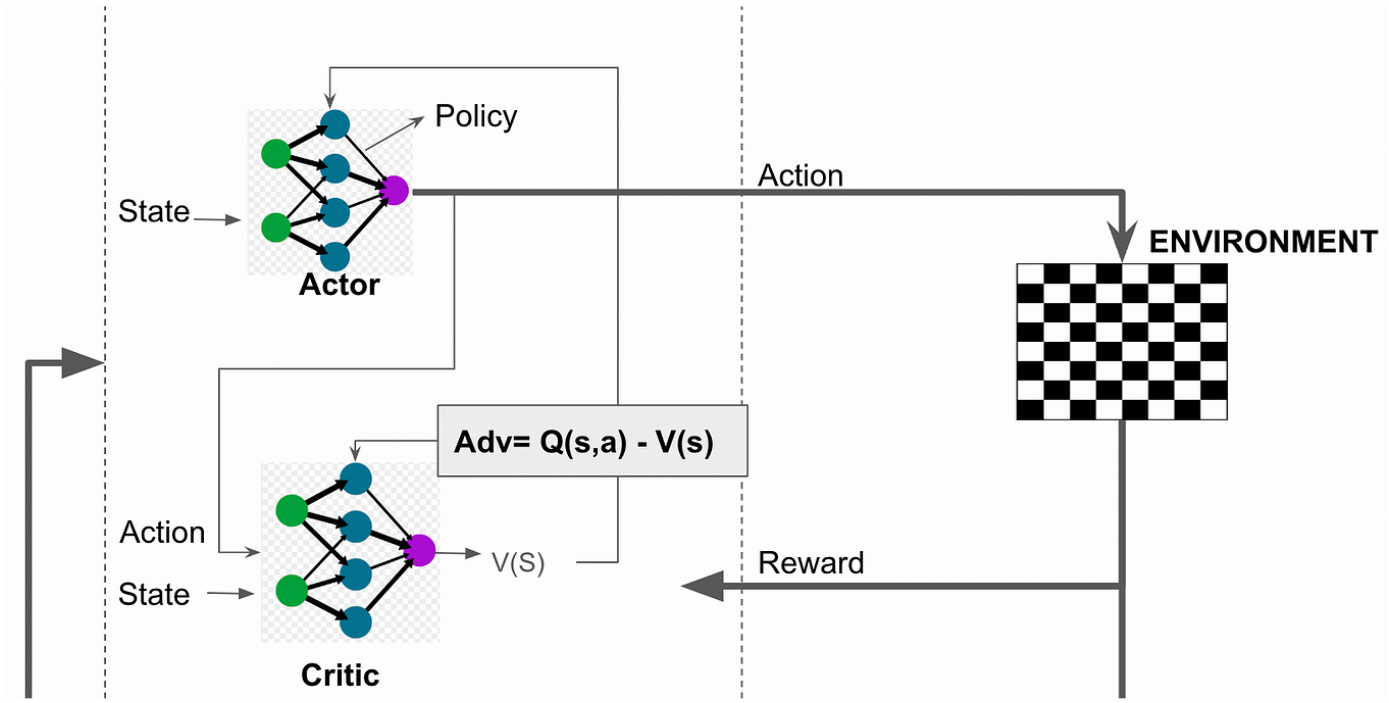


Wouter van Heeswijk, PhD in Towards Data Science

Proximal Policy Optimization (PPO) Explained

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

· 13 min read · Nov 29, 2022



Renu Khandelwal

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

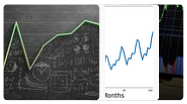
Understanding Actor-Critic Mechanisms, Different Flavors of Actor-Critic Algorithms, and a Simple Implementation in PyTorch

· 6 min read · Feb 20

Lists

Practical Guides to Machine Learning

10 stories · 269 saves



Predictive Modeling w/ Python

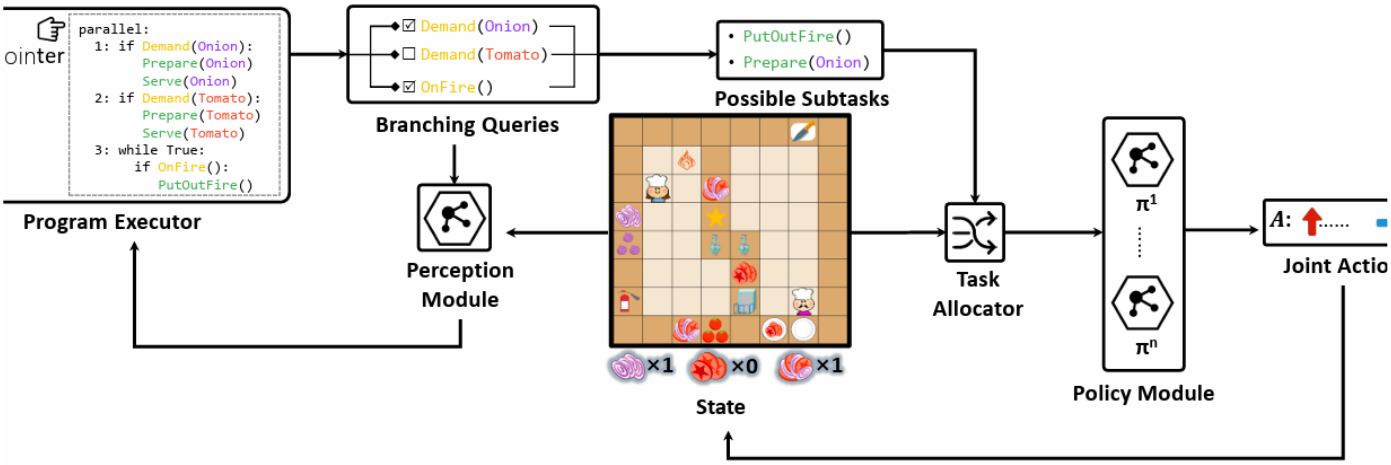
18 stories · 258 saves

Natural Language Processing

494 stories · 127 saves

General Coding Knowledge

20 stories · 204 saves



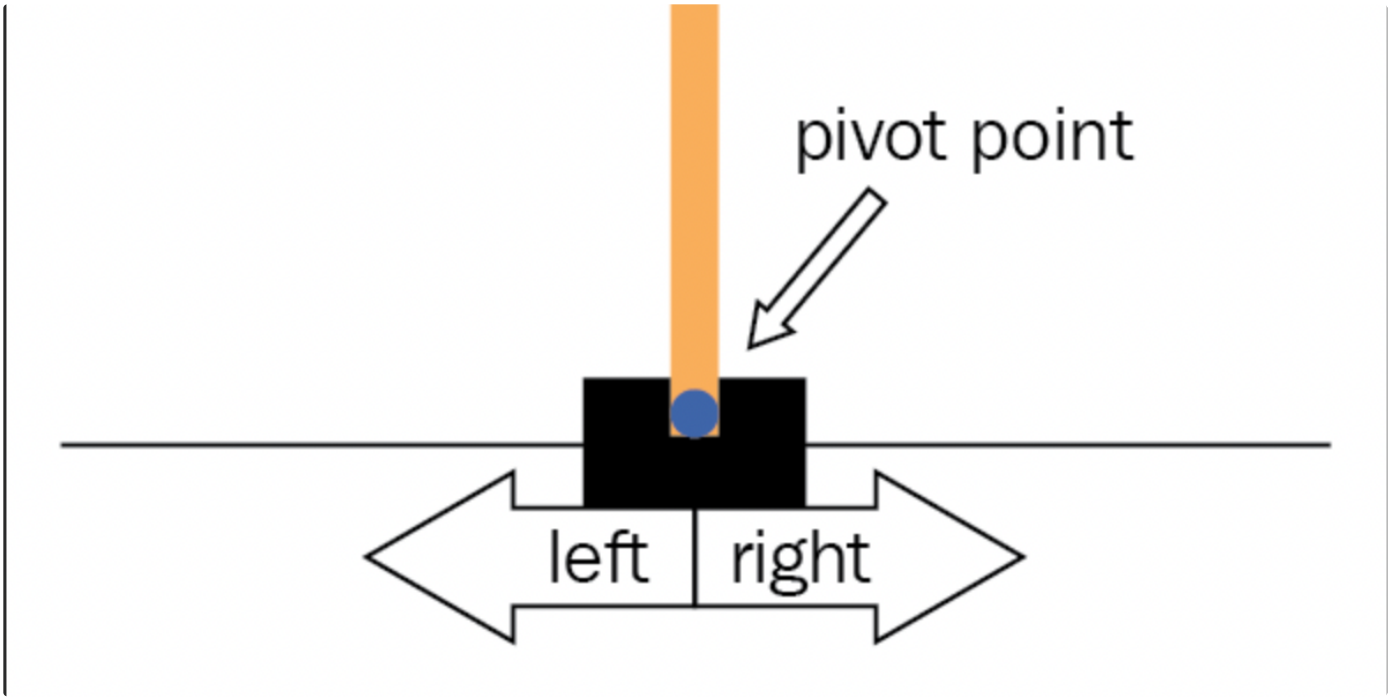
gure 1: **The overall framework of E-MAPP.** E-MAPP includes four components: 1) A percepti odule that maps a query q and the current state s to boolean responses. 2) A program executor th aintains a pool of possible subtasks and updates them according to the perceptive results. 3) A ta ocator that chooses proper subtasks from the subtask pool and assigns those to agents. 4) A poli odule that instructs agents in taking actions to accomplish specific subtasks.

Ming-Hao Hsu

[RL] E-MAPP: Efficient Multi-Agent Reinforcement Learning with Parallel Program Guidance...

Paper Link: E-MAPP: Efficient Multi-Agent Reinforcement Learning with Parallel Program Guidance

3 min read · Jul 21



Samandar Xamidov

Cart Pole Gym using Reinforcement Learning

Welcome to CartPole prooject!

4 min read · Feb 16

Jerry John Thomas

RL Series—Ep 1

This is actually a Reinforcement Learning series of short notes for me to revise when needed. Not intended for noobs but it will be...

6 min read · Jul 27

1

Shruti Dhumne

Mastering Reinforcement Learning with Q-Learning

Introduction:

5 min read · May 11

21

See more recommendations