Understanding Actor Critic Methods and A2C

Important Concepts in Deep Reinforcement Learning

Chris Yoon · Follow

Published in Towards Data Science

6 min read · Feb 6, 2019

Preliminaries

In my previous post, we derived policy gradients and implemented the REINFORCE algorithm (also known as Monte Carlo policy gradients). There are, however, some glaring issues with vanilla policy gradients: noisy gradients and high variance.

But why does that happen?

Recall the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

As in the REINFORCE algorithm, we update the policy parameter through Monte Carlo updates (i.e. taking random samples). This introduces in inherent high variability in log probabilities (log of the policy distribution) and cumulative reward values, because each trajectories during training can deviate from each other at great degrees.

Consequently, the high variability in log probabilities and cumulative reward values will make noisy gradients, and cause unstable learning and/or the policy distribution skewing to a non-optimal direction.

Besides high variance of gradients, another problem with policy gradients occurs trajectories have a cumulative reward of 0. The essence of policy gradient is increasing the probabilities for "good" actions and decreasing those of "bad" actions in the policy distribution; both "goods" and "bad" actions with will not be learned if the cumulative reward is 0.

Overall, these issues contribute to the instability and slow convergence of vanilla policy gradient methods.

Improving policy gradients: Reducing variance with a baseline

One way to reduce variance and increase stability is subtracting the cumulative reward by a baseline:

$$\text{Introducing baseline } b(s):$$

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

Intuitively, making the cumulative reward smaller by subtracting it with a baseline will make smaller gradients, and thus smaller and more stable updates.

Here is a very illustrative explanation, taken from Jerry Liu's post to "Why does the policy gradient method have high variance":

For the sake of hypothetical example, lets say that $\nabla_\theta log \pi_\theta(\tau)$ is $[0.5, 0.2, 0.3]$ respectively for three trajectories, and $r(\tau)$ is $[1000, 1001, 1002]$.

Then the variance of the product of the two terms for these three samples is $Var(0.5 \times 1000, 0.2 \times 1001, 0.3 \times 1002)$ which according to WolframAlpha is around 23286.8.

Now what if we reduce all values of $r(\tau)$ by a constant, say, 1001? Then the variance of the product becomes $Var(0.5 \times 1, 0.2 \times 0, 0.3 \times 1)$, which is 0.1633, a much smaller value.

[Taken from Jerry Liu's post "Why does the policy gradient method have high variance"]

Summary of common baseline functions

The baseline can take various values. The set of equations below illustrates the classic variants of actor critic methods (with respect to REINFORCE). In this post, we will take a look at Q Actor Critic and Advantage Actor Critic.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\, G_t\right] && \text{REINFORCE} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q^w(s,a)\right] && \text{Q Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; A^w(s,a)\right] && \text{Advantage Actor-Critic} \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; \delta\right] && \text{TD Actor-Critic}
\end{aligned}
$$

Image taken from CMU CS10703 lecture slides

Actor Critic Methods

Before we return to baselines, let's first take a look at the vanilla policy gradient again to see how the Actor Critic architecture comes in (and what is really is). Recall that:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t\right]$$

We can then decompose the expectation into:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0, a_0, \ldots, s_t, a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] \mathbb{E}_{r_{t+1}, s_{t+1}, \ldots, r_T, s_T}[G_t]$$

The second expectation term should be familiar; it is the Q value! (If you did not already know this, I would suggest that you read up on value iteration and Q learning).

$$\mathbb{E}_{r_{t+1}, s_{t+1}, \ldots, r_T, s_T}[G_t] = Q(s_t, a_t)$$

Plugging that in, we can rewrite the update equation as such:

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{s_0, a_0, \ldots, s_t, a_t}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)] Q_w(s_t, a_t) \\ &= \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) Q_w(s_t, a_t)]\end{aligned}$$

As we know, the Q value can be learned by parameterizing the Q function with a neural network (denoted by subscript `w` above).

This leads us to ***Actor Critic Methods,*** where:

1. The "Critic" estimates the value function. This could be the action-value (the *Q value*) or state-value (the *V value*).
2. The "Actor" updates the policy distribution in the direction suggested by the Critic (such as with policy gradients).

and both the Critic and Actor functions are parameterized with neural networks. In the derivation above, the Critic neural network parameterizes the Q value — so, it is called ***Q Actor Critic.***

Below is the pseudocode for Q-Actor-Critic:

Algorithm 1 Q Actor Critic

Initialize parameters s, θ, w and learning rates α_θ, α_w; sample $a \sim \pi_\theta(a|s)$.
for $t = 1 \dots T$: **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$
  Then sample the next action $a' \sim \pi_\theta(a'|s')$
  Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \log \pi_\theta(a|s)$; Compute the correction (TD error) for action-value at time t:
    $\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$
  and use it to update the parameters of Q function:
    $w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$
  Move to a $\leftarrow a'$ and s $\leftarrow s'$
end for

Adapted from Lilian Weng's post "Policy Gradient algorithms"

As illustrated, we update both the Critic network and the Value network at each update step.

Back to Baselines

To make an argument from authority (as I was not able to find the reason why), the state-value function makes an optimal baseline function. This is stated in the Carnegie Mellon CS10703 and Berekely CS294 lecture slides, but with no reason provided.

So, using the V function as the baseline function, we subtract the Q value term with the V value. Intuitively, this means *how much better it is to take a specific action compared to the average, general action at the given state*. We will call this value the ***advantage value:***

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

Does that mean we have to construct two neural networks for both the Q value and the V value (in addition to the policy network)? No. That would be very inefficient. Instead, we can use the relationship between the Q and the V from the Bellman optimality equation:

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]$$

So, we can rewrite the advantage as:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

Then, we only have to use one neural network for the V function (parameterized by v above). So we can rewrite the update equation as:

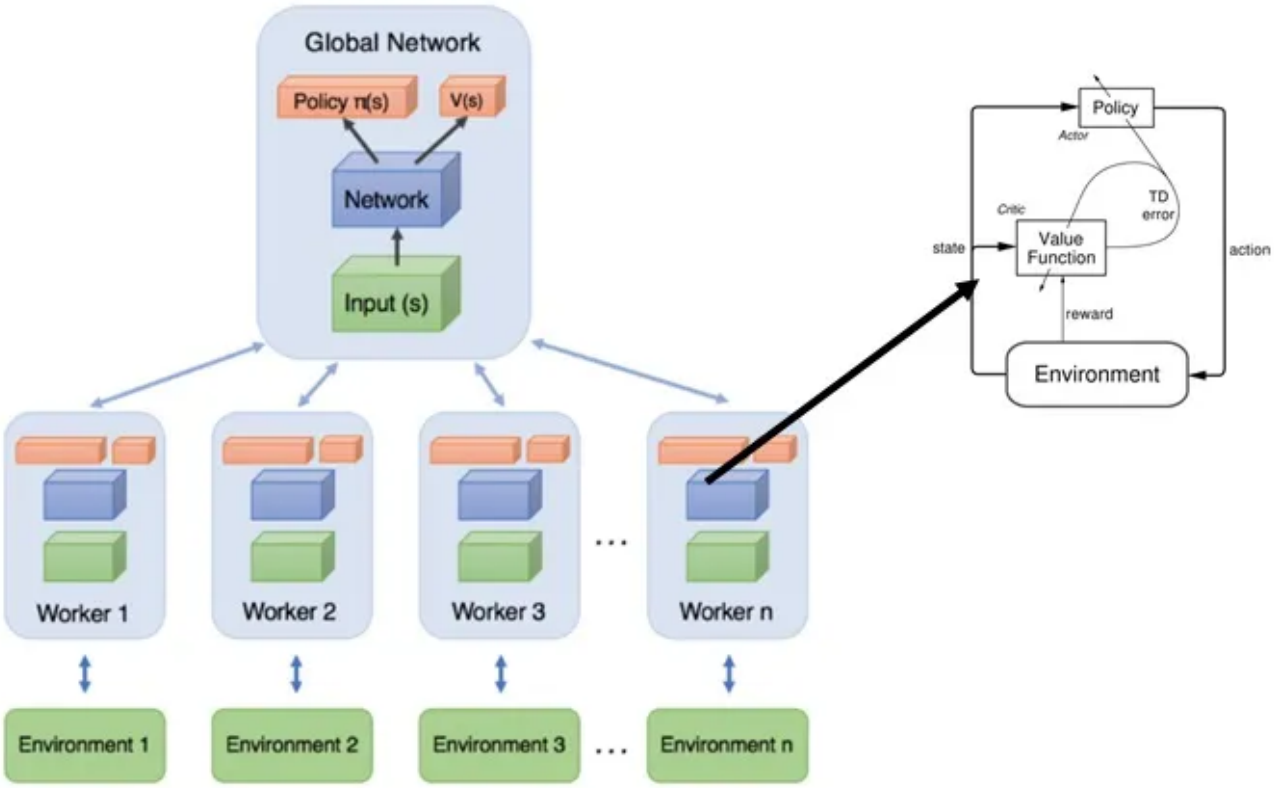
$$\begin{aligned}\nabla_\theta J(\theta) &\sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)) \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\end{aligned}$$

This is the *Advantage Actor Critic.*

Advantage Actor Critic (A2C) v.s. Asynchronous Advantage Actor Critic (A3C)

The *Advantage Actor Critic* has two main variants: the **Asynchronous Advantage Actor Critic** (A3C) and the **Advantage Actor Critic** (A2C).

A3C was introduced in Deepmind's paper "Asynchronous Methods for Deep Reinforcement Learning" (Mnih et al, 2016). In essence, A3C implements *parallel training* where multiple workers in *parallel environments independently* update a *global* value function—hence "asynchronous." One key benefit of having asynchronous actors is effective and efficient exploration of the state space.



High Level Architecture of A3C (image taken from GroundAI blog post)

A2C is like A3C but without the asynchronous part; this means a single-worker variant of the A3C. It was empirically found that A2C produces comparable performance to A3C while being more efficient. According to this OpenAI blog post, researchers aren't completely sure if or how the asynchrony benefits learning:

> *After reading the paper, AI researchers wondered whether the asynchrony led to improved performance (e.g. "perhaps the added noise would provide some regularization or exploration?"), or if it was just an implementation detail that allowed for faster training with a CPU-based implementation …*
>
> *Our synchronous A2C implementation performs better than our asynchronous implementations — we have not seen any evidence that the noise introduced by asynchrony provides any performance benefit. This A2C implementation is more cost-effective than A3C when using single-GPU machines, and is faster than a CPU-only A3C implementation when using larger policies.*

Anyhow, we will implement the **A2C** in this post as it is more simple in implementation. (This can easily be extended to A3C)

Implementing A2C

So, recall the new update equation, replacing the discounted cumulative award from vanilla policy gradients with the Advantage function:

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

On each learning step, we update both the Actor parameter (with policy gradients and advantage value), and the Critic parameter (with minimizing the mean squared error with the Bellman update equation). Let's see how this looks like in code:

Below are the includes and hyper-parameters:

```
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
import pandas as pd

# hyperparameters
hidden_size = 256
learning_rate = 3e-4

# Constants
GAMMA = 0.99
num_steps = 300
max_episodes = 3000
```

a3c_params.py hosted with ❤ by GitHub view raw

First, let's start with implementing the Actor Critic Network with the following configurations:

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

        self.num_actions = num_actions
        self.critic_linear1 = nn.Linear(num_inputs, hidden_size)
        self.critic_linear2 = nn.Linear(hidden_size, 1)

        self.actor_linear1 = nn.Linear(num_inputs, hidden_size)
        self.actor_linear2 = nn.Linear(hidden_size, num_actions)

    def forward(self, state):
        state = Variable(torch.from_numpy(state).float().unsqueeze(0))
        value = F.relu(self.critic_linear1(state))
        value = self.critic_linear2(value)

        policy_dist = F.relu(self.actor_linear1(state))
        policy_dist = F.softmax(self.actor_linear2(policy_dist), dim=1)

        return value, policy_dist
```


The main loop and update loop, as outlined above:

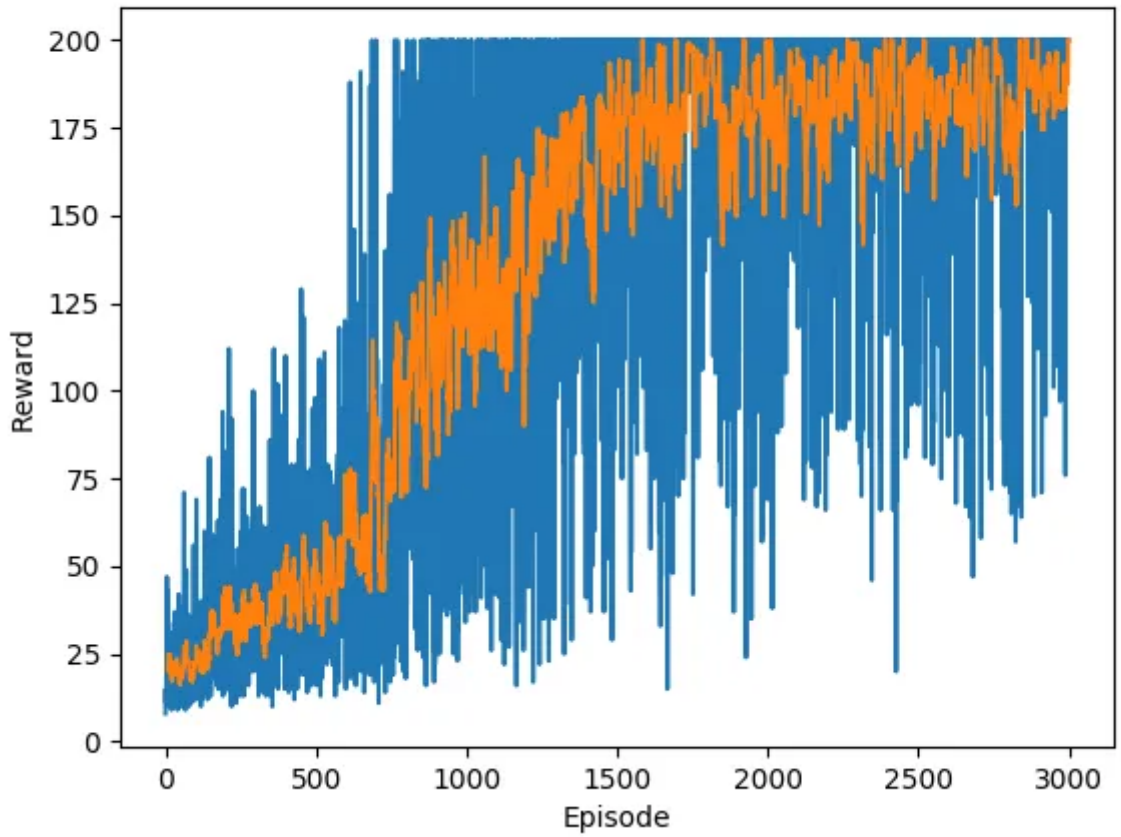
```
def a2c(env):
    num_inputs = env.observation_space.shape[0]
    num_outputs = env.action_space.n

    actor_critic = ActorCritic(num_inputs, num_outputs, hidden_size)
    ac_optimizer = optim.Adam(actor_critic.parameters(), lr=learning_rate)

    all_lengths = []
    average_lengths = []
    all_rewards = []
    entropy_term = 0

    for episode in range(max_episodes):
        log_probs = []
        values = []
        rewards = []

        state = env.reset()
        for steps in range(num_steps):
            value, policy_dist = actor_critic.forward(state)
            value = value.detach().numpy()[0,0]
            dist = policy_dist.detach().numpy()

            action = np.random.choice(num_outputs, p=np.squeeze(dist))
            log_prob = torch.log(policy_dist.squeeze(0)[action])
            entropy = -np.sum(np.mean(dist) * np.log(dist))
            new_state, reward, done, _ = env.step(action)

            rewards.append(reward)
            values.append(value)
            log_probs.append(log_prob)
            entropy_term += entropy
            state = new_state

            if done or steps == num_steps-1:
                Qval, _ = actor_critic.forward(new_state)
                Qval = Qval.detach().numpy()[0,0]
                all_rewards.append(np.sum(rewards))
                all_lengths.append(steps)
                average_lengths.append(np.mean(all_lengths[-10:]))
                if episode % 10 == 0:
                    sys.stdout.write("episode: {}, reward: {}, total length: {}, averag
                break

        # compute Q values
        Qvals = np.zeros_like(values)
        for t in reversed(range(len(rewards))):
            Qval = rewards[t] + GAMMA * Qval
```

```
            Qval = rewards[t] + GAMMA * Qval
            Qvals[t] = Qval

        #update actor critic
        values = torch.FloatTensor(values)
        Qvals = torch.FloatTensor(Qvals)
        log_probs = torch.stack(log_probs)

        advantage = Qvals - values
        actor_loss = (-log_probs * advantage).mean()
        critic_loss = 0.5 * advantage.pow(2).mean()
        ac_loss = actor_loss + critic_loss + 0.001 * entropy_term

        ac_optimizer.zero_grad()
        ac_loss.backward()
        ac_optimizer.step()



    # Plot results
    smoothed_rewards = pd.Series.rolling(pd.Series(all_rewards), 10).mean()
    smoothed_rewards = [elem for elem in smoothed_rewards]
    plt.plot(all_rewards)
    plt.plot(smoothend_rewards)
    plt.plot()
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.show()

    plt.plot(all_lengths)
    plt.plot(average_lengths)
    plt.xlabel('Episode')
    plt.ylabel('Episode length')
    plt.show()
```

Running the code, we can see how the performance improves on the OpenAI Gym CartPole-v0 environment:

```
if __name__ == "__main__":
    env = gym.make("CartPole-v0")
    a2c(env)
```

main.py hosted with ❤ by GitHub view raw



Blue: Raw rewards; Orange: Smoothed rewards

Find the full implementation here:

https://github.com/thechrisyoon08/Reinforcement-Learning/

References:

1. UC Berkeley CS294 Lecture Slides
2. Carnegie Mellon University CS10703 Lecture Slides
3. Lilian Weng's Post on Policy Gradient Algorithms
4. Jerry Liu's answer on Quora Post "Why does the policy gradient method have a high variance"
5. Naver D2 RLCode Lecture Video
6. OpenAI blog post on A2C and ACKTR
7. Diagram from GroundAI's blog post "Figure 4: Schematic representation of Asynchronous Advantage Actor Critic algorithm (A3C) algorithm."

Other Posts:

Check out my other posts on Reinforcement Learning:

- Deriving Policy Gradients and Implementing REINFORCE
- Understanding Actor Critic Methods
- Deep Deterministic Policy Gradients Explained

Thanks for reading!

Machine Learning | Reinforcement Learning | Deep Learning | Data Science | Artificial Intelligence

Follow

Written by Chris Yoon

651 Followers · Writer for Towards Data Science

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

More from Chris Yoon and Towards Data Science

$$\vartheta) = \mathbb{E}[\sum_{t=0}^{T-1} r_t \cdot$$

Chris Yoon

Deriving Policy Gradients and Implementing REINFORCE

Policy gradient methods are ubiquitous in model free reinforcement learning algorithms—they appear frequently in reinforcement learning...

4 min read · Dec 30, 2018

Bex T. in Towards Data Science

130 ML Tricks And Resources Curated Carefully From 3 Years (Plus Free eBook)

Each one is worth your time

✦ · 48 min read · Aug 1

3.1K 10

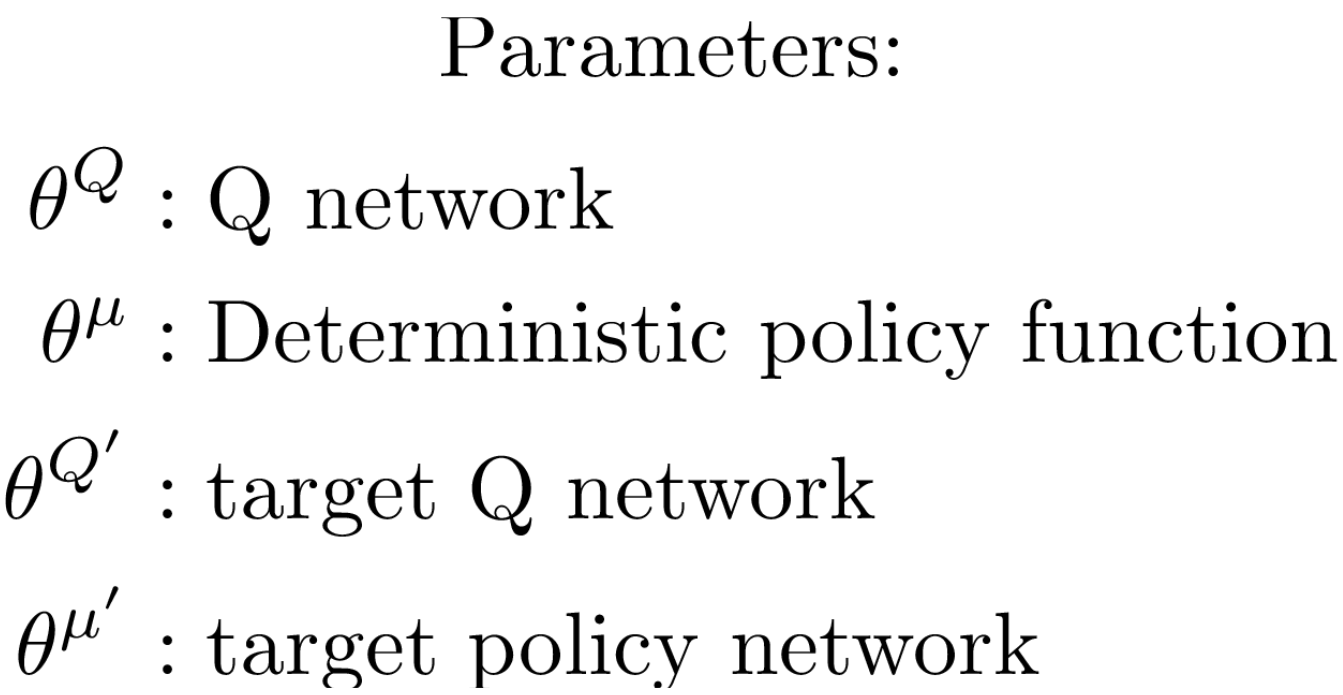
Maxime Labonne in Towards Data Science

Fine-Tune Your Own Llama 2 Model in a Colab Notebook

A practical introduction to LLM fine-tuning

✦ · 12 min read · Jul 25

1.8K 33

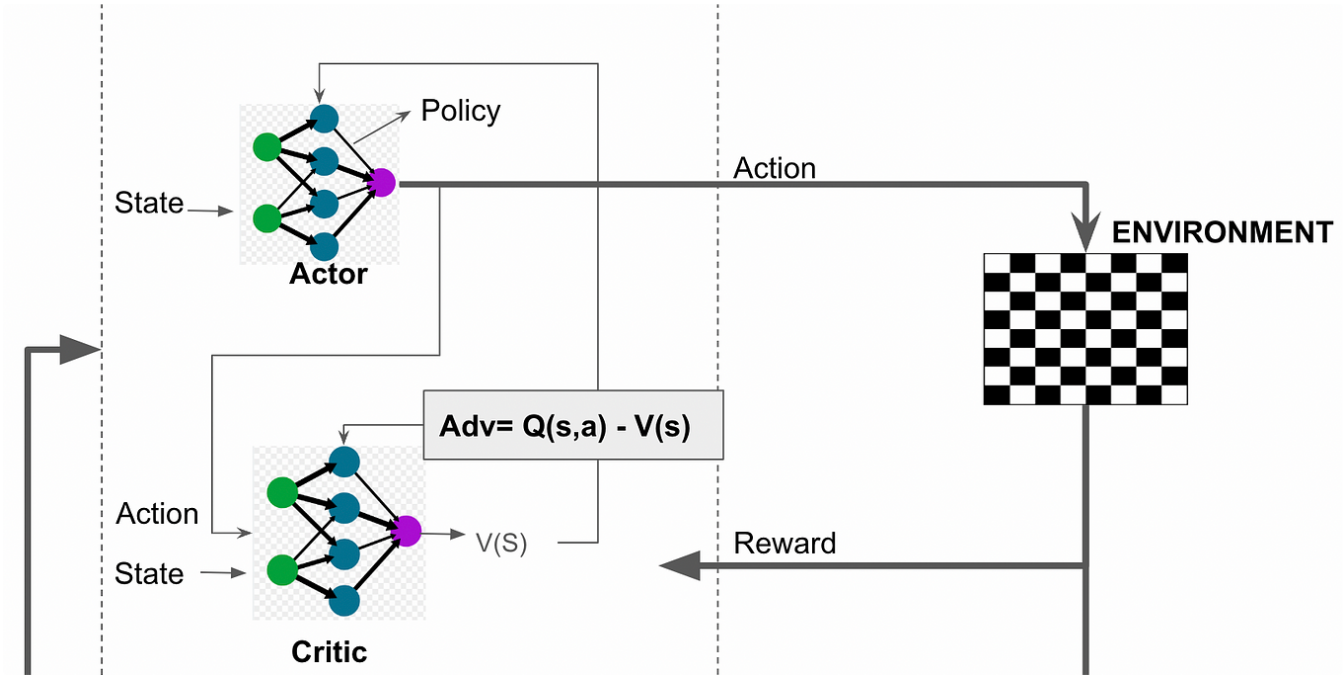


Chris Yoon in Towards Data Science

Deep Deterministic Policy Gradients Explained

Reinforcement Learning in Continuous Action Spaces

6 min read · Mar 20, 2019

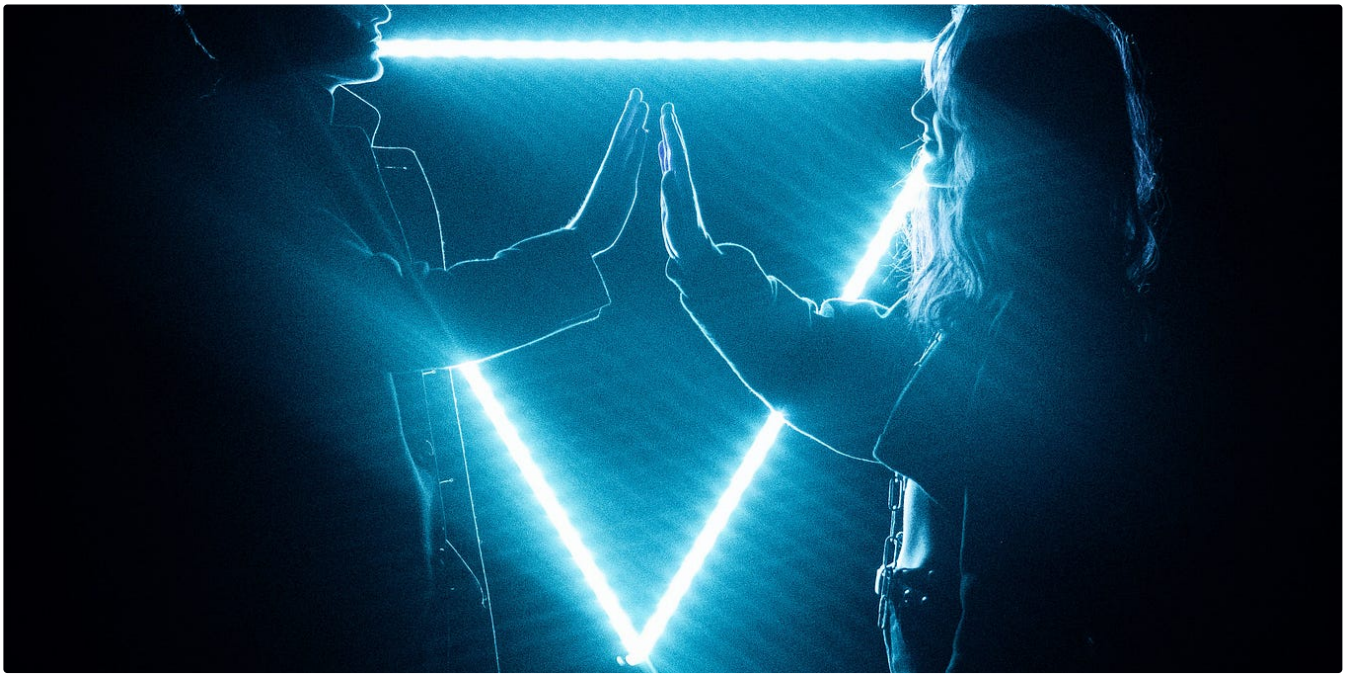


Renu Khandelwal

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



Wouter van Heeswijk, PhD in Towards Data Science

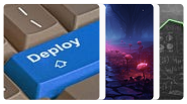
Proximal Policy Optimization (PPO) Explained

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

13 min read · Nov 29, 2022

191 3

Lists



Predictive Modeling w/ Python

20 stories · 313 saves

Natural Language Processing

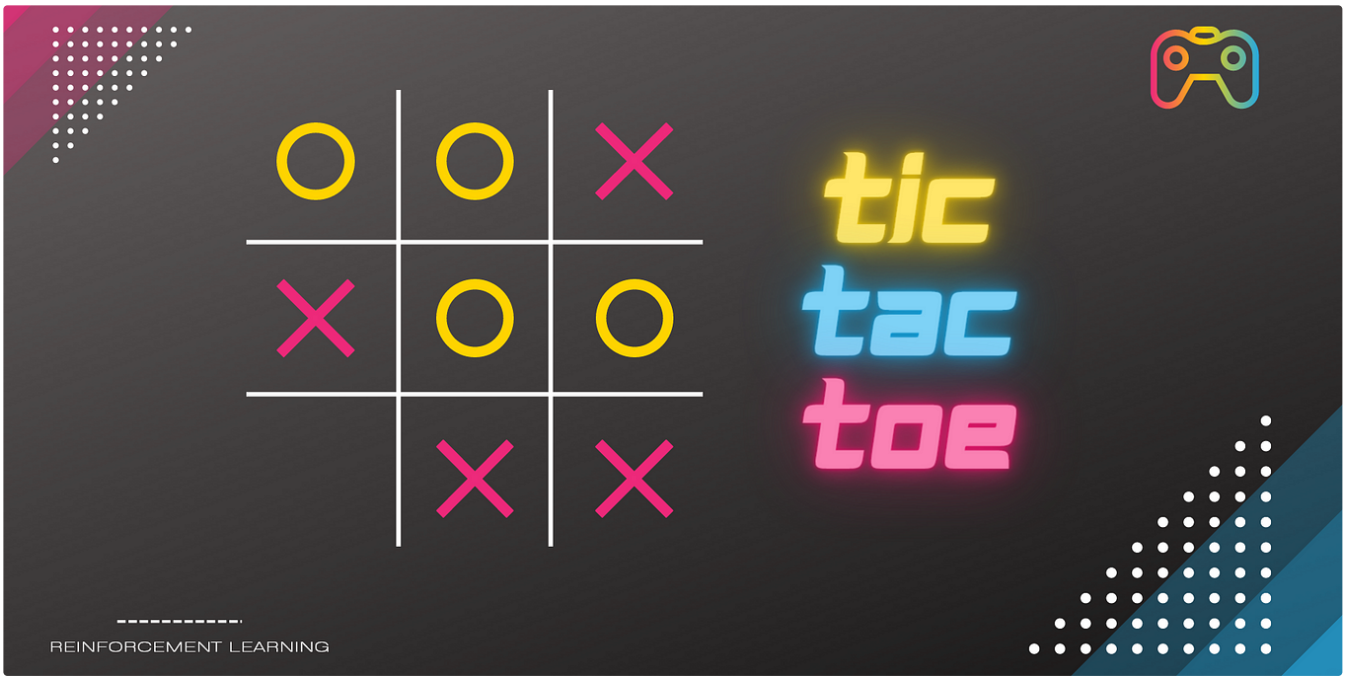
551 stories · 174 saves

Practical Guides to Machine Learning

10 stories · 344 saves

ChatGPT prompts

24 stories · 296 saves



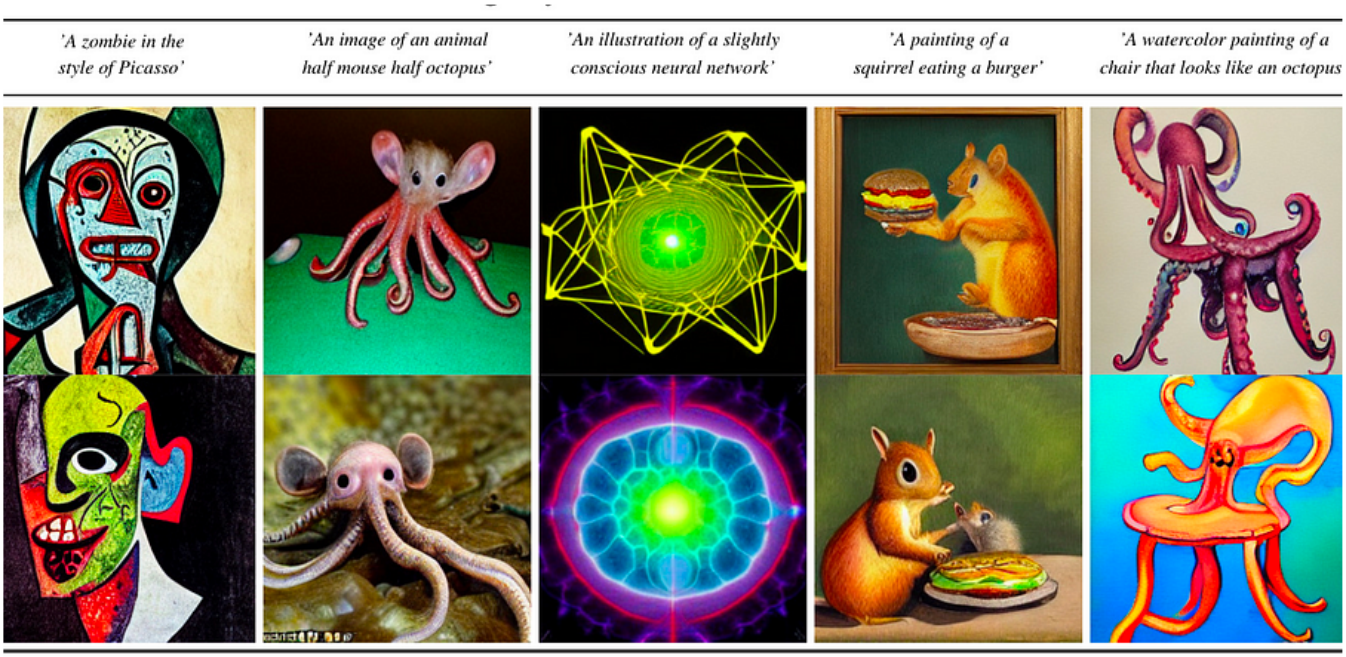
Waleed Mousa in Artificial Intelligence in Plain English

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



Onkar Mishra

Stable Diffusion Explained

How does Stable diffusion work? Explaining the tech behind text to image generation.

6 min read · Jun 8

291 2

cal Programmatic Reinforcement ia Learning to Compose Program

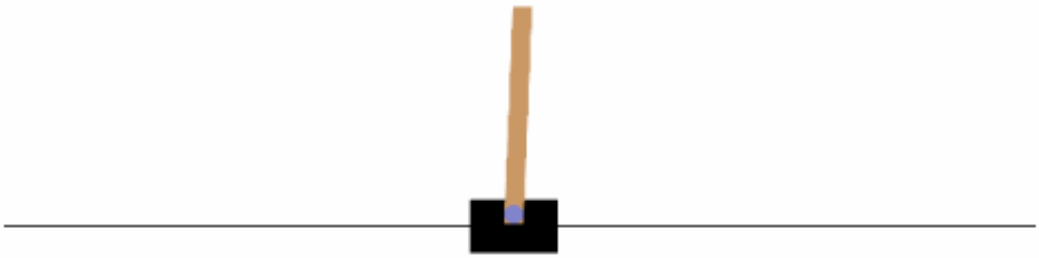
[1] En-Pei Hu [*1] Pu-Jen Cheng [1] Hung-Yi Lee [1]

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



Ludovico Buizza

Solving the CartPole Environment

A (brief) introduction to reinforcement learning, in plain English.

10 min read · Apr 10

 16 

See more recommendations