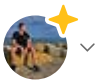



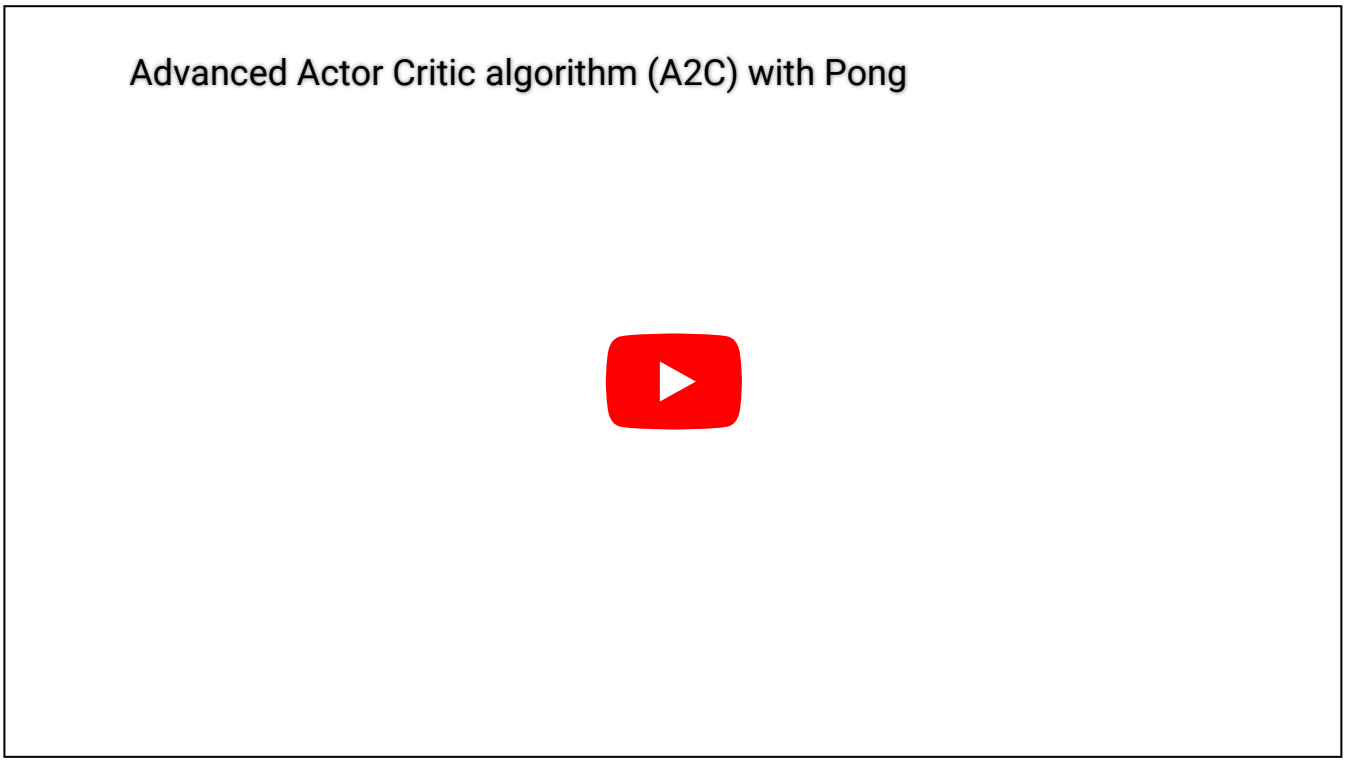
Member-only story

# An Introduction to Advantage Actor-Critic method (A2C)

Learning a Reinforcement Learning algorithm or a ‘hybrid method’ (A2C) that combines value optimization and policy optimization approaches.

Rokas Liuberskis · Follow
Published in Python in Plain English
6 min read · Sep 10, 2021

Listen Share More

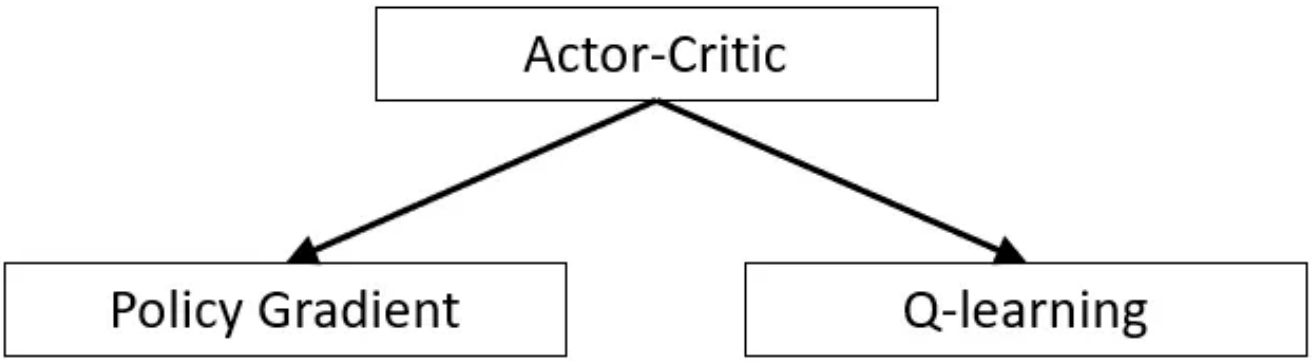



**The Most Advanced Data Science Roadmaps You’ve Ever Seen! Comes with Thousands of Free Learning Resources and ChatGPT Integration!**
https://aigents.co/learn/roadmaps/intro

Since the beginning of this Reinforcement Learning tutorial series, I've covered two different reinforcement learning methods: Value-based methods (Q-learning, Deep Q-learning…) and Policy-based methods (REINFORCE with Policy Gradients).

Both of these methods have considerable drawbacks. That's why, today, I'll try another type of Reinforcement Learning method, which we can call a 'hybrid method': Actor-Critic. The actor-Critic algorithm is a Reinforcement Learning agent that combines value optimization and policy optimization approaches. More specifically, the Actor-Critic combines the Q-learning and Policy Gradient algorithms. The resulting algorithm obtained at the high level involves a cycle that shares features between:

- Actor: a PG algorithm that decides on an action to take;
- Critic: Q-learning algorithm that critiques the action that the Actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay.




The advantage of the Actor-Critic algorithm is that it can solve a broader range of problems than DQN, while it has a lower variance in performance relative to REINFORCE. That said, because of the presence of the PG algorithm within it, the Actor-Critic is still somewhat sampling inefficient.

**The Problem with Policy Gradients:**

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

Recall the policy gradient function:

$$\Delta J(Q) = E_T\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t) G_t\right]$$

The REINFORCE algorithm updates the policy parameter through Monte Carlo updates (i.e., taking random samples). This introduces inherent high variability in log probabilities (log of the policy distribution) and cumulative reward values because each training trajectory can deviate from each other to 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.

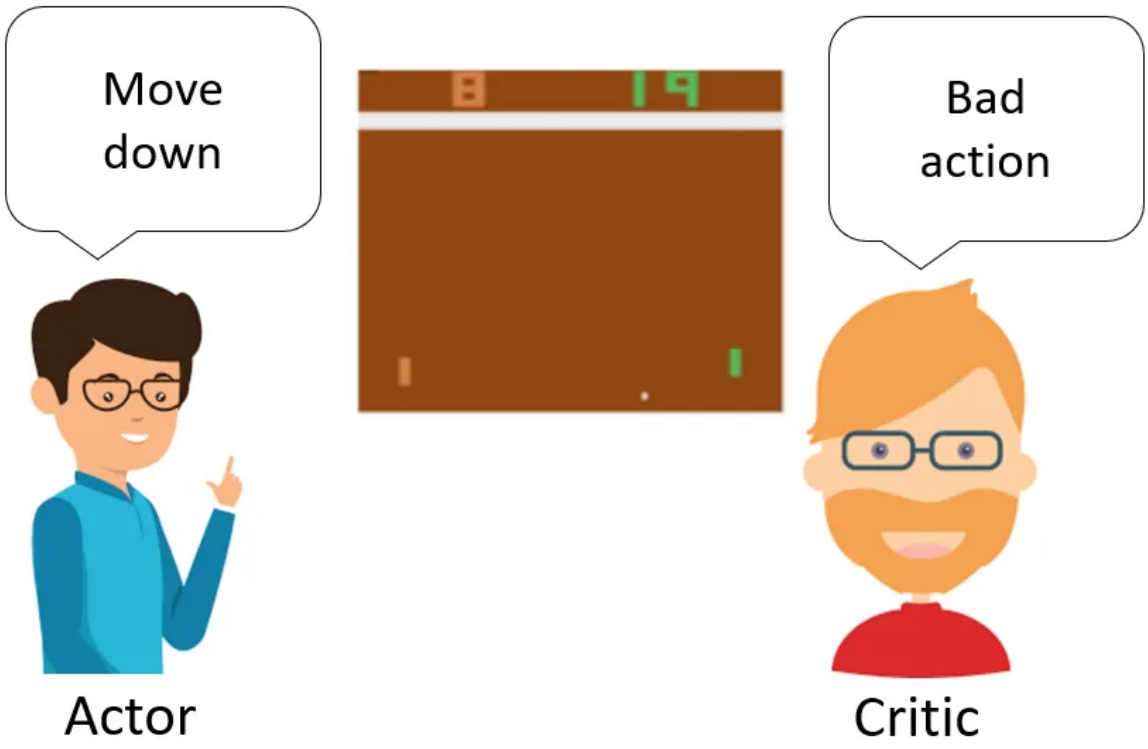
Besides high variance of gradients, another problem with policy gradients occurs trajectories have a cumulative reward of 0. The essence of policy gradient is to increase the probabilities for "good" actions and decrease those of "bad" actions in the policy distribution; both good and bad actions will not be learned if the cumulative reward is 0. Overall, these issues contribute to the instability and slow convergence of vanilla policy gradient methods. One way to reduce variance and increase stability is by subtracting the cumulative reward by a baseline `b(s)`:

$$\Delta J(Q) = E_T\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t)(G_t - b(s_t)\right]$$

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

**How Actor-Critic Works:**

Imagine you play a video game with a friend that provides you some feedback. You're the Actor, and your friend is the Critic:




In the beginning, you don't know how to play, so you try some action randomly. The Critic observes your action and provides feedback. Let's first take a look at the vanilla policy gradient again to see how the Actor-Critic architecture comes in (and what it is):

$$\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_{Q} \log \pi_{Q}(a_t, s_t) G_t\right]$$

We can then decompose the expectation into:

$$\Delta J(Q) = E_{\tau}\left[\sum_{t=0}^{T-1} \nabla_{Q} \log \pi_{Q}(a_t, s_t)\right] E_{r_{t+1}, s_{t+1}, \ldots, r_T s_T}[G_t]$$

The second expectation term should be familiar; it is the Q value!

$$E_{r_{t+1},\, s_{t+1},\, \ldots,\, r_T s_T}\left[G_t\right] = Q(s_t, a_t)$$

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

$$\Delta J(Q) = E_{s_0,\, a_0,\, \ldots,\, s_t,\, a_t}\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t)\right] Q(s_t, a_t) = E_T\left[\sum_{t=0}^{T-1} \nabla_Q \log \pi_Q(a_t, s_t)\right] Q(s_t, a_t)$$

As we know, the Q value can be learned by parameterizing the Q function with a neural network. This leads us to Actor-Critic Methods, where:

- The "Critic" estimates the value function. This could be the action-value (the Q value) or state-value (the V value).
- Critic: Q-learning algorithm that critiques the action that the Actor selected, providing feedback on how to adjust. It can take advantage of efficiency tricks in Q-learning, such as memory replay.

We update both the Critic network and the Value network at each update step.

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

$$A(s_t, a_t) = Q(s_t, a_t) - V(a_t)$$

This is so-called the Advantage Actor-Critic; in code, it looks much more straightforward, you will see.

#### Advantage Actor-Critic Implementation:

I am working on my previous tutorial code; we need to add the Critic model to the same principle. So in Policy Gradient, our model looked following:

To make it Actor-Critic, we add the ‘value’ parameter, and we compile not only the Actor model but and Critic model with ‘mse’ loss:

```python
def OurModel(input_shape, action_space, lr):
    X_input = Input(input_shape)

    X = Flatten(input_shape=input_shape)(X_input)

    X = Dense(512, activation="elu", kernel_initializer='he_uniform')(X)

    action = Dense(action_space, activation="softmax", kernel_initializer='he_uniform')(X)
    value = Dense(1, kernel_initializer='he_uniform')(X)

    Actor = Model(inputs = X_input, outputs = action)
    Actor.compile(loss='categorical_crossentropy', optimizer=RMSprop(lr=lr))

    Critic = Model(inputs = X_input, outputs = value)
    Critic.compile(loss='mse', optimizer=RMSprop(lr=lr))

    return Actor, Critic
```

A2C_reinforcement_learning_1.py hosted with ❤ by GitHub view raw

Another most important function we change is a `def replay(self)`. In policy gradient, it looked following:

```python
def replay(self):
    # reshape memory to appropriate shape for training
    states = np.vstack(self.states)
    actions = np.vstack(self.actions)

    # Compute discounted rewards
    discounted_r = self.discount_rewards(self.rewards)

    # training PG network
    self.Actor.fit(states, actions, sample_weight=discounted_r, epochs=1, verbose=0)
    # reset training memory
    self.states, self.actions, self.rewards = [], [], []
```

A2C_reinforcement_learning_2.py hosted with ❤ by GitHub view raw

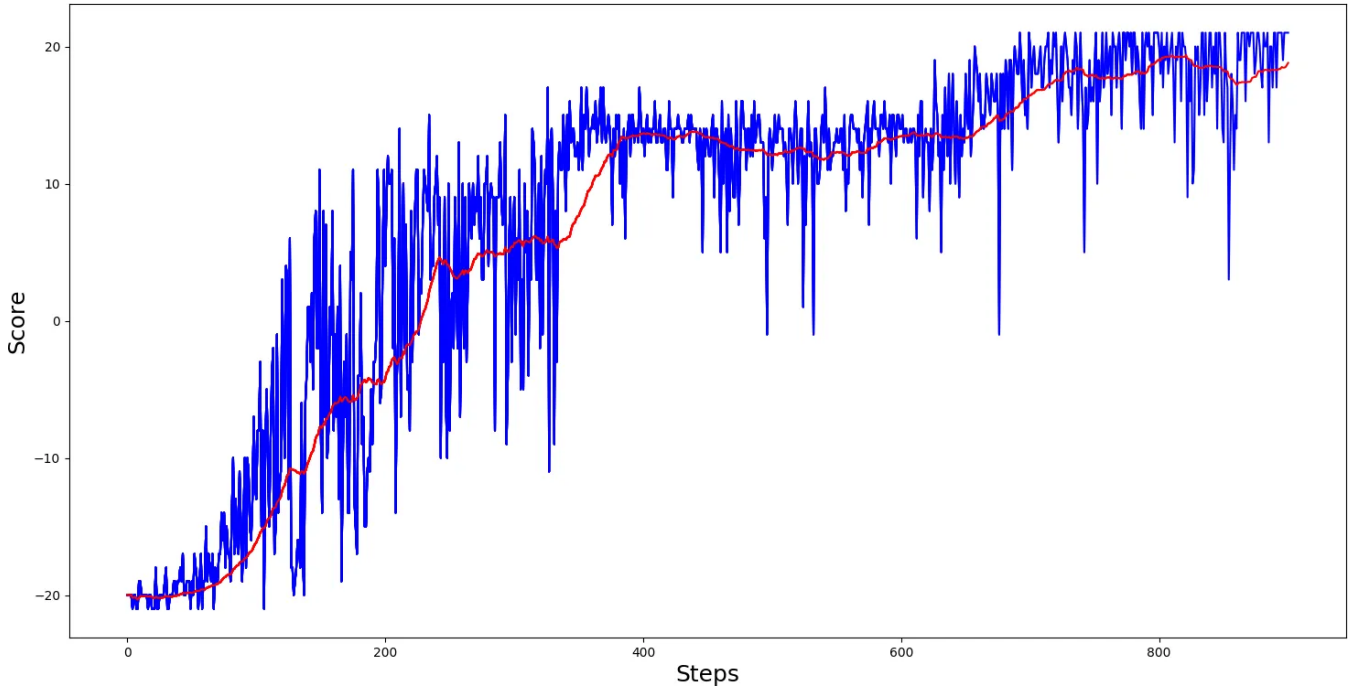
To make it work as an Actor-Critic algorithm, we predict states without the Critic model to get values that we subtract from discounted rewards, and this is how we calculate advantages. And instead of training Actor with discounted rewards, we use advantages, and for the Critic network, we use discounted rewards:

```python
def replay(self):
    # reshape memory to appropriate shape for training
    states = np.vstack(self.states)
    actions = np.vstack(self.actions)

    # Compute discounted rewards
    discounted_r = self.discount_rewards(self.rewards)

    # Get Critic network predictions
    values = self.Critic.predict(states)[:, 0]
    # Compute advantages
    advantages = discounted_r - values
    # training Actor and Critic networks
    self.Actor.fit(states, actions, sample_weight=advantages, epochs=1, verbose=0)
    self.Critic.fit(states, discounted_r, epochs=1, verbose=0)
    # reset training memory
    self.states, self.actions, self.rewards = [], [], []
```

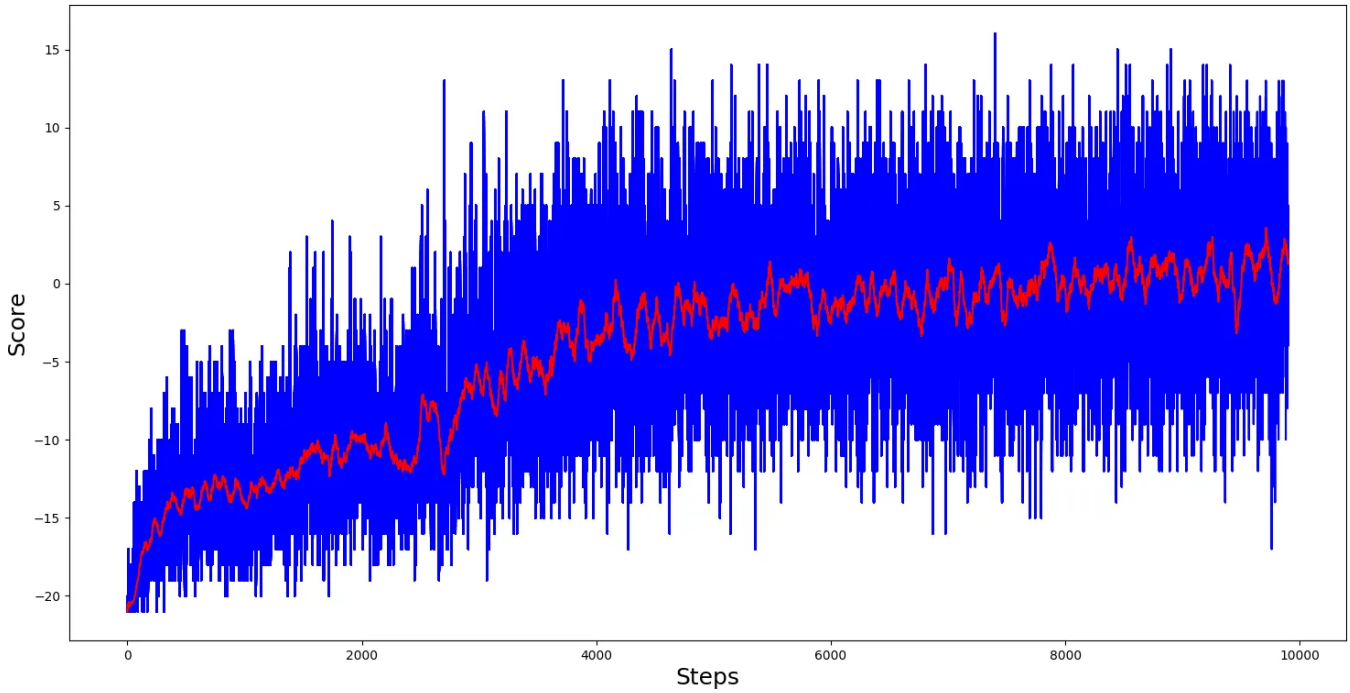
A2C_reinforcement_learning_3.py hosted with ❤ by GitHub view raw

That's it; we just needed to change a few lines of code. Moreover, you can change the 'save' and 'load model' functions. Here is the complete code:

Same as in my previous tutorial, I first trained ‘PongDeterministic-v4’ for 1000 steps, results you can see in bellow graph:




So, from the training results, we can say that the A2C model played pong relatively smoother. Despite that, it took a little longer to reach maximum scores, games of it were much more stable than PG, where we had many spikes. Then I thought, ok, let's give a chance to our 'Pong-v0' environment:




Now our 'Pong-v0' training graph looks much better than in Policy Gradient, much more stable games. But sadly, our average score couldn't get more than 11 scores per game. But keep in mind that I am using one deep layer network; you can play around with architecture.

**Conclusion:**

So, in this tutorial, we implemented a hybrid between value-based algorithms and policy-based algorithms. But we still face a problem, that learning for these models takes a lot of time. So in the next tutorial part, I will implement it as an Asynchronous A2C algorithm. This means that we will run, for example, four environments at once, and we will train the same main model. In theory, this means we will train our agent four times faster, but you will see how it looks in practice in the next tutorial part.

*Originally published at https://pylessons.com/A2C-reinforcement-learning*

*More content at plainenglish.io*

Machine Learning | Artificial Intelligence | Programming | Technology | Education



Follow

## Written by Rokas Liuberskis

339 Followers · Writer for Python in Plain English

Machine learning engineer and enthusiast

**More from Rokas Liuberskis and Python in Plain English**

Rokas Liuberskis

## Transformers and Positional Embedding: A Step-by-Step NLP Tutorial for Mastery

Introduction to Transformers Architecture covering main components, advantages, disadvantages, limitations, etc. In this part, we'll...

· 12 min read · Aug 10

6

Serop Baghdadlian in Python in Plain English

## 10 Python Anti-Patterns You Must Avoid When Writing Clean Code

Don't Repeat My Mistakes—Here is a List of the Most Common Python Pitfalls That Lower Your Code Quality and Efficiency.

✦ · 7 min read · Jul 24

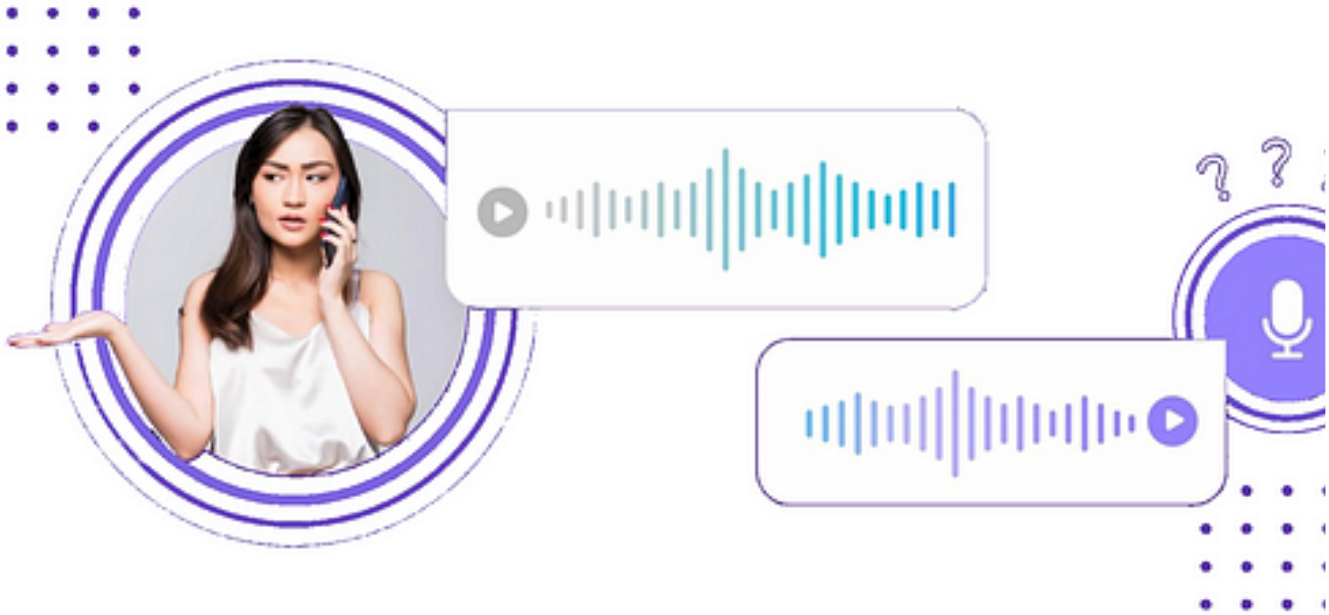
1.1K 16


Lahiru Hewawasam in Python in Plain English

## Top 11 Tools for Microservices Backend Development in 2023

10 min read · May 25

562 9



Rokas Liuberskis in Towards AI

## Introduction to speech recognition with TensorFlow

Master the basics of speech recognition with TensorFlow: Learn how to build and train models, implement real-time audio recognition, and...

· 12 min read · Feb 27

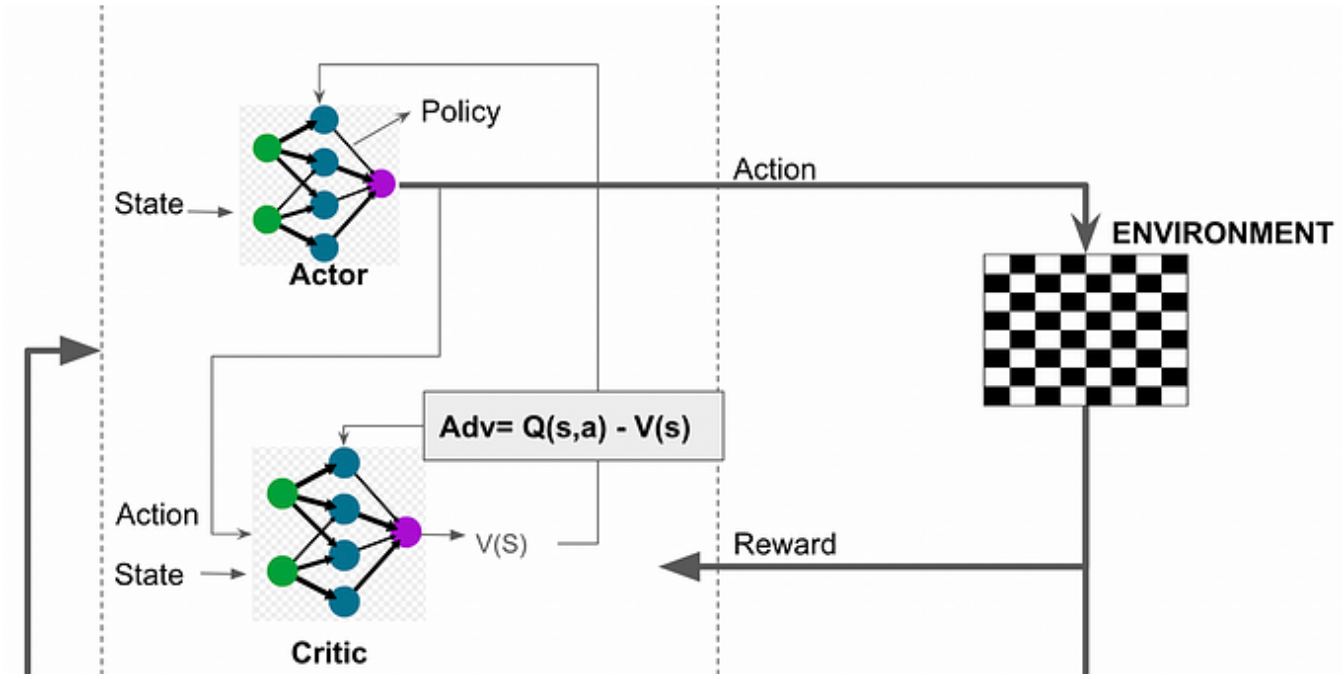
Siwei Causevic in Towards Data Science

# Generalized Advantage Estimation in Reinforcement Learning

Bias and Variance tradeoff in Policy Gradient

· 6 min read · Mar 27






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

**ChatGPT prompts**

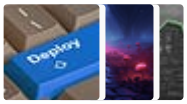
24 stories · 296 saves

**ChatGPT**

21 stories · 124 saves

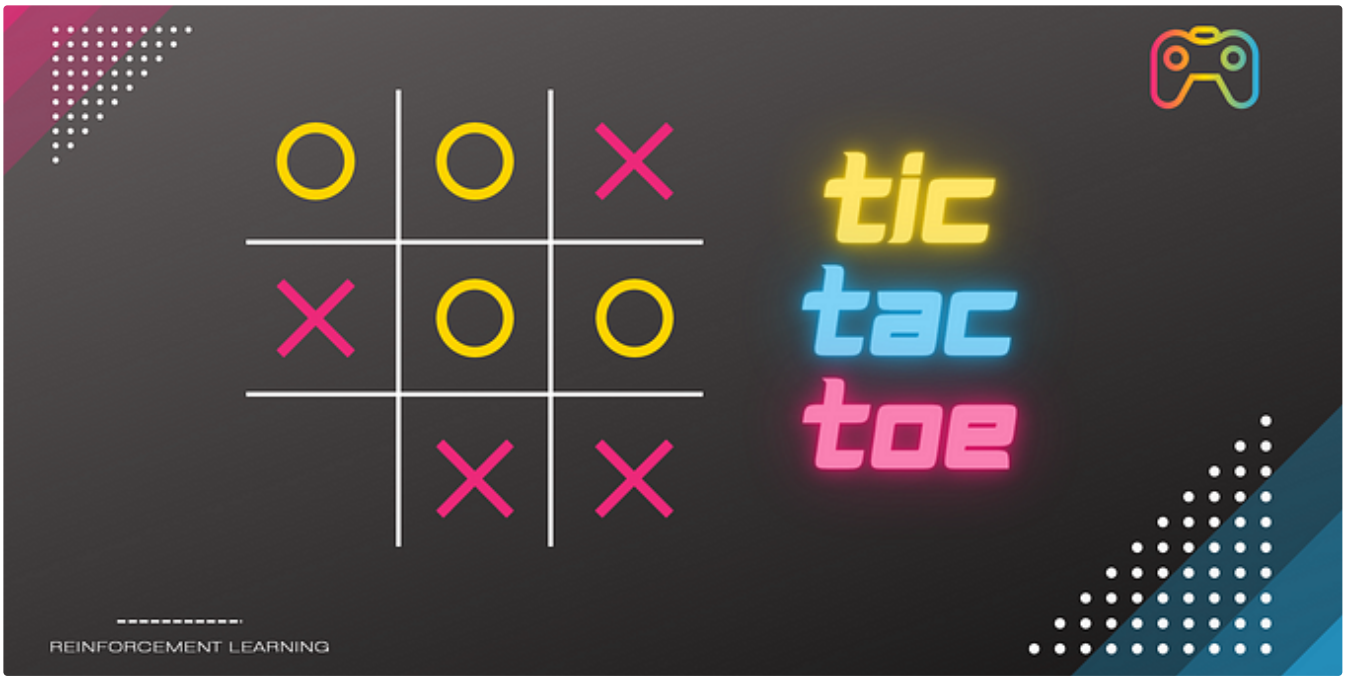
**AI Regulation**

6 stories · 90 saves




**Predictive Modeling w/ Python**

20 stories · 313 saves



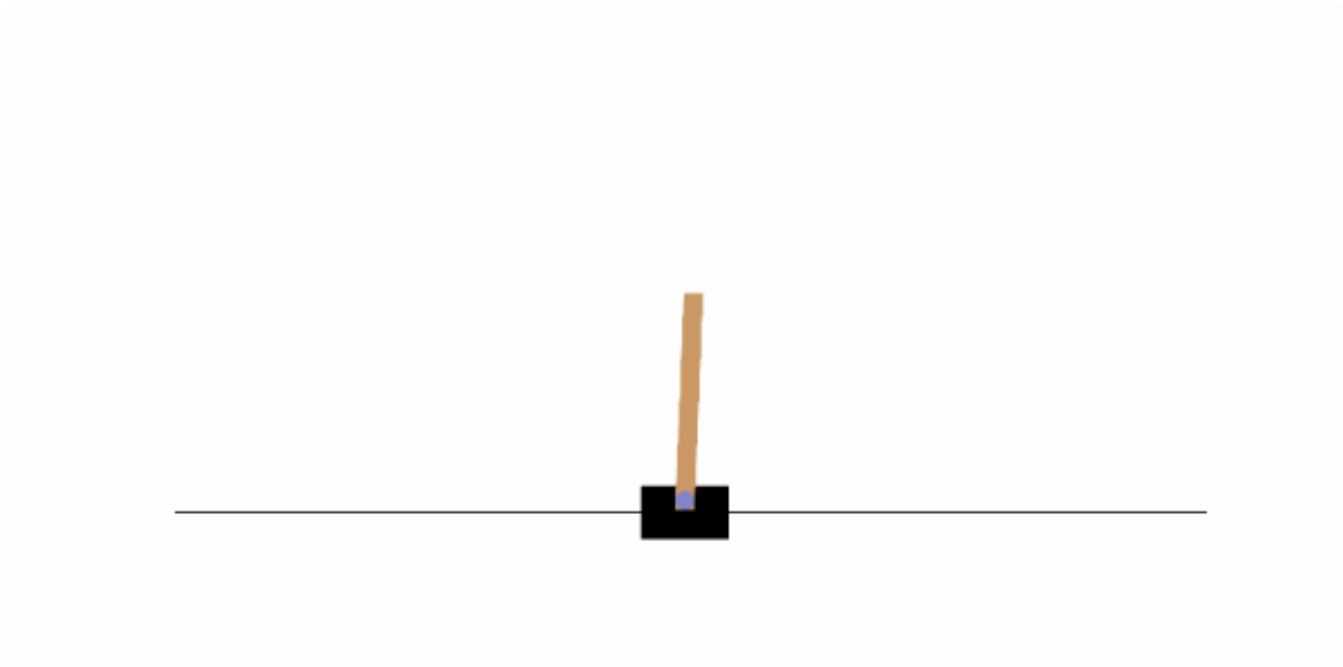

Waleed Mousa in Artificial Intelligence in Plain English

# 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



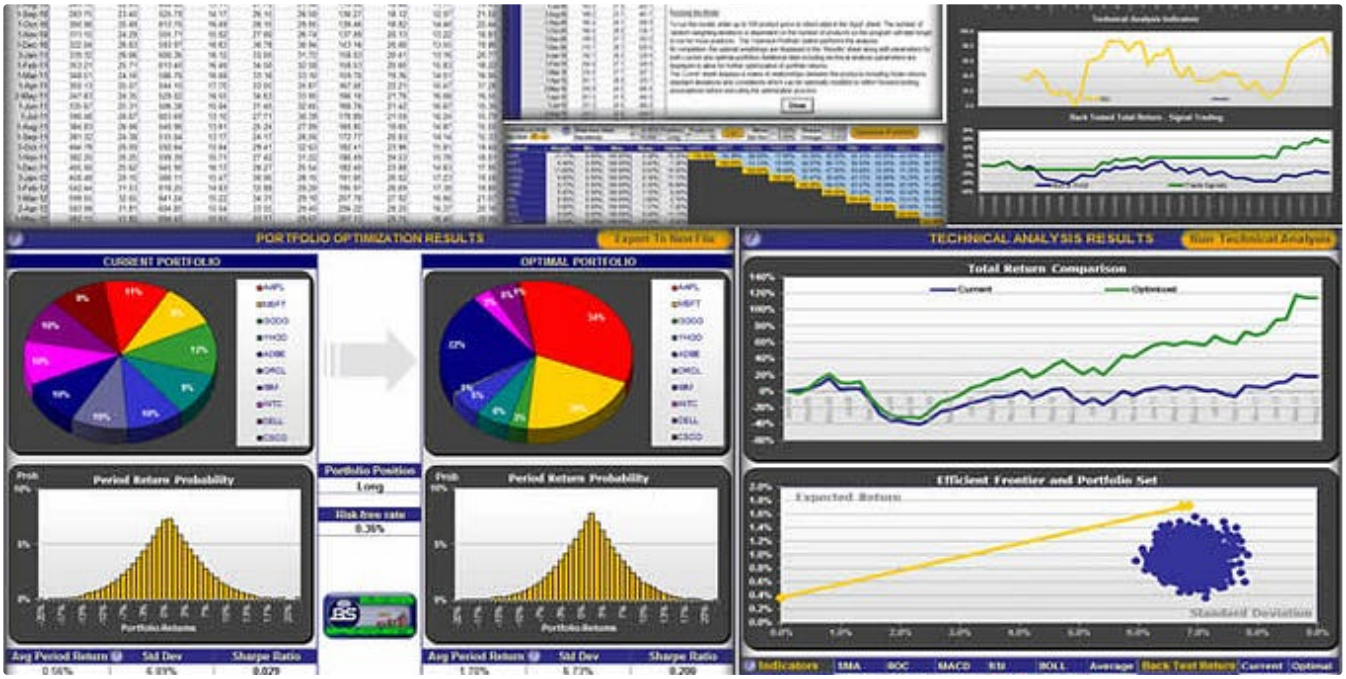
Ludovico Buizza

## Solving the CartPole Environment

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

10 min read · Apr 10

16




Farzana huq

## Applications of machine learning in options trading

The world of options trading is a complex and dynamic domain where investors strive to make informed decisions amidst volatility and...

13 min read · Jun 24

2 1

Michael May

# Deep Learning and Stock Time Series Data

Using Univariate LSTM and CNN-LSTM models to predict stock prices

10 min read · Apr 22

35 3