Advantage Actor-Critic (A2C) algorithm in Reinforcement Learning with Codes and Examples using OpenAI Gym

Combining DQNs and REINFORCE algorithm for training agents

Mehul Gupta · Follow
Published in Data Science in your pocket
8 min read · Apr 14

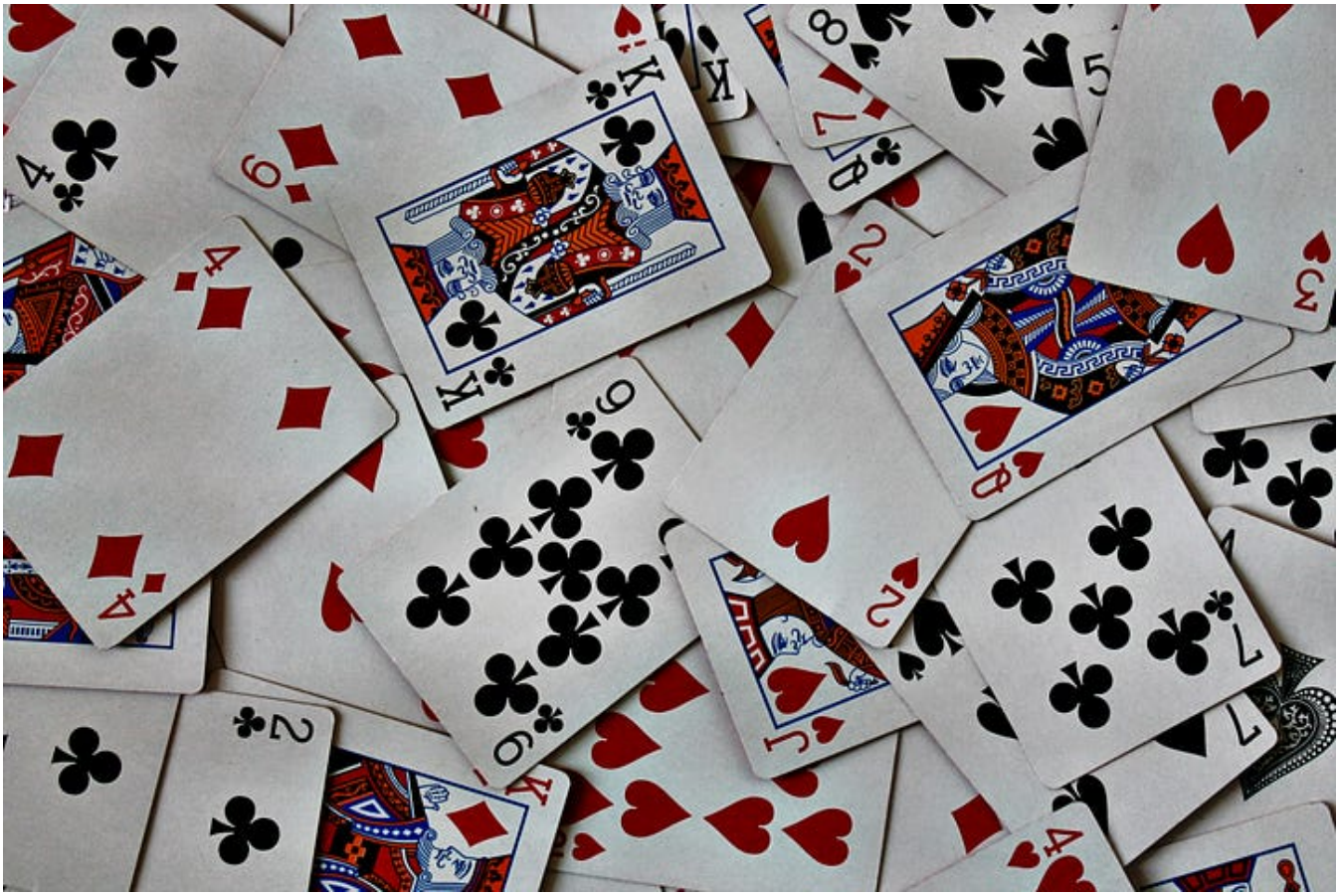


Photo by Amanda Jones on Unsplash

So in my previous posts, we have discussed the following Reinforcement Learning concepts

> *Reinforcement Learning basics*
>
> *Formulating Multi-Armed Bandits (MABs)*
>
> *Monte Carlo with example*
>
> *Temporal Difference learning with SARSA and Q Learning*
>
> *Game dev using reinforcment learning and pygame*
>
> *Contextual bandits with codes*
>
> *Training OpenAI gym envs using REINFORCE algorithm*
>
> *DQNs for training OpenAI gym environments*

Focussing more on the last two discussions, REINFORCE and DQNs, we trained agents using both of these approaches. On focussing closely, there exist a few pitfalls for both the methods

REINFORCE

- The environment must be episodic in nature. What about non-episodic problems? Like Robotics or Trading where there is no end and the agent is engaged continuously

- The agent gets the final reward for every action taken at the end of the episode. If the episode is very long, you may not wish to wait this long to get a reward for action 1 which you will get after taking action 5829!
- Using REINFORCE with environments with sparse rewards (Like MountainCar where every reward is -1 except when you reach the top, you get a 0)

> ***Why?*** *The more frequently we update, the lower the variance will be for the reward. For example: If an episode has 5k+ steps and if we are updating after getting the final reward, if the reward was a fluke, you are going to affect the probability of all the actions in the episodes. Hence, for better training, specially in long episodic environments, it is better to opt incremental training.*

DQNs

- Overheads like Experience Replay and Target & Copy DQNs are required for stable training
- We need to decide over policy apart from the Neural Network which is not the case with REINFORCE where a single Neural Network does the job.
- Also, Experience Replay can be used when the system follows the Markov property of memorylessness i.e. the future state is dependent on the current state and not on any historic state. But we may get situations where this may not get followed and the future is also dependent on historic states.

The question is which is better? Or can we have the best of the two worlds by combining these strategies to form a single algorithm?

Actor-Critic method

The Actor-Critic method does exactly what we wish to have, to take the useful features from both algorithms forming a hybrid that can

- Learn incrementally without waiting for the whole episode to end.
- No Experience Replay is required.
- Stable training

The algorithm that we are going to discuss from the Actor-Critic family is the Advantage Actor-Critic method aka

A2C algorithm

In AC, we would be training two Neural Networks

- Policy Network similar to REINFORCE algorithm
- State-value Network similar to DQN

Hence the name Actor-Critic where Policy Network will act as the main hero and the State-Value Network as the critic. If you have read about GANs, this concept may sound a bit familiar where we have a generator and discriminator involved in an adversarial system.

The loss function for the Policy Gradient algorithm gets updated from

> *loss = -1 x Σlog(probability) x discounted_reward*
>
> *to*
>
> *loss = -1 x Σlog(probability) x (Reward+γV(S`)-V(S))*

And for DQN it remains Mean Squared Error. hence the final loss:

-1 x Σlog(probability)x(Reward+γV(S`)-V(S))+MSE(Actual_V(S),Predicted_V(S))

where the term *(Reward+γV(S`)-V(S))* comes from the State-Value Network which is called as **Advantage** term hence the name Advantage Actor-Critic. If you look closely, this comes from the Bellman equation we used in DQN and Q-Learning. So what we are actually doing is

- Make predictions from both DQN and REINFORCE neural networks.
- Calculate the new loss as it involves terms for both networks
- Backpropagate to both neural networks for weight updation.

One more concept we need to know before jumping onto the code i.e.

N step Learning

Before we jump onto N Step learning, we need to know what is

Online Learning: Updating the agent after every action taken (DQN without Experience Replay)

Monte Carlo: Updating the agent after an episode ends (REINFORCE)

N-step learning is something in between the two where we, as and when the agent takes ‘x’ steps, update the agent where ‘x’ is the threshold we can set up that we will apply in the Actor-Critic method.

The environment

The environment we would training in this time is BlackJack, a card game with the below rules

Blackjack has 2 entities, *a dealer and a player*, with the goal of the game being to obtain a hand with a value as close to 21 as possible without exceeding it. The player

is represented by an agent that makes decisions based on the cards it has been dealt with and the visible card of the dealer. The agent can take one of the following 2 actions:

> ***Hit:*** *The agent requests another card from the dealer to add to its hand.*
>
> ***Stand:*** *The agent is satisfied with its current hand and does not request any more cards.*

The dealer follows a fixed set of rules in the Blackjack environment, where they must hit until they reach a hand value of 17 or more and then stand.

In the Blackjack environment, each card has a point value, with numbered cards having their face value, face cards having a value of 10, and the Ace having a value of either 1 or 11, depending on the agent's current hand value.

The agent's goal is to maximize its expected reward, which is determined by its final hand value compared to the dealer's final hand value. The agent is rewarded with a positive value if it wins, penalized with a negative value if it loses, and receives a neutral reward if the game ends in a draw.

The state is represented by a tuple of 3 elements, (A, B, C) where A=Sum of cards with the player, B=Sum of cards with Dealer, C=Boolean representing whether cards have an Ace

If you haven't played cards ever, this might be a little tricky to understand

Let's get started then

```
import tensorflow as tf
import numpy as np
import gym
import math
from PIL import Image
import pygame, sys
from pygame.locals import *
from tensorflow import keras
from tensorflow.keras.layers import Input, Dense, concatenate
import math
```

2. Initiating the blackjack environment

```
env = gym.make('Blackjack-v1')

input_shape = len(env.observation_space)
num_actions = env.action_space.n
```

3. Designing the Actor-Critic Network

```
# Define input layer
inputs = Input(shape=(input_shape,))

# Define shared hidden layers
hidden1 = Dense(32, activation='relu')(inputs)
hidden2 = Dense(32, activation='relu')(hidden1)

# Define separate output layers
output1 = Dense(num_actions)(hidden2)
output2 = Dense(num_actions, activation='softmax')(hidden2)

model = tf.keras.Model(inputs=inputs, outputs=[output2, output1])

optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
#model = tf.keras.models.load_model('blackjack')
```

As you would have noticed, this network is a Multi-Output Neural Network giving 2 different outputs using the same network, one for the Q-values and the other for action probabilities. We could have taken two separate Neural Networks as well for Critic and Actor respectively. Input shape is state space shape.

4. Declaring a few constants

```
num_episodes = 10000
gamma = tf.cast(tf.constant(0.9),tf.float64)
count = 0
n = 5 #keep n very small
min_loss = math.inf

maxlen = n
states = deque(maxlen=maxlen)
actions = deque(maxlen=maxlen)
rewards = deque(maxlen=maxlen)
```

```
next_states = deque(maxlen=maxlen)
dones = deque(maxlen=maxlen)
```

5. Training loop

```
for episode in range(num_episodes):
    # Reset the environment and get the initial state
    state = list(env.reset())
    state[2] = 1 if state[2] else 0

    # Keep track of the states, actions, and rewards for each step in the episo

    # Run the episode
    while True:
        count=count+1
        # Get the action probabilities from the policy network
        action_probs, _ = model.predict(np.array([state]),verbose=0)

        # Choose an action based on the action probabilities
        action = np.random.choice(num_actions, p=action_probs[0])

        # Take the chosen action and observe the next state and reward
        next_state, reward, done, _ = env.step(action)
        next_state = list(next_state)
        next_state[2] = 1 if next_state[2] else 0
        # Store the current state, action, and reward
        states.append(state)
        actions.append(action)
        rewards.append(reward)
        next_states.append(next_state)
        dones.append(1 if done else 0)

        state = next_state

        # End the episode if the environment is done
        if done:
            done=False
            break

    if count>n:
            count=0
            # Convert the lists of states, actions, and discounted rewards to t
            states_ = tf.convert_to_tensor(states)
            actions_ = tf.convert_to_tensor(actions)
            rewards_ = tf.convert_to_tensor(rewards)
            next_states_ = tf.convert_to_tensor(next_states)
```

```
            with tf.GradientTape(persistent=True) as tape:

                action_probs,q_values1 = model(states_)
                q_value1 = tf.cast(tf.gather(q_values1,actions_,axis=1,batch_d
                _,q_value2 = model(next_states_)
                q_value2 = tf.cast(tf.reduce_max(q_value2),tf.float64)

                target_value = tf.cast(rewards_,tf.float64) + (tf.cast(1,tf.flo
                value_loss = tf.reduce_mean(tf.math.pow(target_value-q_value1,2
                advantage = target_value-q_value1

                action_probs = tf.cast(tf.math.log(tf.gather(action_probs,actio
                loss = -tf.reduce_mean(action_probs * advantage) + value_loss


            grads = tape.gradient(loss, model.trainable_variables)
            optimizer.apply_gradients(zip(grads, model.trainable_variables))


            print('Episode {} done with loss {} !!!!!!'.format(episode, loss))
            if loss<min_loss:
                min_loss = loss
                model.save('blackjack/')
```

- The loop starts off with training for 5000 episodes
- For every episode:

1. Reset environment
2. Simulate the episode by feeding the state to the Actor-Critic model and getting action probabilities (ignore output for q-values)
3. Take action dependent on probabilities and store the current state, action, reward, next state, and done flag (whether the episode ended or not)
4. Set the next state as the current state

When 'N' samples are aggregated, then using GradientTape

- Re-predict action_probabilites and q-values using the model for stored states for the current state tensor
- Get predictions for next-state q-values as well and select the max value

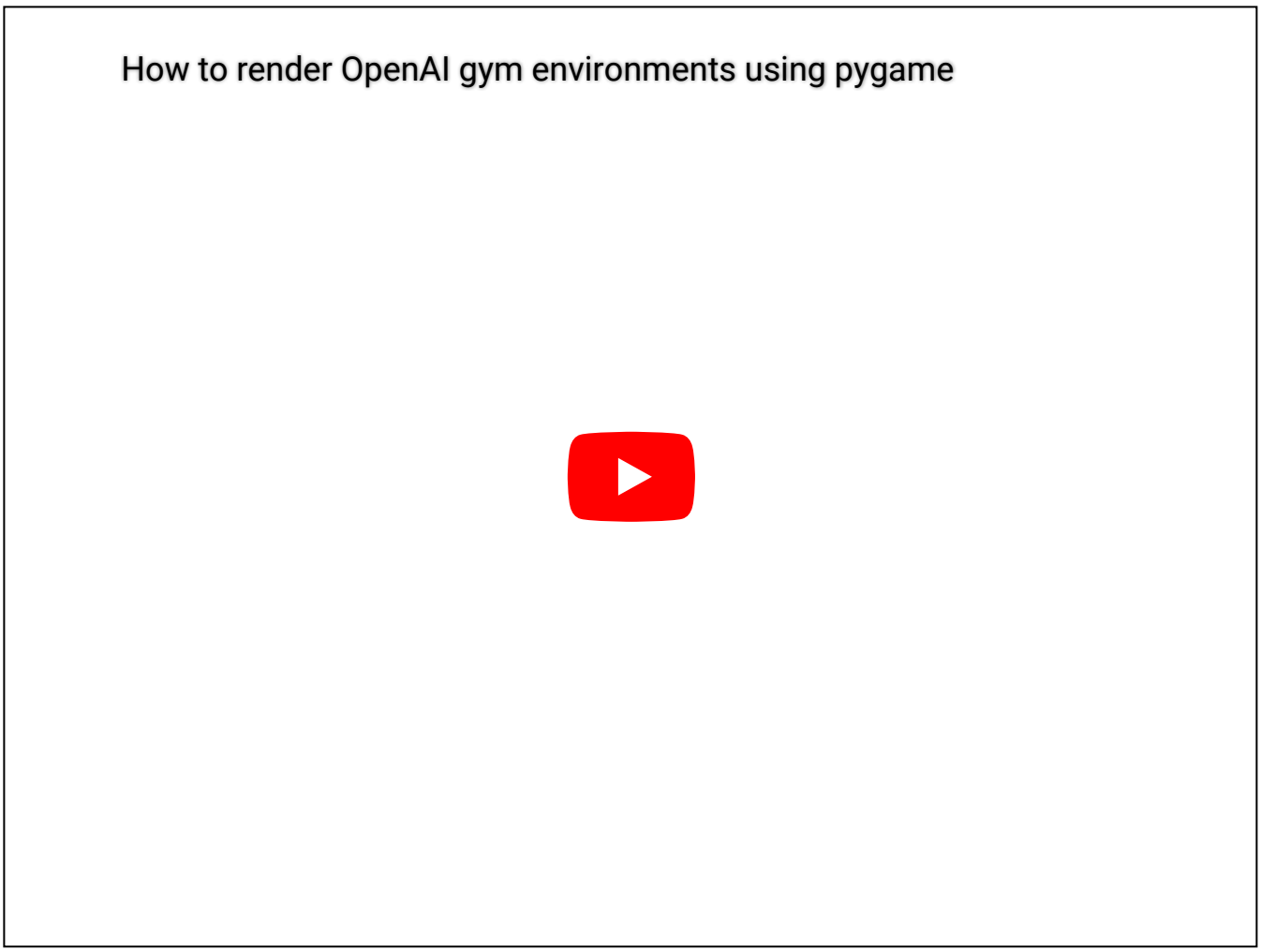
- Calculate target_q_value and Advantage term as discussed
- Calculate loss for both Actor(A Policy Network) and Critic (DQN) and add
- Apply gradients

For the sake of visualization, let's reuse the code we used for REINFORCE and DQN with minor tweaks

```
#pygame essentials
pygame.init()
DISPLAYSURF = pygame.display.set_mode((500,500),0,32)
clock = pygame.time.Clock()
pygame.display.flip()

#openai gym env
env = gym.make('Blackjack-v1')
input_shape = len(env.observation_space)
num_actions = env.action_space.n
state = list(env.reset())
state[2] = 1 if state[2] else 0

done = False
count=0
done=False
steps = 0
#loading trained model
model = tf.keras.models.load_model('blackjack/')
total_wins =0
episodes = 0


def print_summary(text,cood,size):
        font = pygame.font.Font(pygame.font.get_default_font(), size)
        text_surface = font.render(text, True, (0,0,0))
        DISPLAYSURF.blit(text_surface,cood)

while episodes<1000 :
    pygame.event.get()


    for event in pygame.event.get():
                if event.type==QUIT:
                                pygame.quit()
                                raise Exception('training ended')

    action_probs,_ = model.predict(np.array([state]))
    action = np.argmax(action_probs[0])
    next_state, reward, done, info = env.step(action) # take a step in the env
```

```
        image = env.render(mode='rgb_array') # render the environment to the screen

        #convert image to pygame surface object
        image = Image.fromarray(image,'RGB')
        mode,size,data = image.mode,image.size,image.tobytes()
        image = pygame.image.fromstring(data, size, mode)

        DISPLAYSURF.blit(image,(0,0))
        pygame.display.update()
        clock.tick(100)
        if done:
            print_summary('Agent {} game'.format('won' if reward>0 else 'lost'),(50
            if reward>0:
                total_wins+=1

            state = list(env.reset())
            state[2] = 1 if state[2] else 0

            pygame.display.update()
            pygame.time.delay(10)
            episodes+=1
        pygame.time.delay(10)
        next_state=list(next_state)
        next_state[2] = 1 if state[2] else 0
        state = next_state

    pygame.quit()
```

For understanding this code, this video will help



On analyzing, the win % stood at ~38% for the trained agent compared to 26% for an agent with random action. Even ChatGPT believes the win% to be fine so the agent did learn something.

So far, we have covered agents with Discrete action space. What about continuous action space? Stay tuned

Artificial Intelligence Machine Learning Data Science Reinforcement Learning OpenAI

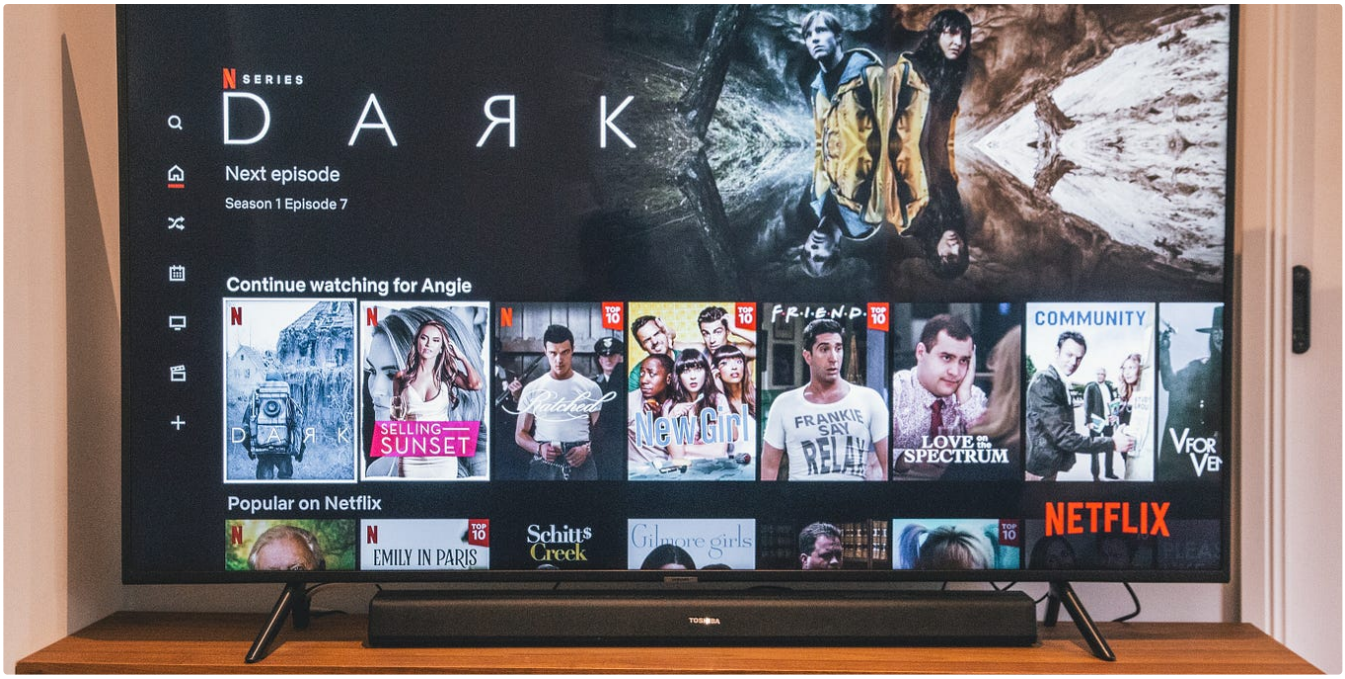
Follow

Written by Mehul Gupta

1.3K Followers · Writer for Data Science in your pocket

Top AI Writer @Medium | Data Scientist @ DBS Bank | Youtube: https://www.youtube.com/channel/UCQoNosQTIxiMTL9C-gvFdjA

More from Mehul Gupta and Data Science in your pocket



Mehul Gupta in Data Science in your pocket

Recommendation Systems using Neural Collaborative Filtering (NCF) explained with codes

Understanding the maths behind NCF

4 min read · Jul 28

56

Mehul Gupta in Data Science in your pocket

Attention is all you need: understanding with example

‘Attention is all you need’ has been amongst the breakthrough papers that have just revolutionized the way research in NLP was progressing...

14 min read · May 4, 2021

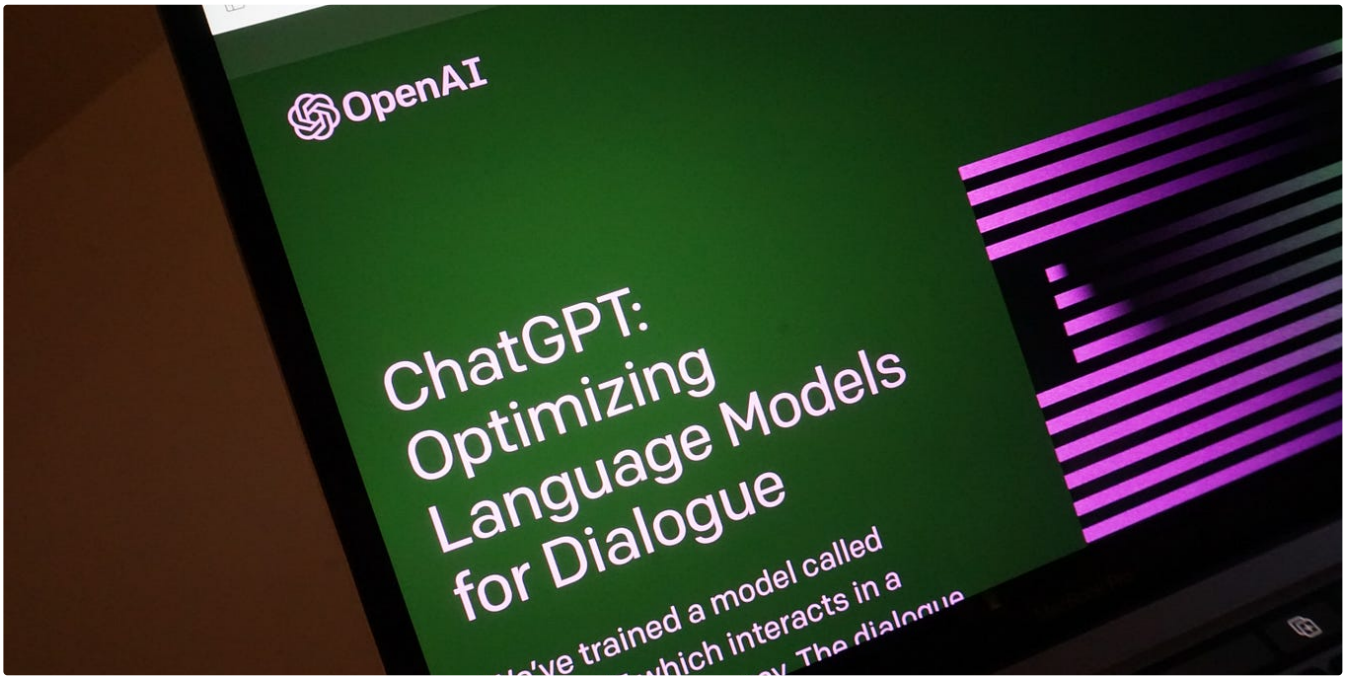
Mehul Gupta in Data Science in your pocket

Auto Regressive Distributed Lag (ARDL) time series forecasting model explained

The mathematical explanation for ARDL with example

5 min read · Jul 12, 2022

140 3



Mehul Gupta in Data Science in your pocket

Langchain tutorials for newbies

Langchain use cases with demo explained

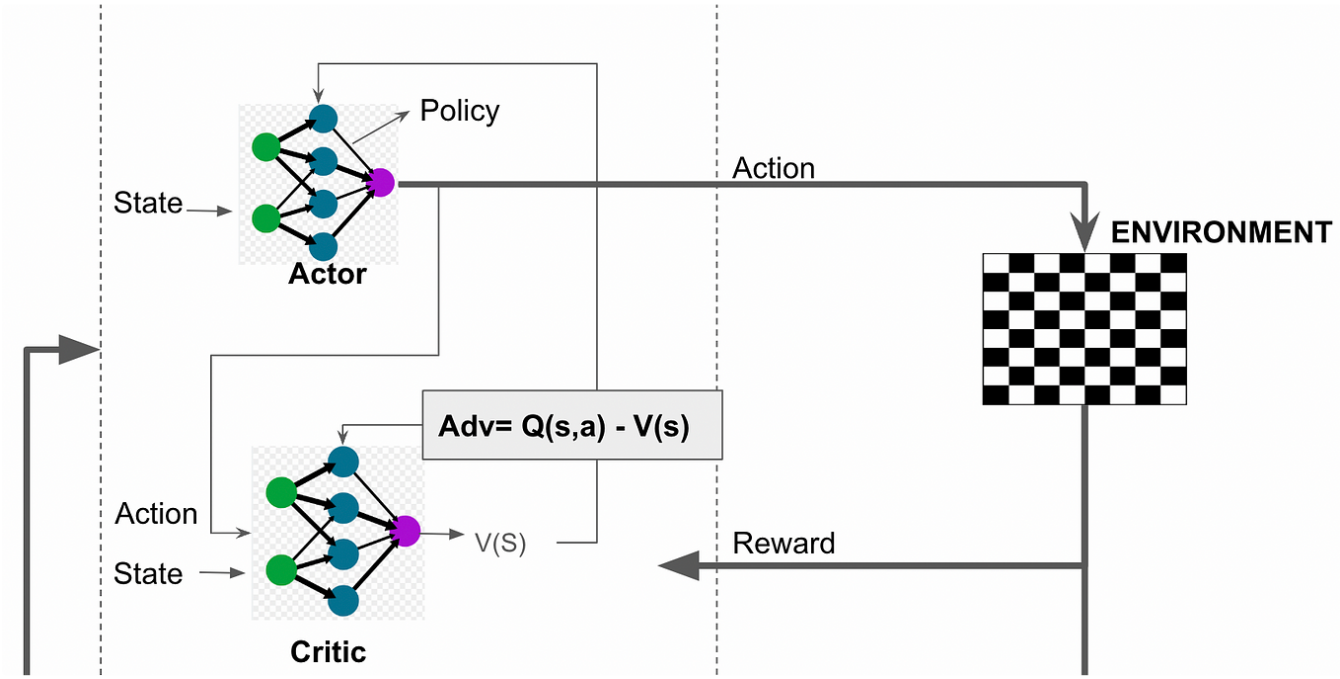
3 min read · Aug 20

29

See all from Mehul Gupta

See all from Data Science in your pocket

Recommended from Medium



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

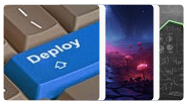
Siwei Causevic in Towards Data Science

Generalized Advantage Estimation in Reinforcement Learning

Bias and Variance tradeoff in Policy Gradient

· 6 min read · Mar 27

Lists



Predictive Modeling w/ Python

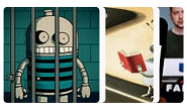
20 stories · 313 saves

Natural Language Processing

551 stories · 174 saves

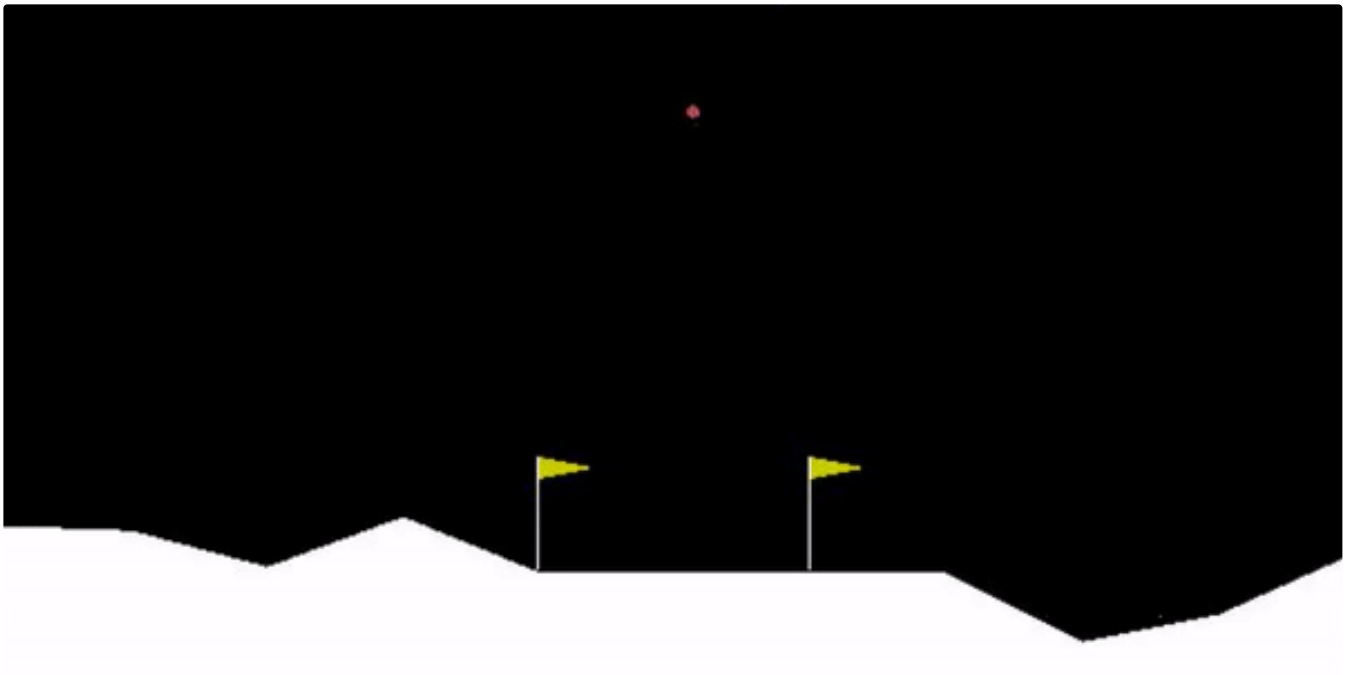
Practical Guides to Machine Learning

10 stories · 344 saves



AI Regulation

6 stories · 90 saves

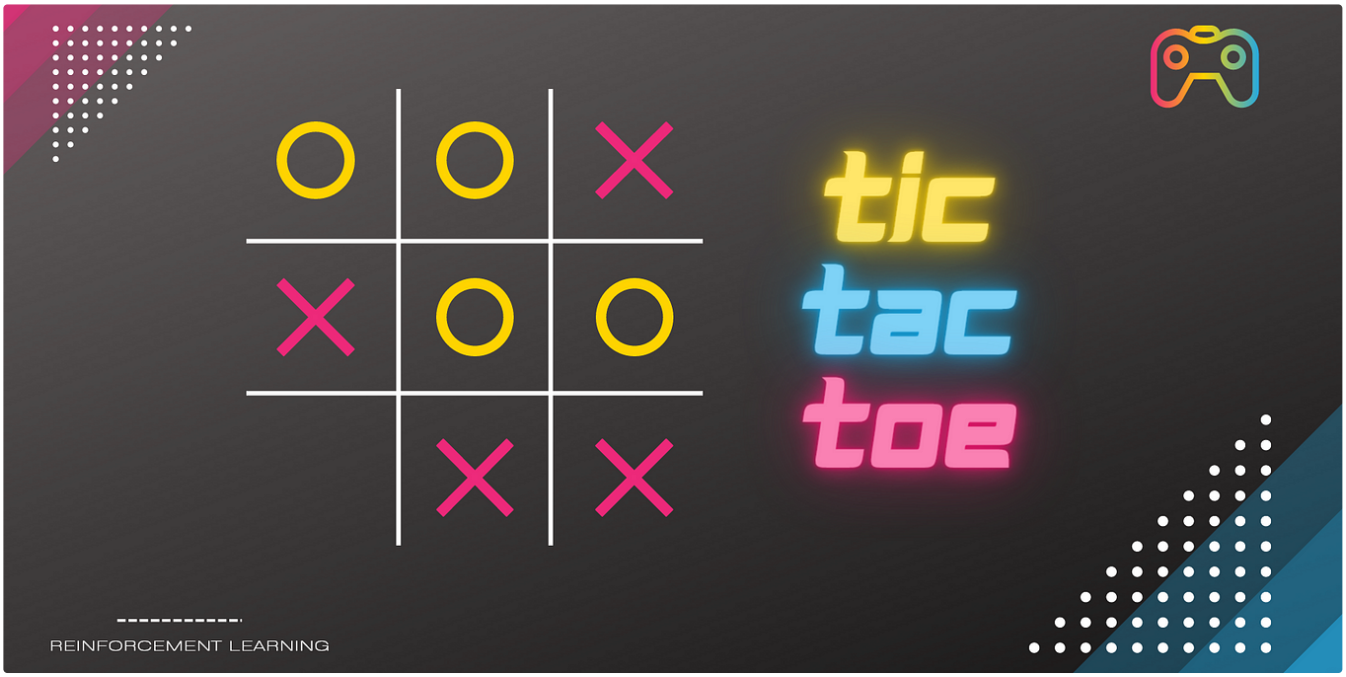


Mehul Gupta in Data Science in your pocket

Training OpenAI gym environments using REINFORCE algorithm in reinforcement learning

Policy gradient methods explained with codes

8 min read · Mar 26



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

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

$$
\begin{aligned}
^{,\lambda)} &:= (1-\lambda)\left(\hat{A}_t^{(1)} + \lambda\hat{A}_t^{(2)} + \lambda^2\hat{A}_t^{(3)} + \dots\right) \\
&= (1-\lambda)\left(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) \right. \\
&= (1-\lambda)(\delta_t^V(1+\lambda+\lambda^2+\dots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\lambda^3+\dots) \\
&\qquad + \gamma^2\delta_{t+2}^V(\lambda^2+\lambda^3+\lambda^4+\dots) + \dots) \\
&= (1-\lambda)\left(\delta_t^V\left(\frac{1}{1-\lambda}\right) + \gamma\delta_{t+1}^V\left(\frac{\lambda}{1-\lambda}\right) + \gamma^2\delta_{t+2}^V\left(\frac{\lambda^2}{1-\lambda}\right)\right. \\
&= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V
\end{aligned}
$$

Z Zhirui Xia

Coding PPO from Scratch with PyTorch (Part 4/4)

Welcome to Part 4 of our series, where we will briefly discuss some of the most common optimization tricks for Proximal Policy Optimization...

8 min read · Aug 16