Reinforcement Learning with PyTorch : A Beginner-Friendly Guide

Hey Learner's! Today we are going to learn about Reinforcement Learning , which is very interesting and easy to understand , Also we will see how to use pytorch with RL.

Before diving deep into coding part let's discuss and understand about the concepts of Rl thoroughly .

How to understand Reinforcement Learning ?

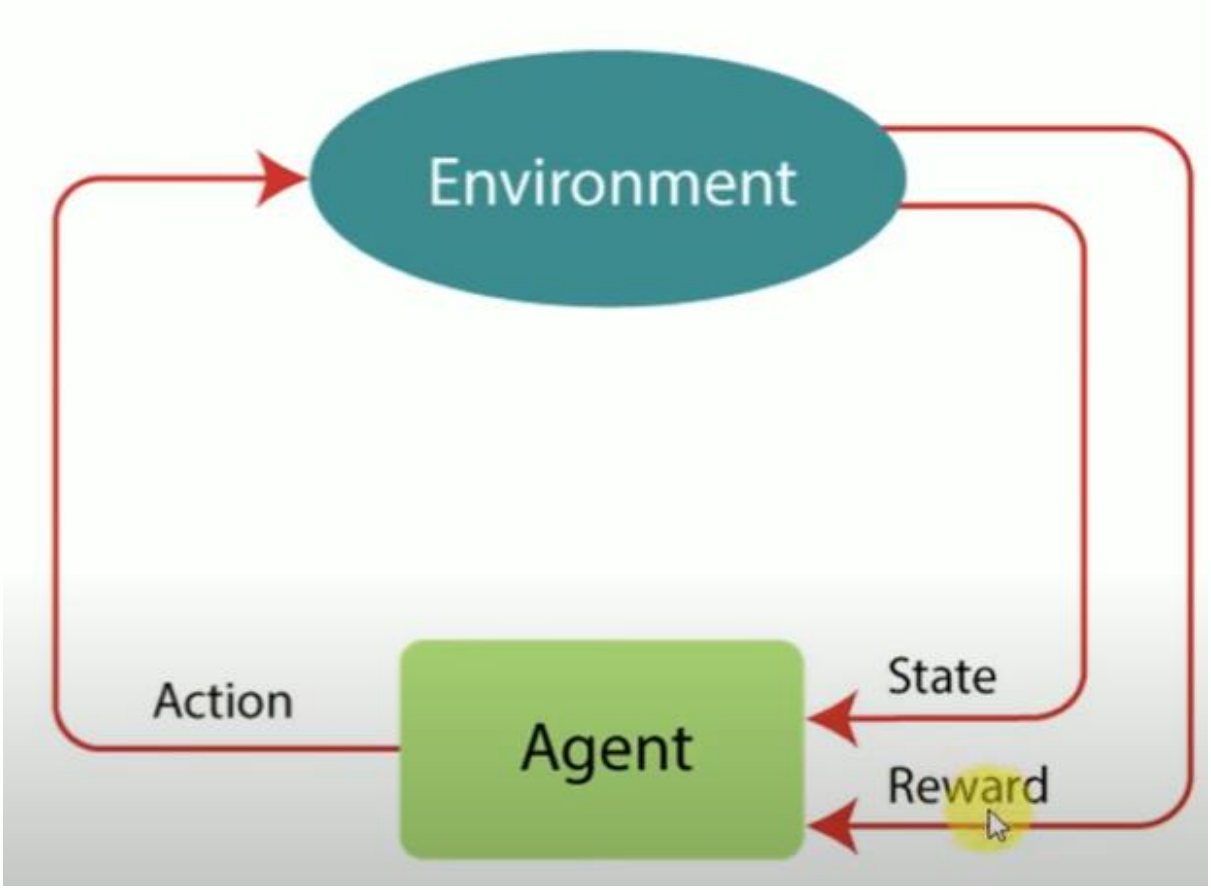
Training a dog come's under Reinforcement Learning

For every right action the Dog is rewarded and for every wrong action nothing is given to the dog , so by this way dog get's understand when it's rewarded the

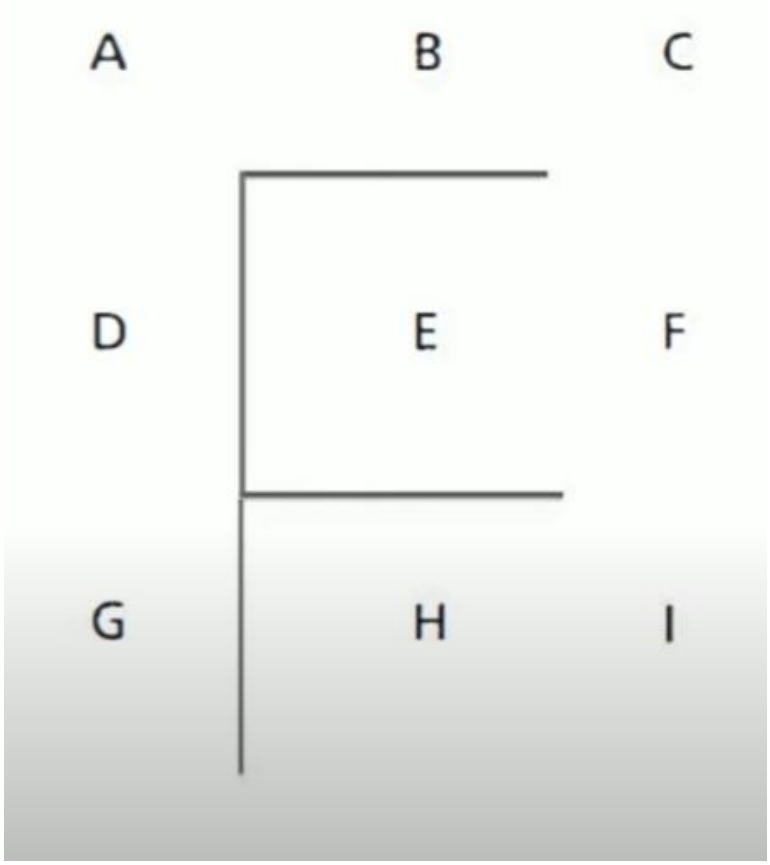
actions of dog is right and when it’s not been rewarded it understands something wrong has done and it will never repeat the false action once again.

Reinforcement Learning is a feedback based machine learning approach in this case the agent will look into the environment based on the environment situation he will perform some actions for each action he will be given a feedback in the form of Rewards. for example let us assume that he has performed some correct action then he will be given a positive feedback if he has performed the negative action he will be given a negative feedback in this case or it's also called as the penalty in this case.

Let us take an example here let us assume that the agent will look into this particular environmental situation based on that he will perform an action here so whenever he performs an action the agent will go from one state to other state for each of this particular action he will be given a reward that is also called as a feedback in this case the reward may be positive for example if he has performed the correct action he will be given positive reward it may be negative also that is whenever he has performed a incorrect action he will be given a negative reward in this case



Let’s try to understand this with an example assume the situation where G is the gold state here and



the agent is present in state E here the goal is the agent has to go from this particular E to G here and he has to follow an optimal path over here let us say that when he is present in this particular State he can take only one action that is the right moment he cannot go up he cannot go left he cannot go down in this case so he has to perform this particular action so when he perform this particular action because it is a correct action he will be given some positive reward here now when he comes to this particular thing he will look into this particular environment and then he will perform one of the action the one action is either he can perform up or he can perform down here whenever he perform up the meaning is again he is going towards this particular G the meaning is it's a correct action for this one also he will be given a positive reward or a positive feedback here but when he performed this action that is from F to I the meaning is this is an incorrect action for this one he will be given a negative feedback so whenever a negative feedback is given he will understand that he should not follow such path in future over here so that is how reinforcement learning works in this case.

In reinforcement learning as I said earlier the agent will interact with this particular environment and then he will identify one of the possible action and then he will perform that particular action over here for each of those particular action he will be given a positive or negative feedback in terms of rewards. The main goal in this case is to get the maximum rewards as as much as possible over here in a reinforcement learning we don't have any label data that's the reason he has to learn through his experience for example if he is getting a positive reward the meaning is it's a correct action if he is getting a negative reward the meaning is it's a incorrect action so through this experience only the agent will learn in this case because we don't have anything called as a label data

Using PyTorch Is A Great Choice With RL

PyTorch is an excellent choice for implementing Reinforcement Learning (RL) because of the following reasons:

1. Dynamic Computation Graphs:

PyTorch allows the creation of dynamic computation graphs, which means you can change the model's behavior on the fly. This flexibility is crucial in RL, where the environment and agent interactions may not always follow a predefined structure.

2. Ease of Debugging:

PyTorch operates more like standard Python, making it easier to debug your RL code. You can use tools like print() or Python debuggers directly without any hassle.

3. Wide Support for Neural Networks:

RL often relies on neural networks for approximating policies or value functions. PyTorch's neural network module (torch.nn) provides prebuilt layers and utilities, simplifying the process of building and training models.

4. Integration with Libraries:PyTorch integrates seamlessly with popular RL libraries such as Stable-Baselines3 and OpenAI Gym, making it easier to set up RL environments and implement algorithms.

5. Efficient GPU Support:

PyTorch has built-in support for GPUs, enabling faster computations. RL algorithms, especially those involving deep learning, benefit significantly from GPU acceleration.

6. Community and Resources:

PyTorch has a large and active community, with abundant tutorials, forums, and documentation. Beginners and experts alike can find support, making it easier to implement RL projects.

7. Freedom for Customization:

Unlike some libraries with rigid APIs, PyTorch offers greater flexibility for customizing RL algorithms. This makes it an ideal choice for experimenting with novel RL techniques or research purposes.

8. Strong Research Adoption:

PyTorch is widely used in academic research, including RL. Many state-of-the-art RL papers and implementations use PyTorch, ensuring access to cutting-edge developments.

By leveraging these strengths, PyTorch provides the perfect blend of flexibility, performance, and usability for reinforcement learning projects.

Implementing Reinforcement Learning using PyTorch

```
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
import matplotlib.pyplot as plt
```

Imports necessary libraries, including gym for the environment, torch for neural network and optimization, numpy for numerical operations, and matplotlib for plotting.

```
episode_rewards = []
```

A list to store the total reward for each episode, used later to visualize the learning curve.

Define Policy Network

The Policy Network in this context is a neural network designed to map states (observations from the environment) to actions. It consists of two linear layers with ReLU activation in between and a final Softmax layer to produce a probability distribution over possible actions. Given a state as input, it outputs the probabilities of taking each action in that state. This probabilistic approach allows for exploration of the action space, as actions are sampled according to their probabilities, enabling the agent to learn which actions are most beneficial. The Policy Network is the agent's "brain," deciding how to act based on its current understanding of the environment, which it improves upon iteratively through training using rewards received from the environment.

```
class PolicyNetwork(nn.Module):
    def __init__(self):
        super(PolicyNetwork, self).__init__()
        self.fc = nn.Sequential(
            nn.Linear(4, 128),
            nn.ReLU(),
            nn.Linear(128, 2),
            nn.Softmax(dim=-1),
        )

    def forward(self, x):
        return self.fc(x)
```

Calculate Discounted Rewards

Calculates the discounted rewards for each time step in an episode, emphasizing the importance of immediate rewards over future rewards.

```
def compute_discounted_rewards(rewards, gamma=0.99):
    discounted_rewards = []
    R = 0
    for r in reversed(rewards):
        R = r + gamma * R
        discounted_rewards.insert(0, R)
    discounted_rewards = torch.tensor(discounted_rewards)
    discounted_rewards = (discounted_rewards - discounted_rewards.mean()) /
(discounted_rewards.std() + 1e-5)
    return discounted_rewards
```

Training Loop

The main function where the environment is interacted with, the policy network is trained using the rewards collected, and the optimizer updates the network's parameters based on the policy gradient.

```
def train(env, policy, optimizer, episodes=1000):
    for episode in range(episodes):
        state = env.reset()
        log_probs = []
        rewards = []
        done = False

        while not done:
            state = torch.FloatTensor(state).unsqueeze(0)
            probs = policy(state)
            m = Categorical(probs)
            action = m.sample()
            state, reward, done, _ = env.step(action.item())

            log_probs.append(m.log_prob(action))
            rewards.append(reward)
            # Inside the train function, after an episode ends:

            if done:
                episode_rewards.append(sum(rewards))
                discounted_rewards = compute_discounted_rewards(rewards)
                policy_loss = []
                for log_prob, Gt in zip(log_probs, discounted_rewards):
                    policy_loss.append(-log_prob * Gt)
                optimizer.zero_grad()
                policy_loss = torch.cat(policy_loss).sum()
                policy_loss.backward()
                optimizer.step()

                if episode % 50 == 0:
                    print(f"Episode {episode}, Total Reward: {sum(rewards)}")
                break

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

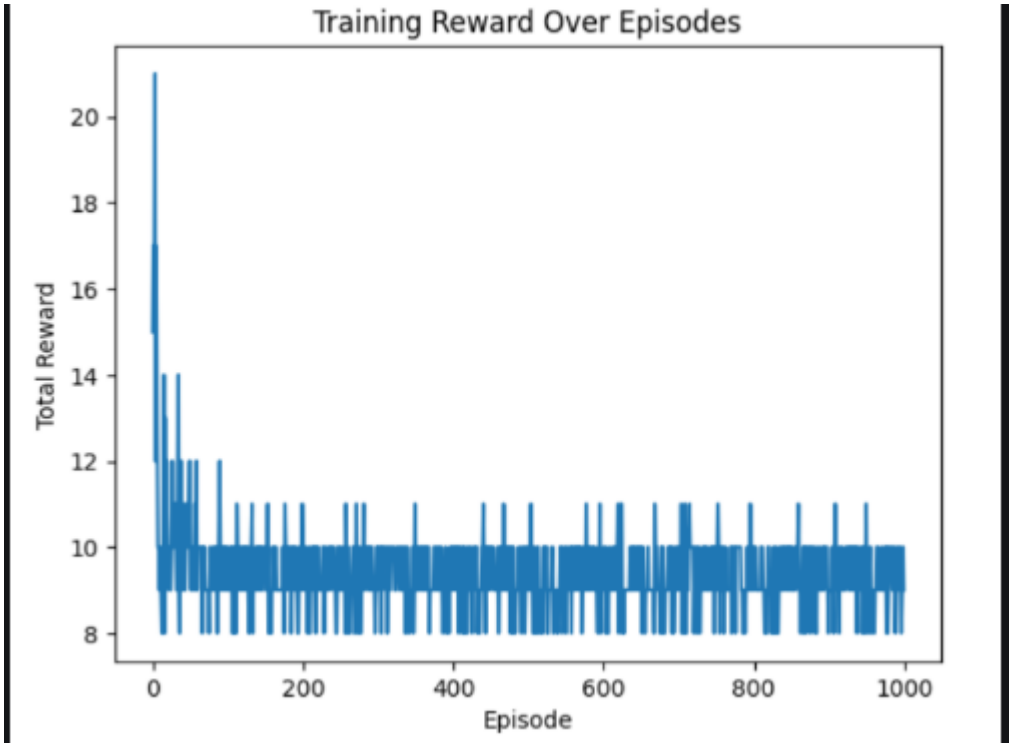
Plotting the Learning Curve

```
plt.plot(episode_rewards)
plt.title('Training Reward Over Episodes')
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```

OUTPUT

```
Episode 0, Total Reward: 15.0
Episode 50, Total Reward: 10.0
Episode 100, Total Reward: 9.0
Episode 150, Total Reward: 10.0
Episode 200, Total Reward: 10.0
Episode 250, Total Reward: 10.0
Episode 300, Total Reward: 10.0
Episode 350, Total Reward: 9.0
Episode 400, Total Reward: 10.0
Episode 450, Total Reward: 10.0
Episode 500, Total Reward: 9.0
Episode 550, Total Reward: 8.0
Episode 600, Total Reward: 10.0
Episode 650, Total Reward: 10.0
Episode 700, Total Reward: 10.0
Episode 750, Total Reward: 9.0
Episode 800, Total Reward: 9.0
Episode 850, Total Reward: 10.0
Episode 900, Total Reward: 9.0
Episode 950, Total Reward: 9.0
```

After training, the total rewards per episode are plotted to visualize the learning progress.



The graph shows the total reward per episode for a reinforcement learning agent across 1,000 episodes. The reward starts high but decreases and stabilizes, indicating the agent may not be improving over time.

Reinforcement Learning vs. Supervised Learning

Reinforcement Learning (RL) and Supervised Learning are two fundamental approaches in Machine Learning, each with distinct purposes and methodologies. Reinforcement Learning focuses on training an agent to make decisions by interacting with an environment and learning through trial and error. The agent receives feedback in the form of rewards or penalties for its actions, aiming to maximize cumulative rewards over time. This process often requires balancing exploration (trying new actions) and exploitation (leveraging known actions for rewards). For instance, RL is widely used in game-playing AI (like AlphaGo), robotics, and dynamic pricing systems. Imagine teaching a dog to perform tricks. You give it rewards (treats) for good behaviour (e.g., sitting) and no reward or a mild penalty for not performing as expected. Over time, the dog learns the tricks that maximize its rewards.

In contrast, Supervised Learning relies on a labeled dataset, where input-output pairs are explicitly provided. The model learns a direct mapping from inputs to outputs by minimizing the error between its predictions and the given labels. Feedback in Supervised Learning is immediate and provided for each example during training. This makes it highly effective for tasks like image classification, spam detection, and sentiment analysis, where the dataset provides clear guidance. Imagine teaching a student with a set of flashcards. Each flashcard has a question (input) and the correct answer (label). The student learns by studying these examples until they can predict the correct answers consistently.

The key difference lies in the feedback mechanism: RL learns from delayed rewards after interacting with an environment, while Supervised Learning uses immediate feedback from labeled data. Moreover, RL often involves a trial-and-error process that is computationally intensive and environment-dependent, whereas Supervised Learning is typically more straightforward to implement with well-prepared datasets. Together, these approaches address different problem domains, showcasing the versatility of machine learning in solving complex challenges.

Reinforcement Learning (RL), combined with the flexibility and power of PyTorch, enables solutions for complex, dynamic problems across various fields, including robotics and game-playing AI. By utilizing PyTorch's dynamic computation graphs, straightforward debugging, and GPU acceleration, developers can efficiently implement RL algorithms tailored to real-world challenges. As RL technology continues to progress, mastering tools like PyTorch will be essential for driving innovation in artificial intelligence. Whether you are a beginner or an experienced developer, now is the ideal time to explore the exciting world of Reinforcement Learning with PyTorch and start building intelligent agents that can learn and adapt to their environments.

Note To Developer’s

When using PyTorch for Reinforcement Learning (RL), it’s vital to understand key concepts like rewards, states, actions, and the exploration-exploitation trade-off. PyTorch’s dynamic computation graphs allow for flexible implementation and

experimentation with RL algorithms, facilitating easier debugging and modifications.

Using prebuilt libraries like Stable-Baselines3 can save time on standard RL tasks. However, challenges such as unstable training and high variance in policy gradients might arise. Techniques like reward normalization, experience replay, and gradient clipping can help improve stability. To ensure efficient training, utilize environments like OpenAI Gym or Unity ML-Agents, and prioritize proper state representation and reward shaping. Employ suitable exploration strategies, such as epsilon-greedy or noise injection, and leverage GPU acceleration for faster training.

Regularly monitor progress with tools like Matplotlib or TensorBoard to track trends and policy behaviors. Tuning hyperparameters, including the learning rate and discount factor, is crucial for optimal RL performance.