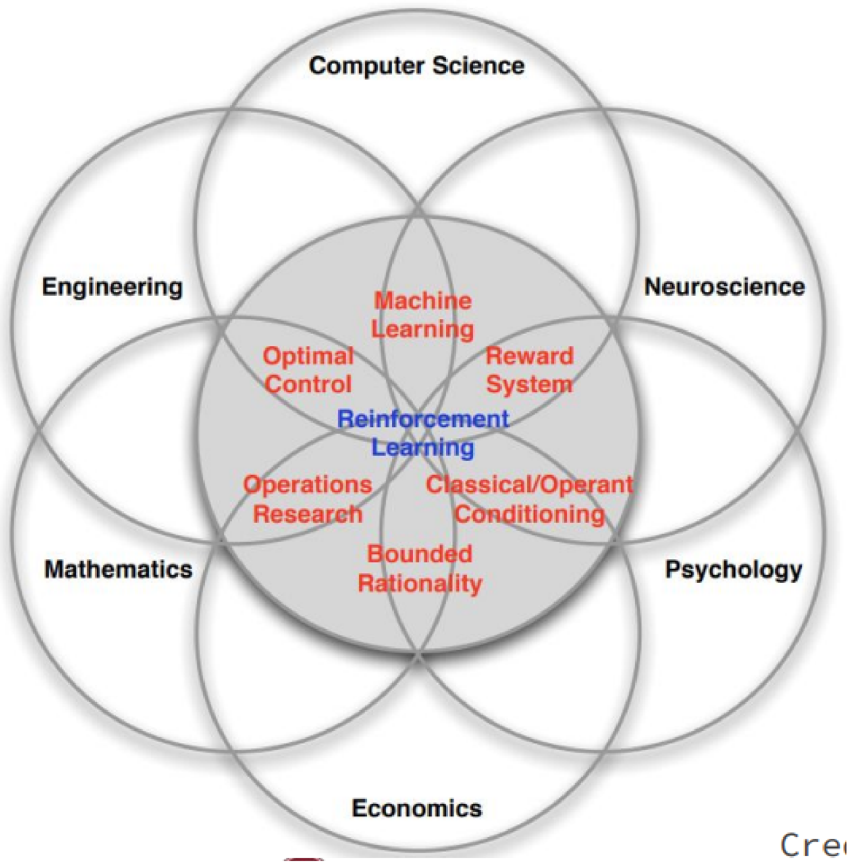


Reinforcement Learning

Assignment 3

Yusupha Juwara
juwara.1936515@studenti.uniroma1.it

December 14, 2023

Contents

Chapter 1

Theory Part

Question 1.1: Suppose you have an environment with 2 possible actions and a 2-d state representation ($x(s) \in R^2$). Consider the 1-step Actor-Critic Algorithm with the following policy and action-state value function approximators:

$$\pi_\theta(a=1|s) = \sigma(\theta^T x(s)) = \frac{1}{1+e^{-(\theta^T x(s))}}$$

$$Q_w(s, a=0) = w_0^T x(s)$$

$$Q_w(s, a=1) = w_1^T x(s)$$

Given

$$w_0 = (0.8, 1)^T.$$

$$w_1 = (0.4, 0)^T$$

$$\theta_0 = (1, 0.5)^T.$$

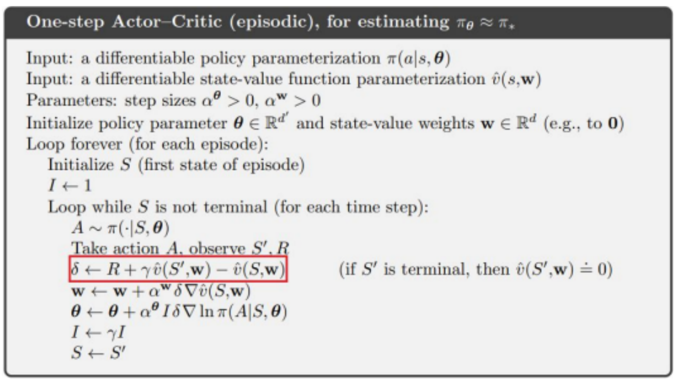
$$\alpha_w = \alpha_\theta = \alpha = 0.1$$

$$\gamma = 0.9$$

and the following transition:

$$x(s_0) = (1,0)^T, a_0 = 0, r_1 = 0, x(s_1) = (0,1)^T, a_1 = 1$$

Compute new values of w_0, w_1 and θ after the transition.



```
One-step Actor–Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
        Take action $A$, observe $S'$, $R$
        $\boxed{\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})}$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

Figure 1.1: 1-step Actor-Critic algorithm using state-value function. Source: prof slides

Answer 1.1: From the algorithm in Figure 1.1, it uses the value function. But we know that the value function is the expectation over the policy of action-value function.[2] That is,

$$V(s) = \mathbb{E}_{a\sim\pi(.|s)}Q(s,a)$$

However, as in this exercise, we already have all the ingredients ready to mimic the algorithm. We already have the states and the actions chosen. So, we can use the action-value functions directly into the update rules.

Thus, we have:

$$Q_w(s_0,a_0) = w_0^T x(s_0) = \begin{bmatrix}0.8\\1\end{bmatrix}^T \begin{bmatrix}1\\0\end{bmatrix} = 0.8$$

$$Q_w(s_1,a_1) = w_1^T x(s_1) = \begin{bmatrix}0.4\\0\end{bmatrix}^T \begin{bmatrix}0\\1\end{bmatrix} = 0$$

$$\text{Let: } \begin{cases} \textbf{target} = \overbrace{r}^{0} + \overbrace{\gamma}^{0.9} \overbrace{Q_w(s_1,a_1)}^{0} = 0 + 0.9 \times 0 = 0 \\ \delta = \textbf{error} = \overbrace{\textbf{target}}^{0} - \overbrace{Q_w(s_0,a_0)}^{0.8} = 0 - 0.8 = -0.8 \\ \textbf{grad} = \nabla_\theta \log \pi_\theta(a_0|s_0) = \nabla_\theta \log(1 - \pi_\theta(a_1|s_0)) \\ = \nabla_\theta \log\left(1 - \frac{1}{1+e^{-(\theta^T x(s_0))}}\right) = \nabla_\theta \log\left(\frac{e^{-(\theta^T x(s_0))}}{1+e^{-(\theta^T x(s_0))}}\right) \\ \therefore\ \textbf{grad} = -\dfrac{x(s_0)}{\mathrm{e}^{-\theta^T x(s_0)} + 1} \text{ how derived? see later!} \\ = -\dfrac{\begin{bmatrix}1\\0\end{bmatrix}}{e^{-\begin{bmatrix}1\\0.5\end{bmatrix}^T \begin{bmatrix}1\\0\end{bmatrix}}+1} = -\dfrac{\begin{bmatrix}1\\0\end{bmatrix}}{e^{-1}+1} = -\dfrac{\begin{bmatrix}e\\0\end{bmatrix}}{e+1} \\ \therefore\ \textbf{grad} = \begin{bmatrix}-\frac{e}{e+1}\\0\end{bmatrix} \end{cases}$$

$$w_0 = w_0 + \alpha \times \delta \times \nabla_{w_0} w_0^T x(s_0) = w_0 + \alpha \times \delta \times x(s_0)$$

$$= \begin{bmatrix}0.8\\1\end{bmatrix} + \begin{bmatrix}-0.08\\0\end{bmatrix} = \begin{bmatrix}0.72\\1\end{bmatrix}$$

$$w_1 = w_1 + \alpha \times \delta \times \overbrace{\nabla_{w_1} w_0^T x(s_0)}^{0} = w_1 + \alpha \times \delta \times [0,0]^T$$

$$= \begin{bmatrix}0.4\\0\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}0.4\\0\end{bmatrix}$$

$$\theta = \theta + \alpha \times \overbrace{\gamma^{t=0}}^{1} \times \delta \times \textbf{ grad } = \begin{bmatrix}1\\0.5\end{bmatrix} + 0.1 \times -0.8 \begin{bmatrix}-\frac{e}{e+1}\\0\end{bmatrix}$$

$$= \begin{bmatrix}1\\0.5\end{bmatrix} + \begin{bmatrix}\frac{0.08\times e}{e+1}\\0\end{bmatrix} = \begin{bmatrix}1 + \frac{0.08\times e}{1+e}\\0.5\end{bmatrix} = \begin{bmatrix}1.058\\0.5\end{bmatrix}$$

If in case you are also interested in how I derived the derivative of the grad already defined above, below illustrates its derivation. This is a simple derivation using the **Quotient rule**.

$$\frac{d}{d\theta}\log\left[\frac{e^{-\theta^T x(s_0)}}{e^{-\theta^T x(s_0)}+1}\right]$$
$$=\frac{d}{d\theta}\log(e^{-\theta^T x(s_0)})-\frac{d}{d\theta}\log(e^{-\theta^T x(s_0)}+1)$$
$$=\frac{-x(s_0)e^{-\theta x(s_0)}}{e^{-\theta x(s_0)}}-\frac{-x(s_0)e^{-\theta x(s_0)}}{1+e^{-\theta x(s_0)}}$$

Simplifying gives:

$$\therefore\ \textbf{grad} \ = -\frac{x(s_0)}{\mathrm{e}^{-\theta^T x(s_0)}+1}$$

Chapter 2

Code Report Assignment 2

2.1 Introduction

Proximal Policy Optimization (PPO) stands out as one of the most popular RL algorithms. As a policy gradient algorithm, PPO directly optimizes the policy to maximize the expected cumulative reward. Its versatility is evident in successful applications across domains like robotic control, game playing, etc.

In this report, I will be writing about PPO applied to Gymnasium's Car Racing environment.

2.2 Car Racing Environment Overview

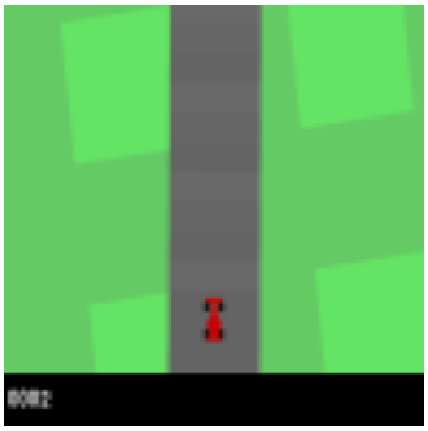


Figure 2.1: Car racing version 2

A brief overview of the CarRacing-V2 environment as taken from their website:

The easiest control task to learn from pixels – a top-down racing environment. The generated track is random every episode.

Some indicators are shown at the bottom of each frame window along with the state RGB buffer. From left to right: true speed, four ABS sensors, steering wheel position, and gyroscope. To play yourself.

This latter paragraph suggests that we can crop the frames to remove the **indicators** and, thus, have a frame size of 84×84.

2.2.1 Action Space

The action space is either continuous or discrete, representing the control inputs for the car. The agent can control the car's acceleration, steering, and braking.

```
Box([-1. 0. 0.], 1.0, (3,), float32)
```

If continuous there are 3 actions:

1. **steering**, -1 is full left, $+1$ is full right
2. **gas**
3. **breaking**

If discrete there are 5 actions:

1. **do nothing**
2. **steer left**
3. **steer right**
4. **gas**
5. **brake**

2.2.2 Observation Space

```
Box(0, 255, (96, 96, 3), uint8)
```

- **Image:** A top-down 96x96 RGB image of the car and race track.

2.2.3 Transition Dynamics

The CarRacing environment is governed by realistic car physics and dynamics. The transition dynamics involve the car's acceleration, steering, and braking inputs, influencing its position and velocity on the track.

2.2.4 Reward

The objective is to drive the car skillfully around the track. The agent receives positive rewards for covering distance and completing laps. Going off-track result in negative rewards. The goal is to maximize the cumulative reward over time.

The reward is -0.1 every frame and $\frac{+1000}{N}$ for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is $1000 - 0.1 \times 732 = 926.8$ points.[1]

2.2.5 Starting State

The car starts at rest in the center of the road.[1]

2.2.6 Episode Termination

The episode finishes when all the tiles are visited. The car can also go outside the playfield – that is, far off the track, in which case it will receive -100 reward and die.[1]

This is the unscaled version of the architecture, where the input is not cropped; similar to the architecture when the input is cropped, except that sizes differ.

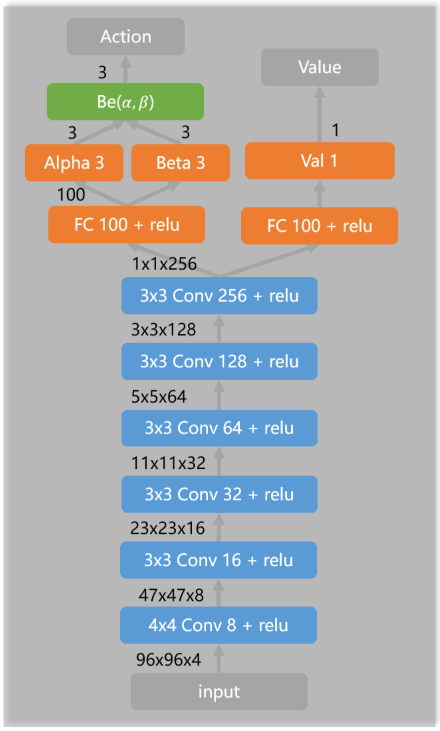


Figure 2.2: Network Architecture

2.3 Data Preprocessing

- As mentioned earlier in the environment overview section, each frame has some indicators that can be cropped out without any performance hit, and this, in turn, reduces the input size from 96 to 84×84. I have devised two different architectures, where one of them uses the scaled version if scale=True and the other the original, full version.
- Secondly, the first 50 frames of each episode is literally just zooming into the initial position of the car. So, we can skip that many frames from the beginning of each episode.
- Furthermore, I gray scaled the frames to remove the color information.
- How to handle temporal information? That is, only a single frame provides a spatial information but not a temporal one. To handle that, I stack 4 frames, just like in the **Atari** games.

```
class Env(gym.Wrapper):
    ...
    @staticmethod
    def rgb2gray(img, scale=scale):
        if scale:
            img = img[:84, 6:90] #
            ↪ CarRacing-v2-specific cropping

        gray = cv2.cvtColor(img,
        ↪ cv2.COLOR_RGB2GRAY) / 255.0
        return gray
    ...
```

2.4 Replay Buffer

- This stores the transitions upto "buffer_capacity" size.
- Once the capacity is full, then the agent learns by updating its weights. Thus, the online learning occurs at every "buffer_capacity" step.
- Furthermore, I find it logical to empty the buffer every time a new episode starts.
- However, if the episode ends before the new buffer is full of transitions, I still update the agent with whatever the buffer has before emptying the it for a new episode.

By transition, I mean a data type of the form:

```
transition = np.dtype(
    [
    ('s', np.float32, (img_stack, 96, 96)),
    ('s_', np.float32, (img_stack, 96, 96)),
    ('a', np.float32, (3,)),
    ('a_logp', np.float32, (3,)),
    ('r', np.float32),
    ('term_or_trunc', np.bool_),
    ('term', np.bool_)
    ])

self.buffer = np.empty(self.buffer_capacity,
↪ dtype=transition)
```

2.4.1 Proximal Policy Optimization (PPO) Overview

Proximal Policy Optimization (PPO) is an on-policy reinforcement learning algorithm that learns from its own actions and experiences. PPO maximizes a "surrogate" objective to approximate the expected return. This approach enhances stability and reduces the risk of overfitting.

The core of PPO is built on policy gradients, utilizing the gradient of the policy with respect to the expected return for policy updates. To prevent divergence, PPO introduces a trust region constraint, limiting the magnitude of policy changes in each update. This constraint contributes to the stability and efficiency of the PPO algorithm.

2.4.2 Policy Gradient Methods

Policy gradient methods involve computing an estimator of the policy gradient and applying it in a stochastic gradient ascent algorithm. The commonly used gradient estimator is given by:

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

where π_θ is a stochastic policy, and $\hat{A}_t$ is an estimator of the advantage function at timestep t. the estimator $\hat{g}$ is obtained by differentiating the objective:

$$L_{PG}(\theta) = \mathbb{E}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

While it is appealing to perform multiple steps of optimization on this loss L^{PG} using the same trajectory, doing so is not well-justified, and empirically it often leads to destructively large policy updates.[3]

2.4.3 Trust Region Methods

In Trust Region Policy Optimization (TRPO), an objective function, the "surrogate" objective, is maximized subject to a constraint on the policy update size:

$$\max_\theta \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \hat{A}_t \right]$$

subject to

$$\mathbb{E}_t \text{KL}[\pi_{\text{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \leq \delta$$

.

TRPO uses a hard constraint due to challenges in choosing a fixed penalty coefficient that performs well across different problems.

2.4.4 Clipped Surrogate Objective

Let $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$, where $r(\theta_{\text{old}}) = 1$. The surrogate objective is:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right]$$

where CPI refers to conservative policy iteration.

This objective penalizes changes to the policy that move $r_t(\theta)$ away from 1, enhancing stability.

2.4.5 Adaptive KL Penalty Coefficient

An alternative to the clipped surrogate objective is using a penalty on KL divergence and adapting the penalty coefficient to achieve a target KL divergence d_{targ} each policy update. The algorithm performs the following steps:

- Optimize the KL-penalized objective: $L_{KLPEN}(\theta) = \mathbb{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \hat{A}_t - \beta \text{KL}[\pi_{\text{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$
- Compute $d = \mathbb{E}_t \left[\text{KL}[\pi_{\text{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right]$
- If $d < \frac{d_{\text{targ}}}{1.5}$, $\beta \leftarrow \frac{\beta}{2}$
- If $d > d_{\text{targ}} \times 1.5$, $\beta \leftarrow \beta \times 2$

The updated β is used for the next policy update.

2.5 PPO Algorithm

For the algorithm section, I will quote from the paper:

> If using a neural network architecture that shares parameters between the policy and value function, a loss function that combines the policy surrogate and a value function error term must be used. This objective can be further augmented by adding an entropy bonus to ensure sufficient exploration.[3]

Combining these terms, we obtain the following objective, which is (approximately) maximized each iteration:

$$L^{\text{CLIP + VF + S}}(\theta) = \mathbb{E}_t \left[L^{\text{CLIP}}(\theta)_t - c_1 L^{\text{VF}}(\theta)_t + c_2 S[\pi_\theta](s_t) \right]$$

where c_1, c_2 are coefficients, S denotes an entropy bonus, and L_{VF} is a squared-error loss $(V_\theta(s_t) - V^t_{\text{targ}})^2$.

In the code, since I share the feature extraction layers between the policy and the value networks, but heed to the recommendation of **Sergey Levine** of UC Berkeley, I update the value network differently with a different learning rate from the policy network. He said that for network stability, it is better to have a different network for the actor and the critic.[2]. Doing that, the value and policy does not have the same loss anymore! So, the value loss is just as specified in the paper, while the policy loss is the combination of the L^{CLIP} and the entropy term to encourage exploration.

```python
adv_index = adv[index]
surr1 = ratio * adv_index
surr2 = torch.clamp(ratio, 1 - self.clip_rate, 1
↪   + self.clip_rate) * adv_index
policy_loss = torch.min(surr1, surr2)
policy_loss = -(policy_loss +
↪   entropy_term).mean() # gradient ascent, not
↪   descent

self.policy_optimizer.zero_grad()
policy_loss.backward()
self.policy_optimizer.step()

value_loss = (self.net(s[index])[1] -
↪   td_target[index]).pow(2)
value_loss = (c1*value_loss).mean() # gradient
↪   ascent, not descent

self.value_optimizer.zero_grad()
value_loss.backward()
self.value_optimizer.step()
```

Run the policy for T timesteps (where T is much less than the episode length) and use the collected samples for an update. This style requires an advantage estimator that does not look beyond timestep T.[3]

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \ldots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T)$$

where t specifies the time index in $[0, T]$ within a given length-T trajectory segment. Generalizing this choice, a truncated version of generalized advantage estimation is used, which reduces to Equation (10) when $\lambda = 1$:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$.

The PPO algorithm follows an iterative process with the following steps:

1. **Collect experience:** The agent interacts with the environment, collecting experience tuples.

```python
ef store(self, transition):
    return self.buffer.store(transition)

...
buffer = self.buffer.get_transitions()
s = torch.tensor(buffer['s'].copy(), dtype=torch.float).to(device)
a = torch.tensor(buffer['a'].copy(), dtype=torch.float).to(device)
r = torch.tensor(buffer['r'].copy(), dtype=torch.float).to(device).view(-1, 1)
s_ = torch.tensor(buffer['s_'].copy(), dtype=torch.float).to(device)
old_a_logp = torch.tensor(buffer['a_logp'].copy(), dtype=torch.float).to(device).view(-1, 1)
term_or_trunc = torch.tensor(buffer['term_or_trunc'].copy(), dtype=torch.bool).to(device)
term = torch.tensor(buffer['term'].copy(), dtype=torch.bool).to(device)
...
...
...
```

2. **Calculate advantage function:** A measure of the goodness of taking a specific action in a given state.

```python
@torch.no_grad()
def advantage(self, transitions):

'''Use TD+GAE+LongTrajectory to compute Advantage and TD target'''

s, s_, a, old_a_logp, r, term_or_trunc, term = transitions

v_ = self.net(s_)[1]
v  = self.net(s)[1]

'''term for TD_target and Adv'''
deltas = r + gamma * v_ * (~term) - v # ~term = 1-term
deltas = deltas.cpu().flatten().numpy()
adv = [0]

'''term_trunc for GAE'''
for dlt, mask in zip(deltas[::-1], term_or_trunc.cpu().flatten().numpy()[::-1]):
    advantage = dlt + gamma * lambd * adv[-1] * (~mask)
    adv.append(advantage)
adv.reverse()
adv = copy.deepcopy(adv[0:-1])
adv = torch.tensor(adv).unsqueeze(1).float().to(device)
td_target = adv + v
adv = (adv - adv.mean()) / ((adv.std()+1e-4))  #sometimes helps

return adv, td_target
```

3. **Calculate policy gradient:** The gradient of the policy with respect to the expected return.

4. **Update the policy:** The policy is updated using the policy gradient and the trust region constraint.

The PPO algorithm offers benefits such as stability, efficiency, and generality, making it suitable for a wide range of reinforcement learning problems. The use of policy gradients and the trust region constraint distinguishes PPO from other algorithms, contributing to its robust performance in various environments.

Bibliography

[1] Gymnasium. Deep reinforcement learning. Accessed: December 13, 2023, https://gymnasium.farama.org/environments/box2d/car_racing/.

[2] Sergey Levine. Deep reinforcement learning. Accessed: November 20, 2023, https://rail.eecs.berkeley.edu/deeprlcourse/.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.