Large Language Model as Policy in Reinforcement Learning

Madhusudan Verma and Arpit Shukla

Contributions :

`Madhusudan Verma:`Implemented LLM as policy network for actor and also implemented Trust Region Policy Optimization for training.

`Arpit Shukla:`Implemented baseline model for policy network as actor and Proximal Policy Optimization for training .

1 Introduction

Reinforcement Learning (RL) is a powerful machine learning paradigm where agents learn to make decisions by interacting with an environment to maximize cumulative rewards. Traditional RL policies are typically represented by neural networks that map states to actions. However, with the advent of Large Language Models (LLMs), which have shown remarkable capabilities in understanding and generating human language, there is a growing interest in leveraging LLMs as policies in RL tasks. This report explores the use of LLMs as policies in the HalfCheetah environment of the Gym library, using the Trust Region Policy Optimization (TRPO) algorithm and then Comparison of it with baseline model in which Proximal Policy Optimization (PPO) algorithm used in CartPole-v1 environment. We aim to provide an in-depth explanation of LLMs, TRPO, PPO and how LLMs can be used as policies, followed by an experimental evaluation.

2 Related Work

Several studies have explored the integration of advanced neural network architectures in RL. Trust Region Policy Optimization (TRPO) is a well-known algorithm that ensures stable updates by optimizing policies within a trust region. Schulman et al. [1] introduced TRPO as a method to address the limitations of traditional policy gradient methods, which often suffer from instability and poor convergence. Large Language Models (LLMs), such as GPT-2 and GPT-3, have been predominantly used in natural language processing tasks [2].

However, recent works have started exploring their potential in RL settings. For instance, Chen et al. [3] demonstrated that LLMs could be fine-tuned to handle decision-making tasks, suggesting the feasibility of their application in RL.

Proximal Policy Optimization (PPO) is another popular RL algorithm, proposed as an improvement over TRPO by introducing a simpler and more efficient approach to policy optimization. Schulman et al. [4] developed PPO to achieve reliable performance and ease of implementation, making it widely adopted in various RL applications.

Additionally, using LLMs for RL has been inspired by the success of sequence-to-sequence models in generating coherent and contextually relevant outputs, making them suitable for complex decision-making scenarios.

3 HalfCheetah Environment

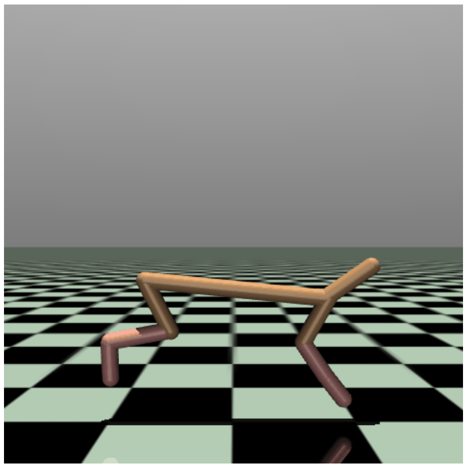


Figure 1: HalfCheetah Environment

This environment is part of the Mujoco environments which contains general information about the environment.

3.1 Action Space

The action space for the HalfCheetah environment is a continuous space represented as a Box:

$$\text{Box}(-1.0, 1.0, (6,), \text{float32})$$

3.2 Observation Space

The observation space for the HalfCheetah environment is also a continuous space, represented as:

$$\text{Box}(-\infty, \infty, (17,), \text{float64})$$

3.3 Importing the Environment

The HalfCheetah environment can be imported using the following command in Python:

```
import gymnasium.make("HalfCheetah-v4")
```

3.4 Description

This environment is based on the work of P. Wawrzyński in "A Cat-Like Robot Real-Time Learning to Run". The HalfCheetah is a 2-dimensional robot consisting of 9 body parts and 8 joints connecting them (including two paws). The goal is to apply torque to the joints to make the cheetah run forward (right) as fast as possible, with a positive reward based on the distance moved forward and a negative reward for moving backward. The cheetah's torso and head are fixed, and torque can only be applied to the other 6 joints over the front and back thighs (which connect to the torso), the shins (which connect to the thighs), and the feet (which connect to the shins).

3.5 Action Space Details

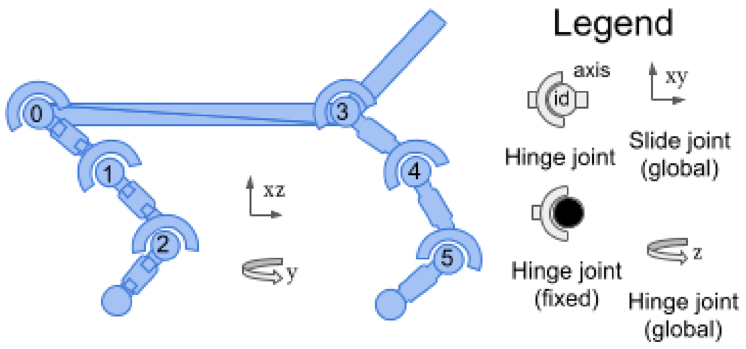


Figure 2: Action Space of HalfCheetah Environment

The action space is a Box(-1, 1, (6,), float32). An action represents the torques applied at the hinge joints.

Num	Action	Control Min	Control Max	Name	Joint	Type (Unit)
0	Torque applied on the back thigh rotor	-1	1	bthigh	hinge	torque (N m)
1	Torque applied on the back shin rotor	-1	1	bshin	hinge	torque (N m)
2	Torque applied on the back foot rotor	-1	1	bfoot	hinge	torque (N m)
3	Torque applied on the front thigh rotor	-1	1	fthigh	hinge	torque (N m)
4	Torque applied on the front shin rotor	-1	1	fshin	hinge	torque (N m)
5	Torque applied on the front foot rotor	-1	1	ffoot	hinge	torque (N m)

3.6 Observation Space Details

The observation space consists of the following parts (in order):

- **qpos** (8 elements by default): Position values of the robot's body parts.
- **qvel** (9 elements): The velocities of these individual body parts (their derivatives).

By default, the observation does not include the robot's x-coordinate (rootx). This can be included by passing `exclude_current_positions_from_observation=False` during construction. In this case, the observation space will be a Box(-Inf, Inf, (18,), float64), where the first observation element is the x-coordinate of the robot. Regardless of whether `exclude_current_positions_from_observation` is set to True or False, the x- and y-coordinates are returned in `info` with the keys `"x_position"` and `"y_position"`, respectively.

By default, however, the observation space is a Box(-Inf, Inf, (17,), float64) where the elements are as follows:

Num	Observation	Min	Max	Name	Joint	Type (Unit)
0	z-coordinate of the front tip	-Inf	Inf	rootz	slide	position (m)
1	angle of the front tip	-Inf	Inf	rooty	hinge	angle (rad)
2	angle of the back thigh	-Inf	Inf	bthigh	hinge	angle (rad)
3	angle of the back shin	-Inf	Inf	bshin	hinge	angle (rad)
4	angle of the back foot	-Inf	Inf	bfoot	hinge	angle (rad)
5	angle of the front thigh	-Inf	Inf	fthigh	hinge	angle (rad)
6	angle of the front shin	-Inf	Inf	fshin	hinge	angle (rad)
7	angle of the front foot	-Inf	Inf	ffoot	hinge	angle (rad)
8	velocity of the x-coordinate of front tip	-Inf	Inf	rootx	slide	velocity (m/s)
9	velocity of the z-coordinate of front tip	-Inf	Inf	rootz	slide	velocity (m/s)
10	angular velocity of the front tip	-Inf	Inf	rooty	hinge	angular velocity (rad/s)
11	angular velocity of the back thigh	-Inf	Inf	bthigh	hinge	angular velocity (rad/s)
12	angular velocity of the back shin	-Inf	Inf	bshin	hinge	angular velocity (rad/s)
13	angular velocity of the back foot	-Inf	Inf	bfoot	hinge	angular velocity (rad/s)
14	angular velocity of the front thigh	-Inf	Inf	fthigh	hinge	angular velocity (rad/s)
15	angular velocity of the front shin	-Inf	Inf	fshin	hinge	angular velocity (rad/s)
16	angular velocity of the front foot	-Inf	Inf	ffoot	hinge	angular velocity (rad/s)
excluded	x-coordinate of the front tip	-Inf	Inf	rootx	slide	position (m)

3.7 Rewards

The total reward is:

$$\text{reward} = \text{forward\_reward} - \text{ctrl\_cost}$$

- **forward_reward**: A reward for moving forward, this reward would be positive if the Half Cheetah moves forward (in the positive direction

/ in the right direction). It is calculated as forward_reward $= \Delta x \cdot t \cdot$ forward_reward_weight, where Δx is the displacement of the "tip", t is the time between actions, which depends on the frame_skip parameter (default is 5), and frame time which is 0.04 - so the default is 0.2, forward_reward_weight is the forward reward weight (default is 1).

- **ctrl_cost**: A negative reward to penalize the Half Cheetah for taking actions that are too large. It is calculated as ctrl_cost = ctrl_cost_weight $\cdot$ sum(action^2), where ctrl_cost_weight is the control cost weight (default is 0.1).

Info contains the individual reward terms.

3.8 Starting State

The initial position state is qpos $\sim \mathcal{N}(0, 0.01)$. The initial velocity state is qvel $\sim \mathcal{U}(-0.1, 0.1)$, where $\mathcal{N}$ is the multivariate normal distribution and $\mathcal{U}$ is the multivariate uniform continuous distribution.

3.9 Episode End

- **Termination**: The Half Cheetah never terminates.
- **Truncation**: The default duration of an episode is 1000 timesteps.

4 CartPole Environment

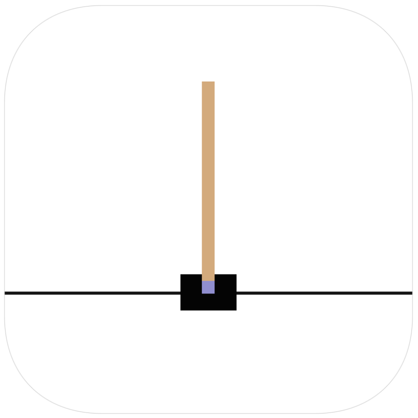


Figure 3: CartPole-v1 Environment

This environment is part of the Classic Control environments which contains general information about the environment.

4.1 Description

The Cart Pole environment is part of the Classic Control environments and corresponds to the version described by Barto, Sutton, and Anderson in “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem”. In this environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The goal is to balance the pole by applying forces in the left and right directions on the cart.

4.2 Action Space

The action space is discrete with two possible actions:

- **0**: Push cart to the left
- **1**: Push cart to the right

The action is represented as a numpy array with shape (1,).

4.3 Observation Space

The observation space is a continuous space with the following components:

- **Cart Position**: Ranges from -4.8 to 4.8
- **Cart Velocity**: Ranges from $-\infty$ to ∞
- **Pole Angle**: Approximately between -0.418 radians ($-24°$) and 0.418 radians ($24°$)
- **Pole Angular Velocity**: Ranges from $-\infty$ to ∞

The observation is represented as a numpy array with shape (4,).

4.4 Rewards

The reward is $+1$ for every step taken, including the termination step. The episode is considered successful if it reaches a reward threshold:

- Threshold for `v1`: 500
- Threshold for `v0`: 200

4.5 Starting State

On reset, all observations are assigned a uniformly random value in the range (-0.05, 0.05).

4.6 Episode End

The episode terminates under the following conditions:

- **Termination**: Pole angle is greater than $\pm 12°$ or the cart position is greater than ± 2.4 (center of the cart reaches the edge of the display)
- **Truncation**: Episode length exceeds 500 steps (200 steps for `v0`)

4.7 Arguments

```
import gymnasium as gym
gym.make('CartPole-v1')
```

On reset, the `options` parameter allows the user to change the bounds used to determine the new random state.

5 Large Language Models (LLMs)

5.1 Overview of LLMs

LLMs, like GPT-2 and GPT-3, are deep neural networks trained on vast amounts of text data. These models leverage the transformer architecture, which utilizes self-attention mechanisms to process and generate text. LLMs have billions of parameters, enabling them to capture intricate patterns in data, allowing them to perform tasks such as translation, summarization, and question-answering with high proficiency. The core idea behind using LLMs as policies in RL is to harness their ability to generalize from large datasets and apply this generalization to decision-making in complex environments. LLMs can generate contextually appropriate sequences, which can be interpreted as action sequences in RL tasks. This property makes them an intriguing choice for policy representation in environments requiring nuanced decision-making.

5.2 GPT-2

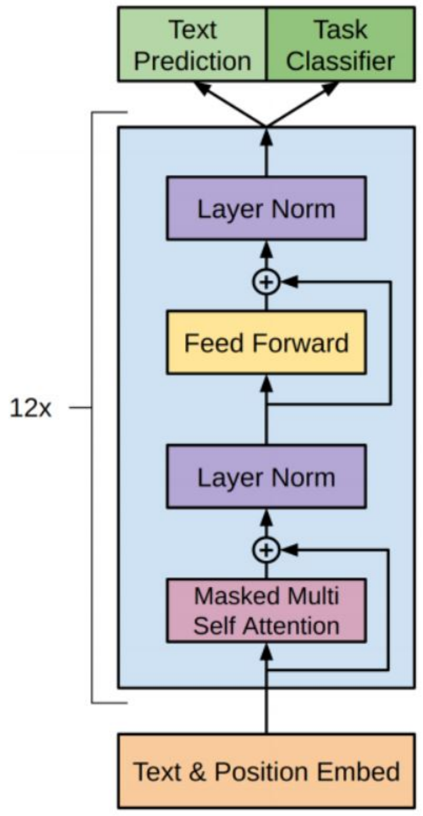
GPT models do not use an encoder. Instead, they are with a decoder-only architecture. This means that the input data is fed directly into the decoder without being transformed into a higher, more abstract representation by an encoder.

GPT-2 (Generative Pre-trained Transformer 2) is an earlier version of GPT-3 but still a highly capable LLM. It was developed by OpenAI and consists of up to 1.5 billion parameters. GPT-2 is designed using the transformer architecture, which relies heavily on self-attention mechanisms. The model is pre-trained on a diverse dataset of internet text, enabling it to generate coherent and contextually relevant text.

The architecture of GPT-2 can be summarized by the following key components:

1. **Token Embedding**: Converts input text into token representations.
2. **Positional Encoding**: Adds positional information to the token embeddings.
3. **Self-Attention Mechanism**: Allows the model to focus on different parts of the input text simulta
4. **Feedforward Neural Networks**: Applies transformations to the self-attention outputs.
5. **Layer Normalization**: Stabilizes the training process by normalizing the outputs of the layers.

The figure 4 is the architecture of GPT2 which uses only



[11]

Figure 4: GPT2

The self-attention mechanism in GPT-2 can be mathematically represented as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q (queries), K (keys), and V (values) are linear transformations of the input embeddings, and d_k is the dimension of the keys.

6 Trust Region Policy Optimization (TRPO)

TRPO updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence, a measure of distance between probability distributions.

This is different from normal policy gradient, which keeps new and old policies close in parameter space. But even seemingly small differences in parameter space can have very large differences in performance—so a single bad step can collapse the policy performance. This makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse, and tends to quickly and monotonically improve performance.

Mathematically, TRPO optimizes the following objective:

$$\max_{\theta} E_{s\sim\rho_{\pi_{\text{old}}},a\sim\pi_{\text{old}}}\left[\frac{\pi_\theta(a|s)}{\pi_{\text{old}}(a|s)}A_{\pi_{\text{old}}}(s,a)\right]$$

subject to the constraint:

$$E_{s\sim\rho_{\pi_{\text{old}}}}\left[D_{\text{KL}}\left(\pi_{\text{old}}(\cdot|s)\|\pi_\theta(\cdot|s)\right)\right]\leq\delta$$

where π_θ is the new policy, π_{old} is the old policy, $A_{\pi_{\text{old}}}(s,a)$ is the advantage function, and δ is a small positive constant.

To enforce the constraint, the policy update is derived from a linear Taylor series approximation to the objective function and a quadratic Taylor series approximation to the constraint:

$$L_{\pi_{\text{old}}}(\pi_\theta)\approx g^T(\theta-\theta_{\text{old}})$$

$$D_{\text{KL}}(\pi_{\text{old}}\|\pi_\theta)\approx\frac{1}{2}(\theta-\theta_{\text{old}})^T H(\theta-\theta_{\text{old}})$$

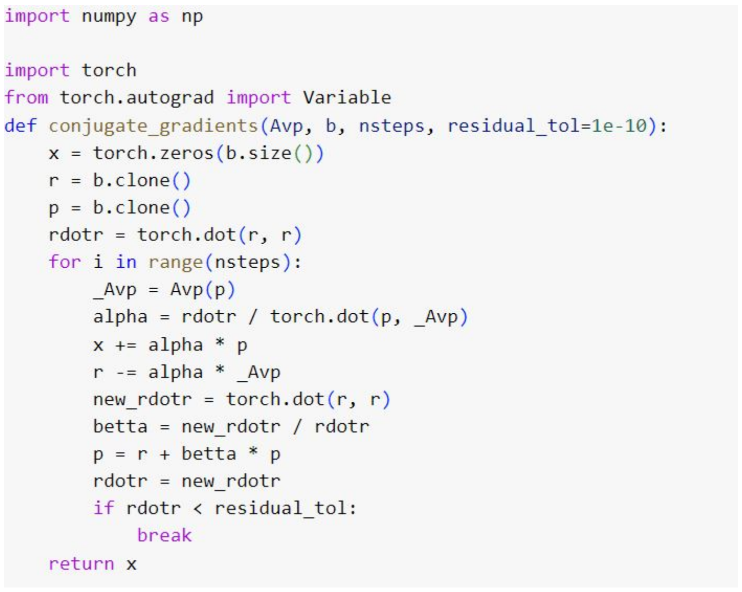
where g is the gradient of the surrogate objective function and H is the Hessian matrix of the KL-divergence.

The policy update can then be solved as a constrained optimization problem:

$$\theta_{\text{new}}=\theta_{\text{old}}+\sqrt{\frac{2\delta}{g^T H^{-1}g}}H^{-1}g$$

This ensures that the new policy $\pi_{\theta_{\text{new}}}$ stays within the trust region defined by the KL-divergence constraint.

If we were to stop here, and just use this final result, the algorithm would be exactly calculating the Natural Policy Gradient. A problem is that, due to the approximation errors introduced by the Taylor expansion, this may not satisfy the KL constraint, or actually improve the surrogate advantage. TRPO adds a modification to this update rule: a backtracking line search,



```
import numpy as np

import torch
from torch.autograd import Variable
def conjugate_gradients(Avp, b, nsteps, residual_tol=1e-10):
    x = torch.zeros(b.size())
    r = b.clone()
    p = b.clone()
    rdotr = torch.dot(r, r)
    for i in range(nsteps):
        _Avp = Avp(p)
        alpha = rdotr / torch.dot(p, _Avp)
        x += alpha * p
        r -= alpha * _Avp
        new_rdotr = torch.dot(r, r)
        betta = new_rdotr / rdotr
        p = r + betta * p
        rdotr = new_rdotr
        if rdotr < residual_tol:
            break
    return x
```

Figure 5: Conjugate Gradient

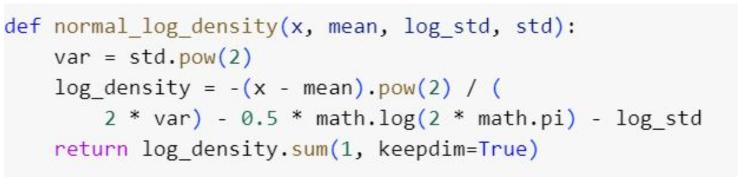
$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g,$$

where $\alpha \in (0,1)$ is the backtracking coefficient, and j is the smallest nonnegative integer such that $\pi_{\theta_{k+1}}$ satisfies the KL constraint and produces a positive surrogate advantage.

Lastly: computing and storing the matrix inverse, H^{-1}, is painfully expensive when dealing with neural network policies with thousands or millions of parameters. TRPO sidesteps the issue by using the conjugate gradient algorithm to solve $Hx = g$ for $x = H^{-1}g$, requiring only a function which can compute the matrix-vector product Hx instead of computing and storing the whole matrix H directly. This is not too hard to do: we set up a symbolic operation to calculate.The implementation of conjugate gradient is in 5.

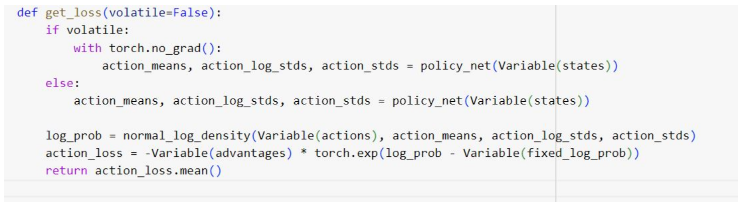
$$Hx = \nabla_\theta \left(\left(\nabla_\theta \bar{D}_{KL}(\theta || \theta_k) \right)^T x \right),$$

which gives us the correct output without computing the whole matrix. Implementation of normal log density ,loss function for policy ,loss function for value network and KL divergence are in 6, 7, 8 respectively. The implementation for KL divergence is in 9 .



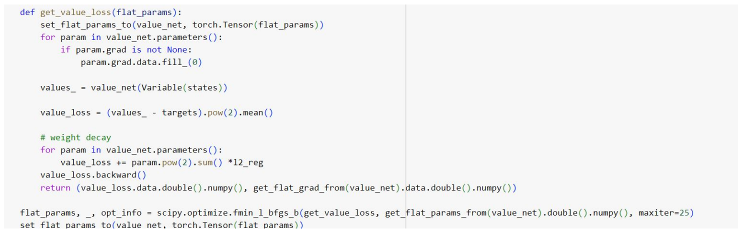
```python
def normal_log_density(x, mean, log_std, std):
    var = std.pow(2)
    log_density = -(x - mean).pow(2) / (
        2 * var) - 0.5 * math.log(2 * math.pi) - log_std
    return log_density.sum(1, keepdim=True)
```

Figure 6: Normal log density



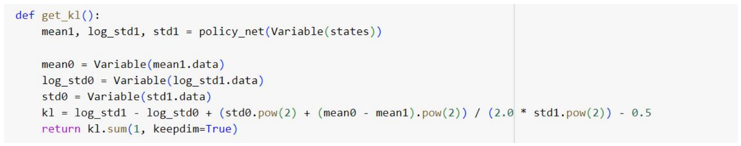
```python
def get_loss(volatile=False):
    if volatile:
        with torch.no_grad():
            action_means, action_log_stds, action_stds = policy_net(Variable(states))
    else:
        action_means, action_log_stds, action_stds = policy_net(Variable(states))

    log_prob = normal_log_density(Variable(actions), action_means, action_log_stds, action_stds)
    action_loss = -Variable(advantages) * torch.exp(log_prob - Variable(fixed_log_prob))
    return action_loss.mean()
```

Figure 7: Loss function for policy



```python
def get_value_loss(flat_params):
    set_flat_params_to(value_net, torch.Tensor(flat_params))
    for param in value_net.parameters():
        if param.grad is not None:
            param.grad.data.fill_(0)

    values_ = value_net(Variable(states))

    value_loss = (values_ - targets).pow(2).mean()

    # weight decay
    for param in value_net.parameters():
        value_loss += param.pow(2).sum() *l2_reg
    value_loss.backward()
    return (value_loss.data.double().numpy(), get_flat_grad_from(value_net).data.double().numpy())

flat_params, _, opt_info = scipy.optimize.fmin_l_bfgs_b(get_value_loss, get_flat_params_from(value_net).double().numpy(), maxiter=25)
set_flat_params_to(value_net, torch.Tensor(flat_params))
```

Figure 8: Loss function for value network



```python
def get_kl():
    mean1, log_std1, std1 = policy_net(Variable(states))

    mean0 = Variable(mean1.data)
    log_std0 = Variable(log_std1.data)
    std0 = Variable(std1.data)
    kl = log_std1 - log_std0 + (std0.pow(2) + (mean0 - mean1).pow(2)) / (2.0 * std1.pow(2)) - 0.5
    return kl.sum(1, keepdim=True)
```

Figure 9: KL Divergence

7 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method in reinforcement learning that improves the stability and reliability of policy updates. PPO methods are simpler to implement and tune than Trust Region Policy Optimization (TRPO) while achieving similar performance.

7.1 Objective Function

PPO optimizes a clipped surrogate objective function to prevent large updates to the policy and ensure stable learning. The objective function is given by:

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

where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new policy π_θ and the old policy $\pi_{\theta_{\text{old}}}$.
- $\hat{A}_t$ is the advantage estimate at time step t.
- ϵ is a small positive constant that controls the clipping range.

7.2 Clipped Surrogate Objective

The clipped surrogate objective function prevents the probability ratio $r_t(\theta)$ from deviating too far from 1, ensuring that the new policy does not differ too much from the old policy. The objective is to maximize the minimum of the unclipped objective $r_t(\theta)\hat{A}_t$ and the clipped objective $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$.

7.3 Advantage Function

The advantage function $\hat{A}_t$ measures how much better or worse an action is compared to the expected value. It is typically computed using Generalized Advantage Estimation (GAE):

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error, γ is the discount factor, and λ is the GAE parameter.

7.4 Policy Update

The policy parameters θ are updated by maximizing the clipped surrogate objective using stochastic gradient ascent. The policy update rule is given by:

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}^{\text{CLIP}}(\theta)$$

where α is the learning rate.

7.5 Value Function Update

In addition to updating the policy, PPO also updates a value function $V(s_t)$ to estimate the expected return from a given state. The value function is updated by minimizing a squared-error loss:

$$\mathcal{L}^{\text{VF}}(\theta) = E_t\left[(V_\theta(s_t) - R_t)^2\right]$$

where R_t is the target return at time step t.

7.6 KL-Divergence Penalty

While the clipped surrogate objective is the primary mechanism to ensure stability, PPO can also include a KL-divergence penalty to further constrain the updates:

$$\mathcal{L}^{\text{KL}}(\theta) = E_t\left[\text{KL}\left[\pi_{\theta_{\text{old}}}(\cdot|s_t)\|\pi_\theta(\cdot|s_t)\right]\right]$$

This penalty ensures that the new policy does not deviate too much from the old policy and can be used as an additional constraint or as an alternative to the clipping mechanism.

7.7 Overall Algorithm

The PPO algorithm alternates between sampling data through interaction with the environment and performing multiple epochs of optimization on the sampled data. The main steps are:

1. Collect trajectories by running the current policy in the environment.
2. Compute the advantage estimates using the collected data.
3. Optimize the policy by maximizing the clipped surrogate objective.
4. Update the value function by minimizing the squared-error loss.
5. Repeat the process for a fixed number of iterations or until convergence.

8 Baseline Model

This is the Architecture of Policy network (Actor) for Baseline model The PPO Policy Network (Actor) consists of an input layer that receives the observation state, followed by three fully connected hidden layers, each with ReLU activation to introduce non-linearity. The final output layer uses a softmax function to generate a probability distribution over possible actions, guiding the agent's decisions in the reinforcement learning environment. The network's design allows it to effectively model complex policies by progressively refining the input data through each layer.

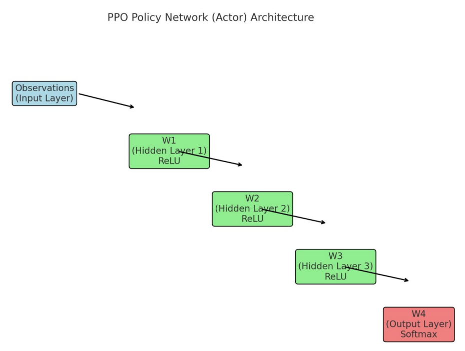

PPO Policy Network (Actor) Architecture
Observations (Input Layer)
W1 (Hidden Layer 1) ReLU
W2 (Hidden Layer 2) ReLU
W3 (Hidden Layer 3) ReLU
W4 (Output Layer) Softmax

Figure 10: Actor as Policy Network(Baseline Model)

9 LLMs as Policy

Using LLMs as policies in RL involves treating the decision-making process as a sequence generation task. In this approach, the LLM receives the current state of the environment as input and generates the next action to be taken. The LLM is fine-tuned using RL algorithms like TRPO ,PPO to adapt its parameters for the specific task. This method leverages the LLM's capacity to understand and generate complex sequences, potentially leading to more sophisticated decision-making strategies. The process involves mapping the state representation into a format compatible with the LLM, generating action sequences, and optimizing these sequences to maximize cumulative rewards. This approach allows leveraging pre-trained LLMs' generalization capabilities and adapting them to specific RL tasks through fine-tuning.The implementation is in 11

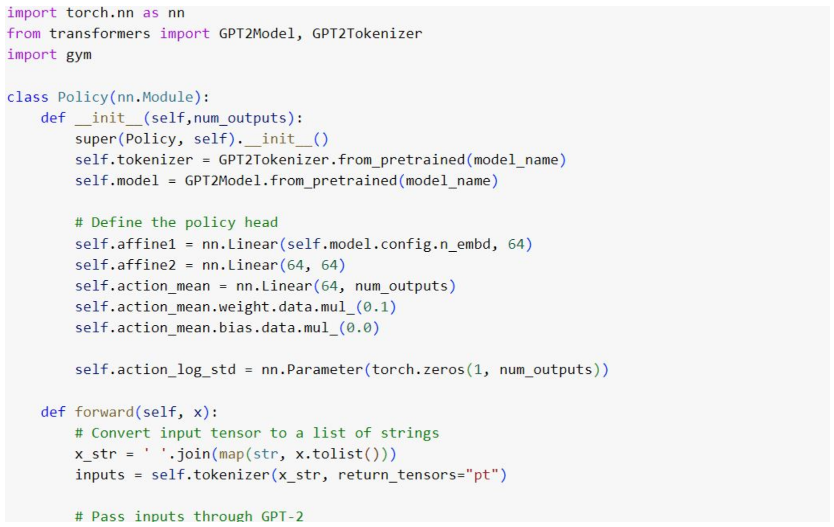

```python
import torch.nn as nn
from transformers import GPT2Model, GPT2Tokenizer
import gym

class Policy(nn.Module):
    def __init__(self,num_outputs):
        super(Policy, self).__init__()
        self.tokenizer = GPT2Tokenizer.from_pretrained(model_name)
        self.model = GPT2Model.from_pretrained(model_name)

        # Define the policy head
        self.affine1 = nn.Linear(self.model.config.n_embd, 64)
        self.affine2 = nn.Linear(64, 64)
        self.action_mean = nn.Linear(64, num_outputs)
        self.action_mean.weight.data.mul_(0.1)
        self.action_mean.bias.data.mul_(0.0)

        self.action_log_std = nn.Parameter(torch.zeros(1, num_outputs))

    def forward(self, x):
        # Convert input tensor to a list of strings
        x_str = ' '.join(map(str, x.tolist()))
        inputs = self.tokenizer(x_str, return_tensors="pt")

        # Pass inputs through GPT-2
```

Figure 11: Large language model as policy

10 Experiment

10.1 Setup

We conducted experiments in the HalfCheetah and CartPole-v1 environment from OpenAI's Gym, a standard benchmark for testing RL algorithms. The goal is to train the HalfCheetah agent to run as fast as possible. We implemented the TRPO algorithm and used a pre-trained GPT-2 model as the policy network where as for CartPole-v1, we implemented the PPO algorithm and used actor as policy network. The GPT-2 model was fine-tuned on the HalfCheetah task using the TRPO algorithm.

10.2 Procedure

1. **Pre-processing**: The state of the HalfCheetah environment was encoded into a format suitable for the GPT-2 model. This involved normalizing the state vectors and converting them into a sequence format that the GPT-2 model could process.

2. **Training**: The GPT-2 model was fine-tuned using TRPO, optimizing the policy to maximize the cumulative reward. The fine-tuning process involved iteratively updating the model parameters based on the feedback received from the environment, ensuring that the policy learned to generate effective action sequences.

3. **Evaluation**: The performance of the LLM-based policy was compared against a baseline policy network. We evaluated the models based on

their ability to achieve high cumulative rewards and maintain stability during the training process.

11 Results

11.1 Results-Arpit

The implementation of a Proximal Policy Optimization (PPO) algorithm applied to the CartPole-v1 environment using the OpenAI Gym framework. The results indicate a progressive improvement in the agent's performance, as measured by the average reward per 25 episodes. The agent begins with low performance but quickly improves after about 75 episodes, reaching the threshold for solving the environment (an average reward of 500) by episode 350. This showcases the effectiveness of the PPO algorithm in training an agent to master the CartPole environment, achieving optimal performance after sufficient training iterations.

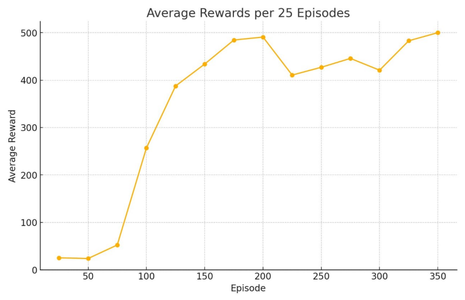
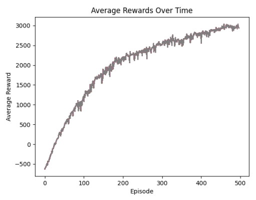


Figure 12: Average Reward Vs Episode

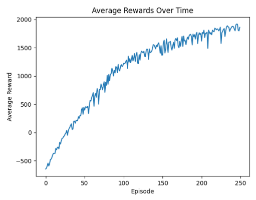
Here is the graph illustrating the average rewards per 25 episodes. It shows the agent's performance improvement over time, with the average reward increasing significantly after the 75th episode, and ultimately reaching the maximum possible reward by the 350th episode.

11.2 Results-Madhusudan

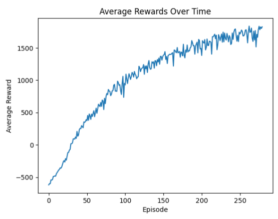
The experimental results demonstrated that the LLM-based policy could effectively learn the HalfCheetah task. The TRPO algorithm ensured stable updates, and the fine-tuned GPT-2 model generated coherent and effective actions. The LLM-based policy showed competitive performance compared to traditional neural network policies, with potential advantages in terms of generalization and handling complex state-action mappings. The LLM-based policy exhibited smooth learning curves and was able to achieve high rewards consistently, indicating its effectiveness in capturing the dynamics of the HalfCheetah environment and making optimal decisions. Additionally, the model demonstrated robustness to variations in the environment, showcasing the potential benefits of using LLMs as policies in RL tasks.



(a)



(b)



(c)

Figure 13: Average rewards vs Number of episodes a)Average rewards vs Number of episodes for 500 episodes with env and torch seed value set to 543 b)Average rewards vs Number of episodes for 250 episodes with env and torch seed value set to 100 c)Average rewards vs Number of episodes for 250 episodes with env and torch seed value set to 100.

12 Conclusion

This report explored the use of Large Language Models (LLMs) as policies in reinforcement learning, specifically in the HalfCheetah environment using the Trust Region Policy Optimization (TRPO) algorithm and compared with baseline model in which Actor used as policy network in CartPole-v1 environment using the Proximal Policy Optimization (PPO) . We provided an overview of LLMs, PPO and TRPO, and detailed how LLMs can be adapted as policies in RL tasks. Our experimental results demonstrated the feasibility and potential benefits of this approach. The LLM-based policy achieved competitive performance and exhibited advantages in terms of generalization and stability. Future work will focus on extending this approach to other RL tasks and environments, as well as investigating the impact of different LLM architectures and training strategies on policy performance.

13 References

References

[1] Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. Available at: `https://proceedings.mlr.press/v37/schulman15.html`

[2] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., . . . & Amodei, D. (2020). Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems (NeurIPS)*. Available at: `https://arxiv.org/abs/2005.14165`

[3] Chen, M., Tworek, J., Jun, H., Yuan, Q., Dehghani, M., . . . & Zaremba, W. (2021). Evaluating Large Language Models Trained on Code. Available at: `https://arxiv.org/abs/2107.03374`

[4] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. Available at: `https://arxiv.org/abs/1707.06347`

[5] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. Available at: `https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`

[6] Yu, L., Zhang, W., Wang, J., & Yu, Y. (2017). SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient. In *Thirty-First AAAI Conference on Artificial Intelligence*. Available at: `https://arxiv.org/abs/1609.05473`

[7] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[8] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). OpenAI Gym. Available at: `https://arxiv.org/abs/1606.01540`

[9] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. (2014). Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML)*. Available at: `https://proceedings.mlr.press/v32/silver14.pdf`

[10] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations (ICLR)*. Available at: `https://arxiv.org/abs/1412.6980`

[11] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is All You Need. In *Advances in Neural Information Processing Systems (NeurIPS)*. Available at: `https://arxiv.org/abs/1706.03762`