An Introduction to Proximal Policy Optimization using Super Mario Brothers

Alexander Hawking

October 2023

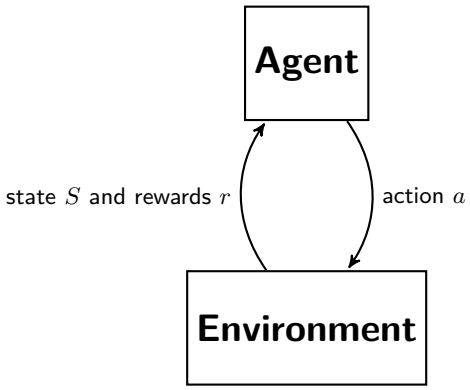
Preamble: This report is designed to develop an understanding of the Proximal Policy Optimization (PPO) algorithm, aimed for individuals who already possess a foundational understanding of Reinforcement Learning (RL). While RL encompasses a vast array of techniques and strategies, PPO has emerged as a notable and effective approach in recent years. To provide a hands-on and tangible perspective, we will be utilizing the classic game of Super Mario Bros as our testbed. This game, with its intricate challenges and diverse environments, offers a comprehensive platform to demonstrate the prowess of PPO. This paper will specifically examine the *CLIP* version of PPO as it is the most widely used version.

1 Introduction to Reinforcement Learning in Game-Playing

In the realm of game-playing artificial intelligence, a plethora of algorithms are built upon the foundational framework known as **reinforcement learning**. This framework is characterized by several components: an environment, states, an agent, a set of possible actions, and a reward mechanism to acknowledge desirable actions [5].

Consider a game like Super Mario Brothers:

- The *state* S depicts the pixel input of the current screen.
- The *agent* (Mario) aims to select an action, such as moving right or jumping, based on the state. This action should ideally maximize the reward.
- Rewards are typically given for progressing in the game without losing a life.
- After the agent takes an action, the game advances to the next state, represented as S'.



A fundamental challenge in devising game-playing algorithms lies in identifying the action that maximizes the reward. Broadly, these algorithms fall into two primary categories [9]:

1. **Value-Based**: Algorithms in this category, like Q-learning, estimate the value of being in a state or the expected return (Q-value) of taking a specific action in a state. The policy, which dictates the agent's behavior, is derived from these values.
2. **Policy-Based**: These algorithms, such as REINFORCE, directly aim to optimize the policy without calculating a value function. The policy is parameterized, and the learning process refines these parameters.

Building upon these categories, there are hybrid methods known as **Actor-Critic** algorithms. In these, the actor proposes an action based on a policy while the critic estimates the value of states or state-action pairs to guide the actor's decisions. The essence of actor-critic methods lies in the simultaneous optimization of both the policy (by the actor) and the value function (by the critic) [6].

One of the prominent actor-critic algorithms is the Proximal Policy Optimization (PPO). In PPO, the actor produces a policy (distribution over actions), and the critic estimates the value of states. The PPO algorithm then adjusts the policy using a surrogate objective that takes advantage of the critic's value estimates, ensuring that the new policy doesn't deviate too much from the old one.

1.1 Python and Essential Libraries

For this project, I've chosen to use Python due to its versatility, extensive libraries, and robust support for machine learning and game environments. Specifically, we are using:

- **Python 3 (version 3.11+)**: The latest stable release, ensuring optimal performance and security [16].
- **PyTorch**: An open-source machine learning library, ideal for building deep learning models [12].
- **NumPy**: A library for numerical operations, offering an array of tools for algebraic computations and matrix operations [7].
- **OpenAI Gym**: A toolkit for developing reinforcement learning algorithms. It provides various environments to test and train agents [3].
- **Super Mario Bros Gym package**: A specialized package to simulate the Super Mario Bros game environment [4].

For setting up the development environment, I recommend using either pip or conda to install and manage the required libraries.

Below is a snippet to initialize the Super Mario Bros game environment using OpenAI Gym:

```
env = gym.make('SuperMarioBros-v0',
↪  apply_api_compatibility=True, render_mode="human")
env = JoypadSpace(env, [["right"], ["right", "A"]])
```

In this setup, we've defined our action space to include only two inputs: **right** and **right and A** (move right and jump). This choice simplifies our training process. While it reduces the range of possible actions, it remains sufficient for completing most levels, striking a balance between complexity and performance.

1.2 Reward Structure

As explained above, the goal in reinforcement learning is to select actions that maximizes rewards. The reward function built into the `gym-super-mario-bros` library is designed with the objective that the agent should move as far to the right as possible, as quickly as possible, without dying. The reward is composed of three separate components [4]:

1. **Instantaneous Velocity,** v: Represents the difference in the agent's x-values between two consecutive states.

   $$v = x_1 - x_0$$

   where x_0 is the x-position before the step and x_1 is the x-position after the step. The possible conditions are:

   $$\begin{aligned} v > 0 &\implies \text{Moving right} \\ v < 0 &\implies \text{Moving left} \\ v = 0 &\implies \text{Not moving} \end{aligned}$$

2. **Clock Penalty,** c: Represents the difference in the game clock between frames. This penalty prevents the agent from standing still.

   $$c = c_0 - c_1$$

   where c_0 is the clock reading before the step and c_1 is the clock reading after the step. The conditions are:

   $$\begin{aligned} c = 0 &\implies \text{No clock tick} \\ c < 0 &\implies \text{Clock tick} \end{aligned}$$

3. **Death Penalty,** d: Penalizes the agent for dying in a state, encouraging the agent to avoid death.

   $$\begin{aligned} d = 0 &\implies \text{Alive} \\ d = -15 &\implies \text{Dead} \end{aligned}$$

The overall reward is then given by:

$$r = v + c + d$$

which is clipped to the range $[-15, 15]$.

2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm introduced by the OpenAI team in 2017. It was designed to address several

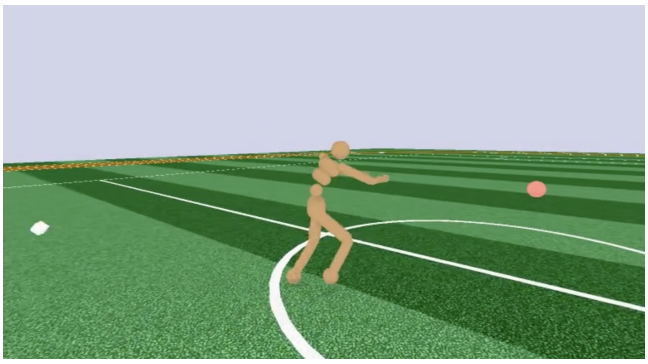


Figure 1: An agent playing Roboschool [10]

shortcomings in traditional policy gradient methods, including inefficiency in policy update methods and issues surrounding stability.

PPO has become the default reinforcement learning algorithm at OpenAI due to its simplicity of implementation and tuning, combined with a performance that rivals or even surpasses more complex state-of-the-art algorithms. A notable application of PPO is the Roboschool environment, as illustrated below. In this setting, an agent learns a myriad of skills—ranging from walking and running to complex maneuvers like recovering from falls—to ultimately reach a target, depicted as a pink sphere [2].

PPO employs a dual neural network strategy using an actor network and a critic network. The actor network's primary role is action selection. Given a state, it predicts the best possible action to take. Conversely, the critic network evaluates the proposed action's potential value. It gauges how beneficial or detrimental an action might be, providing essential feedback to the actor.

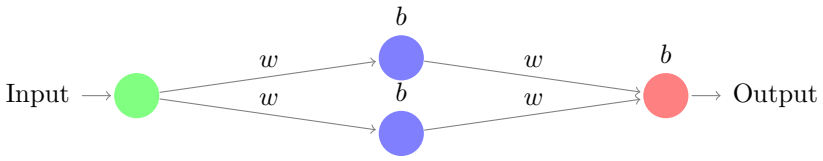
Central to PPO's learning process is its unique loss function. This function not only quantifies the actor's performance but also ensures that the updated policy doesn't deviate drastically from the previous one. This constraint on policy updates is pivotal to PPO's stability. The insights derived from this loss function guide the iterative refinement of both networks, utilizing the process of backward propagation. Through these continuous updates, the agent progressively hones its decision-making abilities, striving for optimal interactions with its environment [14].

2.1 Neural Networks

Neural networks are computational models inspired by the way biological neural networks in the human brain work. They consist of layers of interconnected nodes (or "neurons"). Each connection between neurons can transmit a signal from one neuron to another. The receiving neuron processes the signal and then signals neurons connected to it.

The strength of the connections (weights w) is adjusted during training. Ad-

ditionally, each neuron typically has an associated bias b and activation function. The bias allows the neuron to have outputs other than zero when all its inputs are zeros, and the activation function (like the sigmoid or ReLU function) introduces non-linearity into the model, which enables it to learn from error and make predictions [8].
To understand the basics of neural networks let us consider a simple network with one input node, one hidden layer with two neurons and one output node.



For this function we will use the ReLU (Rectified Linear Unit) activation function which will only return positive inputs or 0 if given a negative input.

$$\text{ReLU}(z) = \max(0, z) \tag{1}$$

Each neuron works by multiplying its input by weight and then adding the bias, before passing this result into activation function [1].

$$\text{output}(x) = \text{ReLU}(\sum_i (x_i \times w_i) - b) \tag{2}$$

So lets consider our simple network with the following parameters:

- Input value: $x = 2$
- Weights: $w_1 = 0.5$, $w_2 = -0.5$, $w_3 = 1$, $w_4 = 1$
- Biases: $b_1 = 0.5$, $b_2 = 1$, $b_{\text{out}} = -0.5$

We can calculate the output of our network given the input 2, as below:

1. **Hidden Layer**:
   Neuron 1:

   $$h_1 = \text{ReLU}(x \cdot w_1 + b_1) = \text{ReLU}(2 \cdot 0.5 + 0.5) = \text{ReLU}(1.5) = 1.5$$

   Neuron 2:

   $$h_2 = \text{ReLU}(x \cdot w_2 + b_2) = \text{ReLU}(2 \cdot (-0.5) + 1) = \text{ReLU}(0) = 0$$

2. **Output Layer**:

   $$\text{output} = h_1 \cdot w_3 + h_2 \cdot w_4 + b_{\text{out}} = 1.5 \cdot 1 + 0 \cdot 1 - 0.5 = 1$$

So, for this simple neural network and given input, weights, and biases, the output is 1. This is a straightforward example, but it demonstrates the basic principle. In practice, neural networks contain many more layers and neurons, and the input could be multidimensional.

The basic process of training a network involves adjusting these weights and biases so that certain inputs give desired outputs. In this case this network is trained with a forward pass, evaluating the output using a **loss** function and then using backwards propagation to adjust the network to reduce the loss. This will be examined further below.

We can use the `nn` class from the `torch` library to create our actor and critic networks in Python [12].

Python Implementation

```
actor = nn.Sequential(
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8,
    ↪  stride=4),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4,
    ↪  stride=2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
    ↪  stride=1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(3136, 512),
    nn.ReLU(),
    nn.Linear(512, env.action_space.n)
)
critic = nn.Sequential(
    nn.Conv2d(in_channels=4, out_channels=32, kernel_size=8,
    ↪  stride=4),
    nn.ReLU(),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4,
    ↪  stride=2),
    nn.ReLU(),
    nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,
    ↪  stride=1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(3136, 512),
    nn.ReLU(),
    nn.Linear(512, 1)
)
```

We can observe that both the actor and critic networks consist of three

convolutional layers followed by two linear layers.

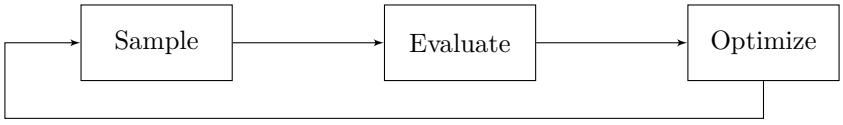
- **Convolutional Layers**: These are fundamental components of Convolutional Neural Networks (CNNs), a class of deep neural networks optimized for processing grid-like data structures, such as images. A convolutional layer performs a mathematical operation named "convolution" that allows the network to discern and learn spatial hierarchies of features. These layers are particularly adept at recognizing patterns in the spatial or temporal domain, making them suitable for tasks such as image and video recognition [11].
- **Linear Layers**: Also known as fully connected (FC) or dense layers, linear layers are the foundational layers we encounter in vanilla neural networks. They are responsible for producing output predictions from the features learned by previous layers.

The network is designed to accept inputs with four channels, corresponding to four stacked greyscale frames from the game (explained further in the pre-processing section). The actor network's output channels correspond to the game's action space, enabling the modeling of a categorical distribution over possible actions. On the other hand, the critic network produces a single output channel, offering an estimate of the state's value.

3 Training Process

Note: the pre-processing methods used in the code are largely based upon those provided in the DDQN tutorial on the PyTorch website

With a foundational understanding of the PPO algorithm, we can explore the methodology behind training a model to play Super Mario Bros. The training process can be broken into three fundamental steps:



The first step in the training process is *sampling*. Here, the model engages with the game, amassing experience which subsequently undergoes evaluation and informs optimization for enhanced performance.

3.1 Pre-processing

Note: The pre-processing strategies utilized in our code predominantly based upon the methods showcased in the DDQN tutorial on the PyTorch website[17]

Training an RL model can be a very computationally expensive process, however there are some adjustments that can be made to improve the efficiency of this process. By making adjustments to the data sampled we can provide our model with minimal input, requiring less calculations and speeding up the learning process. Making these adjustments is known as pre-processing and can include reducing the action space, resizing frames and converting the frames to grey scale.

- **Reducing the action space** involves reducing the number of possible actions the agent can take, reducing the number of different actions that need to be investigated. In traditional super mario bros, there is a D-Pad that can produce actions lefts, right, up and down, as well as two different buttons A (jump) and B (use items). For this example we limit the action space to `["right"]` and `["A", "right"]` (moving jump). These two actions can be used to complete the majority of levels and only using these 2 can greatly speed up the training process.

```
env = JoypadSpace(env, [["right"], ["right", "A"]])
```

- **Resizing** the observation space can reduce the amount of input that the network needs to interpret. High-resolution images can be computationally expensive to process, and often a lower resolution image contains enough information for decision-making in many games or simulations. Fortunately we can resize a given shape using TorchVision's transforms.

```
class ResizeObservation(gym.ObservationWrapper):
    def __init__(self, env, shape):
        super().__init__(env)
        self.shape = (shape, shape)
        obs_shape = self.shape +
        ↪  self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255,
        ↪  shape=obs_shape, dtype=np.uint8)

    def observation(self, observation):
        transformations =
        ↪  transforms.Compose([transforms.Resize(self.shape),
        ↪  transforms.Normalize(0, 255)])
        return transformations(observation).squeeze(0)
```

- An image is **greyscale** if it only has a single colour channel. Generally games produce output with three colour channels *Red, Green* and *Blue* (RGB), however we can convert this into a single grayscale channel, meaning we have less data to interpret. When you convert a color (RGB) image to grayscale, each pixel's red, green, and blue values are combined

into a single shade of gray. The common method to compute the grayscale value G from RGB values is using a weighted sum:

$$G = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{3}$$

However this can be done with TorchVisions `Grayscale()` function.

Python Implementation

```
class GrayScaleObservation(gym.ObservationWrapper):
    def __init__(self, env):
        super().__init__(env)
        self.observation_space = Box(low=0, high=255,
        ↪  shape=self.observation_space.shape[:2],
        ↪  dtype=np.uint8)

    def observation(self, observation):
        transform = transforms.Grayscale()
        return
        ↪  transform(torch.tensor(np.transpose(observation,
        ↪  (2, 0, 1)).copy(), dtype=torch.float))
```

- **Frame skipping** is the final performance optimization method we will examine. In many games or simulations, consecutive frames can be almost identical, so running an agent's decision process on each and every frame can be computationally wasteful, to mitigate this, we can skip a number of frames, only showing the agent every *nth* frame. When the agent takes an action, instead of applying this action to the environment once, the action is applied for skip consecutive frames. The rewards from all these frames are summed up, and only the last observation (frame) is returned to the agent. If the episode terminates (ends) before all skips are done, it breaks early.

Python Implementation

```
class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        total_reward = 0.0
        done = False
        trunc = False
        for _ in range(self._skip):
```

```
            obs, reward, done, trunc, info =
            ↪  self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, trunc, info
```

- The last element of pre-processing to complete is **frame stacking**. This is a process where frames are stacked and given to the network and can help the agent understand movement within the frame. You may have noticed previously that our networks have four input channels, as we stack our frames in groups of four, each input channel is given a frame. With these four elements implemented we can now more efficiently sample and train our model.

Python Implementation

```
env = JoypadSpace(env, [["right"], ["right", "A"]])
env = SkipFrame(env, skip=4)
env = GrayScaleObservation(env)
env = ResizeObservation(env, shape=84)
env = FrameStack(env, num_stack=4)
```

3.2 Sampling

Now that we have pre-processed our data, we can begin the sampling stage. This involves using the model to play the game and learn from experience. We first define our arrays to store the results from our training and then append to these arrays while the game is played.

Pseucode implentation of sampling

```
Input: batch_size, policy_old, env

Initialize arrays: rewards, actions, done, obs, log_pis, values
↪  of size batch_size
FOR i from 0 to batch_size - 1 DO
    Get current observation obs[i]
    Calculate action probability distribution pi and value
    ↪  estimate v using policy_old from obs[i]
    Store v in values[i]
    Sample action a from pi
    Store a in actions[i]
    Store log probability of a in log_pis[i]
    Execute action a in environment to get new obs, reward, and
    ↪  done status
```

```
    Store the reward in rewards[i]
    Render the environment
    IF episode is done THEN
        Reset the episode and environment
    END IF
END FOR
```

After sampling the environment this data is then used to calculate loss and to update the model using backpropagation and optimization methodology.

4 Evaluating the Model

Breakdown of PPO-CLIP framework proposed in Proximal Policy Optimization Algorithms by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov[14]

In the PPO framework, the actor network generates a probability distribution over potential actions, while the critic network provides an estimate of the value for each state.

- Actor Network: Given a state observation s in the form of pixel input data, the actor network computes a probability distribution for the available actions.

  $$\pi(s) = \text{Actor Network}(s)$$

  Here, $\pi(s)$ represents the policy, which is a probability distribution over actions for the input state s.

- Critic Network: The critic network evaluates the value of the state s.

  $$V(S) = \text{Critic Network}(s)$$

  $V(s)$ signifies the estimated value associated with the input state s.

PPO uses a combined loss function to train and evaulate the model.

$$L_t^{\text{CLIP}+V_F+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{4}$$

This loss function can actually be broken into three different parts:

- **Clipped Objective for Actor** L^{clip}
- **Value Function Loss for the Critic** L^{vf}
- **Entropy Regularization** S

4.1 Clipped Objective for Actor

This objective loss function centers around the actor and its ability to select actions that maximize reward.

$$L^{\text{clip}}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta).\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon).\hat{A}) \tag{5}$$

The term $r_t(\theta)$ defines the ratio of probabilities of selecting an action.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta\text{old}}(a_t|s_t)} \tag{6}$$

This ratio compares the likelihood of choosing action a in state s under the current and previous policies. It measures how the policy's preference for a specific action has evolved.

Python Implementation:

```
# Calculate the log probability of actions under the current
↪  policy
log_prob_current_policy = pi.log_prob(samples['actions'])

# Extract the log probability of actions under the old policy
log_prob_old_policy = samples['log_pis']

# Compute the ratio of the old policy over the new policy
ratio = torch.exp(log_prob_current_policy - log_prob_old_policy)
```

The symbol $\hat{A}_t$ symbolizes the advantage at a specific time t.

$$A(s,a) = Q(s,a) - V(s) \tag{7}$$

The value function, $V(s)$, depicts the expected return upon initiating in state s and remaining consistent with the policy:

$$V(s) = \hat{\mathbb{E}}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right] \tag{8}$$

Here, the value of a state embodies the anticipated cumulative rewards while consistently applying the policy from that state, with the discount factor γ modulating the relative significance of immediate versus future rewards. Conversely, $Q(s,a)$ signifies the action-value function, elucidating the expected return from choosing action a in state s and subsequently adhering to the policy. Thus, the advantage $A(s,a)$ quantifies the relative benefit of taking a specific action over merely adhering to the default policy.
Within the PPO algorithm, the advantage is computed using the Generalized Advantage Estimation (GAE) method:

$$\text{GAE}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + ... \tag{9}$$

Where:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{10}$$

This equation incorporates:

- r_t: Reward at time t.
- $V(s_t)$: Critic network's value estimate for state at t.
- γ: Discount factor, influencing the present valuation of future rewards.
- λ: Hyperparameter, balancing bias and variance of advantage estimation.

The product $r_t(\theta) \cdot \hat{A}_t$ in policy gradient methods influences policy updates, steering the policy to favor actions with positive advantages and vice versa.

Python Implementation:

```
gae = 0
for t in reversed(range(len(rewards))):
    # Calculate delta
    delta = rewards[t] + self.gamma * values[t + 1] - values[t]
    # Calculate GAE
    gae = delta + self.gamma * self.lamda * gae
    returns.insert(0, gae + values[t])
# Calculate advantage
adv = np.array(returns) - values[:-1]
```

Traditional policy gradient techniques, like TRPO, optimized a direct "surrogate" objective:

$$L = \hat{\mathbb{E}}_t[r_t(\theta) \cdot \hat{A}_t] \tag{11}$$

However, this approach sometimes induced large shifts in the policy, which could result instability during decision-making. To counteract this PPO introduces a clipping strategy, characterized by a parameter ϵ, commonly set to 0.2. By employing the function $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)$, the policy updates are constrained to ensure that $1 - \epsilon \leq r_t(\theta) \leq 1 + \epsilon$ during loss computation.

Python Implementation

```
clipped_ratio = ratio.clamp(min=1.0 - clip_range, max=1.0 +
↪  clip_range)
clip_loss = torch.min(ratio * sampled_advantages, clipped_ratio
↪  * sampled_advantages)
```

In summary it can be seen The L^{clip} objective in PPO aims to optimize the policy by leveraging the advantage of taken actions, with the clipping mechanism ensuring stability by preventing overly large policy updates that deviate excessively from the previous policy.

4.2 Value Function Loss for the Critic

The Value Function Loss L^{VF} represents the loss evaluation for the critic network. The value function $V^{\pi}(s)$ approximates the expected cumulative reward (or return) from a given state s when following a specific policy. The critic network attempts to model this function $\hat{V}(s)$. The L^{VF} quantifies the difference between the predicted value $\hat{V}(s)$ from the critic network and the actual observed return from that state. A common form for this loss is the Mean Squared Error (MSE) between the predicted values and the observed rewards.

$$L^{\text{VF}} = \mathbb{E}[(\hat{V}(s) - R_t)^2] \tag{12}$$

Where:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{13}$$

The goal during training is to minimize this loss (or the difference between predictions and actual results), which in turn makes the critic's predictions of state values more accurate. Accurate state value predictions are crucial for algorithms like PPO, as they are used in computing advantages, which in turn guide the policy updates.

Python Implementation

```
vf_loss = nn.MSELoss(value, sampled_returns)
```

4.3 Entropy Regularization

The introduction of entropy regularization into the loss function allows the model to explore various actions rather than exploiting the best-known action. This exploration helps in discovering potentially better actions that haven't been tried frequently. The entropy of a policy π quantifies the randomness of the action distributions it produces. A deterministic policy (one that always produces a specific action for a given state) has low entropy, while a completely random policy (equal probability for all actions) has high entropy. The entropy S of a policy π in a state s can be determined using:

$$S(\pi(s)) = -\sum_{a} \pi(a|s) \log \pi(a|s) \tag{14}$$

Entropy regularization is an important way to add randomness and encourage exploration in the early stages of training, encouraging the actor to explore the environment. However, as the policy becomes more confident in selecting actions, the entropy gets lower, leading to less exploration. Entropy can be incorporated using `torches` inbuilt `entropy()` function.

Python Implementation

```
s = pi.entropy()
```

4.4 Reconstructing the Loss Function

Now to revisit the loss function

$$L_t^{\text{CLIP}+V_F+S}(\theta) = E^t \left[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{15}$$

We can also see two coefficients c_1 and c_2, these can be set to determine importance of each element in the loss function. Therefore we can see the combined loss function in PPO is a representation of how our actor performed and is designed to safely improve the policy by taking moderate steps, accurately estimate state values, and maintain a balance between exploration and exploitation.

Python Implementation

```
loss = -clip_loss + 0.5 * vf_loss - 0.01 * s
```

We can see c_1 and c_2 are set to 0.5 and 0.01. The expression is slightly modified in the code as generally in policy gradient methods, the goal is to maximize the expected cumulative reward. The gradient ascent update to increase the expected cumulative reward is equivalent to performing gradient descent on the negative of the reward. Since deep learning libraries are designed to perform gradient descent (i.e., minimize functions), we use the negative of the objective [14].

5 Optimization

Based on tutorials by Adrian Rosebrock and PyTorch [13][15]

After calculating the loss, we can use it to optimize our network using backpropagation. Backpropagation is a supervised learning algorithm used for training artificial neural networks, especially in deep learning models. The core idea is to compute the gradient of the loss function concerning each weight and bias by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule. This gradient indicates how the loss would change if a particular parameter is adjusted by a small amount. A positive gradient suggests that increasing the parameter would lead to an increase in the loss, implying that the parameter should be decreased to minimize the loss. Conversely, a negative gradient suggests that increasing the parameter would reduce the loss. After computing these gradients, an optimization algorithm, the Adaptive Moment Estimation (Adam) algorithm in this case, updates the weights and biases of the network to minimize the loss.

Given a loss function L, the main steps of backpropagation are:

1. **Forward Pass** (sampling): Calculate the output for each neuron from the input layer, through the hidden layers, to the output layer. This will provide the predicted outputs.

2. **Compute the Loss**: Calculate the loss using the predicted outputs and the actual labels.

3. **Backward Pass**: For each node n in the network:

   (a) Calculate the gradient of the loss w.r.t. the output: $\frac{\partial L}{\partial \text{output}_n}$.

   (b) Calculate the output gradient w.r.t. the input: $\frac{\partial \text{output}_n}{\partial \text{input}_n}$.

   (c) Multiply the above values to get the error gradient w.r.t. the input: $\frac{\partial L}{\partial \text{input}_n}$.

   (d) Calculate the input gradient w.r.t. each weight by using the output values of the previous layer: $\frac{\partial \text{input}_n}{\partial w}$.

   (e) Use the chain rule to compute the gradient of the loss with respect to each weight: $\frac{\partial L}{\partial w}$.

4. **Weight Update**: Update the weights in the direction that decreases the loss.

Mathematically, using the chain rule, the gradient of the loss L concerning a weight w in the network is given by:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \text{output}_n} \times \frac{\partial \text{output}_n}{\partial \text{input}_n} \times \frac{\partial \text{input}_n}{\partial w}$$

Where:

- $\frac{\partial L}{\partial \text{output}_n}$ is the gradient of the loss concerning the output of neuron n.
- $\frac{\partial \text{output}_n}{\partial \text{input}_n}$ is the gradient of the neuron's output concerning its input. This is determined by the derivative of the activation function used in neuron n.
- $\frac{\partial \text{input}_n}{\partial w}$ is the gradient of the neuron's input concerning the weight w.

The weight and biases can then be adjusted using an optimization algorithm like Adam.

Python Implementation

```
# Setting the optimizer and learning rates for each network
optimizer = torch.optim.Adam([
    {'params': self.policy.actor.parameters(), 'lr': lr_actor},
    {'params': self.policy.critic.parameters(), 'lr': lr_critic}
], eps=1e-4)
# Function to calculate loss as explained above
loss = calculate_loss(clip_range=clip_range, samples=mini_batch)
optimizer.zero_grad()
# Perform backpropagation
```

```
loss.backward()
# Optimize
optimizer.step()
```

Note that `zero_grad()` clears the gradients of all optimized tensors to prepare for a new optimization step.

One of the distinguishing features of PPO's training methodology is the use of **mini-batches**. After collecting trajectories by interacting with the environment for a certain number of timesteps, these trajectories are divided into **mini-batches** and then shuffled. Training on these mini-batches offers several benefits:

- **Computational Efficiency**: Mini-batch training can be more computationally efficient, especially when leveraging the parallel processing capabilities of modern hardware.
- **Memory Efficiency**: By processing data in smaller chunks, mini-batches can be more memory-friendly.
- **Generalization**: Shuffling the mini-batches ensures the model doesn't see the same sequences repeatedly, promoting better generalization and preventing overfitting to specific sequences.

Python Implementation

```
# Randomize indexes
indexes = torch.randperm(batch_size)
# Loop through mini batches
for start in range(0, batch_size, mini_batch_size):
    mini_batch_indexes = indexes[start: start + mini_batch_size]
    mini_batch = {}
    # Perform optimization here
```

In the context of Proximal Policy Optimization (PPO), maintaining a stable training process is of also very important. To address this, PPO introduces a technique of maintaining two separate copies of the policy network: the current policy π and the old policy π_{old}. The old policy is used to evaluate the importance sampling ratio, ensuring the updated policy doesn't deviate drastically from the old one. After each update, the weights of the current policy are copied over to the old policy, ensuring it remains a lagged version of the current policy. This mechanism helps to strike a balance between exploring new strategies and exploiting what's already known, making the training process more stable and robust. `policy_old.load_state_dict(policy.state_dict())`, is used in the code to load the current policy's weights into the old policy [14].

6 Performance

Training an RL model to tackle intricate problem, like playing a game, is often a lengthy task. In my experience, the model required training for thousands of iterations to achieve consistent success in completing levels. One potential criticism of PPO pertains to the sheer number of hyperparameters available for tuning. The abundance of these parameters can necessitate extensive experimentation, thereby elongating the training process.

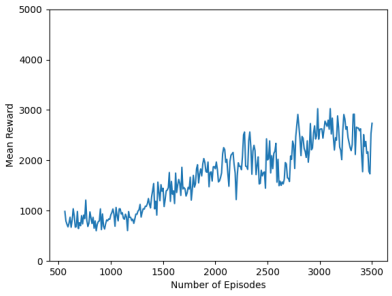


Figure 2: Level 1

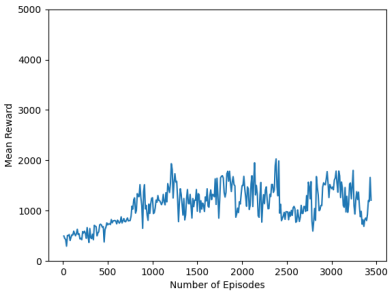


Figure 3: Level 2

The graphs illustrating the evolution of rewards for levels 1 and 2 depict an upward trend over 5,000 episodes. Notably, the onset of rewards for level 2 demonstrates a more gradual growth compared to level 1. This discrepancy can be attributed to the unique challenges the agent encounters in level 2, such as awaiting moving platforms or biding time for enemies to cross so Mario can leap onto them. Given that our action space is solely confined to `["right"]` and `["A", "right"]`, the agent lacks the capability to wait or move backwards. Even though the model eventually devised strategies to surmount these obstacles (like leaping against a wall to decelerate), this highlights a potential constraint arising from a narrowed action space.

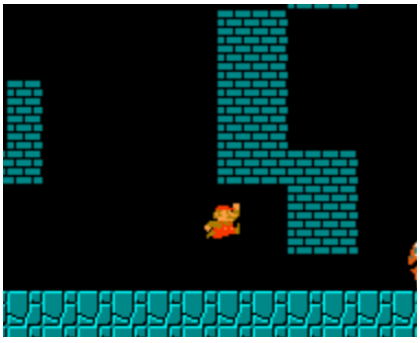


Figure 4: Mario jumping into a wall

7 Conclusion

In this report, we delved into the Proximal Policy Optimization (PPO) algorithm using Super Mario Bros as a testbed. This hands-on approach helps to breakdown PPO's core concepts, emphasizing the actor-critic framework and PPO's stability-centric loss function. Super Mario Bros showcases PPO's effectiveness, highlighting the reward system that propels Mario to advance efficiently. Essential Python libraries used to implement PPO were also discussed.

References

[1] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[2] Open AI. Proximal policy optimization. `https://openai.com/research/openai-baselines-ppo`, 2023.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Kauten Christian. gym-super-mario-bros. `https://github.com/Kautenja/gym-super-mario-bros/tree/master`, 2023.

[5] Sheikh Amir Fayaz, S Jahangeer Sidiq, Majid Zaman, and Muheet Ahmed Butt. Machine learning: An introduction to reinforcement learning. *Machine Learning and Data Science: Fundamentals and Applications*, pages 1–22, 2022.

[6] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1291–1307, 2012.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[8] Sonali B Maind, Priyanka Wankar, et al. Research paper on basic of artificial neural network. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2(1):96–100, 2014.

[9] Abhishek Nandy, Manisha Biswas, Abhishek Nandy, and Manisha Biswas. Reinforcement learning basics. *Reinforcement Learning: With Open AI, TensorFlow and Keras Using Python*, pages 1–18, 2018.

[10] OpenAI. Roboschool. `https://openai.com/research/roboschool`.

[11] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[13] Adrian Rosebrock. Backpropagation from scratch with python. `https://pyimagesearch.com/2021/05/06/backpropagation-from-scratch-with-python/`, 2021.

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

[15] Torch. A gentle introduction to torch.autograd. `https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html`.

[16] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[17] Howard Wang Steven Guo Yuansong Feng, Suraj Subramanian. Train a mario-playing rl agent. `https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html`, 2020.