ML14
MARIO PLAYING AI

Aditya Kshitiz
Aviral Sharma
Misha Jain
Sai Sanjana Reddy Algubelly

Mentors:
Yatharth Gupta
Atharva Mohite

PROBLEM STATEMENT

Our task is to utilize reinforcement learning (RL), to develop an agent capable of autonomously playing Super Mario by learning from interactions with the game environment.

We have built a model that can play different levels at the minimum gameplay time.

TECH STACKS USED

- ML Libraries like Pytorch, Numpy, Matplotlib.
- OpenAI gym for RL Environment.
- Neat-python library for NEAT algorithm.
- Web dev tech stack: HTML, CSS, Javascript.

WHAT
WE DID SO FAR?

- We tried different algorithms: SARSA, DQN, DDQN, Dueling DQN, NEAT, PPO.
- Trained model on different levels using PPO.
- Built a website to showcase our work.

SARSA Algorithm

The SARSA algorithm is based on the Q-learning method, but instead of using the maximum Q-value of the next state (as in Q-learning), SARSA uses the action that the agent actually takes in the next state. The updated rule for SARSA is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

This algorithm required q-table and therefore, couldn't work for the case of Mario.

Deep Q Learning Algorithm

DQN combines RL with deep neural networks to handle high dimensional input spaces . The main idea behind is to use CNN, to estimate the Q values, Which represent expected cumulative reward for taking a particular action.

$$TD_e = Q^*_{online}(s, a)$$

$$a' = argmax_a Q_{online}(s', a)$$

$$TD_t = r + \gamma Q^*_{online}(s', a')$$

For both target and online values, DQN is using same network therefore it can create overestimation of some Q values and the solution for this is Double DQN.

Double Deep Q Learning Algorithm

DDQN is an extension of DQN, to address the problem of overestimation bias. During training, the Q network can sometimes overestimate the Q values, DDQN aims to reduce this by introducing a "target" network that is separate from main Q network.

$$TD_e = Q^*_{online}(s, a)$$

$$a' = argmax_a Q_{online}(s', a)$$

$$TD_t = r + \gamma Q^*_{online}(s', a')$$

By using the target network, DDQN can improve the stability and performance of Q learning.

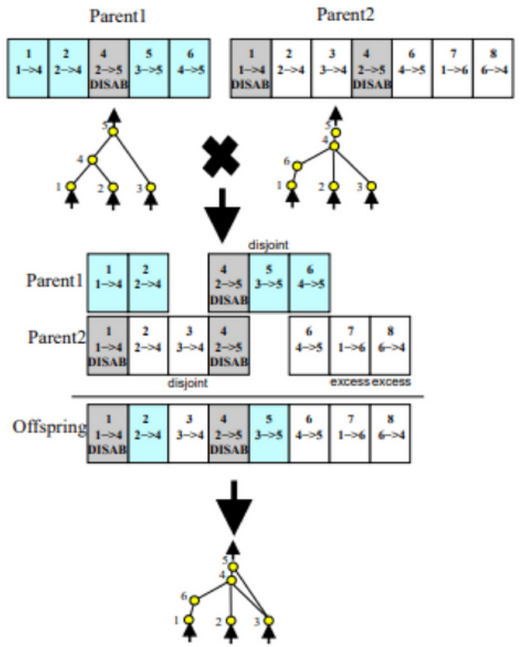
NEAT

NEAT(Neuro Evolution ofAugmenting Topologies) is a popular genetic algorithm-based technique used for evolving artificial neural networks.

NEAT represents a neural network as a "genome", which consists of collection between nodes in the neural network.

NEAT employee various mechanisms for structural innovation. New connections and nodes can be added to the neural network during the evolution process, enabling the network to adapt its architecture over time.

A fitness function is used to evaluate the performance of each individual in the population.In reinforcement learning, the fitness function is based on the agent's performance in the environment



WHY
PROXIMAL POLICY OPTIMIZATION

PPO is an improved version of policy gradient methods that use conservative parameter updates through clipping. Normally,

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \ln \pi_\theta(a|s)]$$

However, such an update rule might result in overshooting from parameter values.

WORKING RULE

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

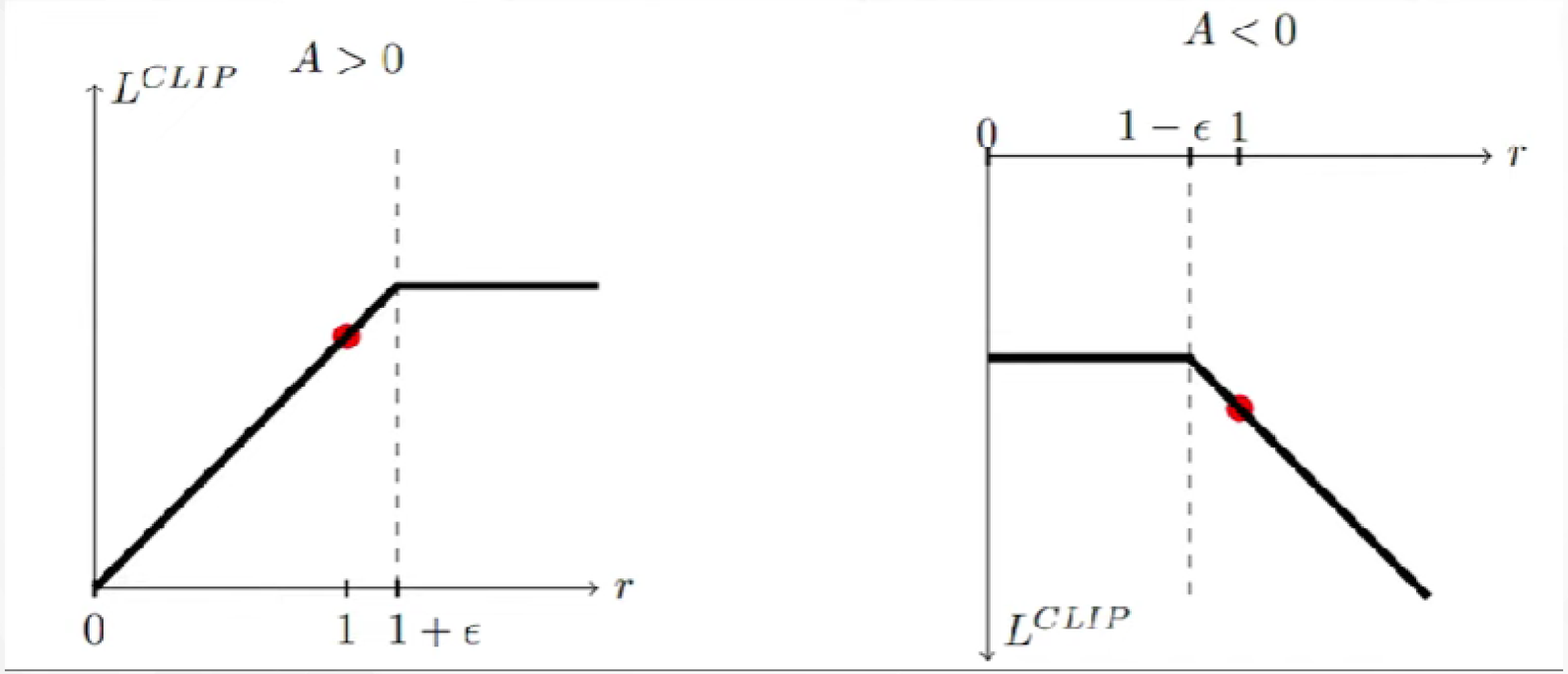
$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

CLIPPED SURROGATE OBJECTIVE



$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}) \right] \right]$$

OTHER HIGHLIGHTS

Calculate how much the policy has changed -> $ratio = \pi_{new} / \pi_{old}$

Express in log form -> $ratio = [\ log(\pi_{new}) - log(\pi_{old})\ ].exp()$

Calculate Actor loss as minimum of two functions ->

$p1 = ratio \cdot advantage$

$p2 = clip(ratio, 1-\varepsilon, 1+\varepsilon) \cdot advantage$ where $\varepsilon=0.2$ is clipping range

$actor\_loss = min(p1,p2)$

Calculate Critic loss as MSE between Returns and Critic value

$critic\_loss = (\ R - V(s)\ )^2$

Calculate Total loss

$total\_loss = critic\_loss \cdot critic\_discount + actor\_loss - entropy$

During the learning phase of the vectorized architecture, the PPO implementation shuffles the indices of the training data of size N∗M and breaks it into mini-batches to compute the gradient and update the policy.

PPO sets the epsilon parameter to 1e-5, which is different from the default epsilon of 1e-8 in PyTorch and 1e-7 in TensorFlow

In PPO, orthogonal initialization outperforms the default Xavier initialization in terms of the highest episodic return achieved.

PPO implements the return target as returns = advantages + values, which corresponds to TD(λ) for value estimation.

THANK YOU