Exploratory Analysis of Reinforcement Learning Algorithms: PPO, A2C, and DQN in Cart-Pole Environment

By: Kundan Kumar, Adepeju Opaleye & Eriyon Adams

Wayne State University

Abstract:

This paper presents the implementation details of different reinforcement learning (RL) algorithms used to manage a Cart-Pole system. This paper investigates the effectiveness of three well-known reinforcement learning (RL) algorithms—Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and Deep Q-Network (DQN)—in the CartPole-v1 environment, which is a widely used benchmark in RL. We employ a technique that consists of executing and instructing each algorithm in this specific setting, subsequently conducting a thorough assessment of their performance, which is determined by the stability and control exhibited in the pole balancing challenge. The results uncover clear behavioral patterns and effectiveness levels for each algorithm, with PPO, A2C, and DQN displaying different capacities in preserving the pole's equilibrium. The project closes by providing a comparison study, which is visually depicted using bar charts and displayed on tensorboard. This analysis highlights the average rewards and standard deviations attained by each method. This comparison offers valuable insights into the algorithm's comparative advantages and limits, providing a detailed knowledge of their suitability in reinforcement learning tasks and leading to well-informed judgments in the field of AI and machine learning.

Background:

In the Reinforcement Learning (RL) paradigm of machine learning, an agent gains knowledge on the best course of action for a task by repeatedly interacting with a dynamic environment that either rewards or punishes the agent for its actions. A semi-supervised learning strategy would be reinforcement learning, in which the environment provides incentives that serve as an indirect source of the supervision signal needed to train the model. Reinforcement learning is more appropriate for learning the dynamic behavior of an agent interacting with its environment than it is for learning static mappings between two sets of input and output variables. With varying degrees of success, several reinforcement learning architectures and methodologies have been released throughout the years. But the success of deep learning algorithms recently has revived reinforcement learning and attracted the attention of scholars, who are now successfully applying this to solve exceedingly complex problems that were previously believed to be intractable. Events like **IBM Watson** winning the Jeopardy game or artificial agents like **AlphaGo** defeating world champion **Lee Sedol** have brought attention to the rise of artificial intelligence, which may soon surpass human cognition. A crucial paradigm for creating intelligent systems that can accumulate knowledge through time is reinforcement learning. The fields of robotics, healthcare, recommender systems, data centers, smart grids, financial markets, and transportation are now seeing an increase in the use of reinforcement algorithms.

Three well-known reinforcement learning (RL) algorithms—Deep Q-Network (DQN), Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO)—within the **CartPole-v1** environment. The goal is to offer a useful manual for applying various aspects of reinforcement learning using OpenAI/Gym, and Python. It will be helpful to scholars and students who are interested in this field. Theoretical details and a mathematical exploration of these concepts have been omitted to make this work concise. Rather, readers are referred to the relevant literature for a greater understanding of these concepts.

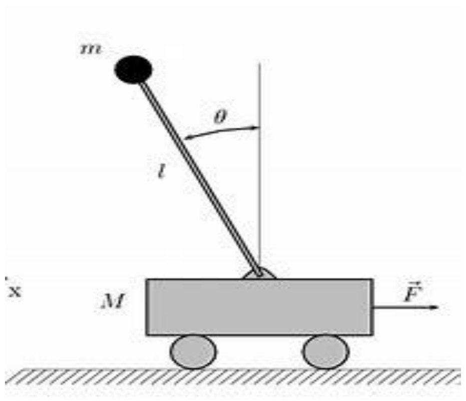


Fig:1- Algorithms for control engineering and reinforcement learning are evaluated in this conventional control environment.

METHODOLOGY:

This is how the remainder of the paper is structured. The next part contains a variety of deep learning ideas along with specifics on how they were implemented.

The System

OpenAI Gym is used to replicate the **Cart-Pole system**. It is constructed from a black cart with a passive pivot joint holding a vertical bar in place shown in Fig2. The wagon may move to the left and right. Getting the car to go left or right without allowing the vertical bar to collapse is the issue. The random action policy may cause the system to behave animatedly if the **code-1** is run. The state vector for system x is a four-dimensional vector with components **{x,x,θ,˙ θ˙}**. The action exists in two states: **left (0)** and **right (1).** A cart position more than **±2.4 cm** from the center, a pole angle greater than **±12**◦ from the **vertical axis**, or an **episode** length greater than **10** are the three conditions that signal the end of the **episode**. The agent receives **one reward** for every action, including the last one. The problem is declared resolved if the average reward is greater than or equal to **195**.

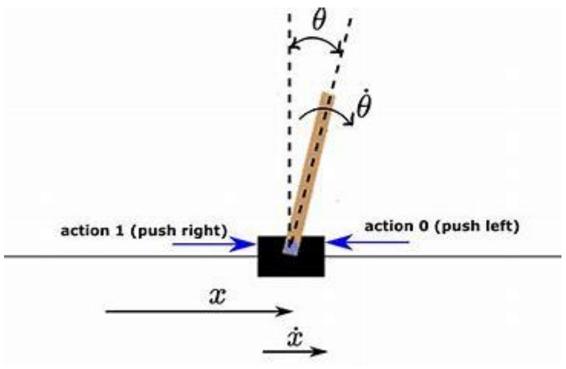


Fig2 -The objective is to balance the pole (pole upright) by applying forces in the left and right direction on the cart for as long as possible.

Code 1:

```
Code Visualize Cartpole Animation
import gym env
environment_name = "CartPole-v1"
env = gym.make(environment_name, render_mode='human')
episodes = 10
scores = []  # List to reward  scores

for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()  # Render the environment
        action = env.action_space.sample()  # This randomly generates an
action that the agent takes
        n_state, reward, done, _ = env.step(action)[:4]

        score += reward

    scores.append(score)
```

The steps follow an outlined process for experimenting with different reinforcement learning algorithms in a virtual environment:

Step 1: Install all required dependencies from stablelines3, gymnasium, Tensorboard, Rise and python library.

Step 2: Run a test experiment:

a) Visualize an untrained agent's behavior in a virtual environment.
b) Show a graphical evaluation of the performance of the untrained agent.

Step 3: Train An Agent (With Proximal Policy Optimization algorithm)

a) Visualize the trained agent's behavior in a virtual environment.
b) Show a graphical evaluation of the performance of the trained PPO agent.
c) Validate agent performance.

Step 4: Train An Agent (With Advantage Actor-Critic)

a) Visualize the trained agent's behavior in a virtual environment.
b) Show a graphical evaluation of the performance of the trained A2C agent.
c) Validate agent performance.

Step 5: Train An Agent (With Deep Q-Network)

a) Visualize the trained agent's behavior in a virtual environment.
b) Show a graphical evaluation of the performance of the trained DQN agent.
c) Validate agent performance.

Step 6: Compared performance across experimented algorithms.

A. Proximal Policy Optimization Algorithms

An estimate of the policy gradient is computed and then plugged into a stochastic gradient ascent algorithm to operate on policy gradient techniques. The gradient estimator with the most use has the form.

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

where At is an estimation of the advantage function at timestep and π_θ is a stochastic strategy. In this instance, an algorithm that alternates between sampling and optimization uses the expectation Et [...] to represent the empirical average across a batch of data. When automated differentiation software is used, implementations create an objective function whose gradient equals the policy. gradient estimator: by differentiating the goal, the estimator g^ is produced.

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \Big[\log \pi_\theta(a_t \mid s_t) \hat{A}_t \Big].$$

While it is appealing to perform multiple steps of optimization on this loss LPG using the same trajectory, doing so is not well-justified, and empirically it often leads to destructively large policy updates.

code to initialize PPO algorithm.

```
# Train the PPO model with TensorBoard logging
total_timesteps_ppo = 10000
ppo_callback = EvalCallback(
    eval_freq=1000,
    best_model_save_path=os.path.join(log_path, 'best_model_ppo'),
    log_path=log_path,
    eval_env=env,
    n_eval_episodes=5,
    deterministic=True
)
model_ppo.learn(total_timesteps=total_timesteps_ppo, callback=ppo_callback,
  tb_log_name='ppo')


# Close the environment
env.close()


# Close TensorBoard writer
writer.close()
```

Algorithm

The surrogate losses from the previous sections may be calculated and distinguished by making a slight modification to a standard policy gradient implementation. When utilizing automated differentiation, the approach involves creating the loss function L^{CLIP} or L^{KLPEN} instead of L^{PG}, and then executing numerous iterations of stochastic gradient ascent to optimize this goal.

To enhance this aim, we may incorporate an entropy bonus to ensure enough investigation. By combining these elements, we get the following objective, which is almost maximized in each iteration:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right], \qquad (9)$$

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

A particular approach to implementing policy gradient, which gained popularity in [11] and is particularly suitable for recurrent neural networks, involves running the policy for a limited number of timesteps (T) that is significantly less than the length of the episode. The gathered samples are then used to perform an update. This approach necessitates the use of an advantage estimator that does not consider any information beyond timestep T. The estimator employed by [11] is

$$\hat{A}_t = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t+1} r_{T-1} + \gamma^{T-t} V(s_T) \qquad (10)$$

t represents the time index inside a trajectory segment of length T, where t is a value between 0 and T. To generalize this decision, we may employ a condensed form of generalized benefit estimation, which simplifies to Equation for λ=1.

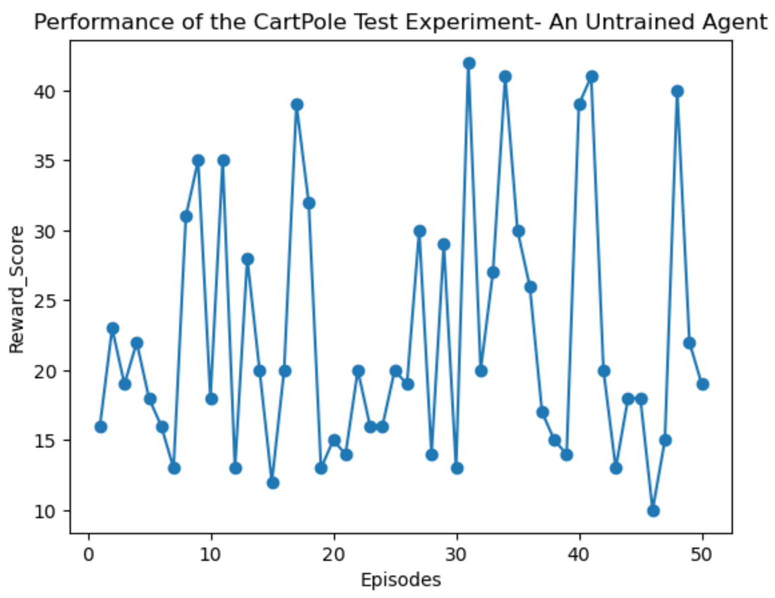
$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1}, \tag{11}$$

$$\text{where} \quad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{12}$$

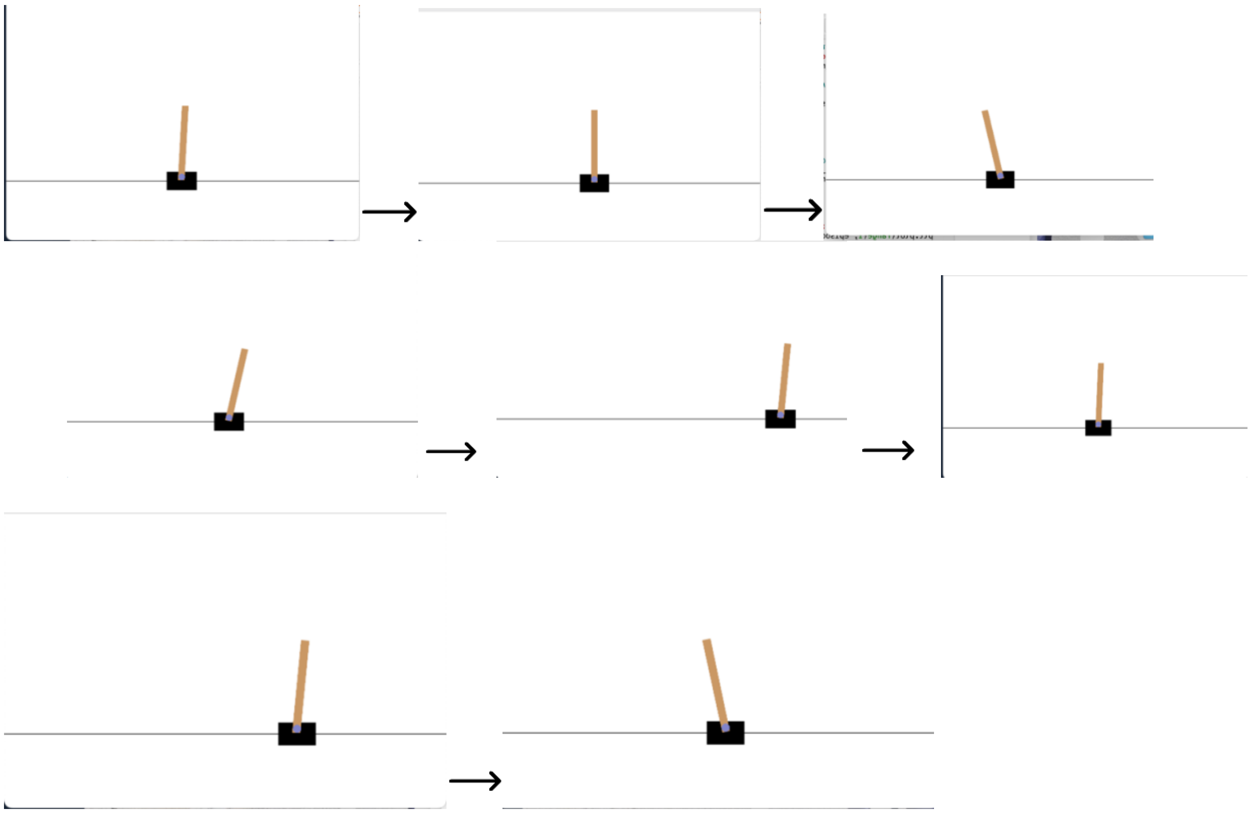
Algorithm 1 PPO, Actor-Critic Style

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., $N$ do
        Run policy $\pi_{\theta_{\text{old}}}$ in environment for $T$ timesteps
        Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \le NT$
    $\theta_{\text{old}} \leftarrow \theta$
end for
```

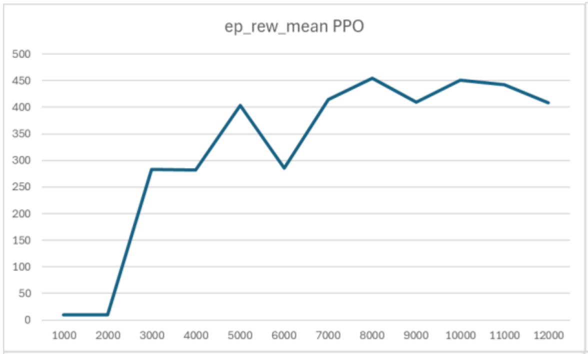
Visualize the untrained agent's behavior in cartpole environment for 50 Episodes.

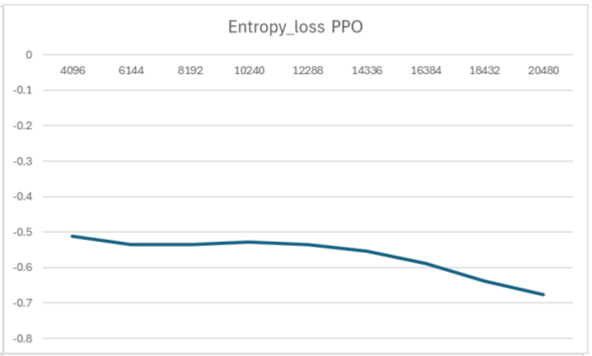


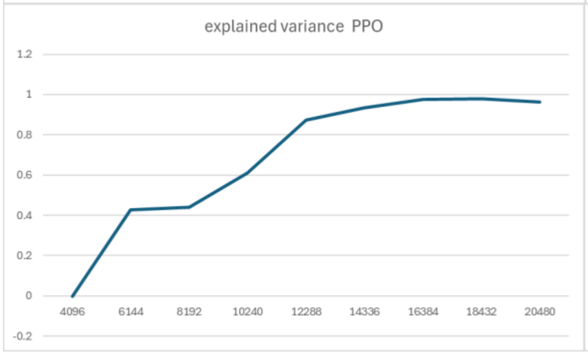
<u>Visualize the trained agent's behavior in the environment & in tensorboard.</u>

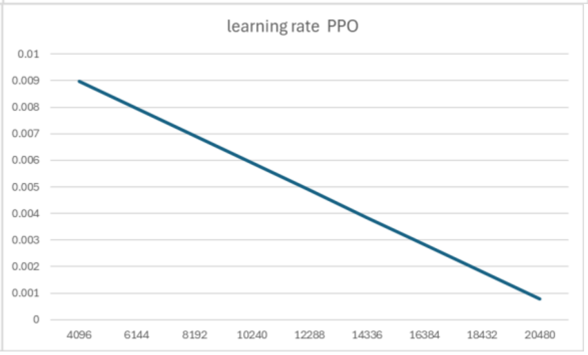


```
Mean Reward: 291.8, Std Reward: 91.93671736580549
```









B. ***Advantage Actor-Critic (A2C) Algorithms:***

Policy-based techniques aim to optimize the policy directly, without relying on a value function. To be more precise, the term "reinforce" falls within the category of Policy-Gradient methods, which is a subclass of Policy-Based techniques. This subclass use Gradient Ascent to estimate the weights of the optimal policy, therefore directly optimizing the policy.

Reinforcement shown efficacy. Nevertheless, there is a significant disparity in the estimates of policy gradient due to the utilization of Monte-Carlo sampling, which involves the computation of the return utilizing an entire episode.

The policy gradient estimation corresponds to the direction of the highest increase in return. In other words, how can we modify the weights in our policies to enhance the probability of undertaking behaviors that provide favorable outcomes? In this lesson, we will go further into the concept of Monte Carlo variance and its impact on the training process. It is worth noting that the Monte Carlo variance leads to slower training due to the necessity of reducing many samples.

In this course, we will explore Actor-Critic approaches, which are a combination of value-based and policy-based methods. This hybrid architecture is designed to enhance training stability by lowering variation.

- A policy-based technique employs an actor to govern the behavior of our agent (Policy-based method)
- A critic evaluates the quality of an actor using a value-based system (Value-Based method)

We will examine one of these hybrid techniques, known as Advantage Actor Critic (A2C), and proce ed to train our agent using Stable-Baselines3 in cartpole situations. We will train a Cartpole to mainta in its alignment.

During the forward pass, the model will receive the state as input and provide action probabilities an d critic value V, which represents the state-dependent value function. The objective is to develop a model that selects actions based on a policy π that maximizes the anticipated reward.

The state representation for CartPole-v0 consists of four variables: cart location, cart velocity, pole a ngle, and pole velocity. The agent has the option to do two actions: pushing the cart to the left (0) or pushing it to the right (1).

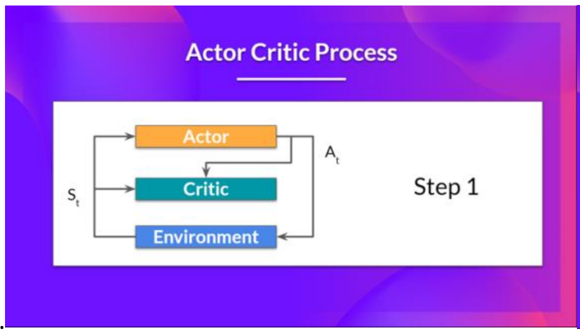
- Code to Initialize A2C Model:

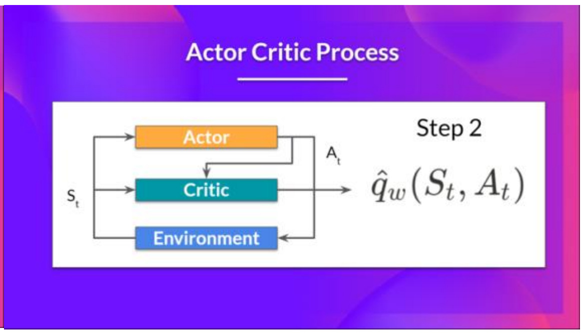
```
# Create a Gym environment
env = gym.make('CartPole-v1', render_mode='human')

# Define the initial learning rate
initial_learning_rate = 0.01

# Create a learning rate schedule using the scheduler
learning_rate_schedule = linear_schedule(initial_learning_rate)

# Initialize TensorBoard writer
```

```
  log_path = os.path.join('Training', 'Logs')
  writer = SummaryWriter(log_dir=log_path)

  # Initialize the A2C model with the custom learning rate schedule and TensorBoard logging
  model_a2c = A2C("MlpPolicy", env, verbose=1, learning_rate=learning_rate_schedule,
tensorboard_log=log_path)

  # Train the A2C model with TensorBoard logging
  total_timesteps_a2c = 10000
  a2c_callback = EvalCallback(
    eval_freq=1000,
    best_model_save_path=os.path.join(log_path, 'best_model_a2c'),
    log_path=log_path,
    eval_env=env,
    n_eval_episodes=5,
    deterministic=True
  )
  model_a2c.learn(total_timesteps=total_timesteps_a2c, callback=a2c_callback, tb_log_name='a2c')

  # Close the environment
  env.close()

  # Close TensorBoard writer
writer.close()
```

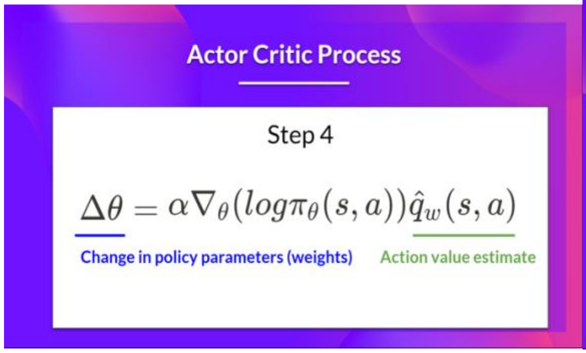
Let us examine the training procedure to comprehend how the Actor and Critic are optimized:

- At each time step, denoted as t, we get the current state St from the environment and transmit it as input through our Actor and Critic models.
- Our Policy takes the state and outputs an action A_t.





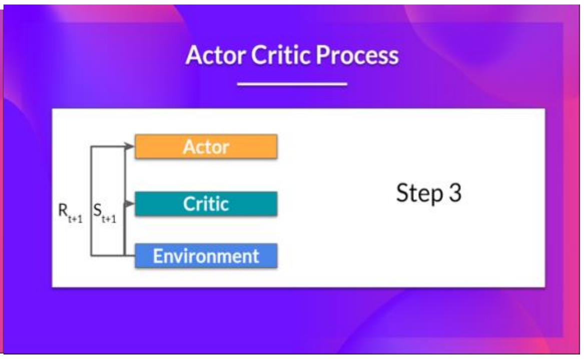
.



Actor Critic Process

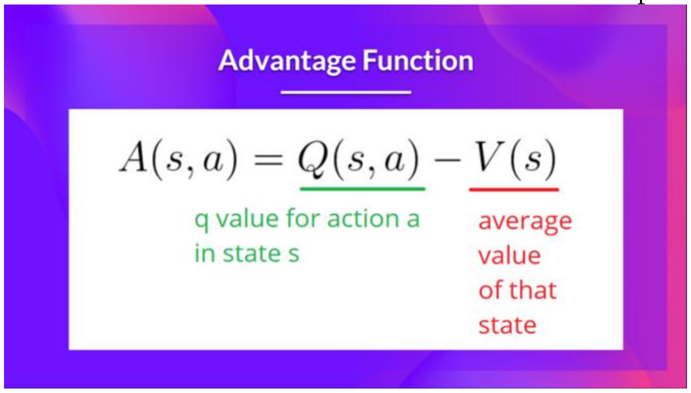
Step 4

$$\underbrace{\Delta\theta}_{\text{Change in policy parameters (weights)}} = \alpha\nabla_\theta(log\pi_\theta(s,a))\underbrace{\hat{q}_w(s,a)}_{\text{Action value estimate}}$$



Adding Advantage in Actor-Critic (A2C)

To enhance the stability of learning, we might utilize the Advantage function as the Critic instead of the Action value function.
The concept revolves on the benefit function, which computes the comparative benefit of an action in relation to other potential actions at a given state. It determines the extent to which doing a particular action at a state is superior to the average value of that state. The process involves calculating the difference between the mean value of the state and the state-action pair.



Advantage Function

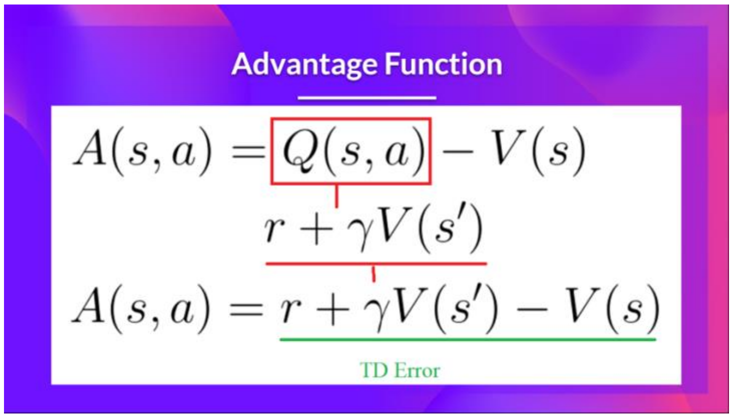
$$A(s,a) = \underbrace{Q(s,a)}_{\text{q value for action a in state s}} - \underbrace{V(s)}_{\text{average value of that state}}$$

In other words, this function calculates the incremental gain we would obtain by executing this action at that particular state, as compared to the average reward we would earn at that state.

The increased incentive surpasses the expected value set by the state

a) Our gradient is pushed in that direction **if A(s,a) > 0**.
b) Our gradient gets pushed the other way **if A(s,a) < 0** (our action performs worse than the average value of that state).

To construct this advantage function, two value functions, **V(s)** and **Q(s,a),** are required, which poses a difficulty. Fortunately, the TD error acts as a dependable estimate of the advantage function.



Advantage Function

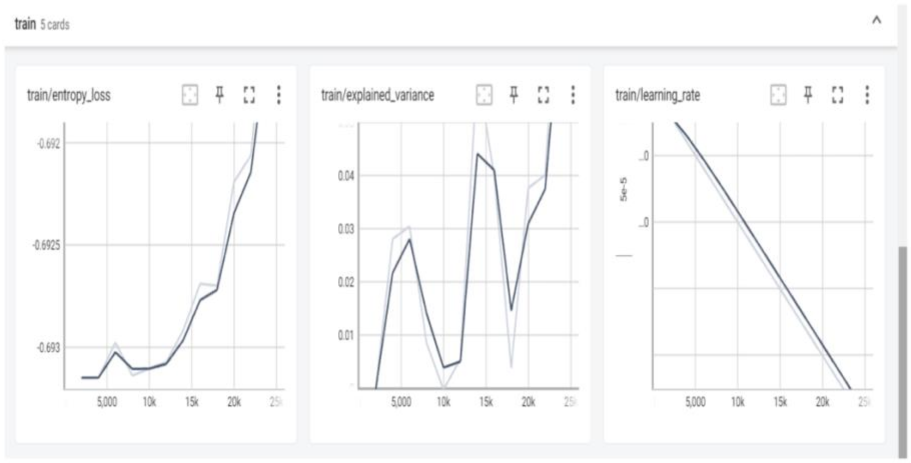
$$A(s,a) = \underbrace{\boxed{Q(s,a)}}_{r + \gamma V(s')} - V(s)$$

$$A(s,a) = \underbrace{r + \gamma V(s') - V(s)}_{\text{TD Error}}$$

Visualize the Model in Tensorboard and trained agent's behavior in cart-pole environment:

Figure: 1B

Entropy loss Explained Variance Learning rate
(Inconsistent with what we expected output for a learning agent)



Learning rate : 0.001
Timesteps: 25000

Figure: 1A

ep_len_mean : Mean episode length (averaged over episodes, 100 by default)

ep_rew_mean : Mean episodic training reward (averaged over episodes, 100 by default)

The mean cumulative episode reward over all agents. Should increase during a successful training session. The general trend in reward should consistently increase over time. Small ups and downs are to be expected. Depending on the complexity of the task, a significant increase in reward may not present itself until millions of steps into the training process.

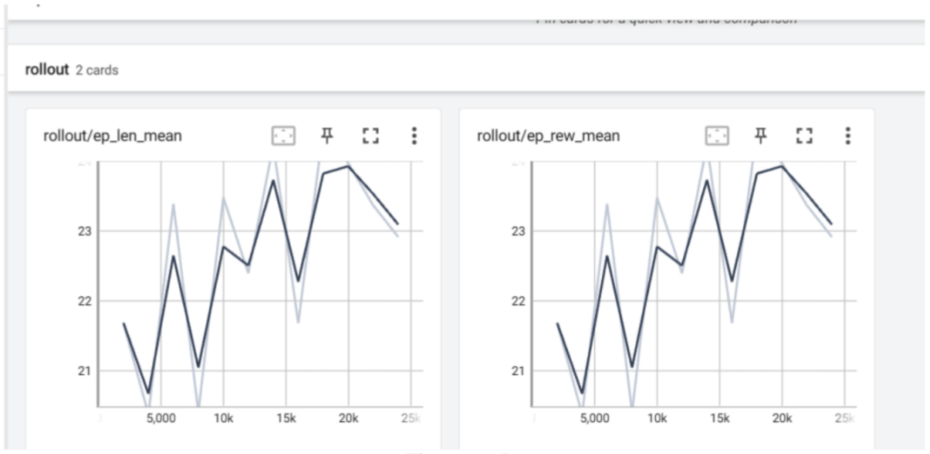


Figure: 1B

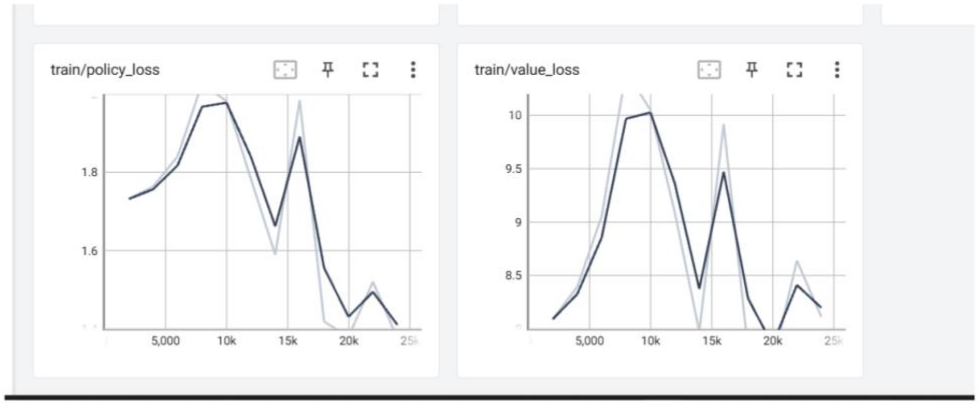
Entropy loss Explained Variance Learning rate
(Inconsistent with what we expected output for a learning agent)

Figure: 1C

Policy loss **Value loss**

(Inconsistent with what we expected output for a learning agent)

Loss function correlates to how well the model can predict the value of each state. This should increase while the agent is learning, and then decrease once the reward stabilizes. When the model is stabilised, t*hese values should increase as the reward increases and then should decrease once reward becomes stable. (There was no stability in the mean reward, check 1)*



Learning rate : 0.001
Timesteps: 5000

Figure: 2A
Sharp increase in mean episode length and reward but stable

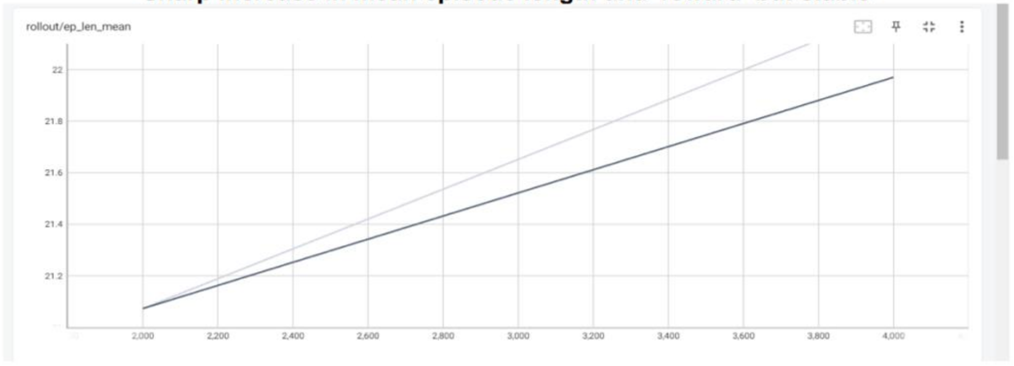


Figure: 2A

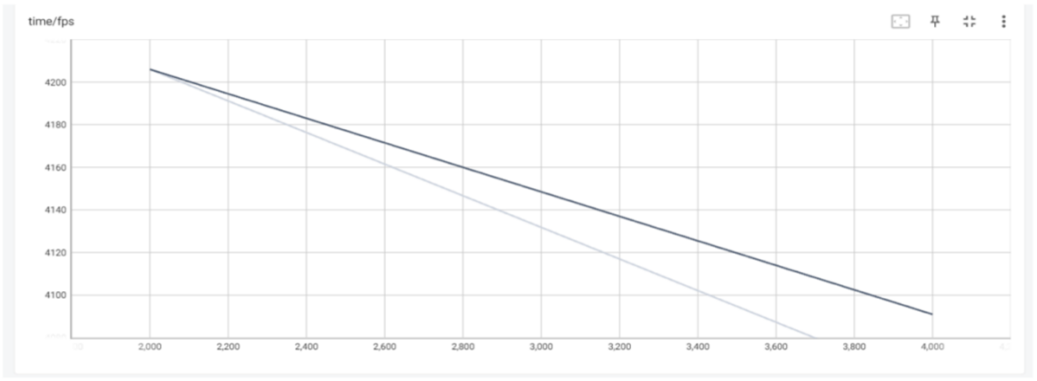


Figure: 2C : Loss Function
As expected but a very sharp decline was observed

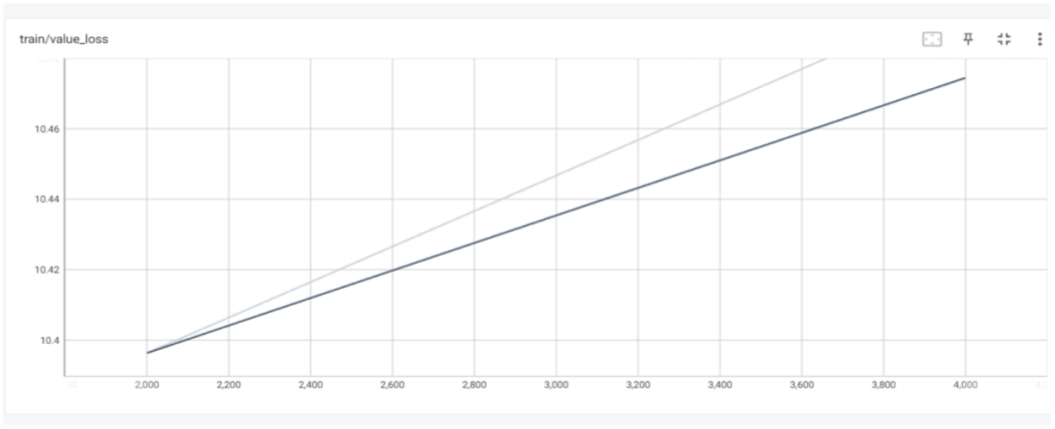


Figure: 2B : Entropy loss
Entropy (degree of disorderliness), sharply decline was observed

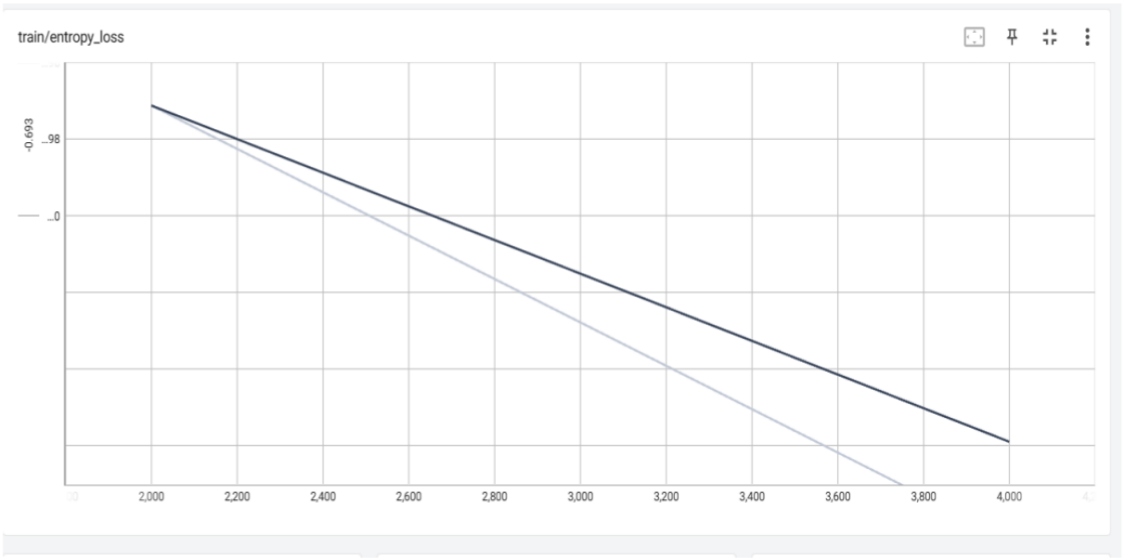


Figure: 2B : Explained Variance
As expected but a very sharp increase was observed

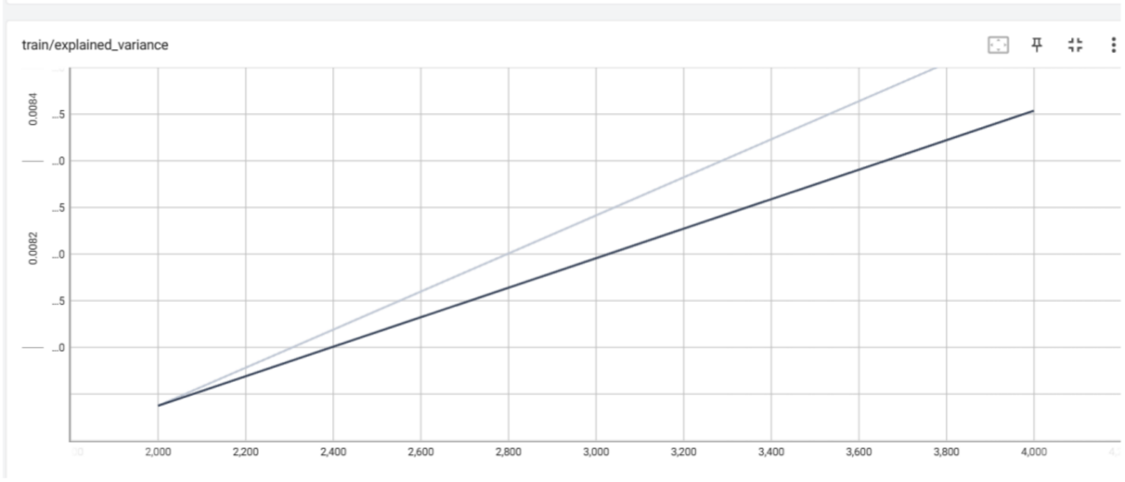


Figure: 2B : Learning Rate
As expected but a very sharp decline was observed

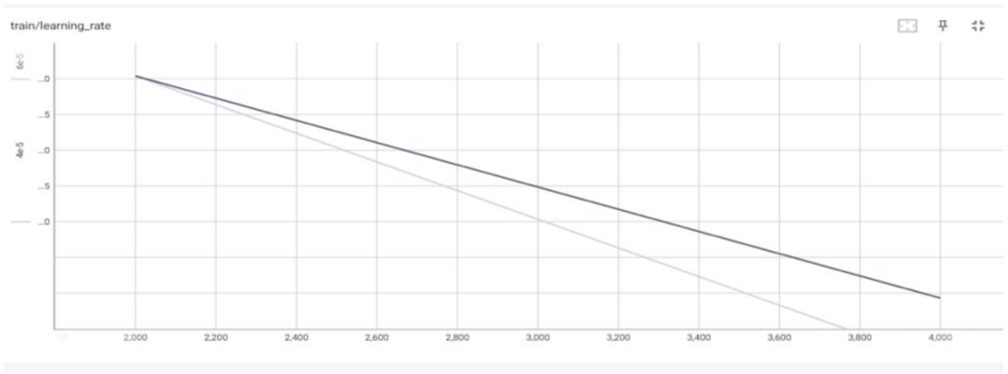
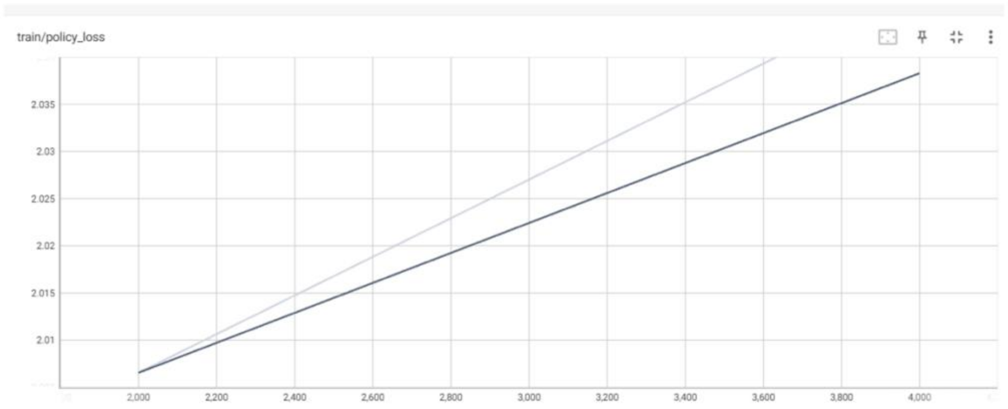


Figure: 2C : Loss Function
As expected but a very sharp decline was observed (Refer to 1C)



Mean Reward: 32.8, Std Reward: 9.877246579892597

C. *Deep Q Learning (DQN) Algorithm:*

Our environment is deterministic, so all equations presented here are also formulated deterministically for the sake of simplicity. Within the reinforcement learning literature, these expectations would also encompass probabilistic shifts in the environment.
Our aim will be to train a policy that tries to maximize the discounted, cumulative reward $\mathbf{R}_{to} = \sum_{t=t0}^{\infty} \gamma \hat{} (\mathbf{t} - \mathbf{to}).\mathbf{r}_t$, where $\mathbf{R}_{to}$ is also known as the return. The discount, **γ,** must be a constant ranging from 0 to 1 to guarantee the convergence of the total. A smaller value of **γ** diminishes the significance of benefits from the distant and unknown future for our agent, in comparison to the rewards that can be confidently anticipated soon. It also motivates agents to prioritize collecting rewards that are closer in time over benefits that are temporally distant in the future.

The fundamental concept underlying Q-learning is the existence of a function **Q*: State × Action →R**, which provides information about the expected return when doing a certain action in a particular state. By leveraging this function, we may straightforwardly devise a policy that maximizes our reward.

$$\pi*(s)=\arg_a\max Q*(s,a)$$

Nevertheless, our knowledge of the world is incomplete, hence we lack access to **Q∗**. However, since neural networks could approximate any function, we can easily construct one and train it to mimic **Q∗**.
Our training update rule will be because every Q function, under a given policy, follows the Bellman equation.

$$Q^\pi(s,a)=r + \gamma Q^\pi(s',\pi(s'))$$

To reduce this mistake, we shall employ the Huber loss function. The Huber loss has characteristics like the mean squared error for small errors but resembles the mean absolute error for big errors. This property enhances its resilience against outliers in situations when the estimates of Q are highly erratic. The calculation is performed on a batch of transitions, B, which is sampled from the replay memory.

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s,a,s',r)\,\in\,B} \mathcal{L}(\delta)$$

$$\text{where} \quad \mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \le 1, \\ |\delta| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

Code to initialize DQN:

```
# Create a Gym environment
env = gym.make('CartPole-v1', render_mode='human')

# Define the initial learning rate
initial_learning_rate = 0.01

# Create a learning rate schedule using the scheduler
learning_rate_schedule = linear_schedule(initial_learning_rate)

# Initialize TensorBoard writer
log_path = os.path.join('Training', 'Logs')
writer = SummaryWriter(log_dir=log_path3)

# Initialize the DQN model with the custom learning rate schedule and TensorBoard logging
model_dqn = DQN("MlpPolicy", env, verbose=1, learning_rate=learning_rate_schedule,
tensorboard_log=log_path)

# Train the DQN model with TensorBoard logging
total_timesteps_dqn = 10000
```

```
dqn_callback = EvalCallback(
    eval_freq=1000,
    best_model_save_path=os.path.join(log_path, 'best_model_dqn'),
    log_path=log_path,
    eval_env=env,
    n_eval_episodes=5,
    deterministic=True
)
model_dqn.learn(total_timesteps=total_timesteps_dqn, callback=dqn_callback, tb_log_name='dqn')

# Make predictions with the trained A2C model
obs = env.reset()
action_dqn, _ = model_dqn.predict(obs)

# Close the environment
env.close()

# Close TensorBoard writer
writer.close()
```

Visualize the model in tensorboard and trained agent's behavior in the cartpole environment:

Learning rate : 0.001
Timesteps: 20000

Figure: 1A

ep_len_mean : Mean episode length (averaged over episodes, 100 by default)

ep_rew_mean : Mean episodic training reward (averaged over episodes, 100 by default)

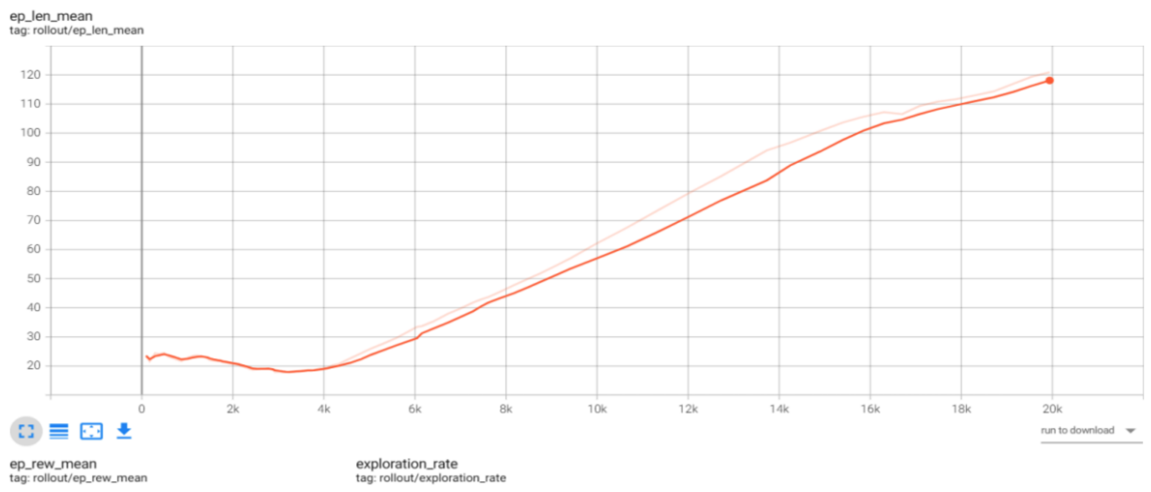


Figure: 1B
Mean_reward

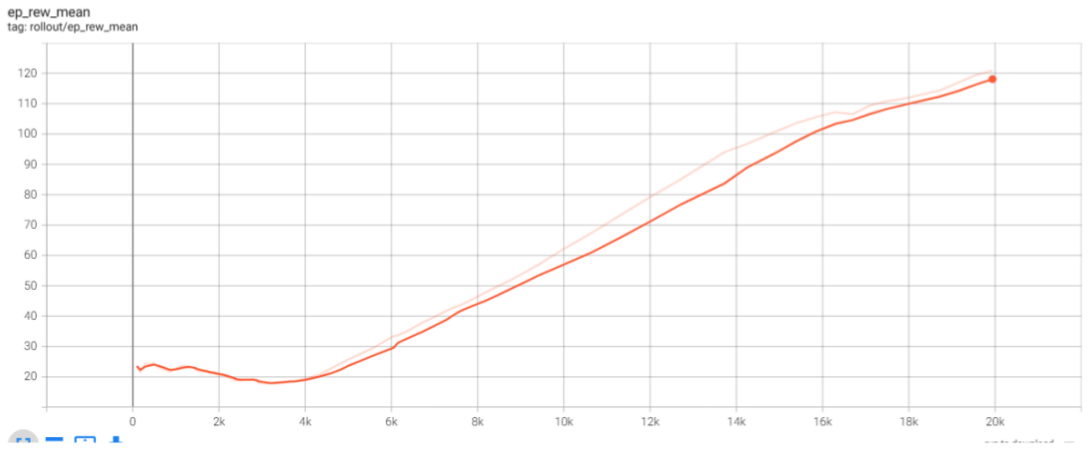


Figure: 2
Exploration rate

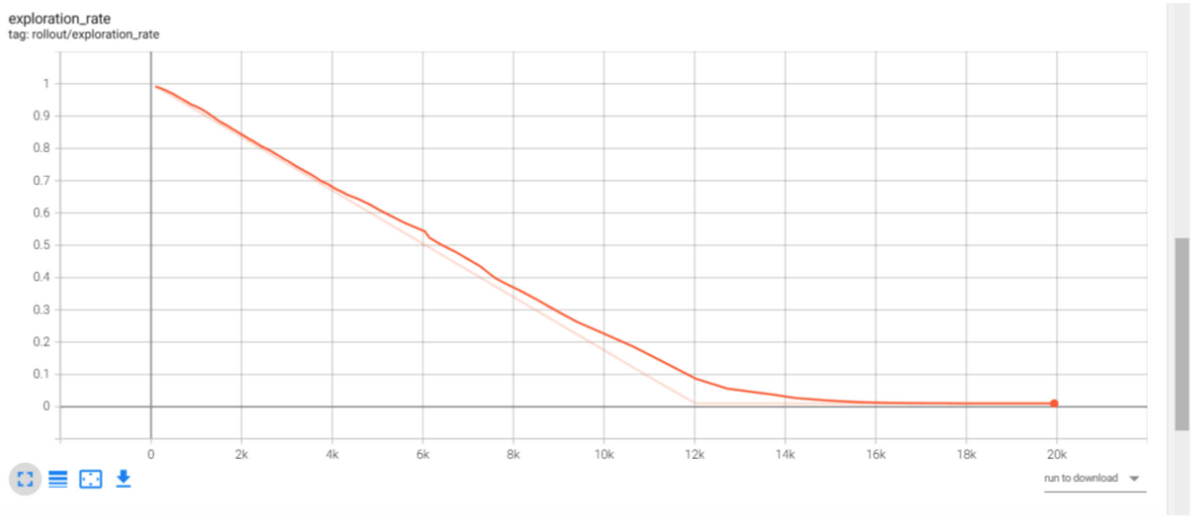


Figure: 3
Learning Rate
As expected with a linear schedule rate

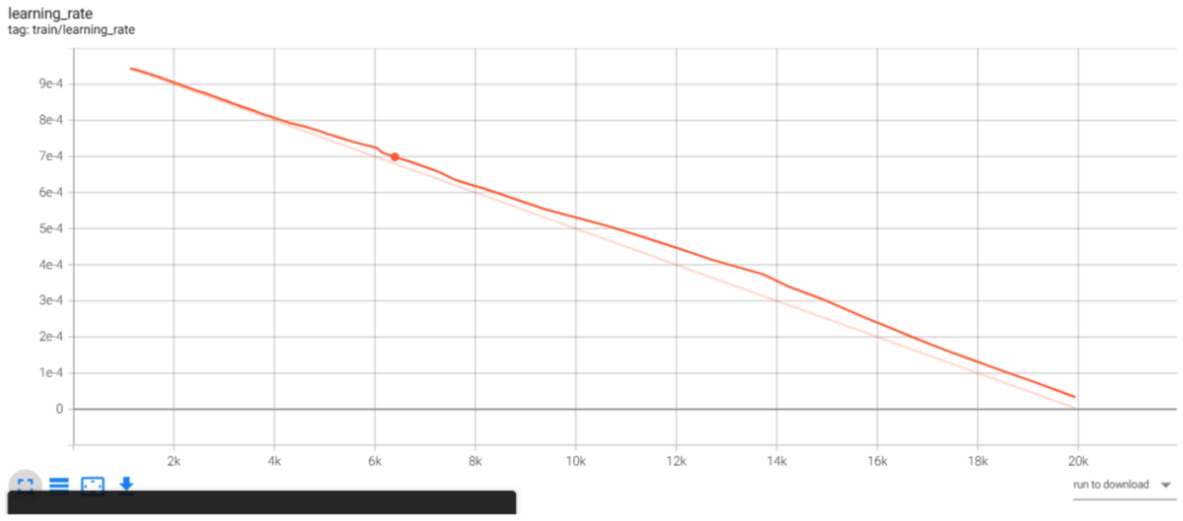
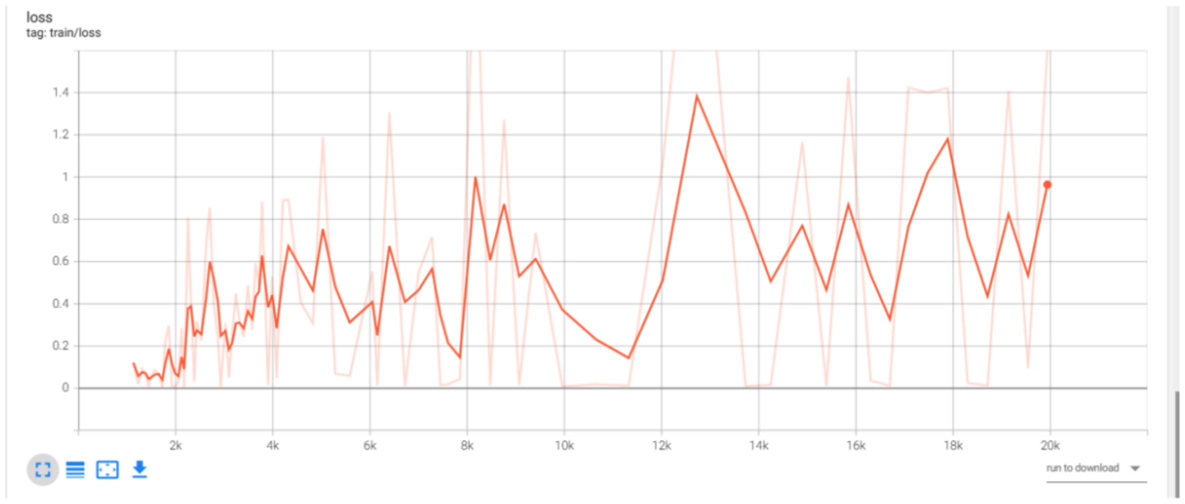


Figure: 4 : Loss Function



Mean Reward: 9.5, Std Reward: 0.5

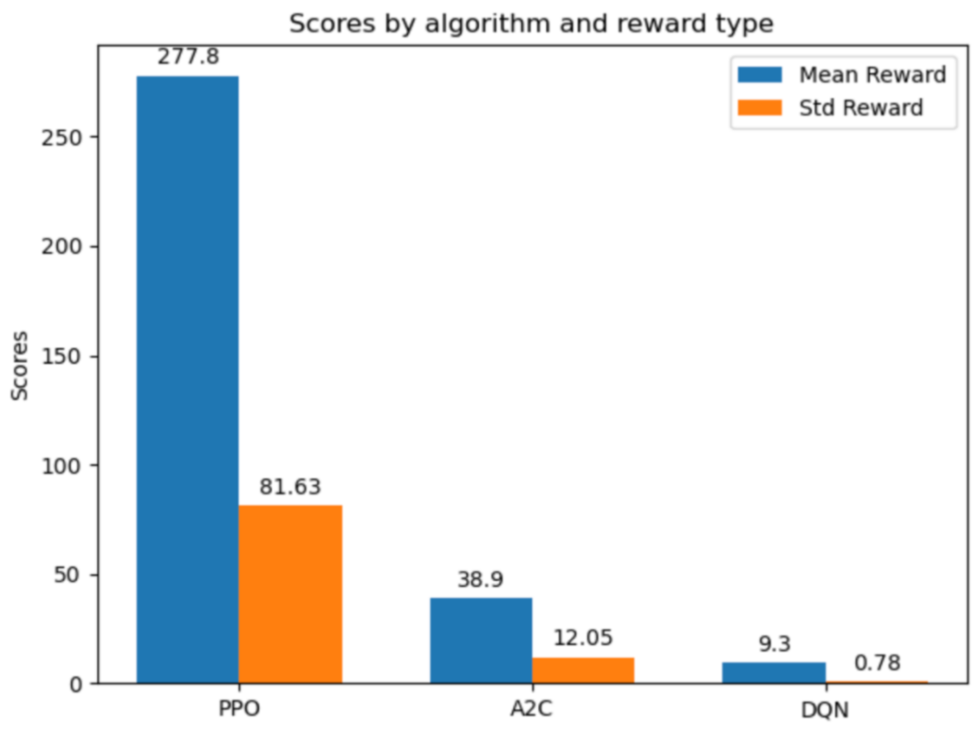
RESULT

This section provides the details of experiments carried out to evaluate the performance of various reinforcement learning models described in the previous sections. This is described next in the following subsections.

A. Software and Hardware Setup

The complete implementation of code is done on Jupyter Notebook. The program is written using p ython with various libraries. It takes about an hour for running episodes on an Apple MacBook M1 WITH 8GB RAM.

B. Performance of various RL models



The image displays a bar chart titled "Scores by algorithm and reward type". There are three differen t algorithms compared on the chart: PPO, A2C, and DQN.

For each algorithm, there are two bars representing two types of rewards:

- Mean Reward (depicted in blue)
- Standard Deviation (Std) Reward (depicted in orange)

Here are the detailed scores for each algorithm:

- PPO has a Mean Reward score of approximately 277.8, which is by far the highest among the three. Its Std Reward score is approximately 81.63.
- A2C has a significantly lower Mean Reward score of around 38.9, with a Std Reward of ab out 12.05.
- DQN has the lowest scores, with a Mean Reward of roughly 9.3 and a Std Reward close to

0.78.

The y-axis represents the "Scores", which is not explicitly labeled with a range but based on the ba r heights extends from 0 to slightly above the highest score, just below 300. The x-axis lists the alg orithms.

The chart shows that PPO outperforms A2C and DQN in terms of the mean reward by a substant ial margin. However, PPO also has a higher standard deviation in rewards, indicating more variabil ity in the reward outcome compared to A2C and DQN. DQN shows the least variability in reward but also the lowest mean reward.

Conclusion

The project aimed to implement and compare three reinforcement learning algorithms: Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), and Deep Q-Network (DQN), within the CartPole-v1 environment, a standard benchmark in RL. The project's intent was to provide a practical guide for those interested in RL, using Python and frameworks like OpenAI/Gym.

Work Completed:

- Implemented the Cart-Pole system simulation using OpenAI Gym.
- Developed code to visualize the Cart-Pole system's behavior under a random action policy.
- Trained and evaluated the PPO, A2C, and DQN algorithms, integrating TensorBoard for performance visualization.
- Conducted a comparative analysis of the algorithms based on mean rewards and standard deviations.

Contribution to Learning:

- The project facilitated a deeper understanding of the practical implementation of RL algorithms.
- It offered hands-on experience with setting up RL environments and training agents within those environments.
- It also contributed to the comprehension of the nuances involved in the performance and behavior of different RL algorithms.

Lessons Learned:

- Algorithmic Performance: The PPO algorithm exhibited superior performance in terms of mean reward but with greater variability, as indicated by a higher standard deviation. This suggests that while PPO can be more effective, it may also be less predictable.
- Challenges in RL: The project underscored the inherent challenges in reinforcement learning, such as balancing exploration and exploitation and the importance of stability in learning.
- Implementation Insights: Practical implementation details, such as the importance of visualizing training progress and the use of callbacks and logging for tracking model performance, were critical takeaways.
- Research Application: The project illustrated the real-world applicability and potential of RL algorithms in various domains, emphasizing the importance of tailoring the choice of algorithm to the specific characteristics of the problem at hand.

Overall, the project not only served as an educational exercise in applying RL algorithms but also provided valuable insights into their relative strengths and weaknesses, contributing to the field of AI and machine learning.

REFERENCES

[1] M. P. Deisenroth, K. Arulkumaran, M. Brundage and A. A. Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*,

[2] Y. Li. Reinforcement Learning Applications. *arXiv preprint arXiv:1908.06973*, 2019.

[3] K. Kavukcuoglu, V. Mnih, D. Silver, J. Veness, , A. A. Rusu, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski,et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015

[4] OpenAI Gym is a platform. Reinforcement learning algorithm development and comparison toolkit. (https://gym.openai.com/.)

[5] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[6] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3- 4):279–292, 1992.

[7]https://huggingface.co/learn/deep-rl-course/unit6/advantage-actor-critic

[8] Vicki Cheung, Greg Brockman, Ludwig Pettersson, John Schulman, Jonas Schneider, Jie Tang Wojciech Zaremba OpenAI: arXiv:1606.01540v1 [cs.LG] 5 Jun 2016

[9] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. arXiv preprint arXiv:1604.06778, 2016.

[10] B. Tanner and A. White. RL-Glue: Language-independent software for reinforcement-learning experiments. J. Mach. Learn. Res., 10:2133–2136, 2009.

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. “Asynchronous methods for deep reinforcement learning”. In: arXiv preprint arXiv:1602.01783 (2016).