ME5418 MACHINE LEARNING IN ROBOTICS

FINAL PROJECT REPORT

Title: Reinforcement Learning for Optimal Racing Line

NAME: MADKAIKAR ATHARVA ATUL

MATRICULATION NO: A0268376M

Department of Mechanical Engineering

National University of Singapore

Semester 1, 2023/2024

INTRODUCTION

This project aims to develop a Reinforcement Learning (RL) based system for an autonomous racing vehicle. The goal of the agent would be to dynamically determine the optimal race line on a track, minimize the lap time and maximize the speed during the lap. Opting for Reinforcement Learning based system was driven by its ability to offer a model free approach which does not require building complex system models of real physical systems. Moreover, RL mimics human like learning through trial and error by actively interacting with the environment and learning from the experience gained by interaction. This allows the agent to explore the environment and learn form a diverse set of experiences. This flexibility allows the agent to refine the understanding of its environment and prepares it for the unforeseen circumstances which is necessary in a dynamic and a fast-paced environment like autonomous racing. This helps the agent to optimize the racing performance and improve its decision-making capabilities without being constrained by predetermined maps.

Beyond the scope of the competition, research and advancements in the field of autonomous racing propagate the development of self-driving cars. With the society moving towards a more autonomous future, these developments will speed up the transition. Moreover, autonomous racing serves as a testing ground for cutting edge technologies in a safe and a controlled environment. Further research and results obtained will help us make the transition to a safer and a more reliable future for self-driving cars in everyday use.

CONVENTIONAL ALGORITHMS:

Conventional planning algorithms, such as RRT* and A* used in conjunction with controllers like Model Predictive Controllers (MPCs) or Pure Pursuit, have long been used in autonomous racing as well as in other applications where path planning, and motion control is involved. One of the primary advantages of these state-of-the-art traditional planning algorithms is that they are robust, reliable and are time tested. Their deterministic nature provides a sense of predictability and are often capable of quick real-time decision making. Moreover, these algorithms can make dynamic adjustments in real-time and provide a smooth trajectory. These also consider the vehicle dynamics allowing for more accurate control of the vehicle.

However, these algorithms also exhibit limitations. The reliance on predetermined maps often obtained through SLAM, introduces computational expenses. The SLAM based maps are often discretized which and don't reflect real world systems that well. Also, the traditional planning algorithms have the tendency to hug close to the corners and usually employ additional path smoothening algorithms like elastic band or artificial force field algorithm which can be computationally expensive. While these algorithms offer a solid foundation, RL provides a model free approach that can seamlessly adapt to the nuances of different tracks and changing conditions.

RL CAST:

The environment for simulation and training of the race car was built from scratch by following the guidelines from OpenAI Gym. The environment was rendered using Pygame. Apart from the modifications made to the reward structure and minimum and maximum constraints set to the steering angle, acceleration and velocity, the rest of the code remains unchanged. [Refer to the Gym Report]

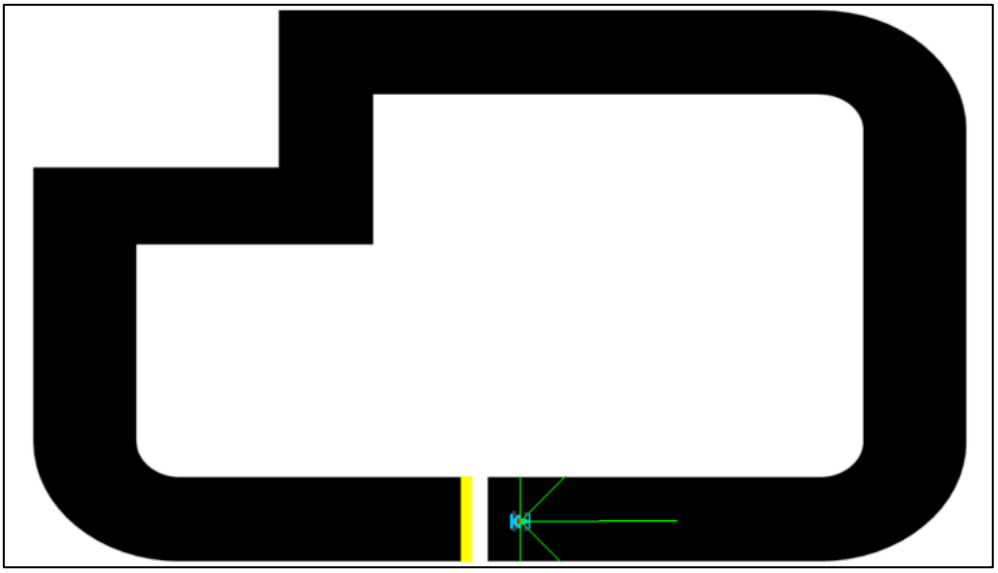

Figure 1: Agent on the track (map1a). Green lines represent the sparse lidar scans. Yellow line represents the finish line.

1. State Space: The observation space is continuous and defines the agent's x and y co-ordinates in the global frame, agent's pose in the local frame, agent velocity in the local frame, and five sparse lidar scans which span from -90 to 90 degrees.
2. Action Space: The action space is continuous and consists of vehicle's steering angle and its acceleration/deceleration. (Max Steering angle set to 30 degrees)
3. Reward Structure:
   - A reward of +0.001 is given for every time step in the simulation.
   - A reward of +0.1 * speed is given for every time step in simulation.
   - A reward of +50 is given every time agent completes 50 steps in the simulation.
   - A reward of +500 is given on completion of the lap.
   - An additional reward of +100 is given if agent finishes the lap faster than its previous fastest lap time.
4. Neural Network: After experimenting with various Neural Network architectures, we opted for a streamlined yet sufficiently deep Actor-Critic Network for the final training and testing.
5. RL Algorithm: For our project, we chose the PPO algorithm as it is effective at handling continuous state and action spaces.

CODE STRUCTURE AND PIPELINE

1. **Neural Network Architecture:**
   The python file "**ACNet.py**" contains the Neural Network code. The code contains two sperate Neural Network models for Actor and Critic used by the Policy Proximal Optimization (PPO) agent. The network code is written keeping flexibility in mind, allowing the user to decide the number of hidden layers and the number of units in each layer. Figure 2 demonstrates the network architecture that was used in the final code. The class **ActorNetwork** produces mean action and standard deviation for the Gaussian distribution from which the actions are sampled, while the class **CriticNetwork** estimates the state-value function. These components collectively form a PPO-based Actor-Critic model for reinforcement learning tasks.

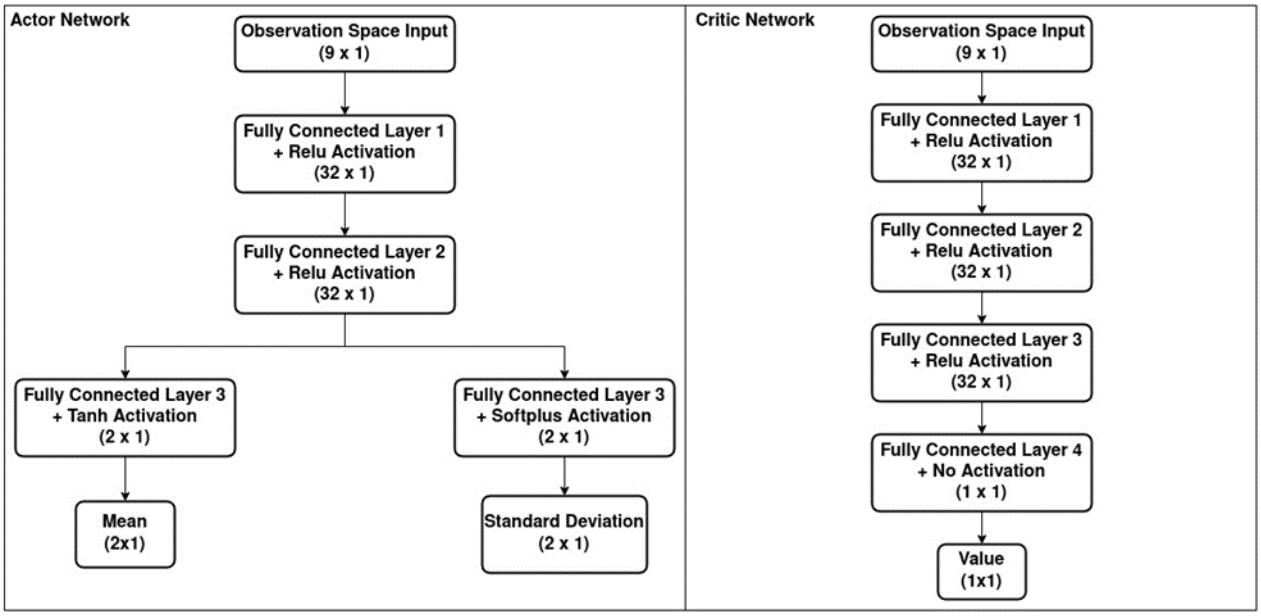

Figure 2: Actor-Critic Network for the PPO Agent

A. Actor Network:
   a. The input to the Actor Network is a normalized Tensor of observations.
   b. The Network utilizes its shared layers for feature extraction through the specified number of hidden layers with **ReLU** activation. (2 hidden layers with 32 units each for the final architecture)
   c. The output of the shared layers is then passed on to two separate branches:
      i. Actor layer for generating mean actions under the current policy. The '**tanh'** activation function is used to bound output in the **range [-1, 1]**.
      ii. Standard deviation layer for giving standard deviation. The **'softplus'** activation in this layer ensures positive values for the standard deviation.

B. Critic Network:
   a. The input to the Critic Network is also a normalized Tensor of observations.
   b. The network uses shared layers for feature extraction with **ReLU** activation. (3 hidden layers with 32 units each for the final architecture).

c. The output layer with no activation is a single value representing the estimated value (state-value) of the given state.

The new network architecture deviated from the initially proposed architecture for the following reasons:

1. Training Complexity:
    a. The initial network architecture was difficult to train. The losses and the gradients would explode despite using gradient clipping. We inferred that allowing the same to network output the policy and state-values was causing the issue because of the substantial difference in output magnitudes of the policy and state-value function which affected the training stability.
    b. By adopting a simpler network and separating the Actor and Critic models, we were able to address the challenge of dealing with significantly different scales in values of the policy and the state functions. In the new separated architecture, the Actor Network provides mean and standard deviation for generating actions while the Critic Network estimates state-values. The separation allows more stable training, as each network now focuses on its specific task which allows for more proportionate updates to the Policy and the Value.
2. Dynamic Standard Deviation:
    a. Initially, we defined the log of standard deviation as a trainable parameter which was updated based on combined losses (policy loss and the value loss). The challenge lay in different scales and gradient imbalances, causing disproportionate updates between policy and critic components eventually leading to convergence issues.
    b. Computing standard deviation dynamically within the Actor Network allowed for a more adaptive adjustment. By enabling the network to learn the standard deviation directly from the data, the updates across policy components were aligned. Also, the standard deviation was now being updated based only on just the policy loss. This not only enhanced stability during training but also ensured that the network learns a more accurate representation of the uncertainty in the action space.

Moreover, the computation of losses and the Gaussian distribution was shifted to the Agent code since it was decided to adopt two separate models for Actor and Critic Networks. This practical adjustment streamlined the implementation.

2. Learning Agent:

The python file "**PPO_Agent.py**" code defines a Proximal Policy Optimization (PPO) agent for reinforcement learning. PPO was chosen for its stability in training with continuous observation and action spaces. PPO is effective and easier to implement compared to alternatives like DDPG or TRPO. Moreover, PPO is inherently exploratory and generates stochastic policies. Furthermore, the clipped objective function in PPO helps prevent large policy updates which ensures smooth convergence and provides stability during training. The objective function is also responsible for maintaining balance between exploration and exploitation which is essential in Reinforcement Learning.

In the code, the class **PPOAgent** represents the PPO Learning Agent. The code is responsible for memory handling, generating a gaussian distribution, sampling an action from the distribution and carry out training for the policy and value by defining the respective losses. Following is a brief overview of the code:

1) **__init__() function:** The class is initialized by providing the actor and critic models along with hyper-parameters like epsilon, policy learning rate, value learning rate, target kl_div, etc. Two Adam Optimizers are initialized to train both, the policy, and the value.

2) **save_models () function:** Saves the actor and critic models during training.

3) **load_models () function:** Loads the trained models to predict actions.

4) **store_memory () function:** Stores the experiences (*states, actions, values, log probabilities and rewards*) to be used for training.

5) **access_memory () function:** Accesses the stored experience in the training loop. The function converts the stored experiences into a NumPy and returns them.

6) **clear_memory () function:** Clears the memory once the model is trained on a batch of experiences.

7) **get_action_value () function:** The function provides a normalized tensor of observations to the actor and critic models individually and gets the mean, standard deviation, and the value function. Based on the mean and the standard deviation, an action is sampled from a Gaussian distribution, the log of its probability is computed. The function returns the sampled action, value and the probability of the action.

8) **get_log_prob () function:** Computes the log of probability of a given action provided the mean and standard deviation of a Gaussian distribution.

9) **train_policy () function:** The function computes the surrogate function and evaluates the policy loss to compute the gradients. The gradients are then used to update the weight of the Policy Network. Based on the number of iterations specified, the function loops through the process of evaluating the policy loss and updating the weights of the Policy Network.

10) **train_value () function:** Similar to the policy training function, this function evaluates the value loss and based on it updates the weight of the Value Network. The value loss is computed the using mean squared error between state-values predicted by the network and the expected returns for that state. The function then applies gradients to update the weights of the Value Network. Like the policy training, the function also loops through the process of evaluating the value loss and updating the Value Network.

The final code differs from the initial agent code in the following ways:

- In the final code, the Gaussian distribution function and the sampled actions are generated within the code itself instead of the Network code generating those in the previous iteration.
- The Policy loss and Value loss are now computed inside the Agent class in their respective training functions instead of being computed in the Network code as explained earlier.
- Moreover, the policy and value networks are now trained separately for the following reasons:
    - Stability in training.
    - The optimal learning rates for training the policy and value may differ. Training the policy and value networks separately using two separate optimizers and different learning rates tends to improve and speed up the learning process.
    - Since the training happens in iterations for a batch of data, policy training can be stopped if the evaluated policy differs too much from the old one which can be evident by computing the kl divergence coefficient. Whereas the training iterations for Value can go on irrespective of whether the policy iterations have been stopped or not.
    - Separation also ensures that large updates to one of the components do not affect the updates to the other component ensuring better and faster convergence.
- The training loop now has been shifted to the trainer code, which will be explained in detail in the following section.

3. Training Process

The code in jupyter notebook file "**trainer.ipynb**" defines a class ***Trainer*** which is responsible for implementing a well-defined training loop for the agent and testing the trained Proximal Policy Optimization (PPO) agent in our gym environment. It is the main component of the training and testing pipeline for our autonomous race car agent. Below is the brief explanation of the code in the ***Trainer*** class:

A. Initialization:
    a. The necessary inputs to instantiate the trainer are the actor model, critic model and the gym environment. The optional parameters include policy learning rate, value learning rate, max number of steps in the environment, number of episodes, batch size, model update frequency, train, etc. If optional parameters are not specified, the trainer instantiates with the default parameters.
    b. The constructor also creates an instantiates the PPO agent by providing the necessary parameters.

B. Training:
    a. If training mode is enabled (**train=True**), it runs a specified number of episodes (*Default: 300*) in the racing environment.
    b. Each episode involves the PPO agent interacting with environment, collecting experiences, and updating the policy and the value networks if the network update conditions are satisfied.
    c. The training process happens within the episode and is guided by the update frequency, batch size and the number of epochs. The training process is explained in detail below:

i. If the update frequency is set to 32, batch size is set to 8 and the number of epochs is set to 5, the agent keeps accumulating experiences throughout the episode until the collected experiences match the update frequency.
ii. Once the number of accumulated experiences equals the update frequency, the expected returns and advantages are computed for the gathered experiences.
iii. The experiences, advantages and returns are then shuffled with proper correspondence and divided into batches, each of size 8 in this instance, resulting in four batches.
iv. Subsequently, each batch is sequentially passed to the '**train policy'** and '**train_value**' methods of the PPO agent, iterating over the specified number of epochs.
v. Following completion of the epochs, the agent's memory is cleared by calling the **'clear_memory'** method of the PPO agent, preparing for the next set of experiences.

d. The cycle repeats through the episode and thus the data is trained online.
e. If the number of experiences collected during an episode is less than the specified update frequency, 32 in this instance, these experiences are carried forward into the next episode. This ensures that data isn't lost, allowing the agent to gradually accumulate experiences until it reaches designated update frequency.
f. The training loop also keeps a track of episodic rewards, policy losses and value losses.
g. During training, the best model weights are saved if the agent reaches the goal with the highest reward (agent finishing the lap time faster than the previous fastest lap time gets a higher reward)

C. Plotting:
a. After training is complete, the average rewards, policy loss and the value loss are plotted.

D. Testing:
a. If testing mode is enabled (**train=False**), the trainer loads the pre-trained model.
b. The trained model / agent is then evaluated on the environment for a specified number of episodes.
c. Episode rewards and number of steps are printed for each episode.

4. Utility Functions:

The code in python file "**Utilities.py**" provides functions for normalizing the observations, calculating discounted rewards, and plotting the learning curve.

1) **normalize_observations () function:** Normalizes the given observation vector based on predefined ranges for each feature.
2) **discount_rewards_1 () function:** Calculates discounted rewards using the specified discount factor (gamma). This function accumulates rewards with discounting.
3) **discount_rewards_2 () function:** Computes discounted rewards based on advantages and values. It adds advantages and values to get the final discounted returns.
4) **Plot_learning_curve () function:** Used to plot the graphs for average rewards, policy loss and the value loss.

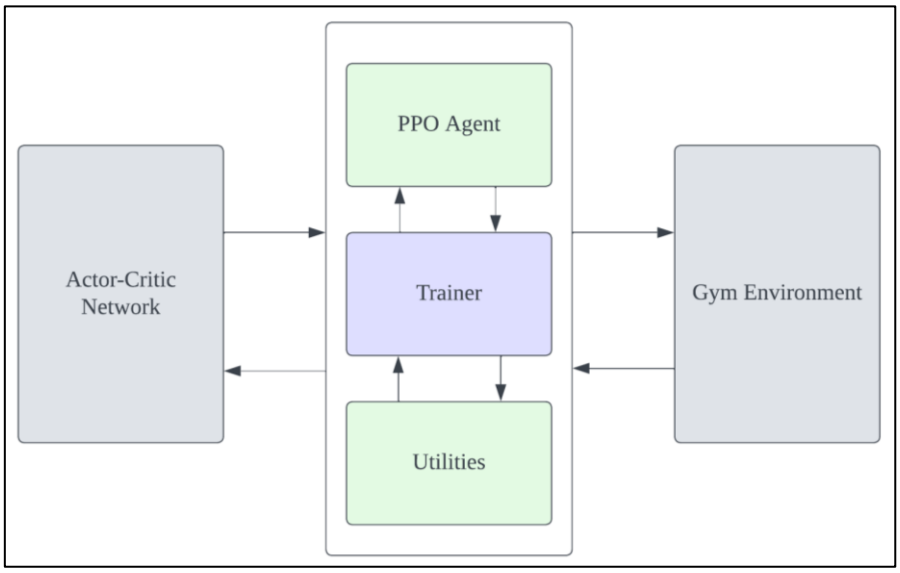

Figure 3: PPO Pipeline

TRAINING AND RESULTS

The model was trained on ***map3*** with the following values of parameters.

1. **max_steps** = 200
2. **num_episodes** = 300
3. **batch_size** = 5
4. **epochs** = 4
5. **update_frequency** = 20
6. **epsilon** = 0.2
7. **target_kl_div** = 0.01
8. **max_policy_iters** = 1
9. **max_value_iters** = 1
10. **policy_lr** = 0.0003
11. **value_lr** = 0.001
12. **max_grad_norm** = 1000

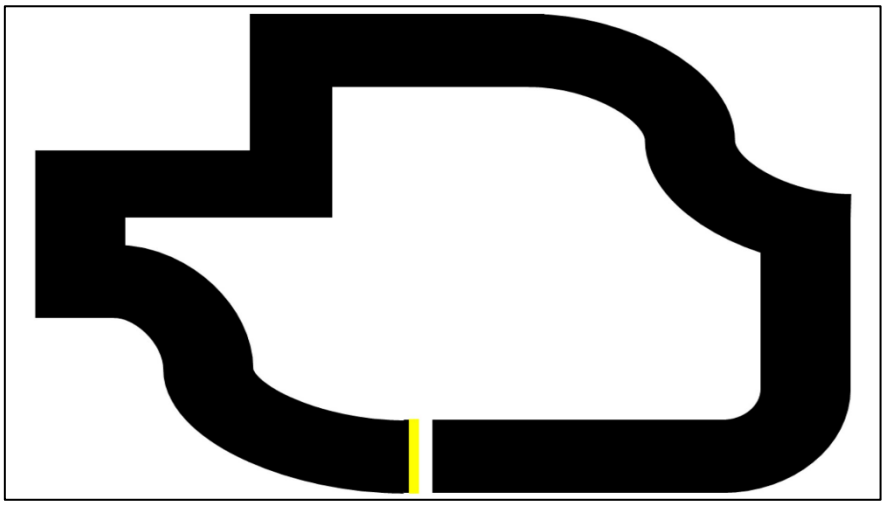

Figure 4: Map3. This track was used for training.

In our observation space with only 9 elements, training proved effective and quick without the need for an extensive number of episodes. We experimented with 200, 300 and 400 episodes and got the best results from one of the 200-episode runs. Model weights were saved when the agent achieved a faster lap time than its previous fastest lap time, ensuring the saved model approached optimality. Post training, we evaluated the model on a different map (**map1a**), not encountered during training. The model performed extremely well on the map it wasn't trained on which attested our objective of generalizing the system. After several iterations of training and testing, we finalized a model, showcasing optimal performance on both ***'map1a'*** and ***'map3'***, navigating without any instances of veering off the track.

The finalized model is utilized as a validation model in the final code. In the "**trainer.ipynb**" file, when the parameter **train** is set to **False** during trainer instantiation, this finalized model is loaded. This model is then employed to race the vehicle on either ***'map1a'*** or ***'map3'***.

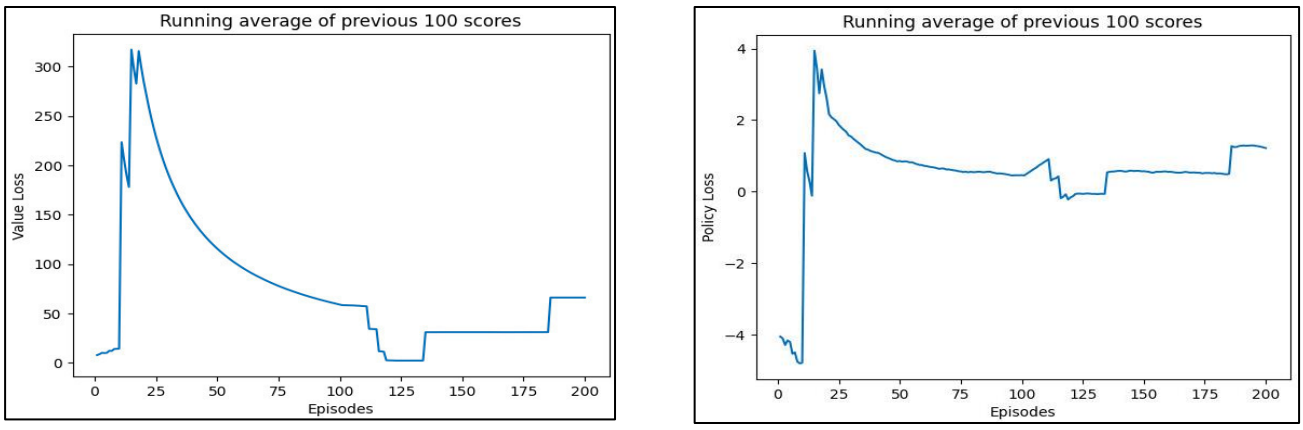

Figure 5: a) Value Loss during training and b) Policy Loss during training

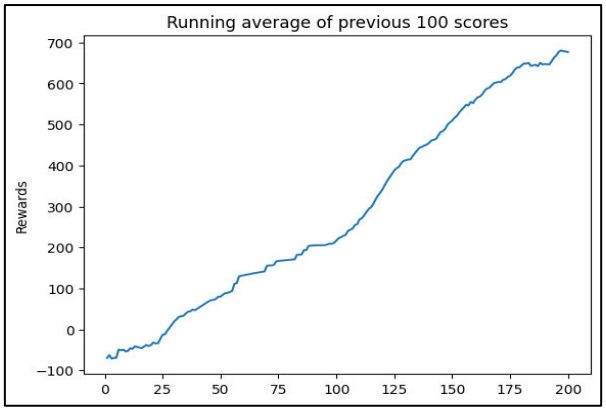

Figure 5: c) Average rewards during training

It is evident from the presented plots of value loss, policy loss, and average rewards that, with ongoing training, the rewards exhibit an upward trend, indicating improved performance as the model refines its policy. Concurrently, both policy loss and value loss exhibit a downward trend, signifying the model's enhanced policy optimization and improved value estimations over the course of training.

The correlation between increasing rewards and decreasing losses suggests that the training process effectively refines the agent's policies and enhances its ability to estimate state values over time. This trend aligns with the reinforcement learning objective of iterative improvement through experience, showcasing the model's adaptability and learning efficacy.

COMPARISON WITH STABLEBASELINES3 PPO

We also conducted a comparative evaluation by training the car using PPO from stablebaselines3 and our implemented PPO. Both agents underwent training on **map3** and were then tested on the same map, as well as on **map1a**, which was not used for training. The results revealed that our agent performed better than stablebaselines3's PPO agent, demonstrating faster lap times on both maps as evident from the comparison videos provided along with the code. Notably, stablebaselines3's PPO agent exhibited a tendency to deviate from the centreline and would slow down a lot around corners because of the evident back and forth yawing. In contrast, our agent maintained proximity to the centreline on straight segments of the track, exhibiting faster speeds around corners, and overall achieving faster lap times.

Comparison videos, along with individual videos displaying the performance of both agents on the provided maps, are available in the "videos" folder.

ADVANTAGES OF OUR APPROACH

- The reinforcement learning (RL) approach, particularly using Proximal Policy Optimization (PPO), provides adaptability to complex and dynamic racing scenarios.
- Unlike the traditional path planning algorithms, RL provides a model-free approach and the agent does not require to know the dynamics of the system and the environment.
- Reinforcement Learning (RL) algorithms do not require to us perform SLAM or incorporate complex localization algorithms to interact with the environment.
- Moreover, there is no need for dedicated controllers like MPCs, LQR controllers, etc.
- Reinforcement Learning is better at managing high dimensional state space and control inputs in the continuous domain.
- Reinforcement Learning promotes end-to-end learning in a sense that the agent directly interacts with its environment and based on the experience, the model weights are updated.
- By learning from experience, the agent gains adaptability to changing conditions.
- The parameters learned on one track can be used to evaluate the agent’s performance on an unseen track. The learned policy can therefore be used in an environment where the conditions and dynamics are different.

LIMITATIONS OF OUR APPROACH

- The current approach and training process can only deal with a single ego vehicle.
- With the current approach, there is no way for the agent to know if it is moving in the opposite direction. This can lead to problems during training since the agent receives the reward for progressing into the environment. Without appropriate flags provided by the environment, it is difficult for the agent to determine whether the direction it is progressing in is correct or not.

- In our environment, the episode terminates once the agent reaches the goal (finishing line). Although, we keep a track of the time it took to complete the lap in the previous episode and use to evaluate the agent's performance in the following episode, the performance of the agent for successive laps within an episode cannot be evaluated.
- In certain scenarios, the agent struggles to explore diverse policies which can lead to the agent moving in a never-ending circular path during the episode.
- The success of the agent at times can depend on the initial weights.

CHALLENGES ENCOUNTERED

- The initial complex network proved challenging to train and exhibited sensitivity to parameter variations. Losses and gradients often exploded, leading to instability, even with gradient clipping measures.
- Having the value and the policy layers in the same network model posed difficulties during training due to the significant difference in the values estimated by the policy and the value networks. This led to disproportionate weight updates causing instability during training and eventually leading to the problem of exploding gradients and losses.
- The training success became dependent on the initial weights of both the networks. Thus, frequent restarts during training were necessary.
- The were instances where the training would start well but would diverge over time.
- Occasionally, initial sampled actions led the agent to learn circular policies as mentioned before, causing continuous circling until episode termination.

REFLECTIONS / LESSONS LEARNED

- In developing this code, we got to experience the complexity of building and training a Reinforcement Learning Model.
- Adapting to a simple yet sufficiently deep Network architecture taught us that sometimes simple things just work!
- The experience highlighted the power of RL, while also exposing its current limitations compared to more robust and reliable traditional planning algorithms. Specifically, the influence of the initial weights of the model on the agent's success highlighted the non-deterministic nature of the RL.
- Initial assumptions about the effectiveness of certain approaches, in our case using a complex network architecture, proved mutable, basically emphasizing the importance of experimentation, and understanding the system. This prompted us to reconsider the network architecture and make necessary adjustments to both the agent and training strategies.
- The integration and seamless execution of the code demanded substantial time and effort and emphasized the importance of proper planning for implementation and troubleshooting.

POTENTIAL FUTURE WORKS

- Explore the implementation of a model capable of training multiple agents simultaneously. Introduce a competitive element by having these agents race against each other.
- Extend the project by delving into the development of a stable architecture to mitigate training divergence and aim for more consistent and reliable learning outcomes.
- Consider implementing a similar model in a more advanced simulators like CARLA or Airsim. Transitioning to these environments will offer an opportunity to account for more complex car dynamics and realistic environmental factors, providing a more practical setting for the agent.

EXISTING CODES / REFERNCES

- Gym code was written after drawing inspiration from a couple of sources. The core structure of the code followed guidelines from OpenAI gym. Further inspiration for adoption of lidar sensors and track generation was taken from NeuralNine (mentioned in the gym report)
  - https://github.com/NeuralNine/ai-car-simulation
- The code written for the Neural Network and the basic framework for the agent and the trainer draws inspiration from sources listed below:
  - https://www.neuralnet.ai/a-crash-course-in-proximal-policy-optimization/
  - https://www.youtube.com/watch?v=9DO63MSGeNA
  - https://github.com/philtabor/Youtube-Code-Repository/tree/master/ReinforcementLearning/PolicyGradient/PPO/tf2
- The code for the PPO Agent is constructed based on insights gained from a tutorial available on YouTube.
  - https://www.youtube.com/watch?v=HR8kQMTO8bk