Miniproject 2

Group 1: The Dream Team:

Joseph Sepich
Sean Steinle
Shahriar Hossain
Sanjay Mohan Kumar

1. Environment/Infrastructure Setup

- **Infrastructure Challenges:**
  - While the university IT server offers high-performance resources, working remotely (via remote desktop connections) poses its own challenges.
  - On the other hand, the local setup like that of Joseph's computer were limited in terms of storage and computational resources.

- **IT System Setup (University Server):**
  - The IT system had sufficient computational power and resources for running demanding simulations while having the capability to handle extensive training runs and large-scale simulations.
  - Unfortunately, the remote desktop connectivity and latency hampered real-time debugging and interactive sessions. Along with network-based file transfers and data management can add overhead.

- **Local Setup (Joe's Computer):**
  - We had direct access without remote connectivity issues and a faster iteration for code testing and debugging.
  - But having limited storage capacity which was critical in our case for this project because the binary installation of Carla requires an enormous Docker container (approximately 700GB) to import maps. Lower hardware specifications may restrict the complexity of simulations that can be run locally.

2. Building from Source vs. Building from Binary Package

- **Binary Package Installation (Current Setup):**
  - **Map Import Issue:**
    - When installed via binary, Carla provides map import only through a Docker container that encapsulates the entire Unreal Engine environment.
    - This container requires about 700GB of storage, which is unfeasible for Joe's local computer.
  - **Limitations:**
    - Limited flexibility in handling map imports, especially when advanced features (like collision metadata) are needed.

        - Relies on pre-packaged solutions that may not adapt well to our custom simulation requirements.
- **Building from Source:**
    - While building from source we had the greater flexibility in configuring how maps are loaded and managed along with having options to tweak the build to support alternative map import methods, including those that can handle collisions more gracefully.
    - **Current Progress:**
        - One of our team member (Jo) managed to get the source build running and attempted map import with collision data, but encountered issues with the selected map.

3. Map Import Issues and Proposed Solutions

- **Current scenario:**
    - With the binary installation of Carla, the only available method for importing maps is through the large Docker container, which is impractical on Joe's computer due to its 700GB requirement.
    - Our team has managed to import maps, but they "break" (cars seem to either start running in circles or they start sinking).
- **Observations from Other Teams:**
    - We are aware that at least one other team managed to run the Docker container with 700GB, which indicates that resource availability is a critical factor.
    - Other teams might have also built from source to avoid this issue, suggesting that there is a precedent for this approach.

- **Our final development setup**
    - We Train a long policy of 5,000 steps or until convergence! Then tested the model on 3-4 different maps

4. Model Design and Training

- **Model Design**
    - For the design of the overall infrastructure of our codebase we followed the youtube series provided for this project and did some sublte modifications like added loading maps and added saving / loading models for rollouts/eval on different maps. In the following sections we describe the codes that we implemented in the form of training.py and reinforcement_learning.py

- **Simulation Environment & Observations**

    - In the Carla based environment the custom **Environment** class wraps Carla functionalities.
    - The environment spawns a vehicle at a designated spawn point (using a configurable index).

    - Uses a front-facing RGB camera sensor to capture visual observations (the “state” is the camera image).
    - Includes a collision sensor that listens for collision events and logs them.
    - The environment's **reset()** method:
    - Spawns a vehicle at a (potentially random) location.
    - Attaches the camera and collision sensors.
    - Returns the initial front camera image as the starting state.

- **Observation Data:**
    - The state is represented by an 800×600 RGB image.
    - Additional telemetry (such as vehicle velocity and distance from road) is used in computing rewards.

Reinforcement Learning Agent & Neural Network Architecture

- **Agent Structure:**
    - Implemented in the **Agent** class.
    - Uses an Xception-based CNN as a function approximator.
    - The network takes the RGB camera image as input and outputs Q-values for three discrete actions:
        - Turn left, go straight, or turn right.
- **Dual Model Setup:**
    - Maintains a **main model** and a **target model**.
    - The target model’s weights are periodically updated from the main model to stabilize training.
- **Experience Replay:**
    - Stores transitions (state, action, reward, next state, done flag) in a replay memory.
    - Samples mini-batches from the replay memory for training.
    - Uses the Bellman equation to compute target Q-values.
- **Training Process:**
    - Runs in a separate training thread.
    - The agent initially performs a dummy training step to initialize model weights.
    - Uses TensorBoard (via a custom ModifiedTensorBoard callback) for logging training statistics.
    - Target model updates occur after a fixed number of training steps.

Epsilon-Greedy Strategy

- **Action Selection:**
    - Implements an epsilon-greedy policy:
        - With probability **ε** (epsilon), choose a random action to encourage exploration.
        - Otherwise, select the action with the highest predicted Q-value (exploitation).
- **Epsilon Decay:**
    - Starts with a high initial epsilon (favoring exploration).

    - Gradually decays epsilon (using a decay factor, e.g., 0.995 per episode) until reaching a minimum threshold.
    - This decay schedule helps shift from exploration to exploitation as training progresses.

Reward Function & Environment Dynamics

- **Reward Computation:**
    - Rewards are based on several factors:
        - Positive reward for maintaining high speeds (above a threshold).
        - Negative reward for driving off the road (distance to road threshold).
        - Penalizes sharp turns (to prevent circular or erratic driving).
        - Heavy penalty (or termination) if a collision is detected.
- **Episode Termination:**
    - Episodes end if a collision occurs, the vehicle drives off the road, or when the episode time limit (60 seconds) is reached.
    - This encourages the agent to learn policies that maintain safe and efficient driving behaviors.

Overall Training Loop (train.py)

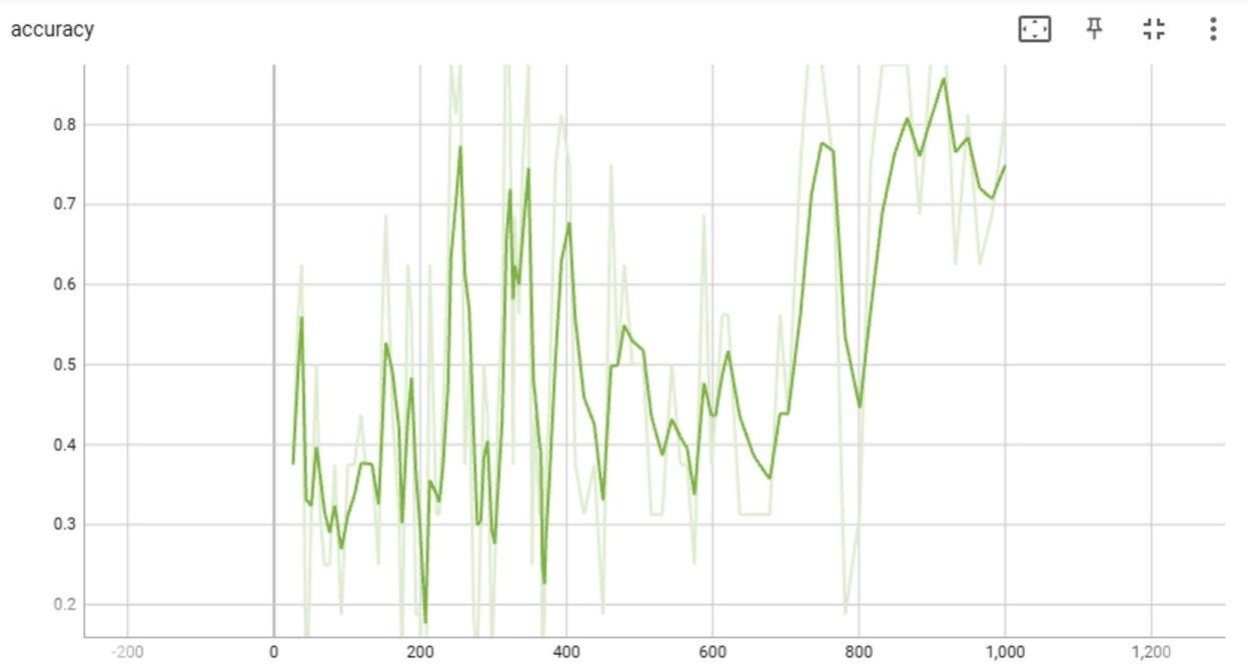
- **Setup & Initialization:**
    - Connects to the Carla server and loads the specified map.
    - Spawns a separate thread to generate traffic in the simulation.
    - Initializes environment, agent, and TensorBoard logging.
- **Episode Loop:**
    - For each episode:
        - Resets the environment to get the initial state.
        - Iteratively selects actions using the epsilon-greedy strategy.
        - Updates the agent's replay memory and accumulates rewards.
        - At episode end, logs statistics and decays epsilon.
        - Periodically saves the model if performance criteria are met.
- **Training Dynamics:**
    - The training and policy updates run concurrently with environment interactions.
    - The design supports continuous learning and adaptation over multiple episodes.

Things we would like to improve

- It would assist our analysis to figure out how to import maps that have proper road collisions. Being able to import custom maps would enable us to create specific scenarios that we could train on such as a certain intersection type or traffic situation like a sudden bottleneck in traffic. I think it would be interesting to try and incorporate waypoint rewards. Perhaps the car would continue to drive if it could obtain rewards for hitting waypoints, which would presumably be recognizable and learned through the imagery.

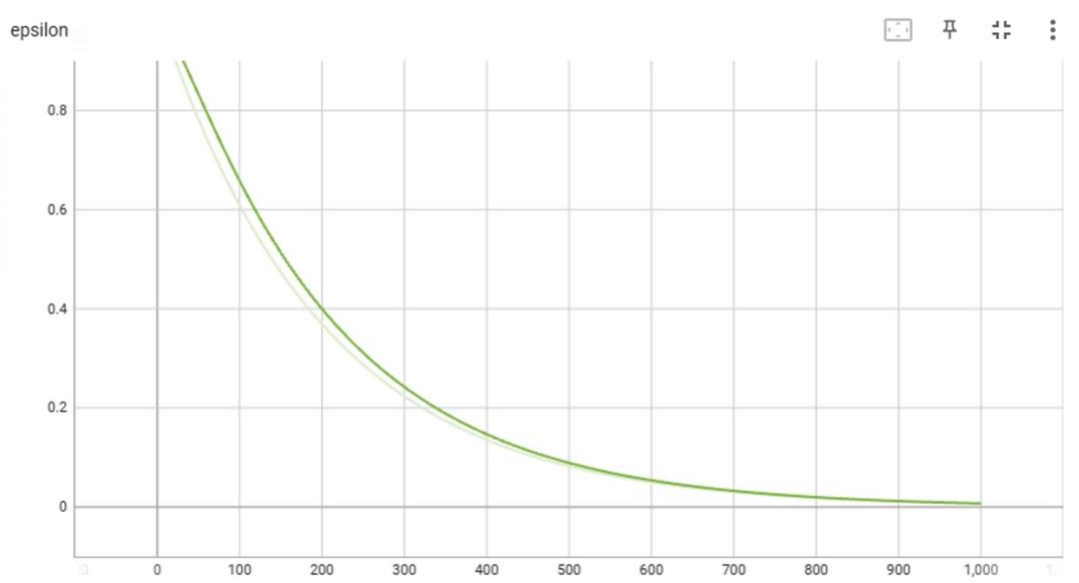
Model performance and analysis

- summarize our performance quantitatively (via tensorboard stats)
- how did reward evolve, how to loss evolve, did we ever converge or should we have kept training, etc
- summarize our performance qualitatively (with the help of Sanjay's notes during training / testing)
- Accuracy plot



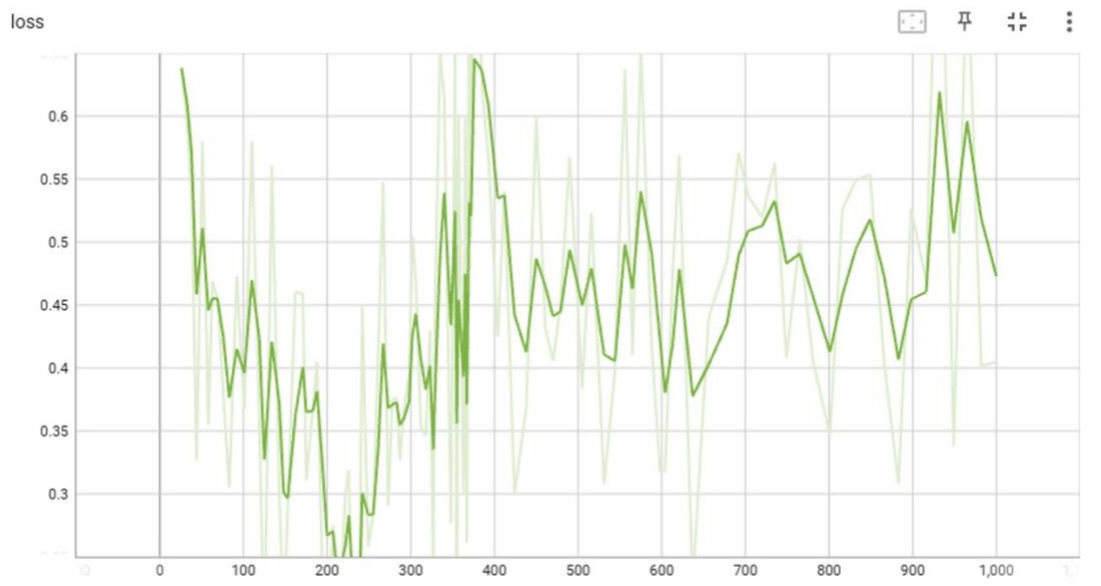
This accuracy plot shows how the model's performance fluctuates over training episodes. While there's an overall upward trend, there are noticeable sharp drops in accuracy, for example, near episode 283, likely caused by significant collisions, poor exploration decisions, or unpredictable spawns. These spikes can occur due to temporary overfitting, noise in reward signals, or unstable behavior during the early stages of training. As training continues, the model appears to recover and generally increase in accuracy past episode 678, suggesting that the agent significantly improved its policy, possibly as a result of the exploration rate decaying and favoring exploitation, or when the agent experienced a breakthrough in handling environmental challenges guiding it toward more optimal driving behavior.

- Epsilon decay plot



The epsilon decay plot, which reflects how exploration gradually gives way to exploitation over the course of training. Starting at 1.0, epsilon exponentially decreases as episodes progress, encouraging the model to rely more on learned behavior rather than random actions.

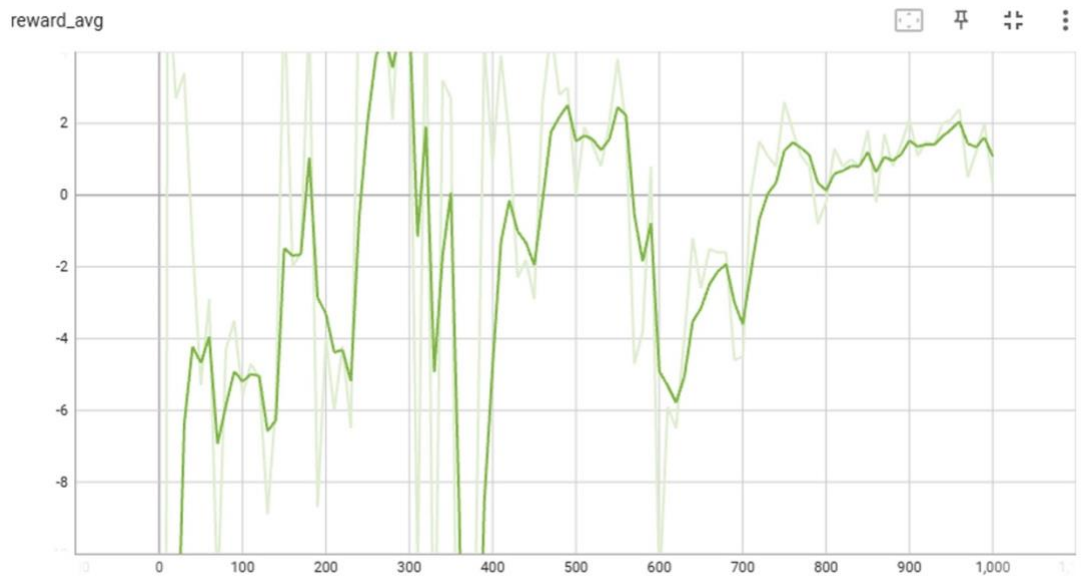
- Loss plot



This loss plot has a general downward trend from the beginning of training until around episode 255, reflecting effective learning and policy refinement during the early stages. However, from episode 255 to 412, there is a noticeable and rapid increase in loss,

suggesting instability, which could be due to increased reliance on exploitation or environmental variance such as collisions or poor spawns. After episode 412, the loss continues to rise but at a slower, more gradual pace up to episode 1000. This plateau could indicate that the model is struggling to generalize further or is converging to a suboptimal policy.

- Average reward plot



This average reward plot shows significant fluctuations in performance over the training episodes. Initially, the model's average reward starts very low, indicating poor performance in early exploration, but it quickly oscillates with both positive and negative values. There appear to be periods of improvement interspersed with drops, suggesting that the agent occasionally experiences setbacks, before recovering. Overall, the performance of the model seems to be increasing, converging to a greater average reward