AIEA Auditor Final Technical Document

Adam Martinez, Neha Hingorani, Shriya Sravani Yellapragada

The work we did as auditors in the AIEA lab involved researching applications of reinforcement learning for autonomous vehicles. Being on the AVs team, we used CARLA, Duckietown, and OpenAI's Gymnasium Car-Racing environments to test and apply the algorithms we developed. Additionally, as part of our work for the Undergraduate research showcase, we also worked on designing reward functions and environment parameters for the lab's custom CARLA environment.

For Task 6, our team developed an Actor-Critic neural network using the PyTorch library for use in the Car-Racing environment. The network consists of a convolution sequence, consisting of three convolution layers, followed by a fully connected sequence, consisting of two linear layers. Finally, the output of the fully connected layer goes to both the actor and critic output layers. The actor returns a float for every action dimension, and the critic returns a value estimation for the action.

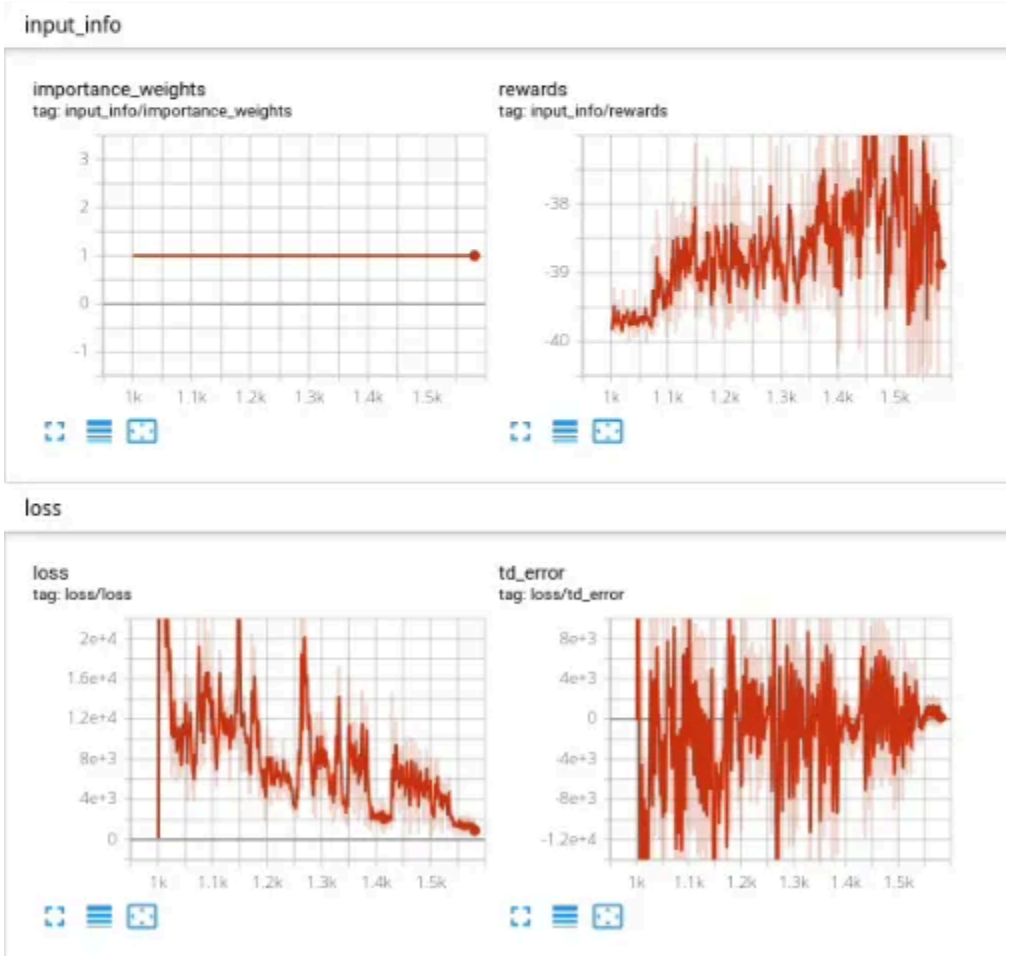
```
class ACNN(nn.Module):
    def __init__(self, action_dimensions):
        super().__init__()

        self.conv = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=window_size, stride=stride),
            nn.ReLU(),
            nn.Conv2d(16, 18, kernel_size=window_size, stride=stride),
            nn.ReLU(),
            nn.Conv2d(18, 32, kernel_size=window_size, stride=stride),
            nn.ReLU()
        )

        self.fc = nn.Sequential(nn.Flatten(), nn.Linear(2592, 1296), nn.ReLU(), nn.Linear(1296, 512), nn.ReLU())

        self.actor = nn.Linear(512, action_dimensions) # Outputs action in 3 dimensions (steering, acceleration, gas)
        self.critic = nn.Linear(512, 1) # Outputs V(s)
```

As for training, the agent picks an action by creating a Normal distribution around the actor values. It uses a flat standard deviation of 0.1. Finally, it samples an action from the distribution and takes it. We also record data for every episode, like the visited states, taken actions, rewards, critic values, and log probability of the selected action. After the episode is completed (agent reaches the "done" state) we begin the optimization process.

For optimization, we start by calculating our advantages by finding the difference between the rewards we got, discounted by γ (returns), and the values our critic predicted. We also combine and condense the recorded actions, states, and log probabilities into tensors. Over the course of four optimization epochs, we sample the model using the stacked state tensor and create a standard distribution around the output just like in training. During optimization, however, we instead find the ratio between the recorded log probabilities and the current log probabilities using the new distribution we just created. Finally we compute our optimization objective by multiplying advantages and our computed ratio. The trick here is to clip the objective by a value epsilon (clamp the value to $1 - \varepsilon$ and $1 + \varepsilon$). This way, the weights of our network won't change by more than ε percent, in our case 0.2. The minimum between the clipped loss and the unclipped loss (objective without policy clip applied) is the policy loss. By adding this to the loss between our returns and the new critic values we get, we find the total loss, and supply it to the Adam optimizer for network weight adjustments with a learning rate of 1e-8.

In the end, this algorithm struggled to achieve positive reward gains. We can identify a variety of factors that may have contributed to this. Firstly, the standard deviation of 0.1 hindered the model's ability to explore diverse actions during early training. Instead, the action values remained too close to the policy, which in the early stages will be very inaccurate. Moreover, the

fact that the standard deviation was fixed was not ideal, it would be better if the standard deviation and the degree of randomness in the agent's actions starts large and decreases over time. Furthermore, the learning rate for the optimizer is very low, and likely made learning unreasonably difficult for the model. Finally, our neural network was misconfigured, as the kernel size and stride (degree of movement of kernel) throughout the input image in our convolution layers remained fixed despite the input size shrinking each layer. This likely resulted in a loss of information for the agent's state interpretations, diminishing its ability to understand and process the features in its current state.



For Task 7, our team built a Deep Q-Network, consisting of a convolution sequence similar to the Actor-Critic, and two fully connected sequences, each containing one linear layer. The model takes in a grayscale (single channel) 96x96 RGB array, and returns a sequence of floats. Each float represents the q-value for each of the possible discrete actions the agent can take.

```
class DQN(nn.Module):
    def __init__(self, action_count):
        super().__init__()

        # For Car-Racing, out input layer ouput shape is a 96x96 image

        self.conv = nn.Sequential(
            # For Greyscale
            nn.Conv2d(1, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        self.fc1 = nn.Sequential(
            nn.Flatten(),
            nn.Linear(4096, 512),
            nn.ReLU()
        )

        self.fc2 = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512, action_count)
        )
```

In order to make a Q-network usable in a continuous environment like Car-Racing, we needed to condense the action space into a few discrete actions the agent is able to choose from.

```
disc_actions = [
    [0, 1.0, 0], # Full forward
    [0, 0, 0.8], # Brake
    [-1.0, 1.0, 0], # Full left
    [-0.5, 1.0, 0], # Half right
    [1.0, 1.0, 0], # Full right
    [0.5, 1.0, 0], # Half right
    [0, 0, 0], # Do nothing
    [0, 1.0, 0.8], # Slow down
]
```

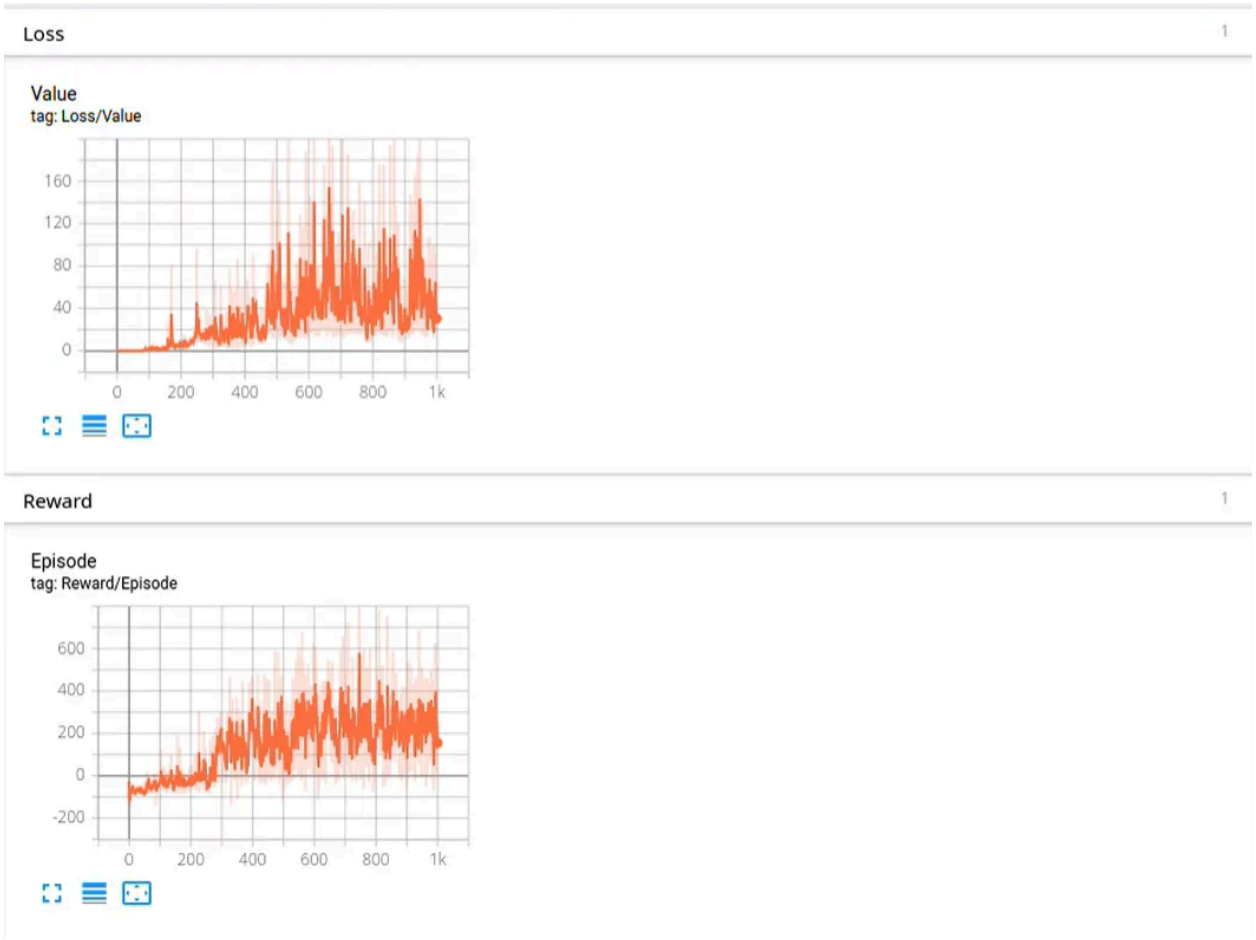
During training, each of these actions is assigned a q-value, or an expected reward for taking that action and each most optimal action after it. Each step, the model will take a random action with probability epsilon (ε), and otherwise will select the action with the highest q-value. This is an instance of epsilon-greedy search, which is useful for encouraging agent exploration before it starts to rely on its policy. Our epsilon value starts very high (ε = 0.9) and decays down to its minimum (ε = 0.1) over the course of 100 “epsilon decay episodes”.

Unlike our PPO algorithm, we optimize every step, instead of after the episode is complete. For this, we compute loss by taking the mean-squared error between the network and the q-value computed by the Bellman Update Equation.

```
def compute_bellman(reward, next_state, gamma, compute_model):
    # Compute Bellman Target

    # We're calling the model but we don't need the gradient - just the return, so no_grad
    with torch.no_grad():
        next_q_values = compute_model(next_state)
        best_next_q, _ = torch.max(next_q_values, dim=1)
        bellman_target = reward + gamma * best_next_q

    return bellman_target
```

Notably, the network we use in the loss calculation (target model) isn’t the same as the one we use for action selection (primary network). To ensure stability, we copy the weights of the

primary network onto the target model every 100 episodes. This means that the target model updates only 10 times per episode, ensuring we have a stable baseline for value predictions. Finally, instead of using the current state each step to compute the difference between the target and the Bellman, we sample a replay buffer after adding the current state to the buffer. The buffer allows us to reuse old experiences and continue to learn off of them. This allows us the agent to make the most of training and improve policy gains by leveraging all of its experience, not just the most recent. We only begin computing loss and optimizing after the replay buffer reaches a certain capacity, to ensure we have a good sample size before starting optimization.

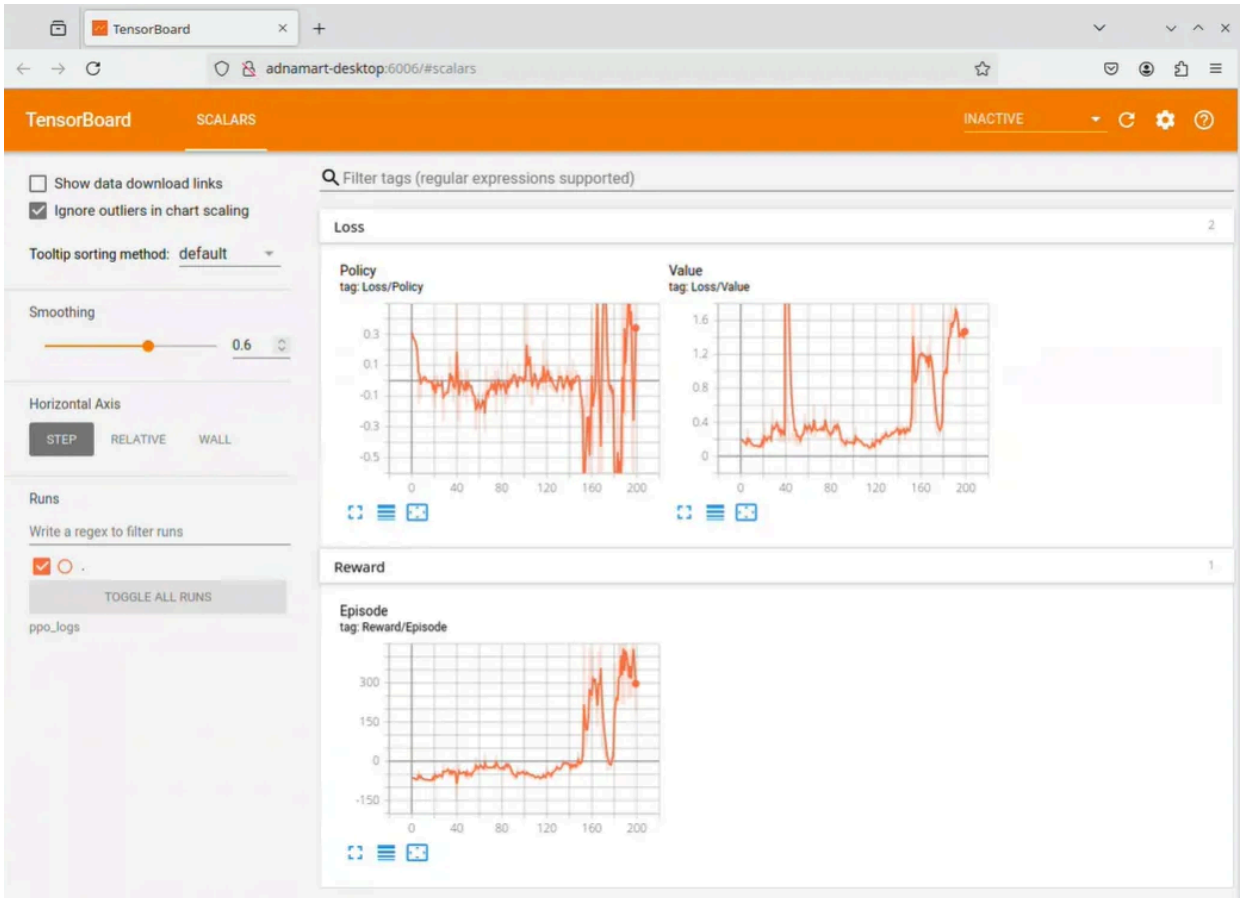


The model ended up performing quite well, managing to maintain a consistent positive reward after 200 steps. We suspect the discretized action space provided the agent with guidance and

support in learning, speeding up the training process and allowing it to learn consistently without falling back on nonsensical or counterintuitive actions. Moreover, the higher learning rate coupled with per-step optimization (as opposed to every episode) and the replay buffer provided it with advantages over our Actor-Critic agent in the previous task.

For the next task, our team got the opportunity to re-visit our PPO algorithm and make some improvements to the network and training parameters. Our improvements include:

- Dynamic standard deviation, starting at 0.5 and decaying down to 0.1 over a number of episodes. This assisted in encouraging exploration during early training, and emphasizing policy exploitation later on.
- Increasing the number of optimization epochs from four to six. This was a likely cause for a slightly slower training speed, but our hope was that our per-episode policy improvements would help balance the performance trade-off
- Normalizing advantages with a fixed standard deviation, preventing exploding weights in the network and helping to further stabilize optimization
- Clipping parameter weights to -1000 and +1000, doing this helped prevent NaN propagation during training, which was an issue we ran into frequently with our first PPO iteration, bounding training and preventing long-term observation due to exploding gradients.
- Converted environment to grayscale, like in our DQN, hastening training and minimizing overhead for state-related operations. Even with our increase in optimization epochs, this allowed our overall training speed to greatly exceed our initial algorithm.



In terms of results, our new algorithm worked much better than our previous attempt, but still struggled with consistently positive rewards. We notice a reward spike after ~160 steps, but the sudden increase followed by an increase and policy and value loss gives us reason to believe this is the result of overfitting. Either way, the results for this algorithm were inconsistent with the expected, gradual increase, warranting further research and optimization.

For our final assignment, we were tasked with creating software capable of testing multiple versions of our PPO algorithm simultaneously. In terms of design, we opted to create a unified data structure “Config” that allows us to easily set hyperparameters in the constructor. Each instance also contains a reference to its own environment (set to grayscale), optimizer, and neural network. Each configuration is trained in parallel, as the algorithm iterates through each configuration each episode. We chose this system over sequential learning, because we wanted to

compare the growth of each algorithm and be able to directly compare them. We leveraged the same training algorithm we used in Task 8.

```
configurations = [
    Config (
        name = "PPO_27",
        action_std_init = 0.5,
        action_std_min = 0.1,
        warmup_episodes = 500,
        decay_episodes = 1000,
        epsilon = 0.2,
        gamma = 0.95,
        epochs = 4,
        learning_rate = 1e-4,
        clip_max_norm = 0.5,
        nn_weight_clamp = 1e3
    ),
    Config (
        name = "PPO_1",
        action_std_init = 0.5,
        action_std_min = 0.1,
        warmup_episodes = 100,
        decay_episodes = 150,
        epsilon = 0.2,
        gamma = 0.8,
        epochs = 6,
        learning_rate = 1e-8,
        clip_max_norm = 0.5,
        nn_weight_clamp = 1e3
    ),
]
```