Reinforcement Learning in the CarRacing-v0 Gym Environment

Group 23; Alex Mistretta, Ryan Mistretta, Kartikay Jain
(asm348, rhm255, kj295)

May 11, 2024

1 Introduction

Autonomous driving has been a hot topic when it comes to the applications of Reinforcement Learning algorithms. This is because by training agents to navigate virtual environments, we can transfer this to real life racing vehicles where improving safety and efficiency on the road is absolutely critical especially for the drivers involved. If we look at auto racing competitions like Formula 1 these algorithms can help cars better navigate race tracks, and make cars adapt fast to dynamic conditions. Primarily, they can optimize driver performance through feedback by suggesting race strategies to drivers given the race track conditions, fuel consumption and tire situations whilst making the competitions safer by automatically preventing the car to act erratically on the track. We focus on RL techniques, specifically Actor-Critic, where we will demonstrate how agents can acquire driving skills which would enable advancements in driver strategies, dynamics of automobiles, simulation and driver assistance in the world of motor sport. The code source and be accessed using this link: https://colab.research.google.com/drive/1QWDICKcLBQME6IxJs8TnfAew899emBO5?usp=sharing

1.1 Car Racing

To model this we will use reinforcement learning for training agents in the racing environment called CarRacing-v0 Gym environment. It is provided by OpenAI's Gym toolkit and the main objective is to make sure a race car can properly go around the track. The current state is represented by a 96x96 RGB image that consists of the track, the car and other features. This rendering also allows you to see the agent's actions. The cars action space is in a continuous setting, and consists of the brake (0 to 1), acceleration (0 to 1), and the wheel position (-1 to 1). The reward function is devised such that a cost is incurred for going off the track and the agent receives a reward at each time step for which it remains on the track. This promotes the car to accelerate as well as stay on the track.

1.2 Actor-Critic for Car Racing

Actor-critic enables the training of agents in driving simulations, such as the CarRacing-v0 Gym environment. This method leverages one of the principles discussed in our class. This method consists of an "actor" component that proposes actions based on a policy that maps states to actions, effectively deciding the behavior of the agent. We also have a "critic", which evaluates the actions taken by the actor by computing a value function. Note that this value function estimates the future rewards the agent can expect from its current state. This combination allows the actor to adjust its policy based on the feedback from the critic, which assesses how good the action was in terms of expected future rewards. This approach not only streamlines the learning process but also proves to be cost-effective for agent training. The following steps outline the setup we will use for actor-critic in the CarRacing-v0 Gym environment.

Expert Trajectories Collection: Collect a dataset of trajectories of various states and actions taken by an expert driver to stay on the track in the environment.

Data Processing: Extraction of relevant actions and states from the expert demonstration dataset in order to make it into a suitable format for learning in the CarRacing-v0 Gym environment.

Model Design: Picking the right model architecture to map between actions and states. Convolutional neural networks are particularly useful as a neural network for data that comes from images.

Model Training: Here the goal is minimize the difference between actions taken by the expert in the race track and those predicted by the model by training the model using the expert trajectories.

Model Evaluation and Fine-tuning: After training, the performance of the model is assessed through the average reward obtained and other metrics for asking how well the agent follows the expert's behaviour. Following this the model architecture could be modified or hyperparameters adjusted to improve model performance.

2 Problem

We will be exploring how to come up with a policy that ensures the car doesn't deviate off the track during turns as well as continually moves in a forward direction. To ensure this, the policy would have to balance the speed of the car, its braking and its steering to help the car remain in the lane. To do this these are the steps we will follow:

- Extract the track boundaries during observation state processing
- Set up the reward function in such a way that the agent is penalized for crashing or going off the track
- Include track boundaries in the model for the state of the environment apart from the position/speed of the car
- Fine-tune hyperparameters such as discount factor, batch size and learning rate to get the optimal training configuration

3 Approach

For our Python environment we used Google Colab. First, we pip installed several packages needed for working with Reinforcement Learning in the CarRacing-v0 Gym Environment. We first installed the Simplified Wrapper and Interface Generator tool which helps in compiling C/C++ extensions for Python packages. Then we installed the Gym package along with the Box2D physics engine which is used in the CarRacing-v0 environment.

We also installed Stable Baselines3 because it helps in the implementation of most Reinforcement Learning algorithms we have learnt in class such as Soft Actor Critic and Proximal Policy Iteration. Additionally, we made sure PyTorch was installed for training RL agents along with its torchvision library given that that the Car Racing environment is image/video based.

We also imported a lot of important modules/libraries needed to set up the environment for running reinforcement learning experiments in the Car Racing gym environment in Google Colab. For example, we imported the sys module to access variables maintained by the interpreter. We also installed the necessary system packages for rendering environments in Google Colab, especially renderlab and colabgymrender which help in rendering Gym environments.

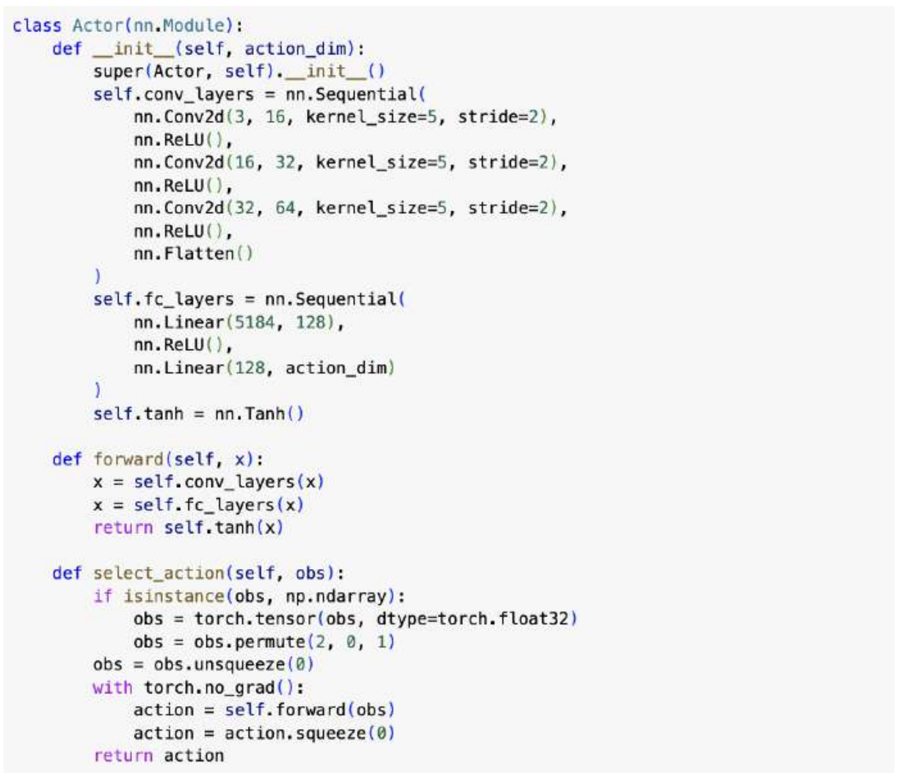
We also installed the moviepy package for create videos of the car and the imageio package for reading and writing image data. Finally, we imported the NumPy library for numerical computing, the Matplotlib library for making visualizations and we also set seeds to ensure the data was reproducible. In summary, we made sure all required dependencies were first installed before running RL experiments on the CarRacing-v0 Gym environment.

Following this, we set up the Gym environment for the CarRacing task, configured it to rend frames as RGB arrays and to save the rendered frames in a specified directory. Then we defined a visualize function which helped visualize the performance of the agent in the CarRacing-v0 Gym environment. It simulates an episode in a specific environment, allowing the agent to interact with it. Then it visualizes the episode through rendering.

3.1 Approach 1

3.1.1 Construction

Actor: We defined the Actor Class as follows:



```
class Actor(nn.Module):
    def __init__(self, action_dim):
        super(Actor, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Flatten()
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(5184, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return self.tanh(x)

    def select_action(self, obs):
        if isinstance(obs, np.ndarray):
            obs = torch.tensor(obs, dtype=torch.float32)
            obs = obs.permute(2, 0, 1)
        obs = obs.unsqueeze(0)
        with torch.no_grad():
            action = self.forward(obs)
            action = action.squeeze(0)
        return action
```

We defined this Actor Class using a convolutions neural network architecture utilizing the nn.Module class provided by PyTorch for modelling neural networks. The reason for using CNN's is that the observation space is an RGB image, and CNN's are great for classifying and working with images. The series of convolutional layers are responsible for extracting features from the input image the fully connected layers could further process the extracted features.
The structure for the convolution layers are as follows:

- First convolution layer: 3 input channels, 16 output channels, 5x5 kernel, stride 2, followed by ReLU activation
- Second convolution layer: 16 input channels, 32 output channels, 5x5 kernel, stride 2, followed by ReLU activation
- Third convolution layer: 32 input channels, 64 output channels, 5x5 kernel, stride 2, followed by ReLU activation
- Flatten the output to prepare for fully connected layers

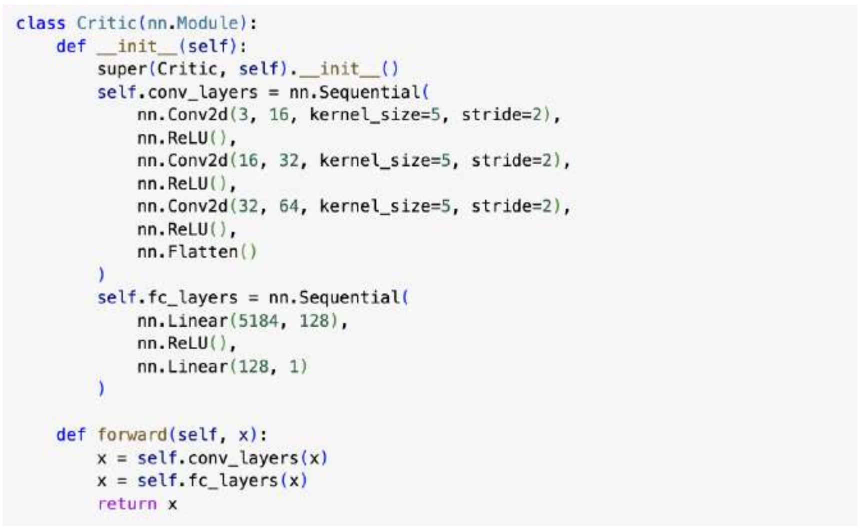
The structure for the fully connected layers are as follows:

- First linear layer: 5184 inputs, 128 outputs, followed by ReLU activation
- Second linear layer: 128 inputs, and action dim outputs (number of actions)

Following this in our code we also implemented a forward pass to ensure the input is propagated through the network. First the input is passed through the convolutional layers and then the fully connected layers. Finally, the Tanh activation function is applied to ensure the action value remain in the valid range for the environment.

We also defined a select action function that selects actions based on the current observation. To do this the function converts the observation to a PyTorch tensor and passes the observation through the network so that it can get the action tensor.

Critic: We defined the Critic Class as follows:



```
class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Flatten()
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(5184, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)
        return x
```

We defined this Critic Class in a similar way as the Actor Class where the PyTorch nn.Module class is used for neural network modeling. The convolutional layers are followed by fully connected layers.
The structure for the convolution layers are as follows:

- First convolution layer: 3 input channels, 16 output channels, 5x5 kernel, stride 2, followed by ReLU activation
- Second convolution layer: 16 input channels, 32 output channels,

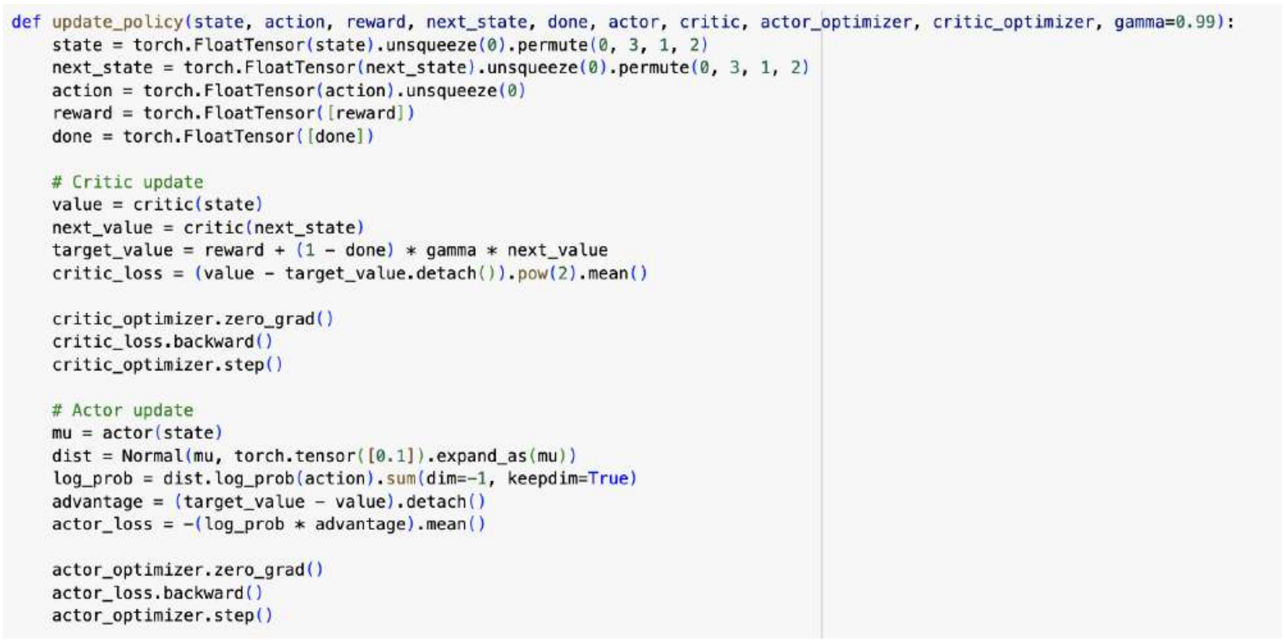
5x5 kernel, stride 2, followed by ReLU activation
- Third convolution layer: 32 input channels, 64 output channels, 5x5 kernel, stride 2, followed by ReLU activation
- Flatten the output to prepare for fully connected layers

The structure for the fully connected layers are as follows:

- First linear layer: 5184 inputs, 128 outputs, followed by ReLU activation
- Second linear layer: 128 inputs, and 1 output

Following this in our code we also implemented a forward pass to ensure the input is propagated through the network. First the input is passed through the convolutional layers and then the fully connected layers.

Policy Update: We implemented the Policy Update as follows:



```
def update_policy(state, action, reward, next_state, done, actor, critic, actor_optimizer, critic_optimizer, gamma=0.99):
    state = torch.FloatTensor(state).unsqueeze(0).permute(0, 3, 1, 2)
    next_state = torch.FloatTensor(next_state).unsqueeze(0).permute(0, 3, 1, 2)
    action = torch.FloatTensor(action).unsqueeze(0)
    reward = torch.FloatTensor([reward])
    done = torch.FloatTensor([done])

    # Critic update
    value = critic(state)
    next_value = critic(next_state)
    target_value = reward + (1 - done) * gamma * next_value
    critic_loss = (value - target_value.detach()).pow(2).mean()

    critic_optimizer.zero_grad()
    critic_loss.backward()
    critic_optimizer.step()

    # Actor update
    mu = actor(state)
    dist = Normal(mu, torch.tensor([0.1]).expand_as(mu))
    log_prob = dist.log_prob(action).sum(dim=-1, keepdim=True)
    advantage = (target_value - value).detach()
    actor_loss = -(log_prob * advantage).mean()

    actor_optimizer.zero_grad()
    actor_loss.backward()
    actor_optimizer.step()
```

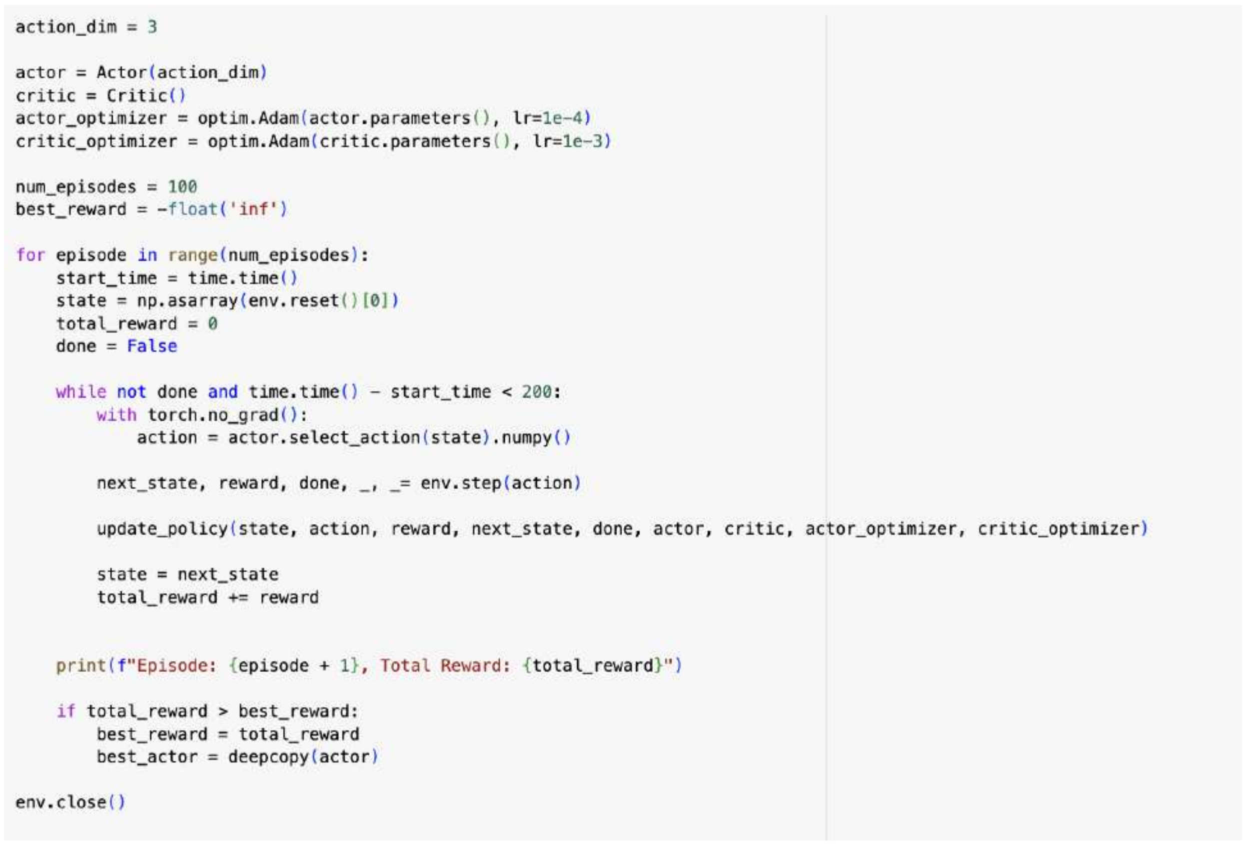
Input Extraction:
We first reshaped the state tensors to match the expected input shape of the neural network and extracted the action, reward, and termination results.

Critic Update:
We then updated the critic which was computed using the estimated state-value for the current state and the next state using its network. We then calculated the target value for the current state-action pair using the Bellman equation. Finally, the critic loss was calculated as the MSE (mean squared error) between the target and estimated value following which we performed back-propagation to update the critic network. Essentially, the critic here estimates the value function.

Actor Update:
For the actor update, we computed the mean action for the current state using the actor

network. Then using a normal distribution with the mean action and standard deviation of 0.1 we computed the log probability of the action under the current policy. The advantage was then calculated as the temporal difference error between the estimated and target value. The actor loss was found through the negative log probability of the action multiplied by the advantage. Finally, we used back-propagation to update the actor network. Overall, the actor updates its policy based on the advantage computed from the critic's evaluation of the state-action pairs.

Training Loop: We implemented the Training Loop as follows:



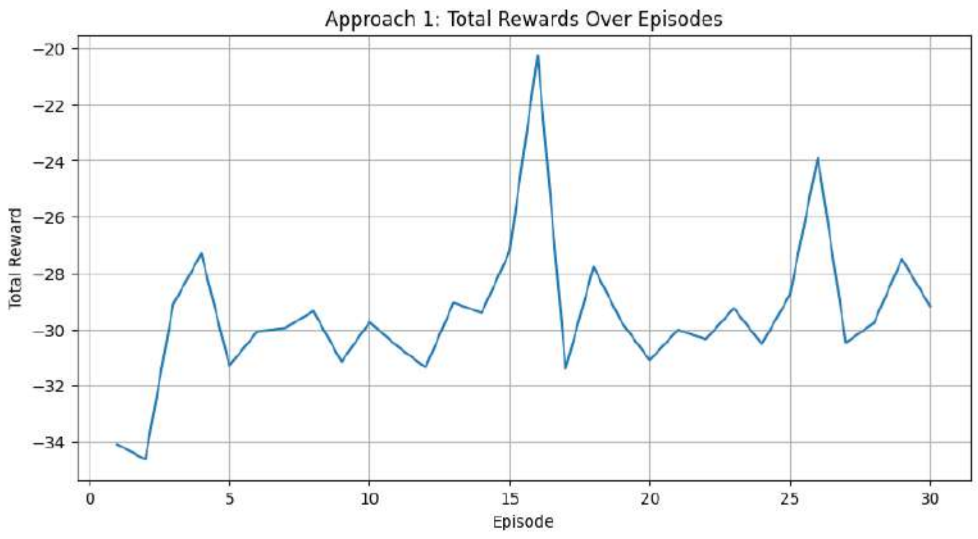
```python
action_dim = 3

actor = Actor(action_dim)
critic = Critic()
actor_optimizer = optim.Adam(actor.parameters(), lr=1e-4)
critic_optimizer = optim.Adam(critic.parameters(), lr=1e-3)

num_episodes = 100
best_reward = -float('inf')

for episode in range(num_episodes):
    start_time = time.time()
    state = np.asarray(env.reset()[0])
    total_reward = 0
    done = False

    while not done and time.time() - start_time < 200:
        with torch.no_grad():
            action = actor.select_action(state).numpy()

        next_state, reward, done, _, _= env.step(action)

        update_policy(state, action, reward, next_state, done, actor, critic, actor_optimizer, critic_optimizer)

        state = next_state
        total_reward += reward


    print(f"Episode: {episode + 1}, Total Reward: {total_reward}")

    if total_reward > best_reward:
        best_reward = total_reward
        best_actor = deepcopy(actor)

env.close()
```

The training loop for the actor-critic RL agent using the above definitions was implemented. The actor and critic networks were initialized as well as the Adam optimizers for both of them (actor optimizer and critic optimizer). We then iterated over a fixed number of episodes and in each episode, we reset the environment and initialize the state. Note thate we have 2 stopping conditions, when the enviroment is terminated and a time constraint. The reason for the time constraint is that we found that the action would often converge at some extreme and not be able to leave the state. In some cases, the car would continue to go in a circle and the enviroment would be unlikely to termiate. We track the total reward obtained in the episode and also if the episode has terminated. We select an action first from the actor network and then step in the environment. From this observed transition, the policy, state, and reward are updated. At the end of each episode, the episode number and the total reward obtained is printed and the best actor is kept.

Overall, the actor-critic agent interacts with the car racing environment, updates the policy based on observed rewards and states so that the performance is improved over multiple episodes.

3.1.2 Results

The results for the first approach are shown in this graph below of total reward as a function of the number of episodes. Note that we used 30 episodes as to minimize the training time when producing the graph (using 100 episodes got similar results).



The graph above shows the total reward as a function of the episode number. We notice that there's a large spike in the graph around episode 16, and medium spikes in the graph at episodes 2 and 26. This is most likely contributed to actor accelerating and increasing reward, and then decelerating while obtaining negative reward from not moving across frames fast enough (or possibly moving off track). This is behavior that we would not have expected, given that our reward function promotes acceleration and staying on track. Looking at our visual simulation, we see that the actor simply stayed at a stand still, occasionally moving at an extremely slow pace. This agrees with the reward function described above. So, this leads us into our next approach: scaling our braking and gas functions in order to account for all possible values that our actor needs.

3.2 Approach 2

The first thing we realized when looking at our expert and the environment again, is that we were improperly scaling the breaking and gas. Previously, we were using the Tanh activation function for steering, breaking, and gas, which scaled all 3 of these between -1 and 1. However, breaking and gas take on values between 0 and 1, meaning we were losing a lot of information using the Tanh function. We therefore did the following:

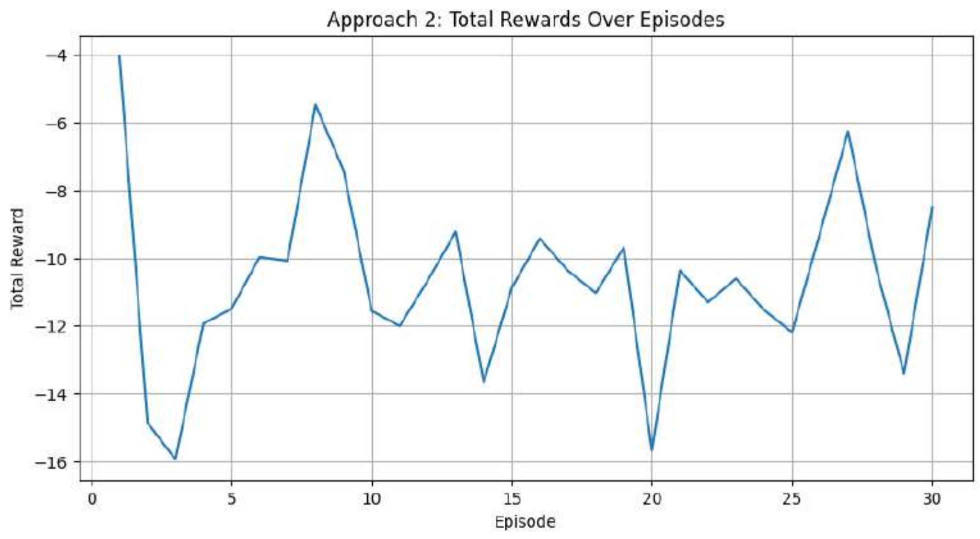
3.2.1 Construction

To scale the gas and brake different than the Tanh function, we would need to extract all three actions. We then would need a different activation function for gas and breaking to scale between 0 and 1; we chose the sigmoid activation as it fit the description well. The was the only thing we changed in this approach.
We re-implemented the Forward Pass as follows:

```
def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)

        steering = torch.tanh(x[:, 0])
        gas = torch.sigmoid(x[:, 1])
        brake = torch.sigmoid(x[:, 2])

        return torch.cat((steering.unsqueeze(1), gas.unsqueeze(1), brake.unsqueeze(1)), dim=1)
```

3.2.2 Results

The results for the second approach are shown in this graph below of total reward as a function of the number of episodes. Note that we used 30 episodes as to minimize the training time when producing the graph (using 100 episodes got similar results).



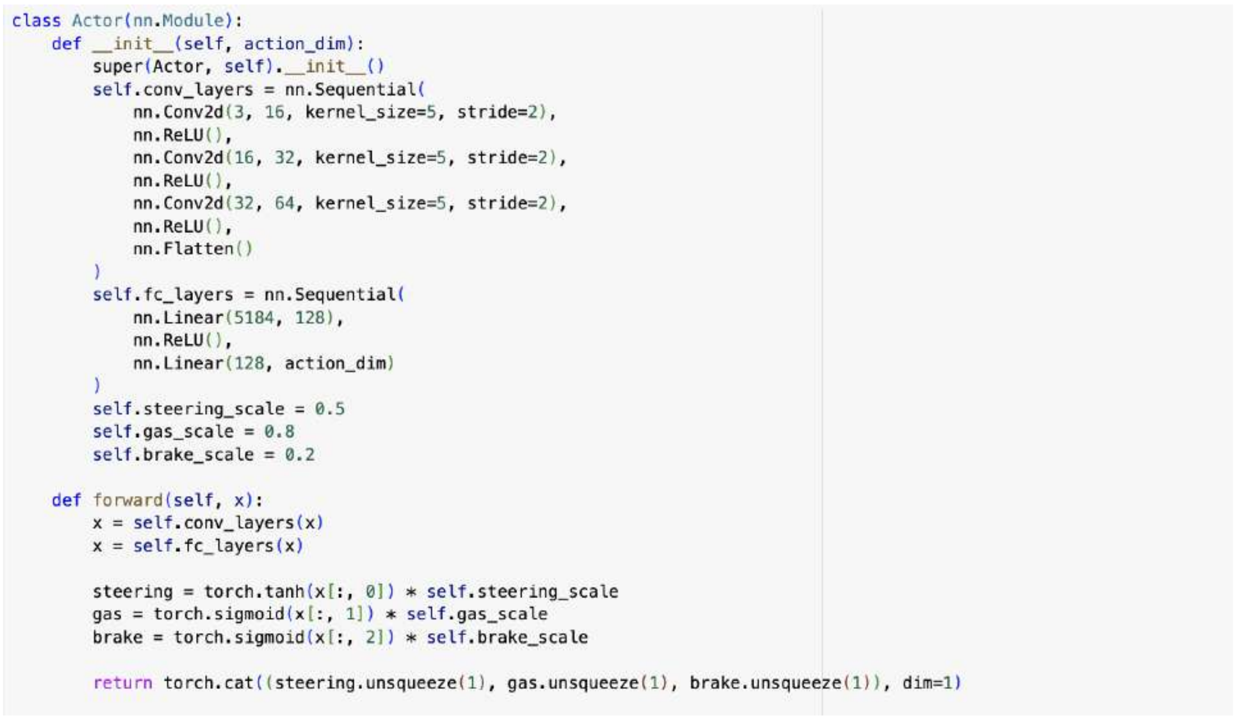
The graph above shows the total reward as a function of the episode number. We notice that the behavior is rather erratic and contains many spikes, both steeply increasing and steep decreasing spikes. This is improvement from the previous approach, as now we see that our actor is more expressive than before. Instead of a plateaued graph, we see a dynamic reward with potential for an actor that can learn. However, there is still work to be done. The reward is not increasing as a episodes increase. This is behavior that we would not have expected, given that our reward function promotes acceleration and staying on track. With our added functionality, our actor can now steer and accelerate correctly. Looking at our visual simulation, we see that the actor simply stayed at a stand still, occasionally moving at an extremely slow pace. This agrees with the reward function described above. So, this leads us into our next approach: adding parameters to our steering, gas, and breaking functions in order to stop each from converging to an extreme.

3.3 Approach 3

We then realized that there was an issue with the actions the actor was getting. When initializing, the actions would quickly find a local minimum reward and as a result, stayed as the same steering, gas, and brake parameters. The steering converged to -1 or 1, while the gas and brake converged to either 0 or 1. Note, it was not exactly these number but something extremely close to them. We therefore did the following:

3.3.1 Construction

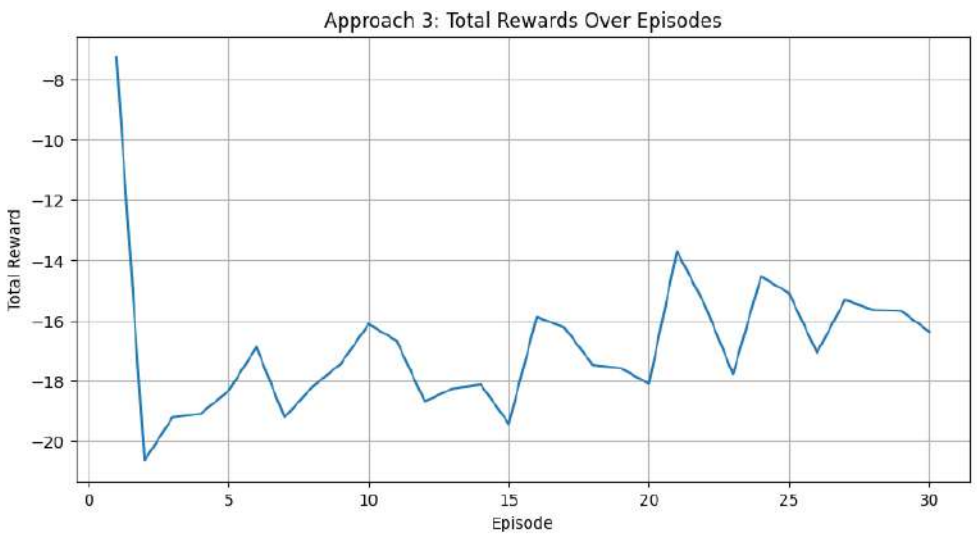
We decided to add more parameters in order to scale the steering, gas, and brake. These parameters should be able to be learned in order to adjust accordingly but our hope was to avoid the parameters converging to the extreme. Instead they would hopefully converge to the parameters initialization but the reward function would not fall into a local minimum of staying still. We initialized the steering scale in order to only allow the steering to be values between -0.5 and 0.5 in order to not allow the car to continually go in a circle. We initialized the gas scale at 0.8 and brake scale at 0.2. The was in order to hopefully enforce the car to move forward rather than staying completely still.
We re-implemented the Actor initialization and Forward Pass as follows:



```python
class Actor(nn.Module):
    def __init__(self, action_dim):
        super(Actor, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(16, 32, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=5, stride=2),
            nn.ReLU(),
            nn.Flatten()
        )
        self.fc_layers = nn.Sequential(
            nn.Linear(5184, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )
        self.steering_scale = 0.5
        self.gas_scale = 0.8
        self.brake_scale = 0.2

    def forward(self, x):
        x = self.conv_layers(x)
        x = self.fc_layers(x)

        steering = torch.tanh(x[:, 0]) * self.steering_scale
        gas = torch.sigmoid(x[:, 1]) * self.gas_scale
        brake = torch.sigmoid(x[:, 2]) * self.brake_scale

        return torch.cat((steering.unsqueeze(1), gas.unsqueeze(1), brake.unsqueeze(1)), dim=1)
```

3.3.2 Results

The results for the third approach are shown in this graph below of total reward as a function of the number of episodes. Note that we used 30 episodes as to minimize the training time when producing the graph (using 100 episodes got similar results).



The graph above shows the total reward as a function of the episode number. We notice that the behavior has improved much from the previous approach. While still containing many spikes, there is a slow increase in reward as the episodes increase. Looking at our visual simulation, we see that the actor simply went in a circle, and went extremely slow. So, this leads us into our next approach: changing the internal reward function to promote desired behavior.

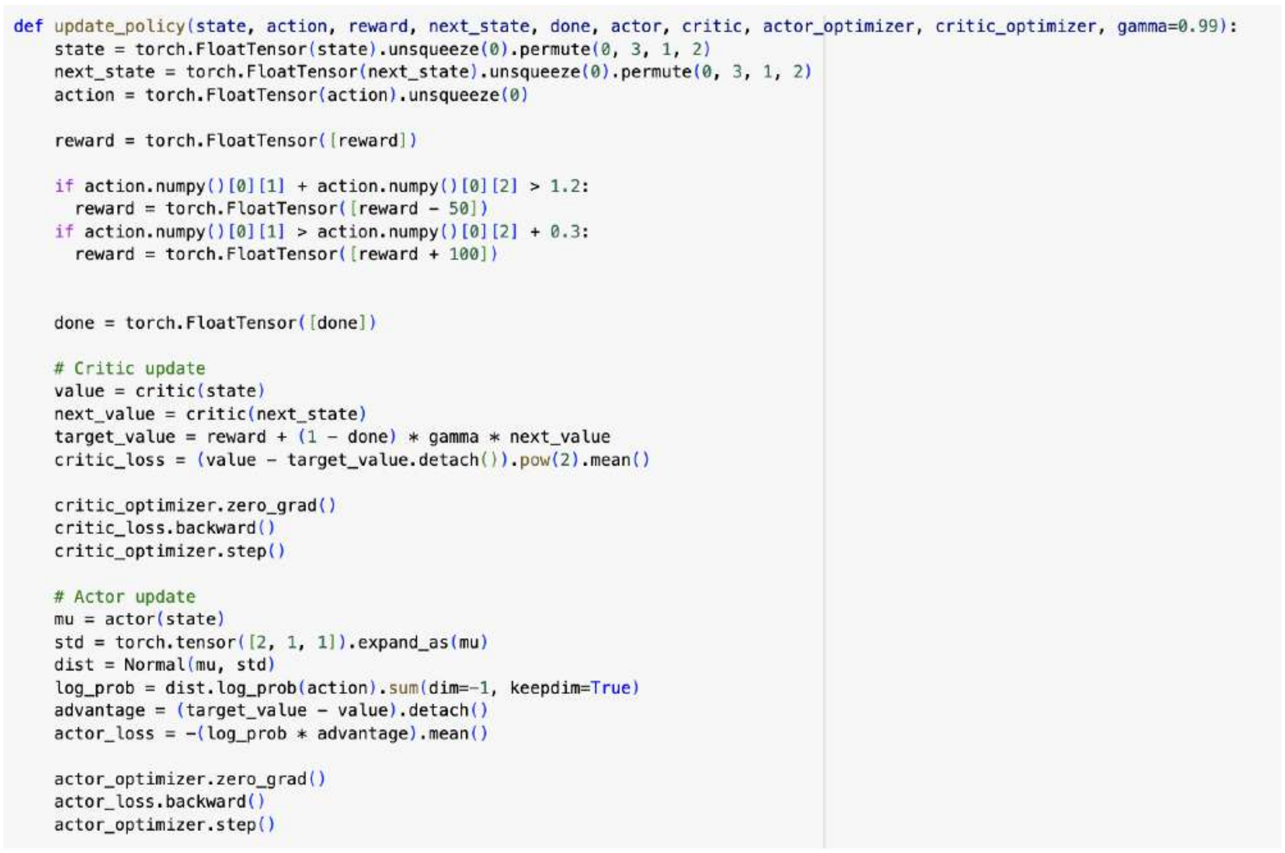
3.4 Approach 4

After approach 3 didn't work as well as expected, we decided we needed to try something else in order to "encourage" the car to move forward and not completely steer to the left or right. We decided the best way to do this was to look at the reward function provided by the environment and edit this to encourage the attributes we want. The current reward function is as follows: "The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track. For example, if you have finished in 732 frames, your reward is 1000 - 0.1*732 = 926.8 points."

3.4.1 Construction

We decided to add a simple reward function as an add-on to the current one. Note that we got rid of the gas scale and brake scale from the previous part as they resulted in no improvement. Our idea was to have decrease in the reward if the sum gas and brake is above a certain threshold (we chose 1.2). The reason for this is that it does not make sense to have both gas and brake as high numbers, and this would hopefully prevent each one to converge to a very high number. Our next idea was to encourage

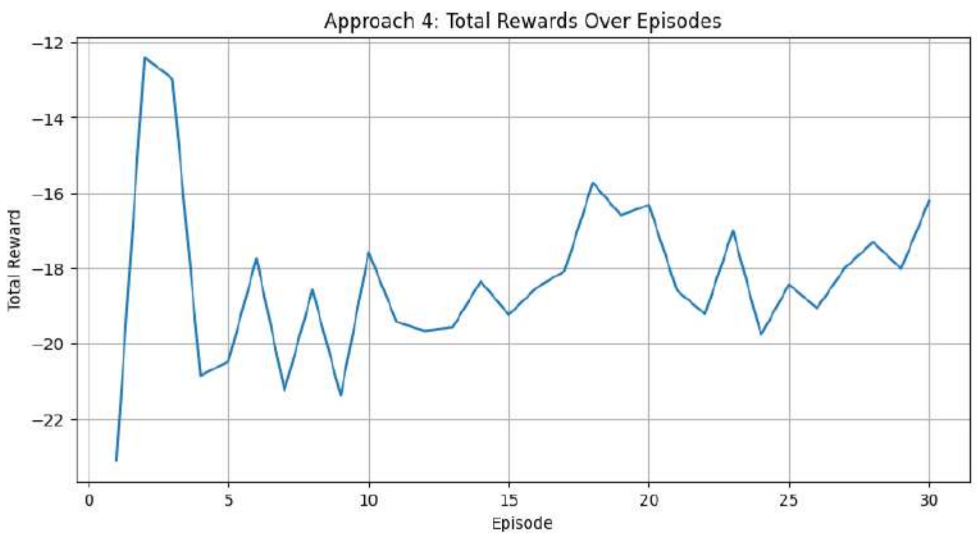
the gas to be above the breaking by a certain number (we chose 0.4). The reason for this was the car would often not move at all and therefore by rewarding having the gas above the breaking then we would further encourage the car to move forward. We also realized that by increasing the standard deviation, it would make the model more likely to fall out of the local minimum of staying still and perhaps explore track more. These updates took place in the update policy function.
We re-implemented the update policy function as follows:



```
def update_policy(state, action, reward, next_state, done, actor, critic, actor_optimizer, critic_optimizer, gamma=0.99):
    state = torch.FloatTensor(state).unsqueeze(0).permute(0, 3, 1, 2)
    next_state = torch.FloatTensor(next_state).unsqueeze(0).permute(0, 3, 1, 2)
    action = torch.FloatTensor(action).unsqueeze(0)

    reward = torch.FloatTensor([reward])

    if action.numpy()[0][1] + action.numpy()[0][2] > 1.2:
      reward = torch.FloatTensor([reward - 50])
    if action.numpy()[0][1] > action.numpy()[0][2] + 0.3:
      reward = torch.FloatTensor([reward + 100])


    done = torch.FloatTensor([done])

    # Critic update
    value = critic(state)
    next_value = critic(next_state)
    target_value = reward + (1 - done) * gamma * next_value
    critic_loss = (value - target_value.detach()).pow(2).mean()

    critic_optimizer.zero_grad()
    critic_loss.backward()
    critic_optimizer.step()

    # Actor update
    mu = actor(state)
    std = torch.tensor([2, 1, 1]).expand_as(mu)
    dist = Normal(mu, std)
    log_prob = dist.log_prob(action).sum(dim=-1, keepdim=True)
    advantage = (target_value - value).detach()
    actor_loss = -(log_prob * advantage).mean()

    actor_optimizer.zero_grad()
    actor_loss.backward()
    actor_optimizer.step()
```

3.4.2 Results

The results for the fourth approach are shown in this graph below of total reward as a function of the number of episodes. Note that we used 30 episodes as to minimize the training time when producing the graph (using 100 episodes got similar results).



The graph above shows the total reward as a function of the episode number. We notice that the behavior has improved much from the previous approach. While still containing many spikes, there is a slow increase in reward as the episodes increase. Looking at our visual simulation, we see that the actor simply went in a circle, and went extremely slow. We hypothesize that training for more episodes could greatly increase the performance of our actor, given that the horizon is in an essentially infinite setting.

4 Overall Results and Future Ideas

From the first iteration, we were able to make significant advancements in training our Actor-Critic model. In the first approach, we set up the Actor class which included the initialization, forward pass, and a function for selecting an actor. We set up the Critic class which included the initialization and forward pass. We also set up the updating policy and training loops. The result of this was the car stayed completely still as it found a local minimum in the reward of not moving at all.

In the second approach, we separated the action into steering, gas and brake and changed the activation function to scale them properly. For steering, we used the Tanh function as to scale it between -1 and 1. For the steering and gas, we used the sigmoid function to scale them between 0 and 1. The result of this was the car still stayed completely still as it found a local minimum in the reward of not moving at all.

In the third approach, we realized that the steering, gas, and brake were all going to -1, 0, or 1 (their extremes) and staying there. This meant that the actor had found a local minimum in the reward and could no "get out" of it. This resulted in the car staying completely still for the entire episode. To fix this, we added more parameters the actor can learn in order to scale down the actions. For steering we initialized a parameter at 0.5 to scale steering between -0.5 and 0.5. For gas we initialized a parameter at 0.8 and brake a parameter at 0.2 in order to initially encourage the car to move forward. The result of this was the parameters of the car still converged to its extreme. This made the car moving forward and steering right, which resulted in the car going continually in a circle for the whole episode.

In the fourth approach, we attempted to change the reward function to directly encourage the car to move forward and not brake. To do this, we decreased the reward if the sum of the breaking and gas actions were above 1.2. We also increased the reward if the gas parameter was at least 0.3 above the breaking parameter. We also tried to increase the standard deviation of the actions, changing from 0.2, 0.1, 0.1 to 2, 1, 1 for steering, gas, and breaking respectively. This was in hope that the car would explore more of the track and not get stuck in a local minimum. This made the car moving forward and steering right, which resulted in the car going continually in a circle for the whole episode.

Due to time constraints, we were unable to make any more progress towards training the expert. If we did, our next approach was to try combining our model heuristic approach of staying in track. This approach would pass the image through a image filter and detect the edges of the road. We would then calculate the distance of the car from the middle of road and reward the car for staying in the middle. We started this implementation as seen in the code but could not get the image segmentation to work. The next model we would try it to pass that as an additional input in the the model to have it train on the addition information.

Overall, we were able to create an Actor-Critic implementation to train an expert on the Car Racing environment. Through the four implementations, our model steadily improved from staying completely still to going continually in circles. If time permitted we could try to add in another heuristic in order to encourage the car to drive straight and between the lines. This is what we expected as it is known to be very hard to train an expert on an environment like the Car Racing, especially in limited time.

Acknowledgments

We would like to thank the TA's (specifically Prithwish) and Professor Choudhury for a great semester and for helping on our final project.

References

[1] Gym Documentation, `https://gymnasium.farama.org/environments/box2d/car_racing/`.