Nafis Abeer, 518 HW1

1.1 a) Was able to use np.load to load in the npy files into observation and action tensors

1.1 b) The training.py file loads in a classification network from network.py, sets an optimizer for the loss function and loads the observations and actions. The purpose behind dividing the data into batches is memory efficiency. We can't fit the entire dataset into GPU memory at once. Batches allow the data to be loaded a bit at a time. We're also converging faster to a minimum of the loss function using Adam and batches. SGD is already faster and less likely to hit a local minima compared to Gradient descent. Adam likely works similarly with batches. Regularization effect of dividing into batches may help overfitting (Answer aided by GPT-4) An epoch is one complete pass through the whole dataset. This means one pass through each of the batches once. Lines 43-48 gets the output from the neural network, compares the loss, zeros out all the gradients so that they are not impacted by the previous batches' gradients. Loss.backward computes the gradient for each of the parameters (weights) with respect to the loss. The optimize.step function updates all the parameters while following the adam rules. total_loss is accumulated for reporting the loss from each epoch. The batch_in and batch_ground truth is reset before the next batch is loaded for the epoch.

1.1 c) Steer takes a value of -1 for steering left and 1 for steering right. Gas takes a value of 1 and Break takes a value of 1. These values are rational and seem like they can be set to discretized values with a maximum magnitude of 1.0. The defined actions initially was selected to be the 9 possible combinations of actions, each with a set discrete value between -1.0 and 1.0 for steering and [0.0, 1.0] for steering and braking. I decided against having steer_slow and steer_hard options. I also opted to just use accelerate, brake steer_right and steer_left to keep the classification options limited and optimize data collection and the models decision. The function definitions are in network.py and their development was debugged with the help of gpt-4.

1.1 d) Decided to keep the 3 channels for the time being to be able to differentiate between the Road and grass colors. Network architecture math to determine number of input and output channels after kernel math and stride calculations were completed with gpt-4.

1.1 e) Hyper parameters being tuned to just mess with the number of channels, adding a convolutional layer, and lowering the learning rate. After observing stagnant training, I've decided that the best way to improve would be to get more data. I don't think the original data was good or diverse enough to begin with. I did find the optimal learning rate for the time being though.

1.1 f) Good training data would include equal amounts of all possible kinds of actions and situations. This means balancing the dataset to not have too many of one action. The Data would include sufficient amounts of actions in every situation. Good data should also include sufficient amounts of imperfect situations to teach the car how to recover. Perfect imitation could lead to overfitting and the robot isn't guaranteed to end up in the same states as the human.

1.2 a) Extract sensor values function grabs the portion of the image that has the speed, breaking and steering values displayed as they are applied. The section also appears to have a score displayed but not extracted. It also has values from the gyroscope. I will note that adding these features only did enough to make sure that the car did not accelerate too fast. I am still seeing an issue of the car just going to the grass and doing donuts.

1.2 b) I had 4 classes and each of their outputs were considered as a part of a threshold to select if we are steering left, right, forward or braking. Ultimately, I had to dynamically set the thresholds after observing the gradients at each iteration. I also had to collect data multiple times and redesign the action to classes and scores to actions functions to incorporate the thresholds of the various scores.

1.2 c) For a regression problem, an MSE loss seemed appropriate. I implemented it as just the different between the user's ground truth action which is hardcoded to a set value and the AI's selected value with is aimed to be between the given range of possible values for each action but the network picks up on these ideal values over time. The advantages of a regression network in that automatic score to action mechanism. We don't have to define hard coded classes, we just drive as we see the world. We could even have more buttons to define steeper actions while driving or maybe increase or decrease steering width with buttons. This would improve the regression model's performance. The classification network could be good to imitate the user perfectly because the class definitions are discretized to match the user's possible inputs. A regression model is more reasonable if we collect enough training data with discrete inputs. I saw the regression better so for my final model, I stuck with the regression model.

1.2 d) My overall performance was based off of the regular regression network vs a regression network that was taking gray scaled inputs and another network that was taking inputs where the images are cropped and zoomed in more. The overall performance improved when using the cut and zoomed model.

1.2 e) I had done some dropout with the initial classification network but saw the performance worsen. This could be due to the small size of the network and limited training data, the model probably didnt have enough capacity to drop off learned weights. I mentioned that I noticed significant improvements when messing with the learning rate. I also got improved performance from changing the network architecture and adding an extra layer of convolution. The learning rate change was encouraged by observing the scores collapsing on a single action so I had to loosen the learning rate and reduce it. The same could be said for why I decided to balance the dataset as well. Improving network architecture and collecting more data was a last ditch effort before implementing dagger.

1.3 a) Proof completed by me. This section is GPT: In the context of imitation learning, behavior cloning (BC) directly learns the mapping from states to actions by minimizing the discrepancy between the policy's actions and the expert's actions. However, BC suffers from compounding errors since the learned policy might encounter states that are off the demonstrated trajectory, leading to potentially poor performance.
On the other hand, DAGGER attempts to rectify this by training the policy not just on the expert's data but also on its own data collected over time. Therefore, it can learn to correct its own mistakes and potentially generalize better to unseen states. In the context of urban driving DAGGER can yield a performance similar to Behavior Cloning when the state distribution encountered during policy execution is similar to the state distribution in the training data. In such cases, the covariate shift is minimal, and both DAGGER and BC could perform similarly. In urban driving, a scenario that could lead to quadratic cost might be when the autonomous vehicle is navigating through densely populated or highly dynamic environments. If the vehicle's policy is not well-optimized, the cost could increase quadratically with the number of time steps, as each wrong decision could lead the vehicle into increasingly complex and challenging situations, further exacerbating the cost.

1.3 b) I think I have the correct code for implementing DAGGER but my data balancing function does not appear to be working properly. This is leading to my inability to train DAGGER algorithm properly. I do think that I thought of the proper way to collect data and train the algorithm over many iterations. My implementation could be found in the main function under the dagger() method. I will submit a much better plot as soon as possible.

1.3 a) assuming $\ell(s,\pi)$ is 0-1 loss and

cost function C bounded in $[0,1]$

let $\mathbb{E}_{s\sim d_{\pi^*}}[\ell(s,\pi)] = \epsilon$

$\rightarrow \Delta C(s)$ $\quad$ $\nearrow C(s,\pi^*)$ $\quad$ $\nearrow C(s,\pi)$

* think of $\ell(s,\pi)$ as difference in cost suffered by optimal policy vs. learned policy given a state

* Then ΔC can be at most 1 b/c C is bounded in $[0,1]$, so the absolute difference b/w two values that are a result of $C(\cdot)$ is $|0-1|$ or $|1-0|$ at the most

* if $\ell(s,\pi)$ is the loss in cost suffered from not following π^* then $\ell(s,\pi)$ is also $= \Delta C(s)$

* since $s\sim d_{\pi^*}$, $\Delta C(s)$ can be expected to equal $\mathbb{E}_{s\sim d_{\pi^*}}[\ell(s,\pi)] = \epsilon$

First we have at a state by state basis:

$$\Delta C(s) = C(s,\pi) - C(s,\pi^*)$$

We expect these values to be

$$\underbrace{\mathbb{E}_{s\sim d_{\pi}^*}[\Delta C(s)]}_{\epsilon} = \mathbb{E}_{s\sim d_{\pi}^*}[C(s,\pi) - C(s,\pi^*)] = \mathbb{E}_{s\sim d_{\pi}^*}[C(s,\pi)] - \mathbb{E}_{s\sim d_{\pi}^*}[C(s,\pi^*)]$$

we rearrange to get

$$\mathbb{E}_{s\sim d_{\pi}^*}[C(s,\pi)] = \mathbb{E}_{s\sim d_{\pi}^*}[C(s,\pi^*)] + \epsilon$$

Over T time steps, the expected cost from following a policy accumulates to $J(\cdot)$

$\hookrightarrow$ $J(\pi) = J(\pi^*) + T\epsilon$ $\swarrow$ loss is accumulated at each step.

since ϵ is at most $= 1$, at every step, we can accumulate a total loss of $T\cdot 1$ at most so:

$$J(\pi) \leq J(\pi^*) + T^2\epsilon$$