Arthur Wang
Jason Ha

EC 518 Homework 2 Report

Disclaimer: We paraphrased certain parts of our report from ChatGPT.

1.1 D: Explain what is 'good' training data? Are there any problems with only perfect driving demonstrations?

"Good" training data is typically data that is beneficial for everyone and aims to contain examples of different driving scenarios. For example, in CARLA, we would typically want data that involves the car obeying all traffic signals like stop signs and speed signs while avoiding collision with both cars and pedestrians. With perfect driving demonstrations, there will be outside factors that can affect the car. One example could be when other people driving may possibly collide with the perfect driving demonstration.

1.2 B: To start with, we formulate the problem as a classification network. Have a look at the actions provided by the expert imitations, there are three controls: acceleration, steering and braking. Which values do they take? Since they are not independent (accelerate and brake simultaneously does not make sense), it is reasonable to define classes of possible actions, like {steer left}, {}, {steer right and brake}, {gas} and so forth. Define the set of action-classes you want to use and write the methods that convert actions to classes and scores to action. Your code should map every action to its class representation which will be a one-hot encoding as well as compute an action from the scores predicted by the network during inference.

The value that acceleration takes is between 0 to 1.
The value that steering takes in is from -1 to 1.
The values that breaking takes in is from 0 to 1.

1.2 D: Implement the forward pass for your network, which is the function forward in network.py. Given an observation, it should return the probability distribution over the action-classes predicted by the network. You can decide whether you want to work with all 3 color channels or convert them to gray-scale beforehand. Motivate your choice briefly. Train your network by running python3 training.py. Can you achieve better results when changing the hyper-parameters? Can you explain this?

We decided to work with all 3 color channels instead of turning the image into a grayscale one. The motivation was that we decided that color can provide some decent information. Specifically, traffic lights being green and red will have more information if the image is kept in color rather than grayscale.

Fine tuning the hyper-parameters has a great effect on the training process. For example, if the learning rate is too high, the model tends to make great gains in terms of loss reduction to begin with but struggles when it potentially jumps around from local minima to local minima. Other things such as batch size have an effect as well. We've noticed that for higher batch values the loss tends to be lower. This can be attributed to the fact that you can get a better estimation of the gradient when sampling more than one observation.

See network.py and training.py.

1.2 E: The module training.py contains the training loop for the network. Read and understand its function 'train'. Why is it necessary to divide the data into batches? What is an epoch? Please answer shortly and precisely.

Dividing the dataset into smaller batches helps fit the data and model parameters into GPU memory, enabling efficient computation without exceeding hardware limits. Also, by increasing the number of samples considered at one time, the model is better able to approximate the actual gradient when converging. An epoch refers to a complete iteration through the entire training dataset in one cycle for training the machine learning model.

1.2 G:

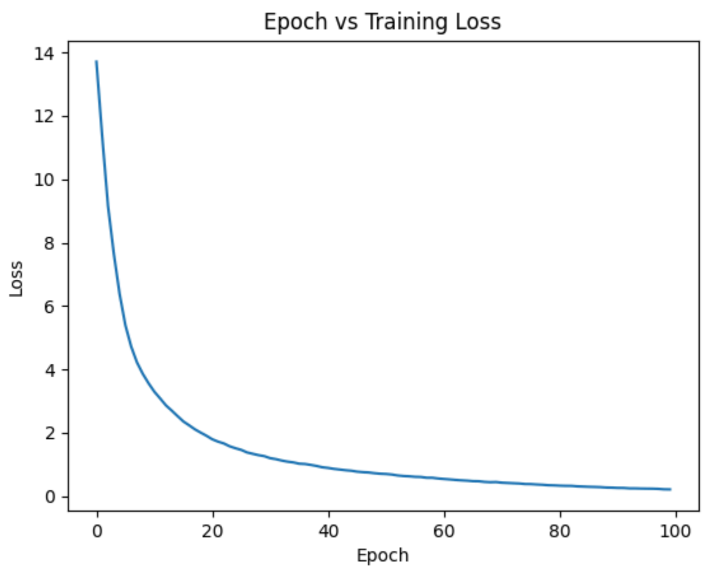

Figure 1. Epoch vs. Training Loss for Best Base Classification

Network & Parameters	**Lowest Loss**
Base Classification (LR = 1e-5)	0.213

1.3 A - Observations: The training data of the network can also contain more information than just the image from the car in the environment. Look at the mounted sensors in manual control.py. What are those sensors? Incorporate it into your network architecture. How does the performance change?

Some of the other sensors that are available are the accelerometer, gyroscope, speedometer and GNSS. We incorporated them by directly concatenating them with the flattened output of the convolutional layers. However, I'm not convinced if this is a good strategy given the output of our convolutional layer is a vector containing tens of thousands of neurons vs the handful from the sensor values.

See training.py and network.py.

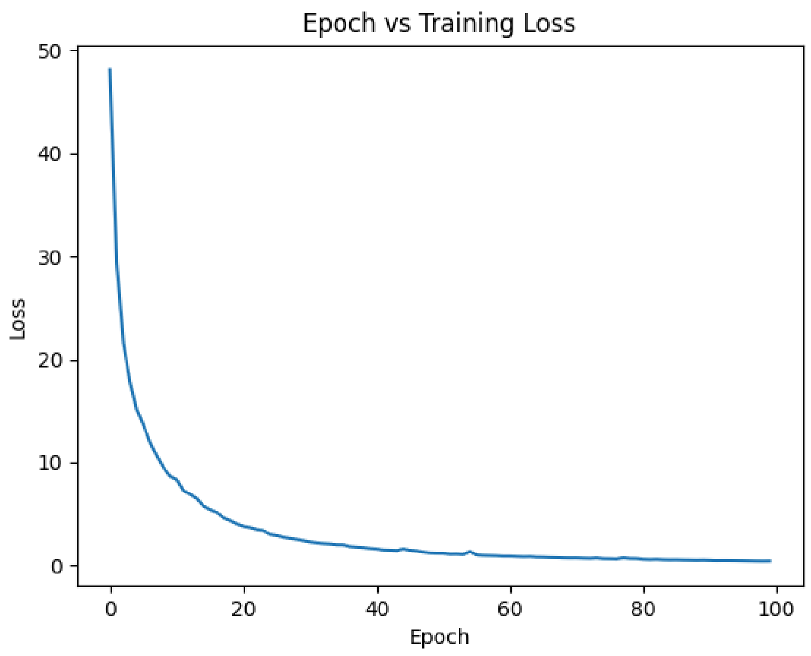

Figure 2. Epoch vs. Training Loss for Best Classifier with Observations, Did not go below 1.0 for Loss

1.3 C - Classification vs. Regression: Formulate the current problem as a regression network. Which loss function is appropriate? What are the advantages / drawbacks compared to the classification networks? Is it reasonable to use a regression approach given our training data?

For the loss function of the regression network, we chose mean squared error. Some advantages of a regression network compared to classification is that the network is able to output action commands that are continuous. This means that the driving behavior of the regression network should be smoother and more precise than the classification network. Related to this is the fact that you do not need to discretize the continuous steering, throttle and braking commands into classes. A drawback is that the network could potentially overfit easier if the data is not varied enough.

See network.py and training.py.

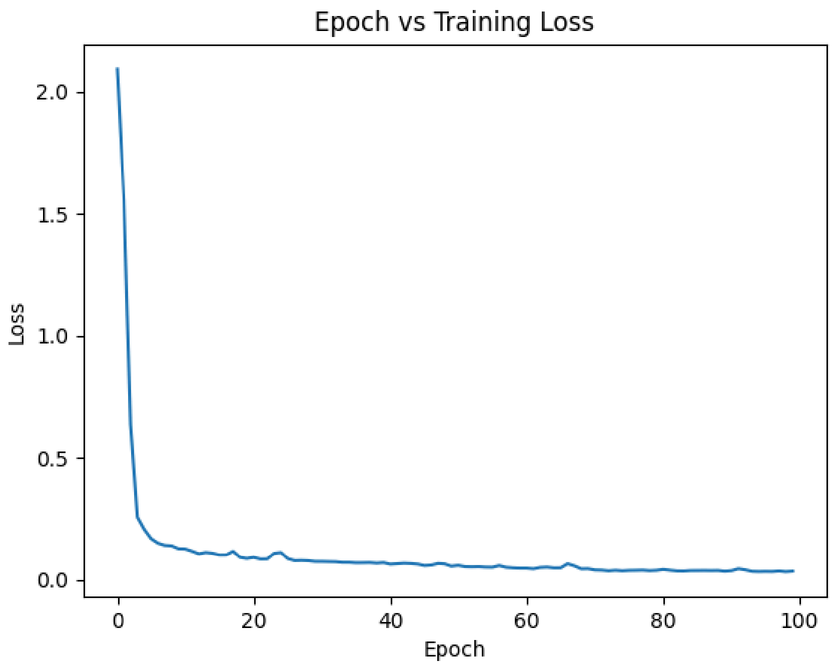

Figure 3. Epoch vs. Training Loss for Best Regression Network (included other fine-tuning such as batch normalization layers)

Network & Parameters	**Lowest Loss**
Regression w/ Batch Norm (LR = 0.75e-5)	0.034

1.3 D - Data augmentation: As discussed in the lecture, the more versatile the training data is, the better generally. Investigate two ways to create more training data with synthetically modified data by augmenting the (observation, action) - pairs the simulator provides. Does the overall performance change?

We planned to investigate augmenting the data by slightly changing the steering, throttle, or brake values for each given image. Another method would be to subtly change the image such as applying rotation, distortion or other transformations to help the model better deal with outliers. An interesting augmentation would be to put fake cars into the image and adjust the control values accordingly. We were unable to complete the data augmentation experiment.

1.3 E - Historical observations: As autonomous driving is a sequential decision-making process, past observations also have a certain impact on current decision-making. Concatenate prior frames as a current observation - do you notice an improvement in driving performance? Why or why not?

We were unable to complete this experiment.

1.3 F - Fine-tuning: What other tricks can be used to improve the performance of the network? You could think of trying different network architectures, learning rate adaptation, dropout-, batch- or instance normalization, different optimizers or class imbalance of the training data. Please try at least two ideas, explain your motivation for trying them and whether they improved the result.

Some other tricks we used were learning rate adaptation and batch normalization. The motivation for learning rate adaptation was that we wanted to have a good balance between large initial reductions in loss and good loss convergence. As such, we started with an initially high learning rate and then reduced it over each epoch. From each training run performed, it did seem to decrease the number of epochs taken to get to a good loss value. However, the final loss wasn't substantially lower than without learning rate adaptation.

The rationale for batch normalization is that by normalizing activations, we can reduce overfitting and help with convergence. Over several training runs, it seems like batch normalization does reduce the overall loss during training.

See network.py and training.py for specifics.

Network & Parameters	**Lowest Loss**
Best Classification w/ Batch w/ LR Scheduling (1e-5 -> 0.5e-5)	0.09
Best Regression w/ Batch w/ LR Scheduling (1.5e-5 -> 0.75e-5)	0.039

1.4 A - DAGGER Failure: What could be done to help DAGGER converge faster? Please also answer when does DAGGER lead to a performance similar to Behavior Cloning? Are there common scenarios in Urban Driving that could lead to quadratic cost?

A couple ways we can help DAGGER converge is through active learning, progressive complexity, and reward shaping. Active learning is used to implement mechanisms, which allow the agent to query the expert for actions in uncertain situations, providing more targeted learning opportunities. Progressive complexity starts training on simpler tasks and gradually increases the complexity. This can help the agent build foundational skills before tackling harder scenarios. Giving DAGGER incentives can better guide the agent's learning and produce more desired behaviors

DAGGER can lead to a performance similar to behavior cloning when the policy being trained approaches the expert's policy closely. As such, the expert and actor are in close agreement and the loss between them becomes smaller. This is even more true if the expert's data represents all the situations the new policy encounters comprehensively. As such, new situations do not provide too much extra information to learn from.

Some common scenarios in urban driving that can lead to quadratic costs are very complex and busy situations such as intersections, large groups of dynamic obstacles and multiple agents interacting with another. If these scenarios occur, there could be compounding errors as time goes on as untrained for circumstances lead to even more untrained for circumstances.

1.4 B - Implement the DAGGER algorithm: Implement the DAGGER algorithm [3] below and apply it to your dataset. Do you observe an improvement in performance? What scenarios benefit most from DAGGER? Compute and plot the regret, what do you observe? Compare the result and report your findings.

We were unable to fully complete the DAgger implementation and got stuck debugging the training step of the network using combined data. However, during the new training epochs it could be seen that there was a significant mse loss when retraining the regression network of around 0.3 during some DAgger iterations.

We were unable to plot the regret as the DAgger implementation never fully trained to completion.

See dagger.py and training.py.

1.5 - Competition

See the model named sdacreg_op34_lr0p75en5.pth which is our competition model. It uses regression and batch_normalization.

See test_agent.py for out implementation of the model in the agent.