BASIL: Branched Architecture Self-Driving Imitation Learning

Woodrow Wang
Stanford University
wwang153@stanford.edu

Tyler Yep
Stanford University
tyep@stanford.edu

Abstract

End-to-end approaches for autonomous driving with imitation learning have become increasingly popular. However, without training on high-level commands, it is not possible to supply the agent with high-level commands indicating a user's intent at test time. To solve this problem, we explore different architectural approaches for supplying high-level driving commands in imitation learning models. Our model is a branched architecture that forks based on the supplied high-level command, making the model supervised via its architecture with shared parameters in its perception layers. We evaluate our model with the Udacity Self-Driving Car Simulator in a grid world with many intersections. To interpret our model's performance, we inspect common errors, activation maps, and adversarial examples.

1. Introduction

Imitation learning has been shown to deliver promising results in training autonomous driving agents, given the ease of obtaining expert demonstrations of human driving. In our problem, we seek to map RGB images with high-level commands to steer/throttle controls for the driving simulator.

Imitation learning, also known as behavioral cloning, has been successful in lane following [2] [4]. The systems, however, have a few critical limitations. The network trained by Bojarski *et al*. only predicts steering angle from perceptual inputs. We adapt our problem to also predict throttle, as we believe it to be central to the goal of safe autonomous driving. More importantly, the Bojarski *et al*.'s network does not consider high-level commands as a feature.

One critical limitation of imitation learning is the assumption that the proper control can be inferred solely from a perceptual input. Namely, in our work, we investigate the issue of an intersection. When a car approaches an intersection, a perceptual input alone does not provide enough information to inform the autonomous agent to go straight, turn left, or turn right. In fact, each of those options are equally valid maneuvers, making the distribution of control multimodal. Pomerleau *et al*. emphasizes that "upon reaching a fork, the network may output two widely discrepant travel directions, one for each choice" [8]. To mitigate this ambiguity, we provide as input the desired high-level command to our network and experiment with varying architectures for optimally handling the high-level commands.

Figure 1. Aerial view of the Intersection Track

2. Related Work

As inspiration for our baseline, Bojarski *et al*. [2] designed a network that maps raw pixels from a front-facing camera to steering commands to tackle the task of lane following. We build on this baseline architecture by incorporating high-level driving commands as inputs to the model to supervise its prediction. We also extend the task to predicting steering angle and throttle to allow our agent to assume full control of the car's control.

We investigate the effectiveness of the branched architecture proposed by Codevilla *et al*. [3]. The model has shown promise in outperforming a network that simply accepts the high-level command as an additional input feature. Instead, the architecture forces a distinction in the learned policy by splitting into sub-modules based on the high-level command. This also allows the model to use its entire representational capacity for learning a policy to control the car autonomously without needing to focus on planning, which an external agent, such as a human user or GPS, can supply. We further explore the work in [4] to examine the lim-

itations of imitation learning and methods to improve the generalization of our model.

Another promising architecture suited for this task is the CNN-LSTM model proposed by Hege *et al*. [6], which uses a recurrent CNN in order to capture temporal dependencies while driving in the CARLA simulator. However, these models can become very computationally expensive, as the CNN-LSTM must compute a series of forward passes using previous timesteps in order to predict the action at the current timestep. Without significant computational power, the latency of predictions makes the driving task much more difficult. As we explain in our problem statement, we limit our task to making a prediction for one image at a time, irrespective of previous timesteps, in order to demonstrate the task completion with low computational cost.

3. Dataset

Our final dataset on the Intersection Track consists of 200,000 labeled images of the front view of the simulated car, taken from three different angles and using various forms of data augmentation. All images were created by driving around our simulated world, with roughly 5 hours of driving data. Our train/dev/test split is 90%/5%/5%.

In order to better interpret our trained model's actions as well as make the task more manageable, we limit the maximum speed on the Intersection Track to 8 mph. Each image is of shape (66, 200) and is cropped to remove the vehicle itself from being present in the image. For each image, we have the corresponding speed, steer, throttle, and high-level intention. The steer values range in the interval (-1, 1) and the throttle values range in the interval (0, 1).

3.1. Simulator

We collected data and conducted all experiments using the open-source Udacity Self-Driving Car Simulator. The vanilla simulator provides a simple interface to drive the car and save a recording. The recordings are saved as a series of RGB images along with labels of the current throttle and steer angle. We modified the simulator to also save the current high-level control setting at each timestep.

For our project, we modified the simulator to accept a high-level control as input (pressing I denotes the high-level intention of going straight, while pressing J or L denotes the high-level intention of taking the next available left or right turn, respectively). We use this interface to help us collect data with high-level intentions as well as supply high-level control during test time.

3.2. Lake vs Intersection Track

In the Udacity simulator, there is a provided Lake Track that is designed for a simple lane following task. The Lake Track, as seen in Figure 3, has curved roads to make the task of lane following non-trivial, but no intersections or two-lane roads. To test models using high level controls, we created a custom grid-like driving world from scratch in Unity in order to train and test our model, which we refer to as the Intersection Track. The Intersection Track, as seen in Figure 1, is entirely flat and includes many identical four-way intersections, well-suited for testing the navigational capacity of our model. All roads in the course have two lanes, and the car is intended to stay on the right side of the road except when making turns. There are no other vehicles in the course. We have scattered trees along the sides of the course and surrounded the map with non-uniform hills to ensure that the model is able to generalize to semi-realistic backgrounds.

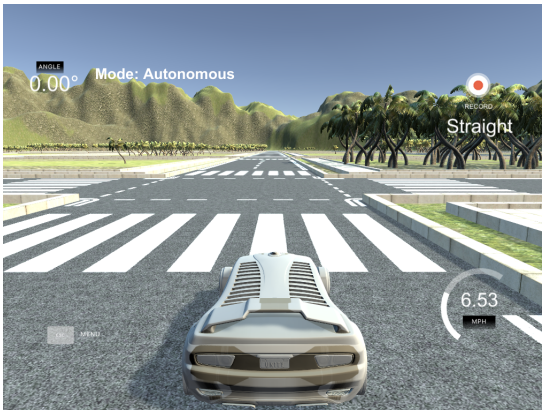

Figure 2. Modified Udacity simulator

Figure 3. Aerial view of the Lake Track

3.3. Data Collection Pipeline

To collect our training dataset, we drove around our simulated test world and recorded all instances when we intended to make a left turn, right turn, or drive straight at an intersection by toggling a key in the simulator. We refrained from crashing into curbs or moving out of our lane except when making turns. We practiced driving around the track before recording any data so that we could drive as close to optimally during data collection as possible. We recorded the data in many traversals through the Intersection Track as

expert demonstrations to be used in imitation learning and obtained a final dataset of 200,000 labeled image examples.

3.4. Data Augmentation

We augment our dataset with several different methods in order to increase the generalizability and robustness of our model. First, when creating batches from our dataset, we randomly choose the images at any given timestep from one of the three cameras in the car. We then correct the steering angle by readjusting with respect to the angle that the image was taken from. This is a crucial step for correcting the steering at any instance our driving agent steps off of the desired trajectory at test time. Additionally, we add random noise to our steering angles to help our model learn to correct itself.

One key problem in our task is the data imbalance between straight-driving and left/right turn images. Since turns are very brief, the dataset is dominated by straight-driving images, even when the high-level control indicates a future left or right turn. Thus, when sampling mini-batches, we ensure that every batch contains a minimum number of "turning" images to balance the data distribution.

4. Problem Statement

Using imitation learning, we seek to design a network that takes as input an observation o_t and predicts an action a_t at each time step t, where the action consists of steer and throttle values. Let our training data be denoted by $D_{train} = \{(o_i, a_i)\}_{i=1}^{N}$ obtained from expert demonstrations. For imitation learning, we assume that the expert exhibits optimal driving behavior and our network seeks to mimic the expert. We seek to learn a function F, parameterized by θ, which takes a set of observations o_i and a high-level command c_i and outputs a predicted action $\hat{a}_i$. Thus, the imitation learning objective conditioned on high-level command becomes:

$$\min_{\theta} \sum_{i} \ell(F(o_i, c_i; \theta), a_i). \tag{1}$$

The observations to our model are RGB images and speed, while the actions consist of steering angle and throttle. We approach this problem as a supervised regression task.

We use MSE as our primary loss function. Thus, we define:

$$MSE(\hat{a}_i, a_i) = \frac{1}{n} \sum_{i}^{n} (\hat{a}_i - a_i)^2. \tag{2}$$

5. Methods

We investigate two main architectural approaches in our work, which we refer to as a Conditioned and Branched approach, respectively. The first approach involves using the high-level command as a one-hot feature in a feed-forward neural network that is concatenated with the output of a perception module. This was suggested in the original convolutional architecture proposed by Bojarski *et al*. [2], and is similar to many conditional models when conditioned on an important label. However, we suspect this approach is limited in that the model is not required to use the feature to accomplish the task of autonomous driving with high-level commands.

As a second more promising approach, we investigate using a branched architecture for conditional imitation learning, as proposed by Codevilla *et al*. [3]. On a high level, our model processes the perceptual input and features at the beginning of the network and branches off into specific sub-modules depending on the high-level command provided. Each module is trained to focus on either driving straight, right, or left. This architecture benefits from parameter sharing among sub-modules in the perception layers and supervising a distinction in learned policy based on the high-level command.

For all models, we minimize the mean squared error between the predicted and true steer and throttle values. While the model in [2] only predicts steer, we choose to predict steer and throttle to give the model full control over the car.

5.1. Nvidia-Conditioned Model

For the Nvidia-Conditioned model, the high-level architecture follows that of Figure 4 (a). The perception module architecture is inspired by Bojarski *et al*.'s network architecture used in their work on DAVE-2 (DARPA Autonomous Vehicle-2) [1]. The first three convolutional layers consist of 5x5 filters and a stride of 2. The last two convolutional layers consist of 3x3 filters and a stride of 1. The measurement module receives the ego vehicle's current speed as a feature. Each convolutional layer follows a scheme of $Conv \rightarrow ReLU \rightarrow Dropout$. The command module consists of a one-hot feature representing the user's high level intention of going straight, turning right, or turning left. The outputs of each of these modules are concatenated and sent through a 4-layer feed-forward network to predict steer and throttle of the car. The hidden layer sizes for the feed-forward network are [100, 50, 10], respectively. Each feed-forward layer except for the last layer follows a scheme of $Linear \rightarrow ReLU \rightarrow Dropout$.

The one-hot high-level-command feature makes the network command-conditional. This feature distinguishes the Nvidia-Conditioned model from the Branched models, as we hope the model will process the feature appropriately to predict the desired steer and throttle, but it is not required to by its architecture.

The Nvidia-Conditioned model has 252,244 trainable parameters, which makes it the most lightweight of our four models.

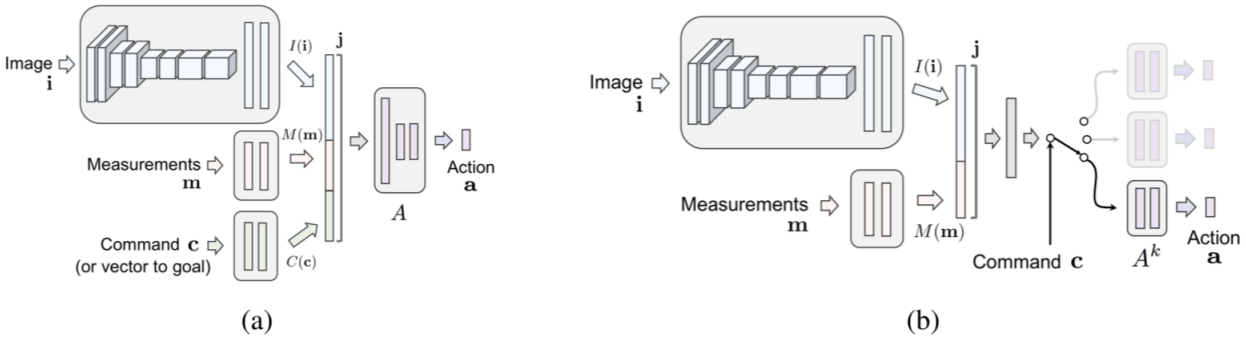

Figure 4. (a) **Conditioned:** the command is used as an additional feature with the measurements and perceptual input. (b) **Branched:** the command decides which module is used for the final control prediction. Figure borrowed from Codevilla *et al*. [3]

.

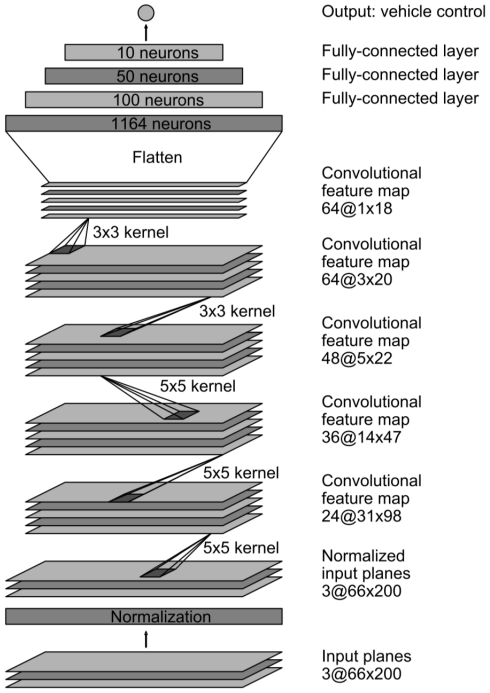

Figure 5. Nvidia-Conditioned network architecture, which we modify as discussed in Section 5.1. Figure borrowed from Bojarski *et al*. [2]

5.2. Nvidia-Branched

For the Nvidia-Branched model, we perform a natural extension of the Nvidia-Conditioned model, but we remove the high-level-command feature and replace it with branched submodules that force distinction in policy through the branched architecture. Instead of relying on the model to learn how to process the command feature appropriately, we explicitly split after the perception module into three feed-forward branches, one for each of the high-level commands.

The perception module is exactly the same architecture as in the Nvidia-Conditioned model. The 4-layer feed-forward network is the same as well, except that there are three separate submodules that are switched between depending on the high-level command.

The Nvidia-Branched model has 493,994 trainable parameters, which is approximately double that of the Nvidia-Conditioned model, but still significantly fewer parameters than both the ResNet18-Branched and ResNet34-Branched architectures.

5.3. ResNet18-Branched

For the ResNet18-Branched model, we seek to leverage the benefits of using a pretrained perception module that potentially can already extract useful features from our input images. Thus, we replace Nvidia's perception module with PyTorch's pretrained ResNet18 on ImageNet [7] [9] [5].

We remove the global average pooling layer at the end of the pretrained ResNet18 network and use the rest of the network as our perception module. Thus, after flattening the output of our perception module, we obtain a 512 dimensional representation of the input image.

For the branched module, we use a 3-layer feed-forward network for each branch with hidden layer sizes of [256, 256], respectively. These hidden layers are inspired by the work of Codevilla *et al*. [3]. Each feed-forward layer except for the last layer follows a scheme of $Linear \rightarrow ReLU \rightarrow Dropout$.

The ResNet18-Branched model has 11,769,414 trainable parameters, which is notably more than the Nvidia-Conditioned model, which makes sense due to the 18 layers in the ResNet perception module. This model benefits from the added expressivity of the ResNet perception module as

well as the pretrained weights on ImageNet, which can help the model recognize features of the road.

5.4. ResNet34-Branched

For the ResNet34-Branched model, we simply replace the ResNet18 perception module with a pretrained ResNet34 perception module. Again, we remove the global average pooling layer at the end of the pretrained ResNet34 network and use the rest of the network as our perception module. The output of the perception module after flattening is then 512 dimensional, which is the same as the output size of the ResNet18. Thus, we use the same architecture as the ResNet18-Branched model for the branched modules.

The ResNet34-Branched model has 21,877,574 trainable parameters, which is nearly double the amount of the ResNet18-Branched model. This makes sense as the perception module has increased from 18 to 34 layers. We have found that the spike in number of parameters can lead to computational limitations discussed further in Section 6.5.

6. Results

6.1. Quantitative Analysis

We performed hyperparameter tuning on a development set of 10,000 examples (5% of our dataset). We evaluated our models with a held-out test set of 10,000 unseen examples (also 5% of our dataset).

We performed a grid search of hyperparameters and mainly focused on tuning the steer correction angle used in data augmentation, dropout, and the number of hidden layers and sizes in our network after the perception module. For our best model, the ResNet18-Branched model, we used an Adam optimizer with a learning rate of 3e-4 and a batch size of 64. The final model consists of a 3-layer feed-forward network with dropout probabilities of [0.0, 0.5, 0.0], respectively.

In terms of numerical results, as seen in Figure 6, we can see that ResNet34-Branched has the lowest MSE on both the train and the development curves. ResNet18-Branched has a very similar performance on the train set, but does not perform as well as ResNet34-Branched on the development set. While both Nvidia-Conditioned and Nvidia-Branched perform worse than the ResNet models, Nvidia-Branched is the better of the two on the development set, which indicates the effectiveness of the branched architecture in solving the intersection problem. As seen in Table 1, in terms of the test set, ResNet34-Branched has the lowest MSE, with ResNet18-Branched, Nvidia-Branched, and Nvidia-Conditioned having slightly higher MSEs, respectively. Note that the ResNet18-Branched and ResNet34-Branched test MSEs are not significantly different, despite ResNet34-Branched having nearly double the number of trainable parameters.

6.2. Waypoint Experiments

We assess our results in Table 1 and Table 2 based on a set of 10 evaluation scenarios designed for a respective track. For each scenario, we evaluate each model's ability to successfully navigate between waypoints in the simulator while counting the number of lane lines crossed, noting the qualitative deviation from the center of the lane, and number of interventions required. We define an intervention to be a need for a manual takeover to recover from a severe misprediction of control onto the original trajectory (note that we allow for multiple potential interventions in each episode). In order to test generalization on the Lake Track, we place the car on a slightly perturbed position at the waypoints to increase the probability of unseen images during scenario evaluation. In order to test generalization on the Intersection Track, we reserve a set of 10 undriven intersections with varying backgrounds across the map for the scenario evaluation.

6.3. Lake Track Experiments

As a preliminary baseline test, we trained the Nvidia-Conditioned model on data obtained from the Lake Track and found that the trained agent could successfully maneuver around the track without collisions. We found that we only needed data on the order of 2 laps around the track (around 2500 examples).

Using the Nvidia-Conditioned baseline model, from Table 2, we can see that our performance in lane following on the Lake Track is significantly better than on the Intersection Track, which is likely because the Lake Track only involves lane following with no complex intersections with multi-modal possible actions.

On the Lake Track, with relatively little training data, we observe impressive visual performance with our baseline agent. The agent keeps to the center of the lane and successfully completes continuous turns around the track. On the Intersection Track, although we provide high-level commands as conditional features, we still see that our baseline trained agent (Nvidia-Conditioned) displays a strong preference towards a single mode of control at an intersection. This encouraged us to investigate the branched architecture in order to help enforce a distinction in policy based on high-level intention. From the results on the Lake Track, it appears as if the perception module is capable of processing the inputs and producing an appropriate control, but the conditioning of the features is not optimal.

6.4. Visual Inspection

Throughout training, we monitored the qualitative performance of our model by inspecting the driving behavior of our model in the Udacity simulator. Some of the notable factors we paid attention to were general stability, need for interventions, and ability to complete episodes as discussed

Intersection Track Results # Trainable Parameters	Nvidia-Conditioned 252,244	Nvidia-Branched 493,994	ResNet18-Branched 11,769,414	ResNet34-Branched 21,877,574
Test MSE	0.0236	0.0193	0.0141	0.0128
# lanes crossed	9	3	1	4
% completion	20%	80%	90%	80%
# interventions	12	2	1	3

Table 1. Evaluation Metrics on the Intersection Track

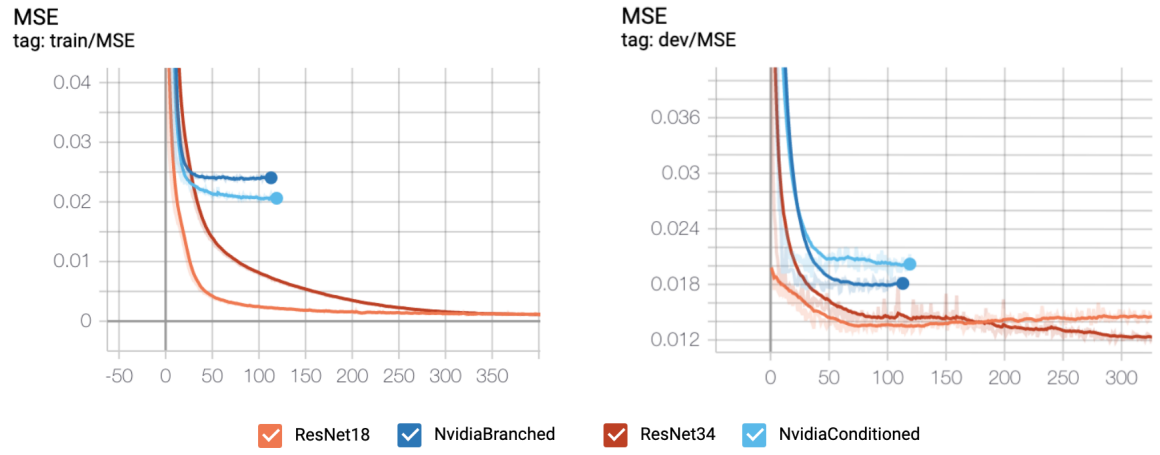

Figure 6. Train and development curves for the branched architecture models

Baseline Results	Lake Track	Intersection Track
Test MSE	0.0226	0.0236
# lanes crossed	1	9
% completion	90%	20%
# interventions	0	12

Table 2. Evaluation metrics comparing performance of Nvidia-Conditioned on the Lake and Intersection Track

in the waypoint experiments in Section 6.2 and summarized in Table 1.

With respect to the waypoint experiments, the ResNet18-Branched model has the best performance, successfully completing 90% of the episodes with only one minor intervention required to help it avoid hitting a curb before a turn. The Nvidia-Branched model performs the second best, but seems to deviate from the center of the lane slightly more than the ResNet18-Branched model. The ResNet18-Branched model drove the most smoothly and nearest to the center of the lane throughout the scenario evaluations.

The waypoint experiments were crucial to understanding and evaluating our model's performance. Although the Nvidia-Conditioned model obtains loss metrics on a similar order of magnitude as the other models, the Nvidia-Conditioned's performance on the waypoint experiments is significantly worse, as it fails to complete almost all turns and becomes heavily biased towards one mode of control, either going straight, turning right, or turning left.

6.5. Computational Limitations

As seen in Table 2, ResNet34-Branched achieves the lowest MSE, but actually performs qualitatively slightly worse than Nvidia-Branched and ResNet18-Branched. Through further investigation, we found that this could be due to a latency issue. Since the ResNet34-Branched contains nearly double the number of parameters compared to ResNet18-Branched, there appears to be a slight delay between the simulator viewing the image and executing the model's next prediction. This latency causes the ResNet34-Branched to perform worse overall qualitatively, as it occasionally fails to complete turns because the steer angles are not changing rapidly enough. While the ResNet34-Branched could perform better with greater compute, we prioritize lightweight models that can accomplish the Intersection Track task over unnecessarily bulky networks with significantly more parameters.

6.6. Activation Maps

In order to interpret our model's predictions, we plotted several activation maps for the ResNet18-Branched, our best model, by showing a heatmap of the first-layer activations. In Figure 7, we include three selected examples of

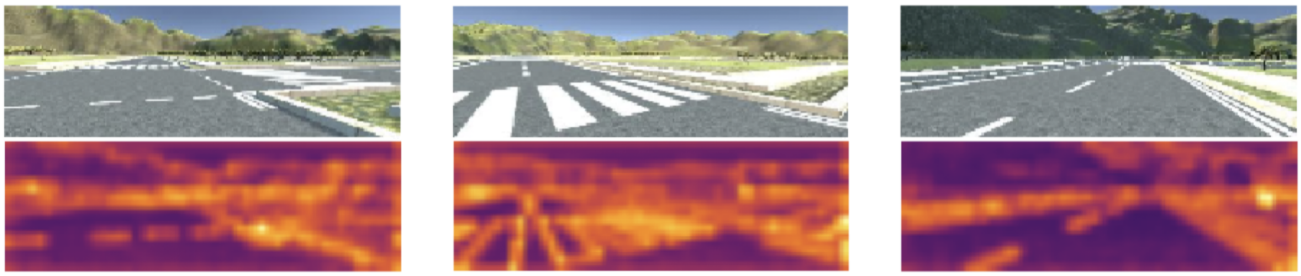

Figure 7. Activation maps for the ResNet18-Branched model. The brighter (more orange-yellow) colors correspond to larger values.

our model's abilities.

In the left image, we can see the vehicle about to complete a right turn. From the activation of the image and input high-level control (turn right), we see the corner of the intersection highlighted, indicating that the model is attending to the corner point, likely to avoid hitting it.

In the center image, we see the outlines of the crosswalk lines in the activation map. Most of the activation maps of our crosswalks show a clear outline of all of the crosswalk stripes. However, in this picture, we see that the right lane of the crosswalk is the brightest part of the map. We can infer from this activation map that the network is attending to the right lane as the location to drive towards and the way to stay in its lane.

In the right image, we see that the network is paying attention to the lane lines and is driving straight. However, we also see that the brightest and most impactful activation is in the right corner - a tree in the distance. We can identify this as a potential weakness with our model, as we would want the model to identify that the most important parts of this image are the lane lines, not a tree in the background.

6.7. Adversarial Examples

In order to examine the vulnerability of our final ResNet18-Branched model, we generated adversarial images that would make our model steer drastically to the right or left when the intended control would be to not steer at all to stay within the lane. We can see the generated adversarial examples in Figure [10]. In order to create the adversarial example that would make the car steer right, we began with an input image with a corresponding near-zero predicted steer and updated the image with the positive gradient of the image with respect to the output steer. We would repeatedly update the image until the predicted steer was above the target adversarial steer. Likewise, we performed a similar method for the adversarial example that would make the car steer left, but we would subtract the gradient so as to decrease the predicted steer.

As seen in Figure [10], the adversarial examples are hard to distinguish from the original image other than a few outstanding pixels, which is concerning for our model to be considered safe and interpretable. Thus, it is important to explore several countermeasures in future work. One such countermeasure is incorporating adversarial examples in the training loop, essentially using the adversarial retraining technique proposed by Goodfellow *et al*. and Huang *et al*. [10]. Another feasible technique is to build an adversarial detection model to be used during test time, as seen in the work of Metzen *et al*. [10]. Considering these methods carefully is crucial to ensuring the deployment of a robust model that behaves predictably and safely.

6.8. Error Analysis

At first, when simply training on data collected from the forward-facing camera inputs of the vehicle without data augmentation, our trained model's performance in the simulator would be extremely vulnerable to slight errors in the predicted control. For example, with the task of driving straight with a steer of zero, a slight non-zero predicted steer at test time would cause the vehicle to diverge from the trained trajectory and crash into an oncoming curb. The data augmentation technique of using three camera angles at train time, suggested in [2], proved to be critical to our model's ability to readjust itself to the center of the lane after a minor prediction error.

Another common failure mode for our model occurs when our model begins too close to a curb while attempting to turn. With the current method of data augmentation, our model is trained to correct itself to be parallel with the lane lines, but not necessarily exclusively driving in the center of the lane. The current data augmentation techniques only add examples of the vehicle readjusting itself when it is at an angle not parallel to the center of the lane. Thus, if the vehicle safely drives parallel to the lane lines but is too close to the curb on the right when attempting to take a right turn, the model occasionally collides with the curb, failing to complete the turn successfully. We could potentially ameliorate this issue by collecting more training data of the vehicle driving back to the center lane whenever it begins to deviate, regardless of the camera angle.

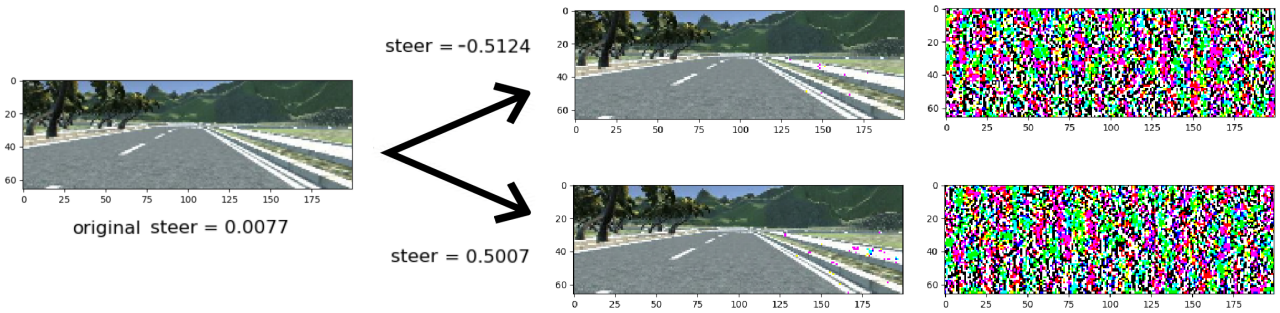

Figure 8. Generated adversarial examples for the ResNet18-Branched model. The left is the original input image. The upper middle is the adversarial image which makes the model steer left. The lower middle is the adversarial image which makes the model steer right. The far right images are the differences between the adversarial images with the original image.

6.9. Generalization

In order to examine generalization within a map, we considered each map, the Lake Track and Intersection Track, in isolation. For each map, we collected training data while deliberately avoiding partitioned areas of the map reserved for testing. Our results for such experiments are in Table 1 and 2. With regards to same map generalization, our models seemed to perform well on unseen examples, which shows promise for inter-map generalization.

During our experiments, we only trained models with data from the Lake Track or data from the Intersection Track exclusively. As a test of inter-map generalization, however, we tried using our final ResNet18-Branched model, trained on the Intersection Track, on the Lake Track. Since the Lake Track has no intersections, we simply used the "Straight" high-level command for these generalization experiments. We found that the ResNet18-Branched model performs a small amount of lane following, even on the curved roads of the Lake Track, which is extremely promising for the generalization ability of our model. Despite starkly contrasting maps in terms of roads, colors, and backgrounds, the driving behavior still seems capable of generalizing, which should certainly be further investigated.

7. Conclusion

Overall, we found that our final ResNet18-Branched model drove the most smoothly and safely based on our evaluation metrics. In our simulator, the model successfully navigates intersections in our grid world using high-level controls, which showcases our model's ability to imitate expert driving and learn the correct starting and stopping times for maneuvers without any temporal information. With the branched architecture, our model is easily able to discern how to apply the learned controls to different intersections in the world.

By analyzing the first layer activations and adversarial examples, we gain insights into the limitations of our model. Specifically, we see that our model may attend to unimportant parts of the image, or become easily fooled by subtle perturbations to the input images. To mitigate these issues, we could expand our dataset immensely and combine examples from different maps in order to help the model generalize to unseen environments. We could also experiment with more expressive models, with the caveat that we would still need to monitor the model's complexity to prevent similar latency issues as we found with using the ResNet34-Branched model.

An important limitation of our work is that our ego vehicle was alone in the environment, with no other dynamic entities in the world. An interesting next step would be to apply our model to a map with multiple vehicles driving along with our ego vehicle. Although we earlier mentioned that Hugh *et al.*'s CNN-LSTM [6] was not a feasible model under our current problem formulation, with more compute, future work on branched architectural imitation learning could examine using a CNN-LSTM in the perception module to capture crucial temporal interactions between vehicles.

8. Contributions

Both Tyler and Woody contributed equally to the project overall. Tyler spent a significant amount of time modifying the Udacity simulator to handle high-level commands. Woody spent a significant amount of time collecting the datasets. Both Tyler and Woody worked together to write the model architectures and parallelized the training by each training models separately. Tyler and Woody both worked together to write the report and create figures.

The first link below is the GitHub repo for the Udacity simulator, which we modified to accept high-level controls at test time. The second link contains the starter code for the files needed to send our model's predicted control to the simulator via client-server response.

GitHub Repositories for original Udacity Self-Driving Car Simulator & starter code:
https://github.com/udacity/self-driving-car-sim/
https://github.com/naokishibuya/car-behavioral-cloning

References

[1] Net-scale technologies, inc. autonomous off-road vehicle control using end-to-end learning. 2004. 3

[2] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. 1, 3, 4, 7

[3] F. Codevilla, M. Müller, A. Dosovitskiy, A. López, and V. Koltun. End-to-end driving via conditional imitation learning. *CoRR*, abs/1710.02410, 2017. 1, 3, 4

[4] F. Codevilla, E. Santana, A. M. López, and A. Gaidon. Exploring the limitations of behavior cloning for autonomous driving. *CoRR*, abs/1904.08980, 2019. 1

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009. 4

[6] H. Haavaldsen, M. Aasboe, and F. Lindseth. Autonomous vehicle control: End-to-end learning in simulated urban environments. *CoRR*, abs/1905.06712, 2019. 2, 8

[7] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 4

[8] T. A. P. Pastor, H. Hoffmann and S. Schaal. Learning and generalization of motor skills by learning from demonstration. *ICRA*, 2009. 1

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017. 4

[10] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017. 7