

# Program Synthesis for Autonomous Driving Decisions

*Yiteng Zhang  
Yang Gao  
Li Erran Li  
Xinyun Chen  
Trevor Darrell*

Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2020-114

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-114.html>

May 29, 2020



Copyright © 2020, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

### Acknowledgement

I would like to thank my research advisor Prof. Trevor Darrell, project mentor Dr. Yang Gao, coauthors, and colleagues at the BAIR lab for their help and advice throughout this project. I would also like to thank my parents and friends for their unconditional love and support.

---

# Program Synthesis for Autonomous Driving Decisions

by Yiteng Zhang

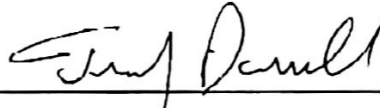
---

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,  
University of California at Berkeley, in partial satisfaction of the requirements for the  
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

### Committee:



---

Professor Trevor Darrell  
Research Advisor

5/29/20

---

(Date)

\*\*\*\*\*



---

Professor Yi Ma  
Second Reader

May 29, 2020

---

(Date)

---

# Program Synthesis for Autonomous Driving Decisions

---

**Yiteng Zhang**  
UC Berkeley  
yiteng@berkeley.edu

**Yang Gao**  
UC Berkeley  
yg@berkeley.edu

**Li Erran Li**  
Scale AI  
erranlli@gmail.com

**Xinyun Chen**  
UC Berkeley  
xinyun.chen@berkeley.edu

**Trevor Darrell**  
UC Berkeley  
trevordarrell@berkeley.edu

## Abstract

There have been many attempts to solve the decision-making problem for the driving behaviors of autonomous vehicles. Early attempts employed empirical human knowledge to define rules explicitly, and thus limited by the diversity of human experience and the depth of human thinking. Recent attempts built neural network models based on Computer Vision and Imitation Learning, and thus vulnerable to unexpected behaviors caused by the black-box system of the model. In this paper, we present an alternative approach to solving the decision-making problem while avoiding the issues above using Program Synthesis. Our model generates human-readable programs to represent the necessary and sufficient conditions for safely executing some chosen driving behavior, and thus significantly increase the model transparency to avoid accidents. At the same time, it is also powered by the virtually unlimited machine learning capacity. Along with our program generation pipeline that generates programs from driving scenes of some chosen driving behavior, we also design a driving simulator that's capable of generating diverse driving scenes efficiently, and a domain-specific language to describe all the conditions we account for. So far, we have built a solid foundation for our approach, and we will continue to tune our model and validate our pipeline on more scenarios.

## 1 Introduction

Autonomous driving companies have traditionally employed empirical rules to decide the driving behaviors for autonomous vehicles. However, there are some major drawbacks to this approach. The rule designer may not cover critical edge cases when designing the driving rules. Even if some edge cases are covered in the design, it may be difficult to test whether the vehicle behaves accordingly, and thus some bugs may stay undiscovered in the codebase. Also, our knowledge is only based on our personal experiences, so they are limited in many ways. Computers may learn hidden rules that are not yet discovered by humans.

As the recent advancement in Imitation Learning and Computer Vision, many have switched to neural-network-based models that make autonomous decisions that mimic human behaviors based on the captured scenes around the vehicle. However, there have been several cases (even life-and-death cases) reported regarding the incapacity of these methods. We cannot capture the omissions in these models beforehand because they are black-box models, as we can only evaluate them by testing them, but not by understating the step-by-step details of how these models make their decisions.

In this work, we focus on increasing the transparency and rationality of driving models by employing a technique called program synthesis. In program synthesis, we are training a model that generates

human-readable programs based on the input-output examples, and then we can both comprehend the program and also evaluate the program on the examples. In our specific problem, given a specified driving behavior, we feed in the driving scenes to our model, and our model generates the conditions for the driving behavior to safely execute. We can read through the program to verify whether it violates our experiences and driving rules. We can also apply these conditions back on to the input driving scenes and check whether the safety of execution predicted by the program matches the ground-truth safety of execution.

Since we use only driving scenes as our training data, we avoid being trapped by the constraints of human knowledge. However, since this also gives the model much more freedom to learn, we need an extensive amount of training data to compensate our model so that the model can learn the correct program generation behavior. Therefore, we design a driving simulator capable of generating a diverse range of driving scenes, so we can use the randomly generated scenes as the training data for our model.

To limit the search scope of our program generation model, we also introduce a domain-specific language (DSL) that is tightly fit to our program scope. The DSL allows describing the conditions on the 10 vehicles around the ego vehicle. For each vehicle, the program can include conditions on 6 different features, including relative displacement and relative velocity to the ego vehicle. These conditions combined specifies the satisfactory requirement for some potential driving behavior.

We have prepared all the components of our pipeline, and we are validating and fine-tuning our model so it can reach a better program generation quality before we move to the next step. We have more steps planned to improve our model and pipeline and we will discuss them in the last section.

## 2 Related Work

### 2.1 Long Short-term Memory

Long short-term memory (LSTM) is a variant of the RNN architecture, in which it is capable of capturing and remembering long-term dependencies (5). This dependency information is encoded in the cell state vector. At the same time, three gates together regulate which pieces of information can move into or out of the cell state vector.

### 2.2 Domain-specific Language

Based on the definition from (4), “a DSL is a language designed to be useful for a specific set of tasks.” A DSL is specified by a set of vocabulary with syntax and grammar rules defined on it. We can use tokens in the vocabulary with correct syntax and grammar rules to build valid sentences in the DSL space.

### 2.3 Program Synthesis

Program synthesis is the process of generating valid programs that satisfy some pre-defined specifications. These specifications may include the syntax and grammar rules from the domain-specific language, and the matching constraints from ground-truth input-output examples. A program and a set of IO examples are considered matching if the program can map each input in the set to its corresponding output.

**Program Synthesis through IO Pairs** (1) provides a great definition for this task: program synthesis aims to compute a synthesizer using the training data of matches between programs and IO example sets such that the synthesizer produces valid programs on the testing data of IO example sets.

**Neural-based Program Synthesis** One common approach to solve program synthesis is to use an encoder-decoder pair (2), where the encoder transforms IO examples into embeddings, and the decoder generates program tokens from the encoded embeddings. The encoder and the decoder are commonly constructed as LSTM networks (1).

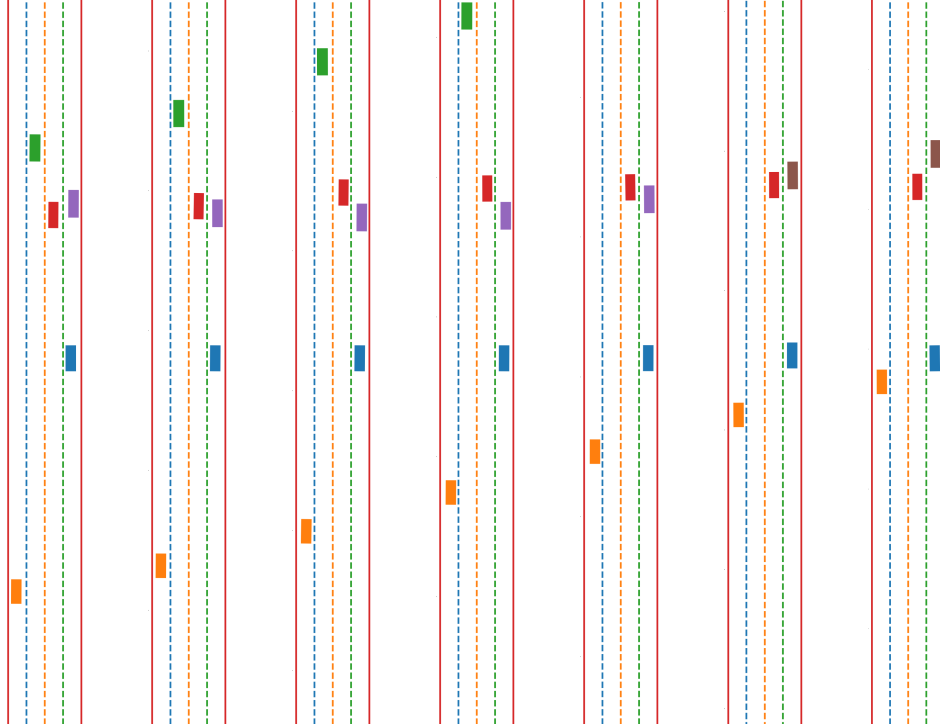


Figure 1: A driving scene with 7 frames from left to right. The ego vehicle (the blue vehicle in the center of the scene) is slowing down to avoid collision with the vehicle in front of it.

### 3 Driving Simulator

We need millions of scenes to support the training of our model. However, there are not sufficient scenes in real-world datasets that present any driving behavior other than lane-following. Therefore, we shift our focus to synthetic data. There have been driving simulators on the market such as the CARLA simulator (3), but they strike too much for realistic rendering of the 3D scenes, and thus consume much GPU power and lead to slow scene generation. Since we are aiming for generating multiple-frame scenes within a second, we then decide to design our own driving simulator with simplified rendering. When we generate a random scene using our simulator, the simulator does not render the 3D world of the scene, but only render a simple top-down view of the scene. This approach reduces the rendering time of scenes to virtually real-time.

The primary goal of our simulator is the ability to generate diverse scenes since diversity is much beneficial to the training of our model. Thus, we try to include as many customization options as possible into our simulator.

#### 3.1 Road Simulation

Our simulator supports the creation of different road styles. For example, it has different configurations for urban roads and freeways. There are also options for customizing the number of lanes on the road, the width of each lane, and even the road speed limit. For example, when we increase the number of lanes, the vehicles may behave more randomly as they have more freedom to change speed and change lanes; when we decrease the lane width, the vehicles may behave more cautiously to avoid collisions.

#### 3.2 Vehicle Simulation

There are many customization options for simulating the vehicles on the road. First, we may specify the density of vehicles on the road to simulate heavy or light traffic, for example, there may be more vehicles in a road segment with the same length on urban roads than on freeways. Also, we

can customize and fine-tune different details of each vehicle in the scene, including location, speed, acceleration, driving direction, driving state, and even vehicle shape (width and length).

**Driving States** Our simulator supports vehicles in different driving states. There are virtually infinite driving states in our simulator. We classify these states into 9 categories to simplify the simulation process, as listed below.

- Lane following while maintaining speed.
- Lane following while speeding up.
- Lane following while slowing down.
- Moving out of a lane towards the left side.
- Moving out of a lane towards the right side.
- Lane crossing from right to left.
- Lane crossing from left to right.
- Merging into a lane from the right side.
- Merging into a lane from the left side.

These categories have causal relations from one to another, and thus we can connect different driving states from these categories to simulate rational driving behaviors for vehicles in the scene.

**Driving Styles** Since some driving states have tendencies to stay or change, we use these tendencies to construct different driving styles. For example, a high-speed driving vehicle will tend to remain a high speed and will be most likely to slow down only to avoid collisions. By adjusting the probabilities of changing states, we can create vehicles with more unique driving styles. For example, by prioritizing the choices of lane following driving states, we create a relatively conservative driving style with a higher tendency of lane following than lane changing.

**Driving Goals** We can optionally set a goal for the vehicles in the scene. For example, a vehicle is driving on the leftmost lane on a freeway and it plans to drive down the freeway, so it needs to move to the rightmost lane. Here, we can set a goal of moving towards the rightmost lane. Another example is when a vehicle tries to drive to the destination as fast as possible. In this case, we can set a goal of higher driving speed. By introducing goals to our simulator, we are increasing both the authenticity and the diversity of our simulation.

## 4 Domain-specific Language

We design a domain-specific language (DSL) that can describe the aggregation of satisfying conditions for certain driving conditions. We only include tokens that are necessary and sufficient to describe all different conditions.

### 4.1 Program Tokens Types

There are 154 different tokens in our DSL, and they are classified into three types: feature-describing tokens (138 tokens), vehicle-describing tokens (10 tokens), and grammar supporting tokens (6 tokens).

**Feature-describing Tokens** These tokens are used to represent feature values of the vehicles around the ego vehicle, for instance, the distance between one adjacent vehicle and the ego vehicle. For each feature, we include 21 discrete values to cut the distribution of common values into 20 windows; we also include a positive infinity value and a negative infinity value for each feature and thus separating all potential values of a feature into 22 windows. We then incorporate a pair of these values in our DSL to describe the satisfying boundaries of a feature. All features and the boundaries of common values are listed below:

- DPX: the component of the displacement between vehicles perpendicular to the driving direction ( $-6m$  to  $6m$ ).

- DPY: the component of the displacement between vehicles parallel to the driving direction ( $-150m$  to  $150m$ ).
- DS: the difference in velocity between vehicles ( $-25m/s$  to  $25m/s$ ).
- DA: the difference in acceleration between vehicles ( $-10m/s^2$  to  $10m/s^2$ ).
- DT: the time it takes for the vehicles to intersect ( $-10s$  to  $10s$ ).
- W: the width of the adjacent vehicle ( $1.65m$  to  $2.05m$ ).

We use these features and values to program construct tokens. For example, the token DPX-0.6 represents a value of  $-0.6m$  in the feature DPX, and the token DS+ represents a value of positive infinity in the feature DS.

Note that we may not need to use all the features above in our training as some features may not be available. For example, we may use only the first two features as our ground truth data. In such cases, we will only use  $23N_{feat}$  different program tokens, where  $N_{feat}$  is the number of features we choose.

**Vehicle-describing Tokens** These tokens are used to differentiate among vehicles around the ego vehicle. In specific, these tokens describe the direction of the adjacent vehicle from the view of the ego vehicle. We include the most commonly related 10 directions, as described below:

- front: the vehicle in front of the ego vehicle.
- back: the vehicle behind the ego vehicle.
- left\_front: the vehicle on the left side of the ego vehicle and surpassing it.
- left\_front\_2: the vehicle in front of left\_front.
- left\_back: the vehicle on the left side of the ego vehicle and not surpassing it.
- left\_back\_2: the vehicle behind left\_back.
- right\_front: the vehicle on the right side of the ego vehicle and surpassing it.
- right\_front\_2: the vehicle in front of right\_front.
- right\_back: the vehicle on the right side of the ego vehicle and not surpassing it.
- right\_back\_2: the vehicle behind right\_back.

**Grammar Supporting Tokens** These tokens are included to improve the syntactical structure and increase the readability of the program, including the and token to connect the conditions of different vehicles, the PROPERTY token to start the description of conditions for a vehicle, two parenthesis tokens p( and p) to bound all conditions for a vehicle, the <s> token to mark the beginning of each program and the \_PAD token to pad shorter programs in the end when programs are trained in batches.

## 4.2 Program Grammar

The goal of our program is to specify the conditions on all features on all adjacent vehicles, such that some defined driving behavior is only safe when all the conditions are satisfied. Therefore, we define our program as the union of conditions on each vehicle, where they are connected by the and token. Each set of conditions on a vehicle starts by declaring the PROPERTY token followed by the vehicle-describing token, and then we list all pairs of feature-describing tokens as condition boundaries for that vehicle inside the p( and p) token. Here is an example of our program<sup>1</sup> with conditions applied to the first 4 features:

```
[‘PROPERTY’, ‘front’, ‘p(‘, ‘DPX-0.6’, ‘DPX3.6’, ‘DPY30.0’,
‘DPY+’, ‘DS-17.5’, ‘DS+’, ‘DA-4.0’, ‘DA+’, ‘p)’, ‘and’,
‘PROPERTY’, ‘back’, ‘p(‘, ‘DPX-5.4’, ‘DPX0.6’, ‘DPY-‘,
‘DPY0.0’, ‘DS-‘, ‘DS10.0’, ‘DA-‘, ‘DA10.0’, ‘p)’, ‘and’,
‘PROPERTY’, ‘left_front’, ‘p(‘, ‘DPX-‘, ‘DPX-0.6’, ‘DPY-90.0’,
‘DPY60.0’, ‘DS-‘, ‘DS15.0’, ‘DA-‘, ‘DA2.0’, ‘p)’, ‘and’,
```

<sup>1</sup>The preceding <s> token and the following \_PAD tokens are not included in the example.



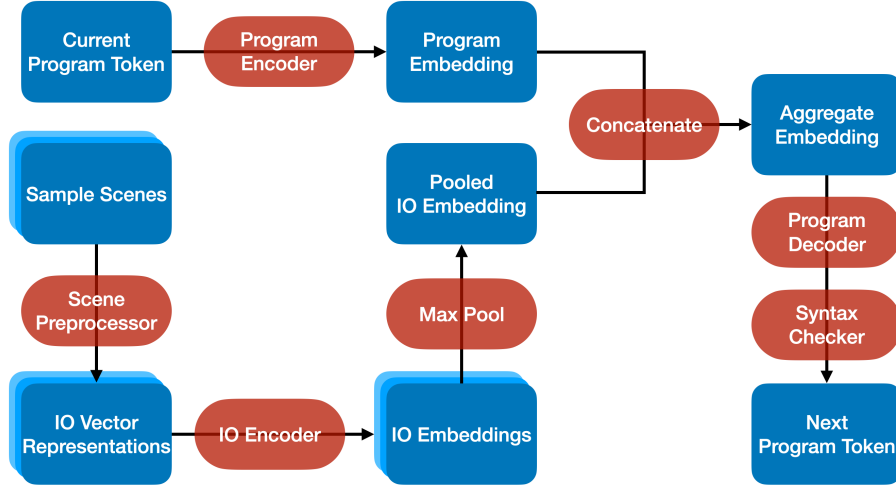


Figure 2: Visualization of our program generation pipeline.

```

‘PROPERTY’, ‘left_front_2’, ‘p(‘, ‘DPX-‘, ‘DPX0.6’, ‘DPY0.0’,
‘DPY+’, ‘DS-5.0’, ‘DS+’, ‘DA-7.0’, ‘DA+’, ‘p)’, ‘and’,
‘PROPERTY’, ‘left_back’, ‘p(‘, ‘DPX-‘, ‘DPX1.2’, ‘DPY-45.0’,
‘DPY45.0’, ‘DS-12.5’, ‘DS15.0’, ‘DA-6.0’, ‘DA6.0’, ‘p)’,
‘and’, ‘PROPERTY’, ‘left_back_2’, ‘p(‘, ‘DPX-‘, ‘DPX2.4’,
‘DPY-‘, ‘DPY15.0’, ‘DS-‘, ‘DS12.5’, ‘DA-‘, ‘DA7.0’, ‘p)’,
‘and’, ‘PROPERTY’, ‘right_front’, ‘p(‘, ‘DPX-2.4’, ‘DPX+’,
‘DPY-120.0’, ‘DPY150.0’, ‘DS0.0’, ‘DS22.5’, ‘DA-6.0’, ‘DA+’,
‘p)’, ‘and’, ‘PROPERTY’, ‘right_front_2’, ‘p(‘, ‘DPX0.0’,
‘DPX+’, ‘DPY0.0’, ‘DPY+’, ‘DS-12.5’, ‘DS+’, ‘DA-8.0’, ‘DA+’,
‘p)’, ‘and’, ‘PROPERTY’, ‘right_back’, ‘p(‘, ‘DPX0.0’, ‘DPX+’,
‘DPY-105.0’, ‘DPY90.0’, ‘DS-12.5’, ‘DS10.0’, ‘DA-3.0’,
‘DA5.0’, ‘p)’, ‘and’, ‘PROPERTY’, ‘right_back_2’, ‘p(‘,
‘DPX0.6’, ‘DPX+’, ‘DPY-‘, ‘DPY30.0’, ‘DS-‘, ‘DS10.0’, ‘DA-‘,
‘DA6.0’, ‘p)’.

```

## 5 Program Generation

The goal of our model is to predict program tokens representing the conditions for some driving behavior given sample scenes with both satisfying and unsatisfying cases. In our pipeline, we generate tokens one by one so that every token can directly depend on its previous token and a representation of the sample scenes. For the generation of each program token, we designed a pipeline with three major steps, as described below.

### 5.1 Extracting Vector Representations from Scenes

Since each program is paired with hundreds or even thousands of sample scenes, it would be infeasible to feed in image representations of the scenes into our model. Thus we design a mechanism to simplify the information in each scene into a simple vector representation using the scene preprocessor.

**Scene Preprocessor** The scene preprocessor takes in a scene from the simulator and converts it into a simple vector representation of the scene with size  $N_{io} = 10N_{feat} + 1$  where  $N_{feat}$  is the number of features we would like to extract from the scene. For each feature, we extract the feature values from the 10 vehicles around the ego-vehicle, and thus we have  $10N_{feat}$  features values in total. Sometimes, there is no vehicle in some direction, we then will make up feature values to mimic a vehicle that is far enough to affect our driving decision. We standardize the feature values so that most

values will lie between -1 and 1. We also include a binary value to indicate the output of the desired program if we treat the scene as an input - whether this scene leads to a satisfactory or unsatisfactory decision for the driving behavior we are learning.

## 5.2 Encoding Scenes Vectors and the Current Program Token

In this step, we prepare the scene vectors and the current program token to generate a single representation of the current state. The scene vectors first go through the IO encoder and become 256-dimensional embeddings. We then perform a max pool on all embeddings to generate a single 256-dimensional IO embedding to represent all scenes. The current program token goes through the program encoder to become a 64-dimensional embedding. We then concatenate the pooled IO embedding and the program embedding to create a 320-dimensional embedding that holds information on both the sample scenes and the current program token.

**IO Encoder** The IO encoder is a model that learns the embedding of a scene from its vector representation. Here we construct the IO encoder as three linear layers with ReLU, where the first layer has an input dimension of  $N_{io}$  and an output dimension of 256, while the second and the third layer have both input dimensions and output dimensions of 256.

**Program Encoder** The program encoder is a `nn.Embedding` module from the `pytorch` library that acts like a look-up table from program tokens to their embeddings. Here we choose the embedding dimension to be 64 which suffices to represent the  $N_{prog} = 23N_{feat} + 10 + 6$  different program tokens in our DSL.

## 5.3 Predicting the Next Program Token

The last step in our pipeline is to use the aggregate embedding to predict the most probable next program token. The aggregate embedding is passed into the program decoder, which produces the probability distribution of the next program token on all  $N_{prog}$  tokens. Since some program tokens may not fit the grammar of our program, we also introduce a syntax checker to mask off incompatible tokens so that only coherent tokens can be chosen as the next program token. Finally, we choose the program token with the highest probability after masking as the next program token.

**Program Decoder** The program decoder generates the probability distribution of the next token from the aggregate embedding. Since the aggregate embedding only includes information about the current program token which is not sufficient for predicting the next token, we designed the main structure of the program decoder as an LSTM RNN so that the previous program tokens will also implicitly affect the next token through the hidden states. Here we choose an LSTM structure with 1 recurrent layer and a hidden size of 256. The output of the LSTM is then fed into a linear layer with an output dimension of  $N_{prog}$  so that output of this linear layer represents the logits on the  $N_{prog}$  program tokens. We finally apply the `softmax` function on these outputs to normalize the probabilities.

**Syntax Checker** The syntax checker is used to enforce the grammar and proper ordering of our program. We keep a record of the syntax state that includes all the program tokens generated so far and update the state accordingly after each new program token is generated. The syntax checker then uses the current state to suggest all the tokens that may come after the current token. The syntax checker checks grammar rules like “feature-describing tokens can only appear between p( and p) tokens” as well as ordering rules like “we specify conditions on adjacent vehicles in the order described in section 4.1”. By applying the ordering rules to our program, we guide the model to focus on extracting the correct feature values at each step instead of exploring the diverse grammar structures of our program.

# 6 Experiments

To test the validity and robustness of our proposed model, we have planned to train our proposed model structure on common driving behaviors. The first driving behavior we choose is lane-changing, and we focus on lane-changing to the left side to avoid ambiguity in the program conditions.

## 6.1 Data

We first generate the ground-truth data for training.

To reach a good level of diversity in programs, we generate 8000 ground-truth programs, in which the first 6000 are used as the training set, the next 1000 are used as the validation set, and the last 1000 are used as the testing set. These programs are generated partially randomly, where we control the choices of tokens in some places while randomly choose compatible tokens in other places. We pre-select tokens in some places so that the generated programs follow the correct grammar and are compatible with human knowledge. For example, the DPX value for the `front` vehicle should have an upper bound of positive infinity, because the vehicle in the front may only affect our driving behavior when close enough, and do not have any effect when far away.

We include 1000 sample scenes for each program we generate, with 500 satisfying cases and 500 unsatisfying cases, so that the model exposed to both the sufficient conditions and necessary conditions of the target program. We generate the sample scenes with a fully random approach, where we randomize the properties of the road, the density of vehicles on the road, and the properties and the driving style of each vehicle in the scene. Thus, each scene has its unique initialization and leads to different program output from the scene. We calculate the first 4 features from our scenes, so  $N_{feat} = 4$ . We use the second frame of each generated scene as our training data and evaluate the program output based on this frame.

## 6.2 Training Details

We apply the program generation pipeline described in the earlier section. This pipeline generates programs one by one. In each step, we feed in the program token generated in the previous step and the 1000 corresponding sample scenes into our pipeline to generate the next program token. We choose a batch size of 32, the largest batch size possible to fit in our CUDA memory. Regarding the learning rate, we set the initial learning rate to be  $1e-3$ , and we set the learning rate decay to be a multiplicative factor of 0.25 applied on the learning rate after every 40 epochs. We run the training for 200 epochs.

## 6.3 Results

For each ground-truth program, our model takes in ground-truth scenes and returns its predicted program. We then evaluate the quality of these predicted programs.

Note that although we use the cross-entropy loss on the tokens during training time, our goal is not to replicate the programs themselves, but to find programs capable of producing the same outputs from the same scenes. Therefore, we designed several metrics to evaluate the different aspects of our trained model.

### 6.3.1 Token Similarity

This evaluation compares predicted programs and ground-truth programs token by token and calculates the rate of them having the same tokens at the same locations. This evaluation gives a sense of the similarity between the predicted programs and the ground-truth programs.

In our current model, the training set reaches a similarity of 1, but the validation set only reaches a similarity of 0.64. Thus our model is overfitted to the training set. We are investigating the causes and fine-tuning the model to solve this problem.

Note that token similarity is not a comprehensive indicator of the performance of our program synthesis model, because different programs may produce the same outputs from the same scenes.

### 6.3.2 Output Accuracy

This evaluation compares with outputs of the ground-truth scenes using the predicted programs and the ground-truth programs. We then calculate the true/false positive rates from these comparisons. This is more compatible with the idea behind program synthesis than basic token similarity evaluation: the goal of program synthesis is not to replicate programs, but to generate programs that produce the same effects.

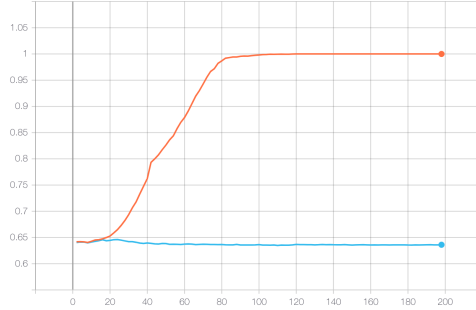
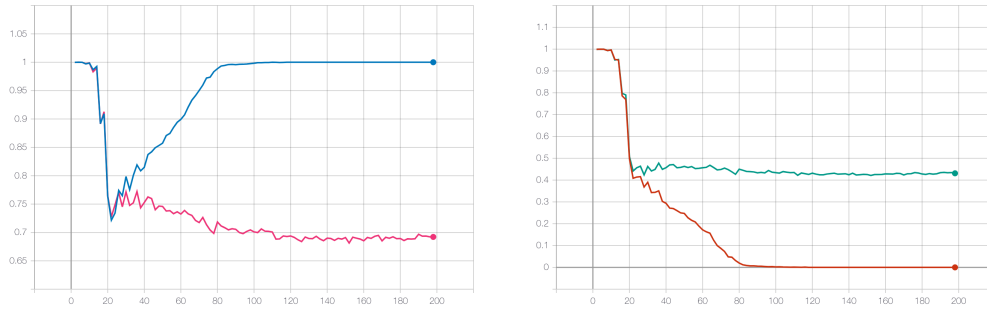


Figure 3: Token similarity on the training set (top) and the validation set (bottom). The x-axis is the number of training epochs, and the y-axis is the similar rate.



(a) True positive rate on the training set (top) and validation set (bottom). (b) False positive rate on the training set (bottom) and validation set (top).

Figure 4: Output accuracy. The x-axis is the number of training epochs, and the y-axis is the true/false positive rate.

Our current model can reach a true positive rate of 1 and a false positive rate of 0 on the training set, but it can only reach a true positive rate of 0.69 and a false positive rate of 0.42 on the validation set. This means that our model can learn the driving conditions behind our programs, but still overfitted to the training set.

## 7 Discussion and Future Works

So far, we have proposed a pipeline for our program generation problem. To complement the pipeline, we have created a driving simulator to generate ground truth scenes and a DSL to describe driving conditions. We have also tested our pipeline on a specific driving behavior - lane changing to the left. Therefore, we have set up a solid foundation for solving the problem.

However, as we see in the experiment results, there are still spaces to improve on the quality of our model, especially the accuracy of program output. We plan to fine-tune hyperparameters and adjust the model complexity so that the model is less overfitting to the training set and thus can reach a better result on the validation and testing set.

We would also like to explore more neural network structures. For example, we may switch to more effective modules for the encoders and the decoder. We are also interested in comparing the result with and without an attention mechanism on the IO vectors and IO embeddings. The attention added to scenes may help the model better focus on the information related to the generation of each token.

As we discussed above, after we reach satisfactory program accuracy on our current driving behavior, we would like to extend our scope and check the compatibility of our model with more common driving behaviors such as lane changing to the right, turning to the left and right, and lane-following. We may even extend our goal to more complex driving behaviors such as overtaking other vehicles on the road.

Once we reach good performance on several driving behaviors, we may also consider adding hierarchy to our current design. This is because there are shared components among the training processes of different driving behaviors, and we may share some parts of the model among all driving behaviors. This can also help with future training on more driving behaviors because there will be a smaller scope to train for each specific driving behavior.

## References

- [1] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. 2018.
- [2] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.
- [3] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.
- [4] Richard C Gronback. *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Pearson Education, 2009.
- [5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.