ME292B Individual Project: Planning

Zaowei Dai 3039643180

2024.4.10

Abstract

This project report details the optimization of a training model for improved prediction accuracy in autonomous driving applications. By adjusting parameters such as batch size, number of steps, and worker threads, and experimenting with network architectures like MobileNetV2 and ResNet-18, significant enhancements were achieved. The report provides a comparative analysis of the models and discusses the effectiveness of various configurations, ultimately concluding with the superior performance of an optimized ResNet-18 model.

1. Introduction

Autonomous driving technologies rely heavily on the accuracy and efficiency of their underlying prediction models. This report explores the optimization of a neural network aimed at improving prediction performance for autonomous vehicles. Through various iterations and configurations, including changes to training parameters and network structures, this study seeks to identify the most effective model settings. By focusing on GPU utilization and data handling, we aim to strike a balance between computational efficiency and model accuracy.

2.Trainer using GPU

To speed up the training, I chose GPU in the trainer settings.

And after testing the basecode and the improved code, I found that performing 7 training sessions would keep the training time from being too long while keeping the model accurate. So I set the **max_epochs = 7** in the trainer.

3. Preliminary improvement on setup of training, validation and test configurations:

3.1 Setup of training configuration

3.1.1 batch size

This parameter determines the amount of data processed per batch. Larger batch sizes improve memory utilization and training speed, but also increase memory

requirements. My GPU has enough memory, so increasing the batch size from **16 to 32.**

3.1.2 num_steps

This parameter defines the total number of steps performed during training. Increasing the number of training steps provides more training opportunities, but also increases the training time. Based on the fact that **4 hours** of training time per epoch has been spent on the base code, the current **200000** training steps are too much, and any further increase would result in excessive training time. So I decrease the training.num_steps from **200000** to **100000.**

3.1.2 num_data_workers

This parameter determines the number of worker threads used during data loading. More worker threads speed up data loading, but also increase CPU resource usage. To speed up training, increase num_data_workers from **4 to 8**.

The modified parameters are shown below：

```
## training config
self.training.batch_size = 32
self.training.num_steps = 100000
self.training.num_data_workers = 8
```

Figure 1: Improved train configuration

3.2 Setup of validation and test configuration

3.2.1 batch size and num_data_workers

To speed up the testing and validation process, grow their batch_size from **16 to 32,** and num_data_workers from **4 to 8**.

3.2.2 every_n_steps

Reducing the number of steps between each validation or test allows for more frequent monitoring of model performance, especially in the early stages of model training. When the training.num_steps decrease from 200000 to 100000, the current setting of validating every **1000** steps is enough to do the validation function.

3.2.3 num_steps_per_epoch

Increasing the number of steps processed in each validation or test can provide a more comprehensive performance estimate, especially if the validation dataset is large. This number is large enough to cover a large percentage of the validation set to obtain a more accurate performance estimate. So keep it not changed.

The modified parameters are shown below：

```
## validation config
self.validation.enabled = True
self.validation.batch_size = 32
self.validation.num_data_workers = 8
self.validation.every_n_steps = 1000
self.validation.num_steps_per_epoch = 100

## test config
self.test.enabled = True
self.test.batch_size = 32
self.test.num_data_workers = 8
self.test.every_n_steps = 1000
self.test.num_steps_per_epoch = 100
```

Figure 2: Improved validation and test configuration

4. Further improvement I on network structure

4.1 model_architecture

4.1.1 mobilenet_v2

Advantage: Designed for mobile and embedded vision applications, the lightweight network is realized through an inverted residual structure. Emphasis is placed on maximizing performance with limited computational resources.

Disadvantage: However, it is a lightweight model for scenarios that require fast execution, which has large shortcomings in model accuracy.

4.1.2 resnet18

Advantage:

Degradation Problem Resistance: In deep networks, more layers theoretically should improve performance. However, excessively deep networks often face performance degradation, where accuracy deteriorates as more layers are added. ResNet addresses this issue through the introduction of residual blocks, enabling effective training even with deep architectures. ResNet-18 maintains good performance through this design.

Strong Generalization: Despite having fewer layers, ResNet-18 demonstrates strong generalization capabilities. It handles various image recognition tasks and achieves commendable results on multiple benchmark datasets, making it a versatile and effective model.

Comparison: (1) Compared to **MobileNetV2**, **Resnet18** is more suitable for application scenarios where higher accuracy is required. (2) By introducing residual learning to address the difficulty of training deep networks, Resnet18 strikes a good balance between model complexity, computational efficiency, and performance.

4.2 history_num_frames

In autonomous driving scenarios, longer history information helps models to better understand the behavioral patterns of traffic participants and improve the accuracy of predicting future states. Especially in complex traffic scenarios, **long time history information** can help models capture **more subtle dynamic changes and interactions.**
It is also important to keep the growth of these parameters **consistent** in order to ensure that the model uses the same length of time window for all entities when considering historical information, thus **simplifying the time series analysis.**

So I increase all the history_num_frames、history_num_frames_ego and history_num_frames_agents from **9 to 12**.

4.3 future_num_frames

This parameter defines how far into the future the model needs to predict the trajectory. Longer prediction ranges can help the autopilot system plan and make decisions earlier, but long-term predictions are usually less accurate.

The current **30**-frame future projection is too long or not accurate enough for practical applications. Because short-term predictions are more important, reduce the future_num_frames to **25** frames.

4.4 decoder.layer_dim

The initial decoder is configured as (), which means that the decoder is simple or may not have additional layers. Considering that the decoder is the part responsible for generating the final prediction from the features, increasing the complexity of the decoder may help improve performance.

I set the decoder.layer_dims = (512, 256, 128), the advantages are as below:

(1) **Improving prediction accuracy.** Deeper network structures are often able to capture higher levels of abstract information, which is particularly important for complex tasks. In the prediction task of autonomous driving, it is crucial to be able to accurately understand the environment and predict future states. Adding layers and tuning dimensions can help models perform better in this regard, thus improving overall prediction accuracy.
(2) Balancing overfitting and underfitting. A balance between the expressive power of the model and the risk of overfitting can be found through progressively smaller layer dimensions. Larger dimensions provide enough model capacity to learn complex features in the data, while progressively smaller dimensions help control model complexity and reduce the risk of overfitting.

The modified parameters are shown below：

```
self.model_architecture = "resnext50_32x4d" #mobilenet_v2 resnet18
self.map_feature_dim = 256
self.history_num_frames = 12
self.history_num_frames_ego = 12
self.history_num_frames_agents = 12
self.future_num_frames = 25
self.step_time = 0.1
self.render_ego_history = False
self.decoder.layer_dims = (512, 256, 128) # you can make the decoder more complex
self.decoder.state_as_input = True
```

Figure 3: Improved network structure

But during the train process, I found that the model is too much complex to train, which cause the train time too long. So in the network structure part, I reverted the parameters into origin version, which is the same as the base_code's.

5. Final improvement on Dataloader parameters

I enabled data shuffle, the advantages are as below:

(1) **Improved generalization ability.** By randomly disrupting the order of the training data at the beginning of each epoch, the model sees a different combination of data for each batch during training. This helps prevent the model from becoming overly dependent on a particular order or distribution of data, thus improving the model's ability to generalize to new, unseen data.
(2) **Preventing overfitting.** Fixed data order can lead to models learning sequential features of the data rather than the true features of the data itself. Shuffling reduces this risk and allows the model to focus more on learning common, generalizable patterns from the data.
(3) **Optimizing convergence speed.** Disrupting the data order can help optimize the model learning process and increase the convergence speed by avoiding bias of the model towards some specific data batches during the training process.

```python
def train_dataloader(self, return_dict = True):
    # Exploration:
    # You can choose the dataloader parameters. For instance, whether to shuffle.

    return DataLoader(
        dataset=self.train_dataset,
        shuffle=True,
        batch_size=self._train_config.training.batch_size,
        num_workers=self._train_config.training.num_data_workers,
        drop_last=True,
        collate_fn=self.train_dataset.get_collate_fn(return_dict=return_dict),
        persistent_workers=True if self._train_config.training.num_data_workers>0 else False
```

Figure 4: Improved train_dataloader

6. Final test result

After training, I have validated the model with validation_data_set, checkpoints result is as below:

Figure 5: Validation result of improved Resnet18 model

From the validation, we can see that the losses_goal_loss is lowered into 0.05112, the losses_prediction_loss is lowered into 0.37298. And the **ego_ADE** decreased into **0.60171,** the ego_FDE decreased into 1.49401.

Then, I test the improved model with test_data_set, the test result is as below:

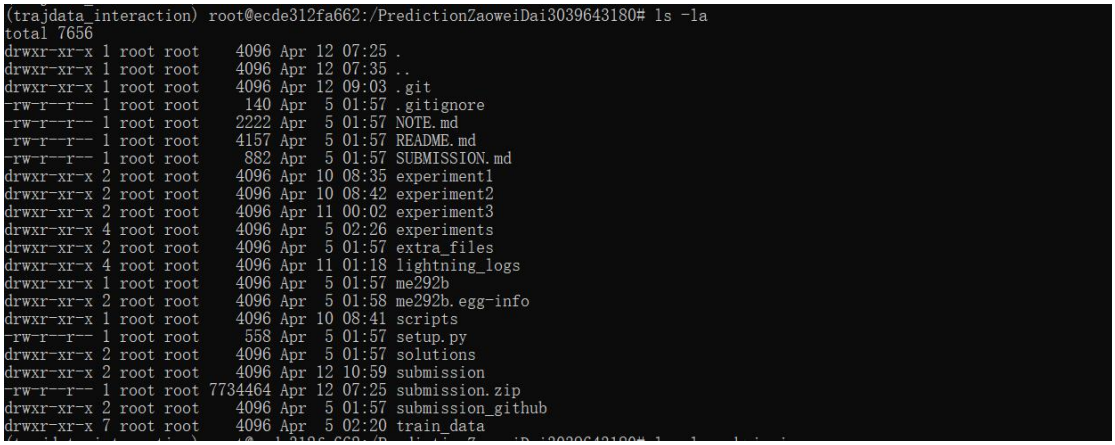

Figure 6: Test result of improved Resnet18 model

Figure 7: minADE and MR of improved Resnest18 model

From the result, it can be seen that final **all_minADE is 0.5438 < 0.73, and the final all_MR is 0.3538 < 0.45, which has passed the test.**

7. Conclusion

The project successfully demonstrated that careful optimization of network parameters and architectures can lead to substantial improvements in model performance for autonomous driving applications. The optimized ResNet-18 model outperformed the baseline and other configurations, showing lower prediction losses and enhanced accuracy in real-world testing scenarios. These findings highlight the importance of parameter tuning and architecture selection in developing robust prediction models for complex applications such as autonomous driving.