Flexi-Compression: A Flexible Model
Compression Method for Autonomous Driving

by

Hantao Liu

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Information Technology: Network Technology

School of Information Technology
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
August, 2021

©Copyright 2021, Hantao Liu
Abstract

Benefiting from the rapid development of convolutional neural networks, computer vision-based autonomous driving technologies are gradually being deployed in vehicles. However, these neural networks typically have a large number of parameters and extremely high computational cost, making them difficult to deploy in autonomous vehicles with limited storage and computational power. In this thesis, we propose an innovative model compression approach to compress convolutional neural networks in autonomous driving algorithms, which we call Flexi-Compression. Flexi-Compression first modifies the model structure by replacing the traditional convolutional layers with our proposed Flexi-CP (flexible compression) module, thus reducing the computation of the convolutional layers. Then, we leverage knowledge distillation to enable the compressed model to quickly acquire the knowledge of the original model. In addition, we use a Flexi-Batch Normalization layer to prune the model and finally further reduce the model size by model quantization. We compress the VGG-16 network (Visual Geometry Group Network) using our proposed model compression algorithm, which is a commonly used backbone network in autonomous driving algorithms. On the CIFAR-10 dataset, our compression method can reduce the parameters of the VGG-16 network by 86% and the computation by 87% with 4% loss of accuracy. To verify the effectiveness of our compression algorithm in real-world applications, we also compress an autonomous driving algorithm and achieve excellent performance.
Acknowledgments

First of all, I would like to express my sincerest gratitude to my advisor, Professor F. Richard Yu, for his dedication, patience, encouragement, and support. Without his guidance, knowledge and expertise, this thesis would not have been possible. I would also like to thank my family for their long term support and encouragement. Finally, I would like to thank my colleagues for their help when I encountered difficulties.
# Table of Contents

Abstract ii

Acknowledgments iii

Table of Contents iv

List of Tables vii

List of Figures viii

List of Abbreviations x

1 Introduction 1

1.1 Research Overview ........................................ 1

1.2 Thesis Contributions ....................................... 3

1.2.1 Accepted Paper ......................................... 4

1.3 Thesis Organization ....................................... 5

2 Autonomous Driving Approaches 7

2.1 Modular Approach ........................................ 7

2.2 End-to-end Approach ...................................... 8

3 Convolutional Neural Networks 11
3.1 Features of CNN ........................................... 12
3.2 Types of Basic Layers of CNN .......................... 13
  3.2.1 Convolutional Layer ................................. 13
  3.2.2 Pooling Layer ......................................... 14
  3.2.3 Activation Layer ...................................... 15
  3.2.4 Fully Connected Layer ............................... 18
  3.2.5 Dropout Layer ......................................... 19
  3.2.6 Batch Normalization Layer ......................... 20
3.3 The Complexity of CNN ................................. 22

4 Network Compression Methods .......................... 24
  4.1 Pruning .................................................. 24
  4.2 Quantization ............................................ 25
  4.3 Knowledge Distillation ................................. 27
  4.4 Low-rank Factorization ................................. 28

5 Proposed Compression Method .......................... 29
  5.1 Flexible Compression ................................. 29
  5.2 Flexi-CP Module ....................................... 29
  5.3 Knowledge Distillation ................................. 33
  5.4 Network Pruning ....................................... 37
  5.5 Quantization ........................................... 39

6 Simulation Results and Discussions .................... 42
  6.1 Experiments on Backbone Networks ................... 42
    6.1.1 Dataset Description ............................... 42
    6.1.2 Evaluation Metric ................................. 43
    6.1.3 Parameter Setup ................................. 44
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Food categories of FOOD-11</td>
<td>43</td>
</tr>
<tr>
<td>6.2</td>
<td>Compression results of VGG-16 network on FOOD-11 after using Flexi-CP module</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>Pruning results with different pruning rates on the FOOD-11 dataset</td>
<td>47</td>
</tr>
<tr>
<td>6.4</td>
<td>Quantization results on the FOOD-11 dataset</td>
<td>48</td>
</tr>
<tr>
<td>6.5</td>
<td>Compression results of VGG-16 network on CIFAR-10 after using Flexi-CP module</td>
<td>49</td>
</tr>
<tr>
<td>6.6</td>
<td>Pruning results on the CIFAR-10 dataset</td>
<td>49</td>
</tr>
<tr>
<td>6.7</td>
<td>Quantization results on the CIFAR-10 dataset</td>
<td>49</td>
</tr>
<tr>
<td>6.8</td>
<td>Compression results of different pruning rate of the autonomous driving model</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Modular approach .................................................. 8
2.2 End-to-end approach [1] ........................................... 9
3.1 The convolutional computation process of input data and convolutional
kernel, figure (a) is the input data, figure (b) is the convolutional kernel,
figure (c) is the result of convolutional computation .................. 14
3.2 Pooling operations .................................................. 15
3.3 ReLU ............................................................... 16
3.4 LeakyReLU ......................................................... 17
3.5 Tanh ............................................................... 17
3.6 Sigmoid ............................................................. 18
5.1 The four step compression process: architecture design, knowledge dist-
tillation, pruning and quantization. .............................. 30
5.2 Architecture diagram of Flexi-CP module ..................... 31
5.3 Teacher-student network knowledge distillation process ........ 35
6.1 Example of Cifar10 dataset ...................................... 44
6.2 Steering angle distribution using keyboard(a) and mouse(b) ....... 52
6.3 Nvidia model architecture ....................................... 54
6.4 VGG based model architecture ................................. 55
6.5 Training results of NvidiaModel and VGG-based-model ........ 57
6.6 Comparison curve between predicted and true values  . . . . . . . . 58
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNN</td>
<td>Deep Neural Networks</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Networks</td>
</tr>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>FLOPs</td>
<td>Floating-point Operations Per second</td>
</tr>
<tr>
<td>LiDAR</td>
<td>Light Detection and Ranging</td>
</tr>
<tr>
<td>ALVINN</td>
<td>Autonomous Land Vehicle In A Neural Network</td>
</tr>
<tr>
<td>BN</td>
<td>Batch Normalization</td>
</tr>
<tr>
<td>VQ</td>
<td>Vector Quantization</td>
</tr>
<tr>
<td>PQ</td>
<td>Product Quantization</td>
</tr>
<tr>
<td>KD</td>
<td>Knowledge Distillation</td>
</tr>
</tbody>
</table>
SVD  Singular Value Decomposition
GAP  Global Average Pooling
MSE  Mean Squared Error
STE  Straight Through Estimator
Adam Adaptive moment estimation
FBN  Flexi-Batch Normalization
Chapter 1

Introduction

1.1 Research Overview

In recent years, deep neural networks have made breakthroughs and become a popular research topic in the fields of artificial intelligence. As one of the most widely used type of deep neural networks, convolutional neural networks have achieved excellent performance in many types of tasks such as computer vision and natural language processing. In 2012, AlexNet [3] won the ImageNet competition, which was a turning point in the history of computer vision. After that, more complex and accurate deep neural networks were proposed, such as VGGNet, GoogleNet [4], ResNet [5], etc. In this process, convolutional neural networks have become deeper and deeper, with AlexNet having only 8 layers, VGGNet having 16 layers, and ResNet having 200 layers. As these networks became deeper, they became more accurate, and they even outperformed humans on certain tasks. For example, ResNet achieved 3.57% error rate in the 2015 ImageNet Competition, surpassing human performance on this dataset.

The outstanding performance of convolutional neural networks in the field of computer vision has led to the widespread interest in autonomous driving algorithms
based on deep learning and computer vision. Autonomous driving refers to vehicles making driving decisions in real time by observing and sensing their surroundings without human intervention. Autonomous vehicles usually use sensor devices including cameras, radar, ultrasound, etc., to perceive the environment. In this thesis, we focus on autonomous driving algorithms based on computer vision and deep learning, which mainly uses cameras to acquire environmental information and make driving decisions through deep learning algorithms. Compared with radar and ultrasound sensors, using cameras to acquire environmental information has many advantages, such as low cost and comprehensive information. Therefore, computer-vision based autonomous driving technology has become the mainstream approach in the field of autonomous driving. Tesla’s Autopilot and Baidu’s Apollo Lite are all vision-based autonomous driving technologies.

Autonomous driving algorithms are highly dependent on computer vision technology at the algorithm level. Although deep learning has made great progress in the field of computer vision in recent years, there are still many problems to be solved to achieve autonomous driving under complex road conditions. One of the main problems is that the backbone network used by autonomous driving algorithms usually has a large number of parameters and require extremely high computational power. AlexNet has over 62 million parameters, while VGGNet has 138 million parameters. Such a large number of parameters requires enormous computational power. Efficient computation of networks is possible in the laboratory with a large number of GPUs, but cost, power consumption and other issues make such solutions infeasible for vehicles. For one thing, a backbone network like VGG-16 requires 526MB of storage space to keep its parameters, which is a considerable challenge for resource-limited autonomous vehicles. Another issue is the inference time of the network. As mentioned earlier, vehicles need real-time feedback on their surroundings during driving,
which requires the model to be capable to perform real-time inference. This is not difficult for laboratories with high computational power GPUs, but for vehicles with low computational power, performing real-time inference is very difficult.

Therefore, how to reduce the size of the neural network and increase its inference speed while guaranteeing its performance becomes a major challenge for deploying autonomous driving algorithm on vehicles. The authors of [6] show that there is significant redundancy in many deep neural networks. Using only a small fraction (5\%) of the total weights is sufficient to predict the remaining weights. It is also suggested in [6] that these remaining weights are not even needed to be learned. That is, only a small fraction of the total weights need to be trained to achieve performance similar to or even better than the original network. Therefore, these networks are highly compressible and if we can find the parameters that are really effective for model inference and discard those that are not, we can greatly reduce the redundancy of the network and achieve the compression and acceleration of the network.

1.2 Thesis Contributions

In this thesis, we investigate compression and acceleration methods for backbone networks of autonomous driving algorithms. We propose a compression process for backbone network. After compressing mainstream backbone networks using this process, we significantly reduce the memory requirements of the network and increase the inference speed with guaranteed accuracy, making the network usable on end devices such as vehicles. Our contributions are summarized as follows:

1. We propose the Flexi-CP (flexible compression) module. The convolutional layer is one of the most important layers in the backbone network. We effectively reduce the parameters as well as the computation of the network by replacing
CHAPTER 1. INTRODUCTION

the convolutional layer in the network with the Flexi-CP module. In our experiments, we use Flexi-CP module for commonly used backbone networks, and obtain good results. For example, we reduce 88% of the computation and 87% of the number of parameters in the VGG network without any significant loss of accuracy.

2. We propose a Flexi-Compression a four step compression process based on Flexi-CP module. Flexi-CP module contains a flexi-batch normalization layer, in which we use the scale factor to prune the filter to achieve network pruning, and after pruning we also quantize the network to further reduce the computational effort and number of parameters.

3. In order to verify that the method proposed in this thesis can also achieve the expected results in practical applications, we compress an autonomous driving algorithm using our proposed compression algorithm. We perform data acquisition through a simulator and trained the original model with the acquired data. The original model is then compressed using Flexi-compression, which reduces the amount of parameters by 95% and the amount of computation by 88% with no significant difference in performance. We deploy the compressed model on a simulated car and achieve good driving performance.

1.2.1 Accepted Paper

The following paper has been accepted:
1.3 Thesis Organization

The rest of this thesis proceeds as follows:

- **Chapter 2**: In this chapter, entitled “Autonomous Driving Approaches”, we begin by describing two mainstream autonomous driving technologies, modular approach and end-to-end approach. Then, we discuss the specifics of both technologies and analyze their advantages and disadvantages.

- **Chapter 3**: In this chapter, entitled “Convolutional Neural Networks”, we first introduce the characteristics of convolutional neural networks, then analyze the functions and characteristics of common layers in convolutional neural networks, and finally introduce the complexity calculation method of convolutional neural networks.

- **Chapter 4**: In this chapter, entitled “Network Compression Methods”, we introduce four commonly used neural network compression methods, namely pruning, quantization, knowledge distillation, and low-rank decomposition.

- **Chapter 5**: In this chapter, entitled “Proposed Compression Method”, we present in detail our proposed compression process: Flexi Compression. First we introduce the structure of the Flexi-CP module and then we present in detail the other parts of the compression process.

- **Chapter 6**: In this chapter, entitled “Simulation Results and Discussions”, we first compress the backbone network with our compression process and analyzed the results, and then we performed simulation experiments in order to verify that our proposed compression process is also effective in practical applications. The simulation results illustrate that our proposed compression flow can significantly reduce the computational and parametric quantities of the network with
no significant performance loss.

- Chapter 7: This chapter summarizes the thesis and suggests possible directions for improvement.
Chapter 2

Autonomous Driving Approaches

Currently, autonomous driving technologies can be broadly classified into two types, namely modular and end-to-end approaches.

2.1 Modular Approach

The modular approach includes key technologies in four areas: perception, planning, decision making, and control. Perception technology refers to the process of collecting information from the environment and extracting relevant knowledge from the vehicle. Perception system needs to process various environmental information [7], including road detection, vehicle detection, pedestrian detection, traffic light detection, etc.

There are two mainstream methods to acquire environmental information, one is to use high-definition cameras to acquire environmental information, and then process it by computer vision algorithms [8]. The other is to use LiDAR to acquire environmental information and then analyze and process the acquired point cloud data [9].

Planning technology [10] refers to the vehicle planning the path in real time based on the road condition information acquired by the perception technology in real time. Decision-making technology [11] means that the vehicle integrates perception and
planning information through algorithms to obtain specific driving decisions. Control technology [12] enables vehicles to execute precise decisions through controllers for timely acceleration, braking and steering. Modular-based autonomous driving systems require manual decoupling of the driving tasks of unmanned vehicles to design multiple sub-modules and require powerful computational power to ensure that each module can respond to changes in the environment in real time. This system implements motion planning and control of unmanned vehicles through regularized logic, leading to a weakened anthropomorphic driving style that affects ride comfort.

2.2 End-to-end Approach

In contrast, end-to-end models are favored by researchers because of their simple structure, ease of use, low cost and anthropomorphism [13]. Benefiting from the rapid
CHAPTER 2. AUTONOMOUS DRIVING APPROACHES

Figure 2.2: End-to-end approach [1]

development of deep learning technologies, end-to-end models are able to derive vehicle control signals directly from vehicle state and external environment information without any human intervention. The earliest end-to-end model is the ALVINN model proposed by Pomerleau, which uses a traditional neural network to obtain a mapping function between image data and steering wheel rotation by collecting environmental images and corresponding steering wheel rotation data during driving. Limited by the conventional neural networks and computational power available at that time, ALVINN was unable to cope with complex traffic environments and could only drive autonomously in the campus. In 2016, NVIDIA trained a deep neural network [1] that could obtain steering commands from a single camera image, and with less than 100 hours of human driving data, the network learned to drive in areas with no lane lines and was able to drive in a variety of weather conditions, including sunny, cloudy, and rainy days. Since then, end-to-end autonomous driving technology has gained the
attention of researchers and has made rapid progress. Both modular and end-to-end approaches require the use of computer vision methods to process environmental information. With the rapid development of deep learning, the use of convolutional neural networks to process environmental information has become the mainstream approach in autonomous driving. However, large neural networks require not only sufficient memory space but also powerful computational power, which is one of the biggest obstacles to deploying autonomous driving systems [14]. In order to reduce the size of these neural networks and speed up their training and inference, many model compression algorithms have been proposed. The compressed autonomous driving systems require less space and computational power and can be deployed in resource-limited autonomous vehicles.
Chapter 3

Convolutional Neural Networks

Convolutional neural network is a very effective method in deep learning theory, the main applications are: target classification, target detection, target segmentation, etc. which is a big data-driven deep learning algorithm. Convolutional neural networks perform well in image recognition, so many computer vision-based autonomous driving technologies use convolutional neural networks to obtain visual features. However, computer vision-based autonomous driving techniques require a convolutional neural network model with a large number of parameters to support, and too large a number of parameters will cause difficulties in engineering applications, so we will conduct research on methods for model compression and acceleration.

In this chapter, we first introduce the basic theory of convolutional neural networks, and give a detailed introduction to some basic layers of a complete convolutional neural network. Then, some common structures in convolutional neural networks are introduced and analyzed.

Finally, we introduce the metrics involved in the model performance such as the number of parameters and computation involved in the model compression and acceleration methods. This chapter provides the theoretical basis for the subsequent research on model compression and acceleration methods.
CHAPTER 3. CONVOLUTIONAL NEURAL NETWORKS

3.1 Features of CNN

The main features of convolutional neural networks are summarized as follows:

- Convolutional neural network is a computationally intensive neural network model [15], and as the depth of the model deepens, the amount of computation and model parameters will keep increasing.

- Compared with traditional machine learning algorithms, convolutional neural networks are able to extract features automatically, thus solving the problem of expression limitations of manually extracted features in traditional computer vision algorithms. With the deepening of the network model, the features extracted by the neural network are more richly expressed, so the convolutional neural network is very suitable for dealing with large-scale image application problems.

- The connection between the convolutional layers in a convolutional neural network is a local connection to mine the spatial local association information in the image, which is reflected in the weight sharing [16] and the receptive field range [17] of the local area in the model. Each neuron only responds strongly to the area within the current receptive field, and does not respond to changes outside the receptive field. By this sparse connection, the parameters of the model are greatly reduced.

- Convolutional neural networks are more suitable for processing large-scale natural image applications than traditional machine learning algorithms because they have better feature extraction capabilities. However, the interpretability of convolutional neural networks is poor, and it is not possible to prove that a model is superior through theoretical derivation, and it is more likely to compare
the advantages and disadvantages of different models through a large number of experiments.

3.2 Types of Basic Layers of CNN

3.2.1 Convolutional Layer

The convolutional layer is one of the most important layers in convolutional neural networks, and its role is to extract features from the input data by convolutional operations. In traditional image algorithms, methods exist to extract image features using convolutional kernels, but these convolutional kernels generally need to be designed manually and can only extract specific features. In contrast, the convolutional layer in convolutional neural networks adaptively adjusts the weights of the convolutional kernels by backpropagating the gradient of the objective function. This allows the convolutional layer to extract valuable features based on the objective function. Next, we briefly describe the convolutional operation, as shown in Figure 3.1. Suppose the input data is the matrix shown in Figure(a), the convolution kernel is Figure(b), the blue part is the area where the convolution kernel performs the first convolution operation, and the blue part of Figure(c) is the output of this convolution.

Benefiting from the convolutional computation, convolutional neural networks have the features of local perceptual field and weight sharing. Compared with multilayer perceptrons, convolutional neural networks have a much smaller number of parameters and are more capable of extracting features.

To extract meaningful features from the input image, convolutional neural networks usually require a large number of convolutional kernels. The number of convolutional kernels is usually a hyperparameter set manually according to experimental experience, so the number of convolutional kernels is usually redundant. Therefore,
we can reduce the number of parameters and computation of the network by filtering out the convolutional kernels of lower value.

### 3.2.2 Pooling Layer

Pooling layer [18], including pooling methods such as average pooling layer, maximum pooling layer, etc., are mainly used in convolutional neural networks to reduce the number of network parameters and computational effort when the input image is relatively large, and the intermediate feature dimensions can be reduced by pooling operations so that the purpose of reducing parameters and computational effort can be achieved. In the target recognition task, the pooling operation can be well adapted to the current recognition task, but in other applications, such as segmentation tasks, the use of pooling layers should be reduced as much as possible because of the resolution
Figure 3.2: Pooling operations

degradation and translation invariance caused by pooling.

As shown in Figure 3.2, the left side is a feature map matrix, and the right side is the output after using the maximum pooling kernel. By pooling operation, we reduce the input feature map by half, which reduces the number of parameters and the computation of the neural network.

3.2.3 Activation Layer

In convolutional neural networks, the activation layer, as the nonlinear layer of the network, is usually placed directly after the convolutional layer or batch normalization layer. Usually, a nonlinear activation function is used as the activation layer function. Using a linear activation function results in a model with insufficient expressiveness and may result in gradient vanishing. The nonlinear activation function is used to increase the nonlinearity of the model by using its nonlinear operation and to prevent the gradient from vanishing. The commonly used nonlinear activation functions are
shown in Figures 3.3, 3.4, 3.5 and 3.6.

The Sigmoid function can guarantee nonlinearity, but it has a saturation region, which causes the gradient of neurons to converge to 0 if the current input is very large or small, thus causing the gradient to vanish and failing to train the network properly.

The Tanh function has a mean value of 0, but it also has a saturation region, which also causes the gradient of neurons to converge to 0 causing the gradient to disappear, making the network difficult to train.
CHAPTER 3. CONVOLUTIONAL NEURAL NETWORKS

Figure 3.4: LeakyReLU

Figure 3.5: Tanh
Compared with the first two functions, the ReLU function is more commonly used, mainly because the derivative of the ReLU function is 1, which is easy to calculate. Secondly, there is no saturation region and no gradient vanishing problem, but the left side of the ReLU function is all 0, which will make some neurons lose their connections, which is its shortcoming.

To address the shortcomings of ReLU, some improved ReLU functions have been proposed [19] [20]. LeakyReLU changes the left side of ReLU to a linear function. This ensures that the gradient vanishing problem does not occur and also does not cause the neurons to lose connectivity.

3.2.4 Fully Connected Layer

In convolutional neural networks, features are extracted by convolutional kernels, and after dimensionality reduction by pooling, they need to be output as classes, but both convolution and pooling operate on images with spatial information, so a fully-connected layer needs to be connected behind the network to extend the original images with spatial information through the fully-connected layer and finally output the corresponding classes. In general, the fully-connected layer accounts for a large
number of parameters in the overall model, so many new networks choose to replace the fully-connected layer using other methods to reduce the number of parameters in the network.

3.2.5 Dropout Layer

The Dropout [21] layer drops some connections randomly with a certain probability with the aim of preventing the model from overfitting [22]. It is equivalent to a regularization operation, which can improve the generalization ability of the model. In each training cycle, although the Dropout probability is the same, but the discarded connections are different, so the model will be different in each training cycle. As shown in Figure 3.7, a is the original neural network, and Fig. b is the network after dropout. From the figure, we can see that the new network becomes sparse compared
with the original network, and each dropout will produce a different network, which makes the generalization ability of our trained neural network stronger and less prone to overfitting.

### 3.2.6 Batch Normalization Layer

The batch normalization layer [23] is widely used in various deep learning models to accelerate the training process of the model, which can help the model converge to better results. Since our proposed compression method involves the parameters of the batch normalization layer, in this subsection, we will introduce the function of the batch normalization layer and the meaning of the parameters in detail.

Normalization is a common method used in the training process of neural networks. Normalization has many benefits, such as unifying the magnitudes of different features, eliminating the adverse effects from abnormal sample data, and accelerating model convergence, etc. For shallow models, normalizing the data during data preprocessing ensures that the output will not change drastically when the model parameters are updated in small increments. However, for deep models with more layers, even if the input data is normalized, small parameter updates propagated through the layers can still cause large fluctuations in the final output. Such fluctuation makes the training of neural networks very difficult, and the proposal of batch normalization solves this problem perfectly. During the model training process, the batch normalization layer continuously calculates the mean and standard deviation of the intermediate layers outputs at the batch scale and uses them to normalize the intermediate layer outputs, so that the fluctuations in the outputs are dissolved and the outputs show a relatively stable distribution, which accelerates the convergence of the network. In convolutional neural networks, the batch normalization layer is usually placed after the convolutional layer and before the activation function. If the
convolutional layer has multiple output channels, it is necessary to perform batch normalization for each channel separately. Suppose our input is a small batch of data consisting of \( b \) samples, and the feature map output from one of the convolutional layers in the model is a 4th rank tensor \( x \in \mathbb{R}^{b \times c \times h \times w} \) where \( c \) is the number of channels of the output feature map, and \( h \) and \( w \) are the height and width of the feature map, respectively. We use the following formula to batch normalize the feature map \( x \). First, the mean and variance on each channel of \( x \) are calculated using equation 3.1 and 3.2. For every channel \( v \in \{1, 2, \ldots, c\} \):

\[
\mu_v = \frac{1}{b \times h \times w} \sum_{i=1}^{b} \sum_{j=1}^{h} \sum_{k=1}^{w} x_{i,j,k,v} \tag{3.1}
\]

\[
\sigma^2_v = \frac{1}{b \times h \times w} \sum_{i=1}^{b} \sum_{j=1}^{h} \sum_{k=1}^{w} (x_{i,j,k,v} - \mu_v)^2 \tag{3.2}
\]

After obtaining the mean and variance of each channel, we normalized \( x \) using equation 3.3.

\[
\hat{x}_v = \frac{x_v - \mu_v}{\sqrt{\sigma^2_v + \varepsilon}} \tag{3.3}
\]

Here \( \varepsilon \) is a very small constant used to ensure that the denominator is not zero. In addition, the batch normalization layer also introduces two sets of learnable parameters \( \gamma \) and \( \beta \), which stretch and offset the normalized data.
\\[ y_v = \gamma \hat{x}^v + \beta \]  

[3.4]

The neural network can judge the previous normalization operation acts as an optimization or not by back propagation and use these two parameters to adjust the effect of normalization. The network retains the possibility of not performing the batch normalization process, and as long as the network learns to set $\gamma$ to $\sigma^2 + \varepsilon$ and $\beta$ to $\mu_v$, the data can be restored to the state before normalization.

### 3.3 The Complexity of CNN

The complexity of the CNN determines the difficulty of the model deployment, and our goal is to reduce the complexity of the large networks by compression methods so that the compressed model can be deployed to autonomous vehicles. In this subsection, we present the methods to compute the model time complexity (FLOPs) and space complexity (parameters).

We first analyzed the complexity of the convolutional layer. Suppose a convolutional layer has $x$ convolutional kernels, each of size $(C_i, H_k, W_k)$, then the number of parameters of this convolutional layer is $X \times C_i \times H_k \times W_k$. Assume that the input feature map size of this layer is $(C_i, H_i, W_i)$, the stride of the convolution kernel in this layer is $S$, and the padding is $P$. Then the size of output feature map can be calculated by equations 3.5 and 3.6. [24]

\[
H_0 = \frac{H_i + 2 \times P - H_k}{S} + 1
\]  

[3.5]
Each element of the output feature map is the result of one convolution operation performed by the convolution kernel. And performing one convolution operation requires $C_i \times H_k \times W_k$ times multiplication and $C_i \times H_k \times W_k$ times addition, so nearly $2 \times C_i \times H_k \times W_k$ floating-point operations (FLOPs) are required. Combined with the number of elements in the output feature map, the convolution layer requires a total of $2 \times C_i \times H_k \times W_k \times C_o \times H_o \times W_o$ FLOPs. [25]

The complexity of the fully connected layer is relatively simple to compute. Suppose the fully connected layer has $X_i$ inputs and $X_o$ neurons as outputs, then the amount of computation is $X_i \times X_o$, and the number of parameters is also $X_i \times X_o$.

It can be seen that the number of parameters of the convolutional layer is much smaller than that of the fully connected layer for the same amount of computation, due to the fact that the weights of the convolutional layer are shared. Weight sharing is also one of the reasons why convolutional neural networks have better generalization ability compared to multilayer perceptron models.
Chapter 4

Network Compression Methods

In recent years, there has been an increasing amount of research on model compression and in this section, we review the four mainstream model compression methods, namely pruning, quantization, knowledge distillation, and low-rank decomposition.

4.1 Pruning

Pruning is by far the most used method in model compression. Generally, we need to find a criterion to judge the importance of parameters and set a threshold to filter out the parameters that need to be pruned, so as to reduce the redundancy of the model while ensuring accuracy.

According to the object of pruning, pruning can be broadly divided into two categories: unstructured pruning and structured pruning. The granularity of unstructured pruning is individual neurons, while structured pruning removes entire channels or filters. Pruning is not a new topic in machine learning and has been studied as early as the 1990s. Early pruning methods were mainly unstructured pruning. Hanson et al. [26] proposed a method to minimize the number of hidden units by decaying the weight of each hidden unit in the network according to its absolute value. Two classic papers in 1990, [27] and [28] proposed the OBS and OBD methods, respectively. They
used the second order derivative of the loss function with respect to the weights as a
criterion to measure the importance of the network parameters and set a threshold
to prune the parameters below the threshold to reduce the number of parameters in
the model.

At that time, neural networks were not a hot topic in machine learning, and
research on model compression was stagnant for a while. However, after 2012, as
more and more deep neural networks were proposed, the need for model compression
grew and more new pruning methods were proposed. Most of the research in recent
years has focused on structured pruning. Li et al. (2016) [29] used the absolute value
of the weights as a criterion to remove the entire filters in the network. Based on the
widespread use of the batch normalization layers, [30] added a channel-wise scaling
factor to the BN layers as a criterion to determine the importance of channels, and
then pruned the channels with small scaling factors.

4.2 Quantization

Similar to model pruning, there have been studies on model quantization since the
last century, such as [31] proposed by Fiesler et al. in 1990. After the rise of deep
learning, more related studies started to appear.

The most straightforward way to quantify the model is rounding. Gupta et al. [32]
proposed two rounding methods for converting 32-bits float-point numbers to 16-bit
fixed-point numbers. One is round-to-nearest and the other is stochastic rounding,
that is, rounding up or down with a certain probability, which is an unbiased
estimation. Another intuitive approach is to consider quantization as a data com-
pression problem. The quantization part in [33] is to cluster the weights, generate
the codebook, and then utilize weight sharing to achieve weight compression. Vector
quantization [34] is a common data compression method, and the basic idea is to map
the number of d-dimensional spaces to the codewords in the codebook. K-means and
product quantization are considered as such methods. PQ(product quantization) is
mainly for the case where the space is very large and the codewords grow exponen-
tially. It divides the original vector space into M low-dimensional subspaces, each
of which is quantized and finally the codewords are concatenated. Gong et al. [34]
introduced VQ into the weight quantization of neural networks. In [35], Stock et
al. improved the performance of PQ by minimizing the reconstruction error of the
activation layer. [36] proposed that randomly selecting a portion of the weight for
quantization in each iteration can reduce the effect of the gradient estimation bias
cauaed by the Straight-Through Estimator [37].

Ultra-low precision quantization has been a hot topic in recent years. Ultra-low
precision quantization here refers to 1-bit or 2-bit quantization. This type of quanti-
zation causes a very large loss of precision and therefore requires some unconventional
means to maintain precision. The most extreme ultra-low precision quantization is to
limit the bit width to 1 bit, that is, to binarize the network. In addition to reducing
the model size by a factor of tens, more appealingly, this can replace time-consuming
multiplication with hardware-friendly bit manipulation. [38] suggests quantizing the
weight to a triple value(-1, 0, +1). To solve the problem of training quantized models,
the authors took the approach of maintaining both high precision and low precision
weights, where the accumulation of weights is done under high precision and low preci-
sion weights are obtained by quantization. Rastegari et al. [39] proposed two efficient
approximations to standard convolutional neural networks: Binary-Weight-Networks
and XNOR-Networks.
4.3 Knowledge Distillation

The concept of knowledge distillation was first introduced by Hinton in [40], they were inspired by [41], in which the authors claimed that the knowledge acquired by a large ensemble of models can be transferred to a single small model. Hinton introduced soft targets related to the teacher’s network as part of the total loss to induce the student’s network to train for knowledge transfer. For the classification problem, we need to make the softmax distribution of the model match the true labels. By introducing soft targets, we only need to make the new model match the softmax distribution of the original model with the same input. The soft target is derived by dividing the predicted output of the teacher network by the temperature parameter and then applying a softmax transformation. The softmax distribution of the trained teacher model contains a certain amount of knowledge. By designing the total loss as a weighted average of the cross-entropy of the soft and hard target (true labels), the student network is able to learn the knowledge of the teacher network faster and better. In the early stage of training, we set the weight coefficients of the soft targets to be larger in order to make it easier for the student network to identify simple samples. However, in the later stage of training we need to reduce the weights of the soft targets appropriately so that the real annotations can help the student network to identify difficult samples.

In recent years, there are some extension work on knowledge distillation. The work in [42] proposed a method for training networks called Hint-based training. It is difficult to train a very deep neural network directly, and the authors proposed to choose the soft target of the middle layers of the teacher network as a guide to perform supervised learning on the middle layer of the student network, which can help the student network to obtain better training results. [43] argued that the direct transfer of feature maps as knowledge from teacher network to student network is too rigid
and ineffective. Therefore, they proposed to use the attention map as knowledge to transfer, expecting the student network to focus on the area that the teacher network focuses on.

4.4 Low-rank Factorization

The main computation of deep neural network lies in convolution computation, which is essentially a matrix operation. Through the study of linear algebra, we know that matrix decomposition methods such as SVD singular value decomposition can effectively reduce the computation of matrix computation. For high-dimensional matrices in neural networks, we need to use Tensor decomposition methods to accelerate and compress them. CP-Decomposition [44], Tucker decomposition, Tensor Train decomposition [45] and Block Term decomposition [46] are all typical tensor decomposition methods. However, in recent years, low-rank methods have become less popular. One of the most important reasons is that more and more networks are now using smaller convolution kernels (1x1, 3x3). It is difficult to achieve network acceleration and compression by performing matrix decomposition on small convolution kernels.
Chapter 5

Proposed Compression Method

Likened mentioned earlier, our goal is to reduce the storage space and computational power required for backbone networks such as VGG-16 network in autonomous driving systems, so that the compressed autonomous driving system can be deployed on edge computing devices such as vehicles. To achieve this goal, we propose “Flexible Compression”: a four-step compression process.

5.1 Flexible Compression

In this subsection, we first introduce the Flexi-CP module we designed and present our model compression process based on the Flexi-CP module. Then, we describe the other parts of the proposed compression process.

5.2 Flexi-CP Module

The structure of our proposed Flexi-CP module is as follows. The Flexi-CP module consists of a depth-wise convolution layer, a Flexi-batch normalization layer, and a point-wise convolution layer. In the following, we describe in detail how to use our
proposed module for model compression.

As we mentioned before in the knowledge distillation part, before utilizing knowledge distillation, we need to find a small model as a student model that minimizes the loss of accuracy of the model during the knowledge distillation process while minimizing the number of parameters. In this thesis, we will use our proposed Flexi-CP module to replace the convolutional layers of the model. Subsequent experiments demonstrate that our proposed structure is effective in reducing the number of parameters and the computation of the network. Next, we will first explain the theory of depth-wise separable convolution used in our proposed structure, and then analyze by theoretical calculations that our proposed structure can effectively reduce the number of parameters and the computation of the network.

Suppose we perform a conventional convolution (no padding, stride set to 1) of a image(12 x 12 x 3) with a 5x5 convolution kernel. After convolution we will get an 8x8 image(12 - 5 + 1 = 8). Since the image generally has 3 channels, our convolution

Figure 5.1: The four step compression process: architecture design, knowledge distillation, pruning and quantization.
kernel also needs to have 3 channels to match, so the shape of the convolution kernel is 5 x 5 x 3. Typically, after the convolution operation, we want to get the image with more features, so we use more convolution kernels to get different features of the image (here we use 256 convolution kernels). At this point we have 256 x 5 x 5 x 3 = 19200 parameters with 256 x 5 x 5 x 3 x 8 x 8 = 1228800 FLOPs.

We will then use depth-wise separable convolution to achieve the same result. The depth-wise separable convolution can be divided into two steps. The first step is depth-wise convolution. In this step, we use the same number of convolution kernels as the number of channels of the image. Each convolution kernel only convolves with the corresponding channel. We will end up with three 8 x 8 x 1 images, and by merging these three images, we will get one 8x8x3 image.

The second step is point-wise convolution. After depth-wise convolution, we successfully convert the 12 x 12 x 3 image to 8 x 8 x 3. Now we need to increase the channels of this image to get the same result as the conventional convolution. Point-wise convolution uses 1 x 1 convolution kernels with the same number of channels as
the input image, while the number of convolution kernels is the same as the number of channels desired. Therefore, we need 256 1 x 1 x 3 convolution kernels, each convolving with the input image (8 x 8 x 3). Finally, we get the same results as the conventional convolution. The number of parameters required for this convolution structure is $3 \times 5 \times 5 \times 1 + 256 \times 1 \times 1 \times 3 = 843$ with $3 \times 5 \times 5 \times 1 \times 3 \times 8 \times 8 + 256 \times 1 \times 1 \times 3 \times 8 \times 8 = 53952$ FLOPs. From the above theoretical calculation, we can observe that using depth-wise separable convolution instead of the original convolution layers can significantly reduce the number of parameters and computation of the model.

Unlike the original depth-wise separable convolution structure, we add a flexi batch normalization layer in the middle of the two convolution layers, which allows us to further compress the model and flexibly adjust the size of the model to suit different needs by adjusting the hyperparameters. Since flexi batch normalization layer is mainly used for model pruning, we will explain the working principle and role of flexi-batch normalization in detail in the later section.

In conventional convolutional neural networks, the convolutional layer is responsible for feature extraction of the image. After obtaining the feature map, the traditional approach is to connect the convolutional layers to the fully connected layer and then perform the activation function to classify the image. As we know from the previous section on model complexity calculation in the background, the number of parameters in the fully connected layer is much larger compared to the convolution layer for the same amount of computation. For the VGG-16 model, the number of parameters of the last three fully connected layers accounts for 89.36% of the whole model. The excessive number of parameters in the fully connected layers causes the model to be easily overfitted. Therefore, many approaches have been proposed to specifically address this problem, such as dropout. In our network we leverage Global
Average Pooling (GAP) to reduce the number of parameters and to alleviate overfitting. Also, GAP preserves the spatial and semantic information extracted from the previous convolutional and pooling layers.

GAP (Global Average Pooling) was first proposed by Lin et al. [47]. The difference between global average pooling and average pooling lies in the definition of the global. In contrast to global average pooling, traditional pooling can be referred to as local average pooling. Both global and local are used to describe the pooling window. “Local” means taking the average of a sub-region of the feature map and sliding that sub-region until the entire feature map is covered, while global averages the entire feature map. The final output of pooling is still 10 feature maps, with only one element left in each feature map, and that pixel is the average value obtained.

5.3 Knowledge Distillation

Through the previous step, we obtained the student model with a smaller number of parameters. In this step, we will first use a teacher model with a larger number of parameters to learn the dataset, and then leverage knowledge distillation to extract valid information from the teacher model to better train the student model. As we mentioned earlier, the goal of knowledge distillation is to make the distributions of the SoftMax outputs of the student and teacher models as close as possible. In traditional SoftMax function, we would first expand the gap between logits using the natural exponent $e$ and then normalize. This produces a distribution that approximates argmax, with the output being a near one-hot vector, where one value is large and the others are small. If the student model learns this SoftMax distribution directly, it is essentially no different from learning the positive labels directly. In fact, in the SoftMax layer, not only do the correct labels carry information, but the negative labels also carry a great deal of information. For example, certain negative labels
correspond to far greater probabilities than others. In the traditional training process, all negative labels are treated uniformly, and thus the knowledge in the negative labels is not utilized at all. Hinton proposed a more generalized SoftMax function [3.7] that enables the SoftMax distribution to amplify the information in the negative labels and utilize the knowledge of the teacher model more effectively.

\[ q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \]  

In this equation, \( q_i \) is the soft target of student model, \( z_i \) is the output logit of the deep neural network before the SoftMax operation, and \( T \) represents the temperature. The original SoftMax function is a special case of \( T \) equal to 1. We can observe from the formula that the larger \( T \) is, the more the output probability distribution of SoftMax tends to be smooth. Therefore, the information carried by the negative labels will be relatively amplified and the model training will focus more on the knowledge of the negative labels.

**Specific Method of Knowledge Distillation**

Knowledge distillation is generally performed in two steps. The first step is to train the teacher model, and the second step is to distill the knowledge from the teacher model(Net-T) to the student model(Net-S) under “high temperature” \( T \). In the first step, we generally choose a model with more parameters and more accurate to be trained at high temperature to get soft target. The second step is the most critical part of knowledge distillation, which we call it high-temperature distillation. In contrast to the general model training, the objective function of high-temperature distillation is obtained by weighting the soft loss(corresponding to soft target) and the hard
Figure 5.3: Teacher-student network knowledge distillation process
loss (corresponding to hard target). The specific formula for the loss function is as follows:

\[ L = \alpha L_{soft} + \beta L_{hard} \]  \hspace{1cm} [3.8]

\[ L_{soft} = -\sum_j^N p_j^T \log(q_j^T) \] \hspace{1cm} [3.9]

\[ p_i^T = \frac{\exp(v_i/T)}{\sum_k^N \exp(v_k/T)} \] \hspace{1cm} [3.10]

\[ q_i^T = \frac{\exp(z_i/T)}{\sum_k^N \exp(z_k/T)} \] \hspace{1cm} [3.11]

We use the SoftMax distribution generated by Net-T at high temperature as the soft target. The cross-entropy of the SoftMax output of the Net-S and soft target is the first part of the loss function \( L_{soft} \). The second part of the loss function \( L_{hard} \) is the cross-entropy of the SoftMax output of Net-S at \( T=1 \) with the ground truth labels. The second part of the loss function is necessary because Net-T also has a certain negative rate and using positive labels can effectively reduce the possibility of errors being propagated to Net-S.

**Temperature Selection**

The choice of temperature depends on how much attention the student model paid to negative labels during training. At lower temperatures, the model pays less attention to negative labels, especially those that are significantly below the mean [48]. In contrast, when the temperature is higher, the values corresponding to the negative labels increase relatively, and Net-S gives more attention to the negative labels. In fact, the negative labels contain certain information, especially those with significantly
higher values than the average. However, the training process of Net-T determines that the negative labels will have more disturbing information, and the lower the value of the negative labels, the less reliable their information will be. Therefore, the choice of temperature depends more on experience, and is essentially a trade-off between two things: learning from negative labels and preventing the effect of noise in negative labels. Hinton found experimentally that the student model was able to perform best when $T=20$, and we followed this hyperparameter configuration in our experiments.

5.4 Network Pruning

In the previous step, we performed knowledge distillation of the teacher network to obtain a student model with fewer parameters. In this section, we will use our proposed Flexi-CP module to further reduce the number of parameters of the network. As we mentioned earlier, network pruning is divided into structured pruning and unstructured pruning. Considering that unstructured pruning can only bring acceleration on specific hardware, practical deployment can be cumbersome. Structured pruning methods, on the other hand, can bring acceleration on general hardware, and thus have received more attention in recent years. There are two main types of structured pruning, one is layer-level pruning, and the other is channel-level pruning. Layer-level pruning requires pruning of the entire layer, which is less flexible. Moreover, when the network is deep enough, layer-level pruning brings a great loss of accuracy. Channel-level pruning, on the other hand, is more flexible and has less impact on the model accuracy. Therefore, in this thesis, we choose the channel-level pruning method. Like most network pruning methods, we need to find a criterion to measure the importance of channels, so that we can prune out those channels that are not important.
Flexi-batch Normalization Layer

When designing the Flexi-CP module, we added a Flexi-batch normalization layer between the depth-wise convolution layer and the point-wise convolutional layer. As mentioned in the background knowledge part, the batch normalization layer introduces two learnable parameters $\gamma$ and $\beta$ after performing batch normalization. These two parameters are introduced to restore the expressiveness of the data itself, and they perform a linear transformation on the normalized data. Experimental results of Liu [30] show that the scaling factor $\gamma$ in the batch normalization layer contains information that can reflect the importance of the filter, so here we leverage $\gamma$ of the trained model as the channel importance measure for flexible pruning of the model.

In Flex-batch normalization layer, we utilize the $\gamma$ to sort all channels and prune the channels with smaller $\gamma$ values according to the pruning rate. Meanwhile, Flexi-batch normalization layer will pass the indexes of pruned channels up to the depth-wise convolution layer to mask the pruned channels, and similarly down to the point-wise convolution layer.
Algorithm 1: Flexi-Pruning

**Input:** Net-O (original Network), $\gamma$ (scaling factors), $\alpha$ (pruning rate)

**Output:** Net-P (pruned Network)

Initialize the network Net-O;

Initialize the Net-P according to $\alpha$;

for each layers of Net-O do

Get all parameters of Net-O;

Find the $\gamma$ of the current layer;

Sort the channels by the value of $\gamma$;

Perform pruning based on $\alpha$;

Pass index of prunned channel to adjacent layers;

Prune adjacent layers;

end

Load the pruned parameters into Net-P;

Save Net-P;

Fine tune Net-P;

return Net-P;

There are two strategies for choosing the pruning rate, one is to choose a higher pruning rate in one step, and the other is to use a lower pruning rate for multiple cuts to finally achieve similar pruning results as the former. Frankle et al. [49] found experimentally that multiple iterations of pruning can obtain a lighter sub-network structure with relatively small loss of accuracy. Therefore, we leverage iterative small-scale pruning when performing network pruning, and found that it outperforms single large-scale pruning in terms of accuracy by comparison.

### 5.5 Quantization

As we mentioned in the literature review section, quantization is a very common method for model compression. To further compress our model, we also use quantization in the last step of our proposed compression process. In this thesis, we try to compress the model parameters to 16 bits and 8 bits.
First, we present the 16-bits approach. In PyTorch, the machine learning framework we use, the model parameters are stored using a 32-bit FloatTensor. PyTorch has three float types, float16, float32, and float64, so we can directly convert the 32-bit model parameters from float32 to float16.

Since PyTorch does not provide 8-bit floats, we need to design our own algorithm to quantize the parameters to 8-bit. As shown in equation 3.12, we first record the maximum and minimum values of each weight, then apply min-max normalization to each weight and multiply by $2^8 - 1$. Finally rounded and saved with the np.uint8 data type(). Here we not only save the quantized parameters, but also the maximum and minimum values of this set of parameters in the dictionary, so that we can recover the parameters when decoding them later.

$$W' = \text{round} \left( \frac{W - W_{\text{min}}}{W_{\text{max}} - W_{\text{min}}} \times (2^8 - 1) \right)$$ [3.12]

Algorithm 2: encode-8bit

Input: parameters-dict-32bit
Output: parameters-dict-8bit

for name, param in parameters-dict-32bit.items() do
  min = min(param);
  max = max(param);
  param = round((param-min)/(max-min)*(2^8 - 1));
  param = np.uint8(param) parameters-dict-8bit[name] = min,max,param;
end
return parameters-dict-8bit;

After finishing the model training, we save the model parameters, the data type of the parameters is float32 currently. Then we perform parameter quantization to save the parameters as low-precision data types. When invoking the model for inference, we need to use the decoding algorithm to reconvert the low-precision parameters to
high precision, at which time there is a certain loss of precision, resulting in a decrease of accuracy. In a similar way we are able to use 4-bit or even smaller data types to

<table>
<thead>
<tr>
<th>Algorithm 3: decode-8bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> parameters-dict-8bit</td>
</tr>
<tr>
<td><strong>Output:</strong> parameters-dict-32bit</td>
</tr>
<tr>
<td><strong>for</strong> name, param in parameters-dict-8bit.items() <strong>do</strong></td>
</tr>
<tr>
<td>min, max, param = param;</td>
</tr>
<tr>
<td>param = np.float64(param);</td>
</tr>
<tr>
<td>param = (param / 255 * (max - min)) + min</td>
</tr>
<tr>
<td>parameters-dict-32bit[name] = param;</td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
<tr>
<td><strong>return</strong> parameters-dict-8bit;</td>
</tr>
</tbody>
</table>

save the parameters, but the accuracy of such model parameters is too low and will lead to a significant decrease in the accuracy of the model, so we only use 16-bit and 8-bit quantization in our experiments.
Chapter 6

Simulation Results and Discussions

6.1 Experiments on Backbone Networks

In this section, we first introduce the dataset used in the experiments and some hyperparameter settings. Then we verify the effectiveness of the proposed method by comparing it with other similar state-of-the-art methods. Finally, we discuss in detail the role of each component of the compression method proposed in this thesis.

6.1.1 Dataset Description

In this thesis we focus on compression and acceleration of convolutional neural networks in autonomous driving technology. The most common application scenario of convolutional neural networks is to deal with image classification problems, so in our experiments we choose the image classification task to examine the performance of the network. The image classification task is to infer the category to which the content of a given image belongs. For this experiment, two datasets, FOOD-11 and CIFAR-10 were used. In this subsection, we will describe these datasets in detail.

The FOOD-11 is a relatively small dataset with 9866 training images, 3347 testing images, and 3430 validation images. The dataset contains 11 different food categories
Table 6.1: Food categories of FOOD-11

<table>
<thead>
<tr>
<th>Labels</th>
<th>Food Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Bread</td>
</tr>
<tr>
<td>1</td>
<td>Dairy Product</td>
</tr>
<tr>
<td>2</td>
<td>Dessert</td>
</tr>
<tr>
<td>3</td>
<td>Egg</td>
</tr>
<tr>
<td>4</td>
<td>Fried Food</td>
</tr>
<tr>
<td>5</td>
<td>Meat</td>
</tr>
<tr>
<td>6</td>
<td>Noodles</td>
</tr>
<tr>
<td>7</td>
<td>Rice</td>
</tr>
<tr>
<td>8</td>
<td>Seafood</td>
</tr>
<tr>
<td>9</td>
<td>Soup</td>
</tr>
<tr>
<td>10</td>
<td>Vegetable/Fruit</td>
</tr>
</tbody>
</table>

such as bread, dairy product, etc. In FOOD-11, the resolution of the images varies greatly, so appropriate adjustment needs to be made before feeding the images into the network.

The CIFAR-10 dataset consists of 50,000 training images and 10,000 testing images, comprising a total of 10 different categories. Each category has 5,000 training images and 1,000 testing images. These images are all three-channel color images with 32x32 resolution.

6.1.2 Evaluation Metric

In our experiments, three evaluation metrics are used to evaluate the performance of the network: accuracy, number of parameters, and FLOPs.

Accuracy is the most important performance metric of the network. In general, network compression of the baseline network results in some degree of accuracy loss.
The effectiveness of the compression method can be visualized by comparing the number of parameters and FLOPs in the original model and the model after performing model compression.

### 6.1.3 Parameter Setup

We used VGG-16 as our baseline network. The VGG-16 model was proposed by Oxford University Visual Geometry Group, and it achieved second place in the classification event of the ILSCVRC competition. Due to its simplicity and practicality,
many studies have used it as benchmark network.

Adam(Adaptive moment estimation) [50] was used for all training optimizers. Compared with other optimization algorithms, Adam converges faster and can overcome the problems of other optimization algorithms, such as vanishing learning rate and slow convergence. We refer to the author’s configuration for the hyperparameter of the Adam algorithm($\beta_1=0.9$, $\beta_2=0.999$), and to prevent overfitting, we add the L2-norm of the model parameters in the loss function with the coefficient set to $10^{-4}$.

For the experiments on CIFAR-10, we trained models for 60 epochs, the batch size was set to 128, and the initial learning rate was set to 0.001. In addition, we used the widely used data augmentation method to preprocess the input images:

1. Random horizontal flip.
2. Random brightness adjustment.
3. Random contrast adjustment.
4. Random clipping: cropping out a region with a resolution of 24x24 for training.
5. Convert the image to tensor and normalize the value to [0,1].

For the experiment on FOOD-11, we trained models for 30 epochs with the batch size set to 128 and the initial learning rate set to 0.001. Since the resolution of the images in the FOOD-11 dataset varies greatly, we need to resize the images to the same size first. The data augmentation steps for FOOD-11 data are as follows:

1. All images are first resized to 128x128 regardless of aspect ratio and resolution.
2. Random horizontal flip
3. Random rotation(Maximum rotation angle set to 15°).
4. Convert the image to tensor and normalize the value to [0,1].
Table 6.2: Compression results of VGG-16 network on FOOD-11 after using Flexi-CP module

<table>
<thead>
<tr>
<th>Model</th>
<th>Accuracy(%)</th>
<th>Parameters(M)</th>
<th>FLOPs(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>93.29</td>
<td>14.98</td>
<td>15.38</td>
</tr>
<tr>
<td>Flexi-CP</td>
<td>88.75</td>
<td>1.95</td>
<td>2.14</td>
</tr>
<tr>
<td>Flexi-CP-KD</td>
<td>92.47</td>
<td>1.95</td>
<td>2.15</td>
</tr>
<tr>
<td>Pruned-GAP</td>
<td>90.64</td>
<td>2.20</td>
<td>7.74</td>
</tr>
<tr>
<td>Taylor</td>
<td>91.50</td>
<td>2.56</td>
<td>3.94</td>
</tr>
</tbody>
</table>

6.1.4 Experimental Results and Discussions

We first conducted experiments on FOOD-11 dataset with VGG-16 and used it as baseline model. The VGG-16 model we used here is different from the original proposed VGG-16 model. Many works have demonstrated that the last three fully connected layers of VGG-16 can be replaced by a global average pooling, and this replacement does not lose performance compared to the original model while reducing the number of parameters and computational effort. The number of parameters is reduced from 138M to 14.98M after replacing the fully connected layer with the global average pooling. We compressed the baseline model using our proposed compression algorithm, which we refer to as Flexi-CP. We also compressed VGG-16 using compression methods proposed in other papers, Pruned-GAP is the method proposed by [51], Taylor is the method proposed in [52]. To verify the effect of knowledge distillation, we first trained the student model directly once and then trained the student model again using the knowledge distillation method.

As can be seen from the table 6.2, all the compression methods we used in this experiment have some accuracy loss compared to the original network, and this performance loss is reasonable in the compression process. All compression methods reduce the number of parameters and computation of the original model, while our
Table 6.3: Pruning results with different pruning rates on the FOOD-11 dataset

<table>
<thead>
<tr>
<th>Pruning rate</th>
<th>Parameter(M)</th>
<th>Accuracy(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.95</td>
<td>1.85</td>
<td>87.26</td>
</tr>
<tr>
<td>0.9025(iterative 0.952)</td>
<td>1.76</td>
<td>84.08</td>
</tr>
<tr>
<td>0.9025(one-shot)</td>
<td>1.76</td>
<td>80.26</td>
</tr>
<tr>
<td>0.8574(iterative 0.953)</td>
<td>1.67</td>
<td>81.74</td>
</tr>
<tr>
<td>0.8574(one-shot)</td>
<td>1.67</td>
<td>75.67</td>
</tr>
<tr>
<td>0.8145(iterative 0.954)</td>
<td>1.58</td>
<td>77.89</td>
</tr>
<tr>
<td>0.8145(one-shot)</td>
<td>1.58</td>
<td>72.39</td>
</tr>
<tr>
<td>0.7728(iterative 0.955)</td>
<td>1.51</td>
<td>73.70</td>
</tr>
<tr>
<td>0.7728(one-shot)</td>
<td>1.51</td>
<td>69.45</td>
</tr>
</tbody>
</table>

proposed compression method shows better performance. We achieved good compression and acceleration without significant accuracy loss to the model. In addition, we compared the training using traditional method and knowledge distillation method. We notice that after training for 30 epochs, training with knowledge distillation enables the model to learn better and thus reach higher accuracy faster. As shown in the table 6.2, our proposed compression method achieves a higher compression ratio than other methods without significant accuracy loss to the model. After performing the first two steps of our proposed compression process, the classification accuracy of the VGG-16 model is reduced by only 0.8% compared to the baseline model, with 88% less computation and 87% less parameters.

We then pruned the model to observe the number of parameters and the accuracy of the model at different pruning ratio. We compared one-shot pruning and iterative pruning and found that iterative pruning had better pruning results. We set the pruning rate of iterative pruning to 0.95, and performed fine-tuning for 5 epochs after each pruning.
Table 6.4: Quantization results on the FOOD-11 dataset

<table>
<thead>
<tr>
<th>Bit</th>
<th>State_Dict_Size(MB)</th>
<th>Accuracy(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32(VGG)</td>
<td>65.47</td>
<td>93.29</td>
</tr>
<tr>
<td>32(pruned)</td>
<td>7.68</td>
<td>87.26</td>
</tr>
<tr>
<td>16</td>
<td>4.15</td>
<td>86.87</td>
</tr>
<tr>
<td>8</td>
<td>2.24</td>
<td>86.47</td>
</tr>
</tbody>
</table>

From table 6.3, we can notice that pruning the model using our proposed pruning method further reduces the number of parameters in the model and ensures that the accuracy does not drop drastically. Furthermore, we observe that the iterative pruning method reduces the accuracy loss of pruning better than one-shot pruning.

Finally, we performed model quantization on the pruned model (the pruning rate is 0.95) by converting the 32-bit model parameters to 16-bit and 8-bit using the quantization method we mentioned before. The following table shows the accuracy of the quantized model. As shown in the table 6.4, quantization of the model parameters can greatly reduce the memory requirements of the model and has little impact on the model accuracy.

To verify that the compression method proposed in this thesis is still effective on larger datasets, we further conducted compression experiments on VGG-16 with the CIFAR-10 dataset, and the experimental results are shown in the following table.

As can be seen from the table below, when using the CIFAR-10 dataset to train the model, the effectiveness of our proposed compression method is verified despite the increase in the amount of data. After compression, the number of parameters and the FLOPs of the model are reduced by 86% and 87%, respectively, and the compressed model loss only about 4% of accuracy (when the pruning rate is 0.95).
Table 6.5: Compression results of VGG-16 network on CIFAR-10 after using Flexi-CP module

<table>
<thead>
<tr>
<th>Model</th>
<th>Acc</th>
<th>Parameters(M)</th>
<th>FLOPs(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16</td>
<td>88.28%</td>
<td>14.73</td>
<td>314</td>
</tr>
<tr>
<td>Flexi-CP</td>
<td>85.32%</td>
<td>2.21</td>
<td>42</td>
</tr>
<tr>
<td>Flexi-CP-KD</td>
<td>87.58%</td>
<td>2.21</td>
<td>42</td>
</tr>
<tr>
<td>Pruned-GAP</td>
<td>85.64%</td>
<td>2.45</td>
<td>152</td>
</tr>
<tr>
<td>Taylor</td>
<td>86.82%</td>
<td>2.78</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 6.6: Pruning results on the CIFAR-10 dataset

<table>
<thead>
<tr>
<th>Pruning rate</th>
<th>Parameter(M)</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.95</td>
<td>2.15</td>
<td>85.67%</td>
</tr>
<tr>
<td>0.9025(iterative 0.952)</td>
<td>2.09</td>
<td>82.08%</td>
</tr>
<tr>
<td>0.9025(one-shot)</td>
<td>2.09</td>
<td>77.42%</td>
</tr>
<tr>
<td>0.8574(iterative 0.953)</td>
<td>2.00</td>
<td>78.65%</td>
</tr>
<tr>
<td>0.8574(one-shot)</td>
<td>2.00</td>
<td>73.27%</td>
</tr>
<tr>
<td>0.8145(iterative 0.954)</td>
<td>1.92</td>
<td>74.89%</td>
</tr>
<tr>
<td>0.8145(one-shot)</td>
<td>1.92</td>
<td>69.74%</td>
</tr>
<tr>
<td>0.7728(iterative 0.955)</td>
<td>1.85</td>
<td>70.46%</td>
</tr>
<tr>
<td>0.7728(one-shot)</td>
<td>1.85</td>
<td>66.23%</td>
</tr>
</tbody>
</table>

Table 6.7: Quantization results on the CIFAR-10 dataset

<table>
<thead>
<tr>
<th>Bit</th>
<th>State_Dict_Size</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>32(VGG)</td>
<td>63.34MB</td>
<td>88.28%</td>
</tr>
<tr>
<td>32(pruned)</td>
<td>7.47MB</td>
<td>85.67%</td>
</tr>
<tr>
<td>16</td>
<td>3.97MB</td>
<td>85.32%</td>
</tr>
<tr>
<td>8</td>
<td>2.14MB</td>
<td>84.89%</td>
</tr>
</tbody>
</table>
6.2 Applying Compression Algorithm to Autonomous Driving

As we mentioned earlier, although the theory and techniques of deep neural networks have made great progress in recent years, their high computational and storage consumption drawbacks hinder their deployment on devices with limited computational resources such as autonomous vehicles, cell phones, and IoT devices.

In this chapter we apply our proposed model compression algorithm to an autonomous driving model to reduce the memory required by the model and speed up the model inference.

The autonomous driving model we chose for this experiment is the behavioral cloning algorithm proposed by NVIDIA in [1], where they train the CNN to map the images acquired from the front camera directly to the steering angle. They deployed this model on real vehicles and achieve good results. These vehicles were able to automatically recognize road features and drive smoothly regardless of the presence of lane lines on the road. However, at that time, due to the lack of computing power and the fact that network structures like VGGNet and ResNet had not been proposed yet, their proposed model had only a small number of layers and parameters, so the accuracy of the steering angle was low and the ability to handle complex road conditions was also low. Therefore, in our experiments, we replace the Nvidia Model proposed in the NVIDIA’s paper with the VGG-16 model.

6.2.1 Dataset Introduction

To obtain enough training data, we used Udacity’s simulator to obtain image data and steering angle data. In the simulator, we can manipulate the simulated car to drive on the road. The simulated car is equipped with three cameras which record
the left, center, and right perspective images of the vehicle during driving, and the steering angle of the vehicle is also recorded by the central camera. The simulator has two types of operation, keyboard control and mouse (joystick) control, and we used each of these two types of control methods to obtain data and compare the collected data. When the vehicle is controlled using the keyboard, the steering angle data were concentrated in the range of $0 \sim 0.25$, while the vehicle is controlled using the mouse, the steering angle data largely conformed to a normal distribution. We believe this is because the steering angle is consistent when using the mouse control, while it changes abruptly when using the keyboard. The uniformity of the steering angle distribution in the dataset is critical to the model. When the steering angle of the vehicle is concentrated around 0, the model may not learn valuable information. A good dataset should be one in which the frequency of steering signals for clockwise, counterclockwise, and straight ahead are close enough, so we chose to use the dataset collected by the mouse. We tried to keep the car in the middle of the road at first, as this would reduce the probability of the vehicle colliding with the obstacles on either side of the road. However, we found that the data collected in this way was still heavily concentrated around 0, so the trained model might not be sensitive to curve lanes. Therefore, we decided to try to drive more from the edge of the road to the center of the road. This resulted in a much smaller number of zeros in the dataset and a more sensitive model for curve lanes. We used the simulator to obtain a total of 31,272 images recorded by the three cameras on the car during driving.

### 6.2.2 Data Preprocessing

Training a deep neural network requires a large amount of data, and the 30,000 images we collected are still too few for the model, therefore we need to increase the training data so that the training set contains as many road conditions as possible, thus we
use data augmentation to expand our dataset. The data augmentation method we used is as follows:

1. Random flip:
   Due to the high frequency of left turns in the simulator, we have an uneven left-right distribution of steering angles, so here we perform random flip on the training images while inverting the steering angles to make the dataset more symmetrical and allow the model to fully learn the characteristics of turns in both directions.

2. Random brightness:
   Changing the brightness of the image can simulate the road conditions under different lighting conditions, which helps to improve the robustness of the model under different road conditions.

3. Crop and resize:
   We found that each image contains a large portion of features that are not related to road conditions. By cropping out these useless areas and resizing the images, we can effectively compress the training data and speed up the
convergence of the model.

6.2.3 Model Design

After completing the data preprocessing, we need to choose suitable model. In this thesis we compared two different models, the end-to-end convolutional neural network model proposed by NVIDIA and VGG-16-based model. In this section we will briefly introduce these two models and compare the performance of both models on the dataset, and finally perform model compression on the VGG-16 model.

NVIDIA end-to-end Model

Nvidia has been working on deep learning and autonomous driving research, and in [1] they propose an end-to-end autonomous driving model. The following image shows the structure proposed in the paper.

The first layer of the network is the normalization layer, which uses a normalization operation that divides each channel of the image by 255 and subtracts 0.5. In this way, all elements of the image are normalized to between -0.5 and 0.5. Normalization is a preprocessing operation performed before the convolutional layer and performing the normalization operation helps to accelerate the model training on the GPU. The normalization layer is immediately followed by 5 convolutional layers, and here the role of the convolutional layers is feature extraction. The convolutional layers are followed by 3 fully connected layers for training the features extracted from the convolutional layers and finally output the steering control signal (steering angle).

VGG-based model
Figure 6.3: Nvidia model architecture
Compared with the model proposed by NVIDIA, the VGG-based model is more complex. The general idea is to use smaller convolutional kernels and deeper convolutional network (16-19) layers to achieve recognition and extraction of image features. In this experiment, we refer to the structure of the VGG network and deepen the model based on Nvidia model. Below is the structure of the model we designed.

As shown in the figure 7.3, we used the first 4 convolutional layers of the NVIDIA model and added the last 4 convolutional modules of VGG-16 afterwards. We learned from NVIDIA’s experience and add the Normalization layer at the bottom of the model to normalize each channel of the image to speed up the training. We use this
model as our baseline model, and in the next part, we will compress this model using our proposed compression method.

6.2.4 Training

In this section, we will use both the NVIDIA model and the VGG-based model to learn the dataset we obtained from the simulator, both models are trained for 20 epochs. Since our task is to return an appropriate steering angle based on the input image, which is actually a regression problem, here we use MSE (Mean Squared error) as our loss function. We used the commonly used Adam optimizer with the initial learning rate set to 0.001.

We compared the results of the two models after training. In the figure, “Training loss” represents the average of the model’s error over the 20 training epochs, and “Val loss” represents the average error of the model in the validation set. From the Figure 7.4, we can see that the VGG-based model has a smoother training process and less loss than the NVIDIA model. In the validation set, the loss curve of the NVIDIA model fluctuates more, while the loss of the VGG-based model is relatively stable and smaller than that of the NVIDIA model.

Figure 7.5 shows the comparison curves between the predicted and the corresponding true values of the two models, from which we can intuitively observe that VGG-based model fits the true values better compared to NVIDIA model. Then, we loaded the trained models into the simulator to implement the autonomous driving of simulate vehicles and observe the performance of both models. We observed that the steering angle of the NVIDIA model changed more dramatically during driving, which was consistent with its fluctuating loss curve. Compared with the NVIDIA model, the VGG-based model behaves more in line with human driving behavior,
Figure 6.5: Training results of NvidiaModel and VGG-based-model
Figure 6.6: Comparison curve between predicted and true values
steering smoothly and keeping straight when driving in a straight line without excessive maneuvers. Apparently, the VGG-based model outperformed the NVIDIA model in terms of both data and practical use. However, the cost of the improved accuracy is that the number of parameters of VGG-based model reaches 36,311,331 and the model size reaches 138.6MB, which is difficult to support for most edge computing devices. Therefore, if we want to deploy the VGG-based autonomous driving model on real vehicles, we must compress the model. In the next section, we will compress the VGG-based model using our proposed compression method and compare the performance of the compressed model with that of the original model.

6.2.5 Performing Compression

In this experiment, our goal is to return the appropriate steering angle based on the input image, which is essentially a regression problem. Knowledge distillation is not suitable for regression problems, therefore we do not use knowledge distillation in this model compression process. Following our proposed compression process, we first structurally modify the original model by replacing all convolutional layers in the original model except the first layer with our proposed Flexi-CP module, and then replacing the fully connected layer at the bottom with global average pooling. In the second step, we perform pruning operations on the model, using the Flexi-Pruning algorithm to iteratively prune the model according to the pruning rate, which we set to 95% initially. After each pruning, we will fine-tune the model for 5 epochs. In the final step, we save the model parameters and perform quantization operation on the model parameters, compressing the original 32-bit parameters to 8 bits to further compress the model.

As can be seen from the table 7.1, the compressed model using our proposed compression method consumes 95% less memory and 88% less computation than the
Table 6.8: Compression results of different pruning rate of the autonomous driving model

<table>
<thead>
<tr>
<th>Model</th>
<th>Memory Usage</th>
<th>FLOPs</th>
<th>Average Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-based Model</td>
<td>138.6MB</td>
<td>3950M</td>
<td>0.0123</td>
</tr>
<tr>
<td>NVIDIA Model</td>
<td>11.8MB</td>
<td>26.98M</td>
<td>0.0323</td>
</tr>
<tr>
<td>Flexi-CP Model(95%)</td>
<td>4.51MB</td>
<td>489.89M</td>
<td>0.0135</td>
</tr>
<tr>
<td>Flexi-CP Model(90%)</td>
<td>4.36MB</td>
<td>489.89M</td>
<td>0.0157</td>
</tr>
<tr>
<td>Flexi-CP Model(85%)</td>
<td>4.18MB</td>
<td>489.89M</td>
<td>0.0172</td>
</tr>
<tr>
<td>Flexi-CP Model(80%)</td>
<td>4.05MB</td>
<td>489.89M</td>
<td>0.0183</td>
</tr>
</tbody>
</table>

original VGG-based model when the pruning rate is 95%.
Chapter 7

Conclusion and Future Work

In recent years, with the rapid development of deep learning technology, autonomous driving technology based on computer vision has received more and more attention. In order to meet the high precision requirements of autonomous driving, the scale of the backbone network in autonomous driving technology is getting larger and larger. The inflated network size enables the backbone network to handle more complex tasks on the one hand, and brings huge computation and storage consumption on the other, which makes it difficult to deploy autonomous driving technology directly on unmanned vehicles. In this context, this problem investigates the compression and acceleration of the backbone network in autonomous driving and proposes Flexi-compression, a four-step compression process. The main work and contributions of this thesis are as follows.

- We proposed the Flexi-CP module. The convolutional layer is one of the most important layers in the backbone network. We effectively reduce the parameters as well as the computation of the network by replacing the convolutional layer in the network with the Flexi-CP module. In our experiments, we used Flexi-CP module for commonly used backbone networks, and obtained good results. For example, we reduced 88% of the computation and 87% of the number of
parameters in the VGG network without any significant loss of accuracy.

- We proposed a Flexi compression process based on Flexi-CP module. Flexi-CP module contains a flexi-batch normalization layer, in which we use the scale factor to prune the filter to achieve network pruning, and after pruning we also quantized the network to further reduce the computational effort and number of parameters.

- In order to verify that the method proposed in this thesis can also achieve the expected results in practical applications, we compressed an autonomous driving algorithm using our proposed compression algorithm. We performed data acquisition through a simulator and trained the original model with the acquired data. The original model was then compressed using Flexi-compression, which reduced the amount of parameters by 95% and the amount of computation by 88% with no significant difference in performance. We deployed the compressed model on a simulated car and achieved good driving performance.

Based on the work in this thesis, we can also conduct more in-depth research on the problem of compression and acceleration of autonomous driving algorithms in the future along the following directions:

- Adaptive pruning [53] [54]. The pruning ratio of the pruning algorithm proposed in this thesis is fixed, i.e., the pruning ratio of each convolutional layer is the same. However, the redundancy level of each convolutional kernel layer is different, and adjusting the pruning ratio according to the actual situation of each layer by adaptive pruning can provide a better pruning effect.

- Low-precision quantization [55] [56]. Low-precision quantization can further reduce the number of parameters in the network by using lower-precision quantization schemes, but how to compensate for the performance loss caused by
low precision needs further study.

- Network structure automatic design algorithm [57] [58], currently the networks we use are designed by human, most of them are based on their own experience, it takes a long time to try and modify. If we use automatic design algorithms, we can use machine learning to adjust the network results automatically, find the optimal structure of the network quickly, and reduce the number of parameters and the amount of computation of the network.
List of References


Appendix A

Experiment Code

A.1 Data Preprocessing

```python
import pandas as pd
import numpy as np
import cv2
import os
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm
import re
import torch
from torch import nn
import torch.nn.functional as F
from torchvision import transforms as tfms

# read the dataset
```
data_df = pd.read_csv('training_data_plains2/driving_log.csv', header=None)

# get the center, left and right as X data and the steering angle as y data
X = data_df[[0,1,2]].values
y = data_df[3].values
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.33)

print('Train Length:', len(X_train))
print('Val Length:', len(X_val))

# steering angle histogram
plt.hist(y_train, bins=20)
plt.show()

def choose_image(image_paths, steering_angle):
    '''Choose center | left | right and adjust the steering angle accordingly'''
    choice = np.random.choice(3)
    if choice == 0:  # left
        return cv2.imread(image_paths[1].strip()), steering_angle + 0.2
    elif choice == 1:  # right
        return cv2.imread(image_paths[2].strip()), steering_angle - 0.2
    return cv2.imread(image_paths[0].strip()), steering_angle  # center

def random_flip(image, steering_angle):
    '''Randomly flip the image horizontally and adjust steering angle'''
    if np.random.rand() < 0.5:
```python
image = cv2.flip(image, 1)
steering_angle = -steering_angle
return image, steering_angle

def random_brightness(image):
    """Randomly increase or decrease brightness""
    choice = np.random.choice(2)
    if choice == 1:
        return image + np.random.randint(1, 20)
    return image - np.random.randint(0, 30)

def augmentation(image_paths, steering_angle):
    """Generate an augmented image""
    image, steering_angle = choose_image(image_paths, steering_angle)
    image, steering_angle = random_flip(image, steering_angle)
    image = random_brightness(image)
    return image, steering_angle

def preprocess_image(img):
    """Preprocess an image by cropping, resizing and tensorizing""
    img = img[60:-25,:]
    img = cv2.resize(img, (200, 66))
    img = tfms.ToTensor()(img)
    return img

def batch_generator(X_data, y_data, batch_size, is_training=True):
```
'''Generate Training Data'''

images = torch.empty([batch_size, 3, 66, 200])
steers = torch.empty(batch_size)

while True:
    i = 0
    for idx in np.random.permutation(X_data.shape[0]):
        image_paths = X_data[idx]
        steering_angle = y_data[idx]
        # randomly decide to augment or not
        if is_training and np.random.rand() < 0.5:
            image, steering_angle = augmentation(image_paths,
                                                  steering_angle)
        else:
            image = cv2.imread(image_paths[0])
        images[i] = preprocess_image(image)
        steers[i] = steering_angle
        i += 1
    if i == batch_size:
        break
    yield images, steers

A.2 Models

class NvidiaModel(nn.Module):
    def __init__(self):
        super().__init__()
self.conv1 = nn.Conv2d(3, 24, 5, 2)
self.conv2 = nn.Conv2d(24, 36, 5, 2)
self.conv3 = nn.Conv2d(36, 48, 5, 2)
self.conv4 = nn.Conv2d(48, 64, 3, 1)
self.conv5 = nn.Conv2d(64, 64, 3, 1)

self.dense = nn.Sequential(nn.Linear(1152, 100),
nn.ELU(),
nn.Dropout(0.25),
nn.Linear(100, 50),
nn.ELU(),
nn.Linear(50, 10),
nn.ELU(),
nn.Linear(10, 1))

def forward(self, x):
    batch_size = x.size(0)

    # Convolutional Pass
    x = F.elu(self.conv1(x))
    x = F.elu(self.conv2(x))
    x = F.elu(self.conv3(x))
    x = F.elu(self.conv4(x))
    x = F.elu(self.conv5(x))

    # Linear Pass
x = x.reshape(batch_size, -1)

out = self.dense(x)

return out

# VGG based model

class O_VGGModel(nn.Module):

def __init__(self):
    super().__init__()
    self.cnn = nn.Sequential(nn.Conv2d(3, 64, 3, 1, 1),
                              nn.ReLU(),
                              nn.Conv2d(64, 64, 3, 1, 1),
                              nn.ReLU(),
                              nn.MaxPool2d(2, 2),
                              nn.Conv2d(64, 128, 3, 1, 1),  # 48 * 5 * 22
                              nn.ReLU(),
                              nn.Conv2d(128, 128, 3, 1, 1),  # 64 * 3 * 20
                              nn.ReLU(),
                              nn.MaxPool2d(2, 2),
                              nn.Conv2d(128, 256, 3, 1, 1),  # 128 * 1 * 18
                              nn.ReLU(),
                              nn.Conv2d(256, 256, 3, 1, 1),  # 128 * 1 * 18
                              nn.ReLU(),
                              nn.Conv2d(256, 256, 3, 1, 1),  # 256 * 1 * 18
APPENDIX A. EXPERIMENT CODE

```
      nn.ReLU()
      nn.MaxPool2d(2, 2),

      nn.Conv2d(256, 512, 3, 1, 1),  # 256 * 1 * 18
      nn.ReLU(),
      nn.Conv2d(512, 512, 3, 1, 1),  # 512 * 1 * 18
      nn.ReLU(),
      nn.Conv2d(512, 512, 3, 1, 1),  # 512 * 1 * 18
      nn.ReLU(),
      nn.MaxPool2d(2, 2),

      nn.Conv2d(512, 512, 3, 1, 1),  # 512 * 1 * 18
      nn.ReLU(),
      nn.Conv2d(512, 512, 3, 1, 1),  # 512 * 1 * 18
      nn.ReLU(),
      nn.Conv2d(512, 512, 3, 1, 1),  # 512 * 1 * 18
      nn.ReLU(),
      nn.MaxPool2d(2, 2),
      nn.AdaptiveAvgPool2d((1, 1))
    )

    self.dense = nn.Sequential(nn.Linear(512, 512),
      nn.ELU(),
      nn.Linear(512, 11),
      nn.ELU(),
    )
```
def forward(self, x):
    batch_size = x.size(0)

    # Convolutional Pass
    x = self.cnn(x)

    # Linear Pass
    x = x.reshape(batch_size, -1)
    out = self.dense(x)

    return out

# Flexi-CP Model
class C_VGG_Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.cnn = nn.Sequential(nn.Conv2d(3, 64, 3, 1, 1),
                                  nn.ELU(),
                                  nn.Conv2d(64, 64, 3, 1, 1, groups=64),
                                  nn.ELU(),
                                  nn.Conv2d(64, 64, 1),
                                  nn.MaxPool2d(2, 2),
                                  nn.Conv2d(64, 64, 3, 1, 1, groups=64),  # 48 * 5 * 22
APPENDIX A. EXPERIMENT CODE

```
  nn.ELU()
  nn.Conv2d(64, 128, 1),

  nn.Conv2d(128, 128, 3, 1, 1, groups=128),  # 64 * 3 * 20
  nn.ELU()
  nn.Conv2d(128, 128, 1),
  nn.MaxPool2d(2, 2),

  nn.Conv2d(128, 128, 3, 1, 1, groups=128),  # 128 * 1 * 18
  nn.ELU()
  nn.Conv2d(128, 256, 1),

  nn.Conv2d(256, 256, 3, 1, 1, groups=256),  # 128 * 1 * 18
  nn.ELU()
  nn.Conv2d(256, 256, 1),
  nn.Conv2d(256, 256, 3, 1, 1, groups=256),  # 256 * 1 * 18
  nn.ELU()
  nn.Conv2d(256, 256, 1),
  nn.MaxPool2d(2, 2),

  nn.Conv2d(256, 256, 3, 1, 1, groups=256),  # 256 * 1 * 18
  nn.ELU()
  nn.Conv2d(256, 512, 1),

  nn.Conv2d(512, 512, 3, 1, 1, groups=512),  # 512 * 1 * 18
```
APPENDIX A. EXPERIMENT CODE

```
    nn.ELU(),
    nn.Conv2d(512, 512, 1),

    nn.Conv2d(512, 512, 3, 1, 1, groups=512),  # 512 * 1 * 18
    nn.ELU(),
    nn.Conv2d(512, 512, 1),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(512, 512, 3, 1, 1, groups=512),  # 512 * 1 * 18
    nn.ELU(),
    nn.Conv2d(512, 512, 1),

    nn.Conv2d(512, 512, 3, 1, 1, groups=512),  # 512 * 1 * 18
    nn.ELU(),
    nn.Conv2d(512, 512, 1),

    nn.Conv2d(512, 512, 3, 1, 1, groups=512),  # 512 * 1 * 18
    nn.ELU(),
    nn.Conv2d(512, 512, 1),
    nn.MaxPool2d(2, 2),
    nn.AdaptiveAvgPool2d((1, 1)),
)

    self.dense = nn.Sequential(nn.Linear(512 * 1 * 1, 512),
    nn.ELU(),
    nn.Linear(512, 11),
```
APPENDIX A. EXPERIMENT CODE

def forward(self, x):
    batch_size = x.size(0)

    # Convolutional Pass
    x = self.cnn(x)

    # Linear Pass
    x = x.reshape(batch_size, -1)
    out = self.dense(x)

    return out

A.3 Prunning

def networkSlimming(old_model, new_model):
    # get parameters
    params = old_model.state_dict()
    new_params = new_model.state_dict()
    selected_idx = []
    for i in range(8):
        # get parameter gamma of each channel
        importance = params[f'cnn.{i}.1.weight']
        # get the number of channels of old model
        old_dim = len(importance)
# get the number of channels of new model
new_dim = len(new_params[f'cnn.{i}.1.weight'])

# sort channels
ranking = torch.argsort(importance, descending=True)

# Save the index of the remaining channels after pruning in list selected_idx
selected_idx.append(ranking[:new_dim])

now_processed = 1
for (name, p1), (name2, p2) in zip(params.items(), new_params.items()):
    #
    if name.startswith('cnn') and p1.size() != torch.Size([]) and now_processed != len(selected_idx):
        if name.startswith(f'cnn.{now_processed}.3'):
            now_processed += 1
        if name.endswith('3.weight'):
            if len(selected_idx) == now_processed:
                new_params[name] = p1[:, selected_idx[now_processed - 1]]
            else:
                new_params[name] = p1[torch.cat([selected_idx[now_processed], []]), selected_idx[now_processed - 1]]
        else:
            new_params[name] = p1[selected_idx[now_processed]]
    else:
        new_params[name] = p1
A.4 Quantization

```python
def encode8(params, fname):
    custom_dict = {}
    for (name, param) in params.items():
        param = np.float64(param.cpu().numpy())
        if type(param) == np.ndarray:
            min_val = np.min(param)
            max_val = np.max(param)
            param = np.round((param - min_val) / (max_val - min_val) * 255)
            param = np.uint8(param)
        else:
            custom_dict[name] = param
    pickle.dump(custom_dict, open(fname, 'wb'))

def decode8(fname):
    params = pickle.load(open(fname, 'rb'))
    custom_dict = {}
    for (name, param) in params.items():
        if type(param) == tuple:
            min_val, max_val, param = param
            param = np.float64(param)
            param = (param / 255 * (max_val - min_val)) + min_val
            param = torch.tensor(param)
```
```python
    else:
        param = torch.tensor(param)

        custom_dict[name] = param

    return custom_dict
```