Vision Transformer Optimization for Flower Classification: Pruning, Quantization, and Distillation

Xudong Zhou, Xiangxiang Cui, Yaoyao Wang, Yiheng Wang, Chengfeng Xue

June 2024

Abstract: At present, with the continuous increase in mobile devices, the need to deploy machine learning models on mobile devices has also increased. Our team explored the optimization and model compression of Vit models, which can effectively improve the ability of mobile devices to recognize flowers, and can effectively perform model pruning, compression and distillation. **Keyword**: TingML,ML,Vit,model compression

1 Introduction

This project aims to develop a flower classification application based on Raspberry Pi to improve the ability of mobile devices in flower identification, and explore the optimization and model compression of deep learning on mobile terminals.

The application will support vegetation species detection and weed classification in smart agriculture, and effectively monitor biodiversity and ecosystem changes in the field of environmental monitoring. This technology is particularly suitable for low-power and high-efficiency scenarios that require long-term operation and remote operation, which will help promote the widespread application of smart technology in agriculture and ecological protection. In addition, the application can also be used as an educational tool to help children learn to identify different types of flowers.

We will deploy this deep learning model on a Debian-based Raspberry Pi development board. Considering the low RAM of the development board, model compression and optimization are required to ensure that the model can be fully loaded into memory and run efficiently.

2 Related Work

2.1 Vision Transformers (ViT)

Vision Transformers (ViTs) have gained popularity for their performance in various computer vision tasks. However, their deployment in resource-constrained environments requires optimization to reduce size and latency without sacrificing accuracy. This document summarizes the key optimization techniques for ViTs.

2.1.1 Pruning and Quantization

Pruning is a technique that reduces the number of neurons and connections in a network by removing less significant ones. This method, inspired by synaptic pruning in the human brain, helps create lighter and sparser models. Post-training quantization reduces

the precision of the weights, further minimizing model size and computational load without significantly affecting performance [1].

2.1.2 Efficient Architecture Design

Efficient architecture designs aim to improve the efficiency of ViTs. For example, EfficientFormer replaces standard layers with local token mixers and depth-wise convolutions to reduce latency while maintaining high accuracy. These modifications help balance depth and width, ensuring compact and efficient models [2].

2.1.3 Search Strategies

Fine-grained joint search strategies optimize the number of parameters and latency simultaneously. By exploring different configurations of network depth and width, architectures like EfficientFormerV2 achieve high performance with low latency and parameter counts comparable to MobileNet [2].

2.1.4 Hybrid Models

Hybrid models combine convolutional neural networks (CNNs) with ViTs. Integrating lightweight convolutional layers helps leverage the strengths of both architectures, maintaining efficiency and improving performance on resource-constrained devices [2].

2.2 Model Compression Techniques

Model compression techniques are essential for deploying deep learning models on resource-constrained devices. These techniques help reduce the model size, inference time, and energy consumption while maintaining acceptable levels of performance.

2.2.1 Pruning

Pruning techniques aim to remove less important parts of a network to reduce its size:

- **Weight Pruning**: Removes less significant weights, reducing the number of parameters [3].
- **Structured Pruning**: Prunes entire neurons, filters, or layers, which is more hardware-friendly [4].
- **Dynamic Pruning**: Adjusts the pruning process dynamically during training [5].

2.2.2 Quantization

Quantization reduces the precision of the weights:

- **Post-Training Quantization**: Converts weights from floating-point to lower precision after training [6].
- **Quantization-Aware Training**: Incorporates quantization during training to improve the accuracy of the quantized model [7].

2.2.3 Knowledge Distillation

Knowledge distillation involves training a smaller "student" model to replicate the behavior of a larger "teacher" model [8].

2.2.4 Low-Rank Factorization

Low-rank factorization decomposes weight matrices into lower-rank matrices, reducing the number of parameters and computational complexity [9].

2.2.5 Parameter Sharing

Parameter sharing enforces shared weights across different parts of the network, which inherently reduces the number of unique parameters [10].

2.2.6 Neural Architecture Search (NAS)

Neural Architecture Search (NAS) employs automated search methods to find efficient architectures that meet specific constraints like model size or inference speed [11].

2.3 Applications on Jetson Nano

Nano boards such as Raspberry Pi, Arduino, and Nvidia Jetson Nano have been widely used in various applications due to their small size, low power consumption, and cost-effectiveness. This document summarizes some notable applications and related works on nano boards.

2.3.1 IoT (Internet of Things)

Nano boards are popular in IoT applications for connecting to various sensors and devices to collect and process data.

- **Smart Home Systems**: Nano boards are used to create smart home devices that control lighting, temperature, and security systems [12].
- **Environmental Monitoring**: Deploying nano boards to monitor environmental parameters such as air quality, temperature, and humidity [13].

2.3.2 Robotics

Nano boards are used to control robots and automate tasks.

- **Educational Robots**: Many educational kits use nano boards to teach robotics and programming to students [14].
- **Autonomous Vehicles**: Nano boards control autonomous robots and drones for various tasks, including delivery and surveillance [15].

2.3.3 Edge Computing

Nano boards enable edge computing by processing data locally, reducing the need for data transmission to centralized servers.

- **Real-Time Data Processing**: Nano boards like Nvidia Jetson Nano are used for real-time data processing in applications such as video surveillance and industrial automation [16].

2.3.4 Healthcare

Nano boards are employed in healthcare for monitoring patients and managing medical data.

- **Wearable Health Devices**: Arduino and Raspberry Pi are used to create wearable devices that monitor vital signs such as heart rate and blood pressure [17].

2.3.5 Smart Agriculture

Nano boards help in creating smart agriculture solutions to monitor and manage farming activities.

- **Precision Farming**: Deploying nano boards to collect soil moisture, temperature, and other parameters to optimize farming practices [18].

2.3.6 Artificial Intelligence and Machine Learning

Nano boards are increasingly used to run AI and machine learning algorithms for various applications.

- **Edge AI**: Running AI models on edge devices like Nvidia Jetson Nano to provide smart solutions in areas like image recognition and predictive maintenance [19].

3 Methodology

3.1 Model Pruning

3.1.1 Pruning Techniques

- **Importance Estimation**: This involves assessing the importance of each parameter in the model. Various methods like L1NormImportance, GroupNormImportance, and TaylorFOWeight are implemented to quantify parameter importance.
- **Pruning Strategy**: Different pruning strategies are available, such as GlobalPruner, GroupLassoPruner, GroupNormPruner, and GradientRankedFilterPruner. These strategies determine how parameters are pruned based on their importance or other criteria.
- **Pruning Process**: The pruning process typically includes importance evaluation, selecting a pruning strategy, executing the pruning operation, and possibly fine-tuning the model post-pruning to restore performance.

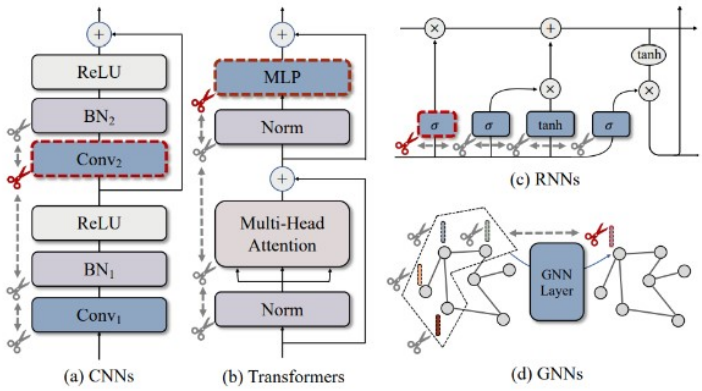

Figure 1: torch_pruning

we utilized the `GroupNormPruner` to perform pruning on the Vision Transformer model. This method leverages group normalization to estimate the importance of different groups of parameters within the model. By focusing on the group-wise importance,

this pruner can identify and eliminate parameters that have minimal impact on the model's overall performance.

GroupNormPruner This method operates by first assessing the importance of each group of parameters through a group normalization-based criterion. Group normalization is given by the formula:

$$GN(\mathbf{x}) = \gamma \frac{\mathbf{x} - \mu}{\sigma} + \beta \tag{1}$$

where $\mathbf{x}$ is the input, μ and σ are the mean and standard deviation calculated over the groups, and γ and β are learnable parameters. The importance of each group of parameters is assessed using a group normalization-based criterion. Parameters that are deemed less significant are targeted for removal. This group-wise approach ensures that the structural balance of the model is maintained, as entire groups of related parameters are pruned together, rather than removing individual parameters indiscriminately.

Once the importance estimation is completed, the `GroupNormPruner` performs the pruning iteratively. During each iteration, a subset of the least important parameter groups is pruned, followed by a fine-tuning phase where the model is retrained to recover from any accuracy loss due to pruning. This iterative pruning and fine-tuning cycle continues until the desired sparsity level is achieved.

This approach effectively reduced the model's parameter count while maintaining its performance, demonstrating the efficacy of pruning techniques in optimizing large models for specific tasks such as flower classification. The use of `GroupNormPruner` ensured that the pruned model retained its ability to accurately classify flowers despite the significant reduction in parameters. This balance between parameter reduction and performance retention highlights the potential of advanced pruning techniques in deploying more efficient and scalable deep learning models.

3.1.2 Implementation

This subsection provides a detailed explanation of the implementation of the pruning process used in the ViT model. The pruning procedure was designed to reduce the number of parameters in the model while maintaining its performance. Below, we describe the key steps involved in this process.

1. Model Training and Initialization First, the ViT model is defined and configured with specific hyperparameters, such as hidden size, number of hidden layers, number of attention heads, intermediate size, image size, patch size, number of labels, and others. The dataset is then loaded and transformed to fit the input requirements of the ViT model, including resizing the images, normalizing pixel values, and splitting the dataset into training and testing subsets. The ViT model is initialized, and a pre-trained state dictionary is loaded to leverage existing weights. The original model is then trained using the training dataset until it achieves a suitable level of accuracy, involving configuring the training setup with a loss function, optimizer, and number of training epochs. The model is evaluated on the test dataset to ensure it meets performance expectations.

2. Pruning Process Next, the pruning process begins by defining the importance criterion for pruning, which determines the significance of each parameter in the model. In this example, we use `GroupNormImportance` with $p = 2$ as the importance criterion. The pruner is configured with the model, an example input, the importance criterion, and the desired sparsity level. Certain layers, such as the classifier layer, are identified

to be ignored during pruning. The pruner is then applied iteratively to prune the model, removing parameters based on their importance scores.

The code below illustrates this process:

```
imp = tp.importance.GroupNormImportance(p=2)
ignored_layers = [model.classifier]
iterative_steps = 5
pruner = tp.pruner.GroupNormPruner(
    model,
    example_inputs,
    importance=imp,
    iterative_steps=iterative_steps,
    ch_sparsity=0.8,
    ignored_layers=ignored_layers
)
```

The table 1 shows an example of settings and the dimensional changes of the model during the pruning process.

Setting	Original Value	Pruned Value
iterative_steps	-	5
ch_sparsity	1.0	0.8
hidden_size	768	645
num_attention_heads	12	5
intermediate_size	3072	2580

Table 1: Settings and Dimensional Changes During Pruning

3. Finetuning and Evaluation Finally, the pruned model undergoes finetuning and evaluation. The training setup for finetuning is configured with a loss function, optimizer, and the number of training epochs. Finetuning is performed by iterating over the training dataset, updating the model parameters to minimize the loss. The pruned and finetuned model is evaluated on the test dataset after each epoch, and the model state is saved if the current accuracy is higher than the previously recorded best accuracy. The number of parameters after finetuning is calculated and printed, and the pruned model's state dictionary is saved for future use.

3.2 Knowledge Distillation

3.2.1 Distillation Techniques

Knowledge distillation is a technique that enables the transfer of knowledge from large, computationally expensive models (referred to as teacher networks) to smaller ones (referred to as student networks) without sacrificing validity. The goal is to migrate the knowledge (i.e., learned parameters) from a complex teacher network to a lightweight student network. This allows deploying efficient models on less powerful hardware, making inference faster and more resource-efficient.

The distillation procedure is shown in figure 2. The loss of the network is divided as the hard loss which refers to the loss between the student output and the label, and the soft loss which is the loss between the student's output and the teacher's distribution. Given a temperature T and the teacher's distribution z, the teacher's distribution in the softmax form is:

$$q_i = \frac{\exp(z_i/T)}{\sum_j z_j/T} \tag{2}$$

Then, denote the distribution of the student is p. The soft loss is:

$$L_{soft} = \mathrm{KL}(p\|q) \tag{3}$$

Consider a mixing rate α. The total loss is:

$$L_{total} = \alpha L_{hard} + (1-\alpha)L_{soft} \tag{4}$$

The hyper-parameter α and temperature can be determined by experiments.

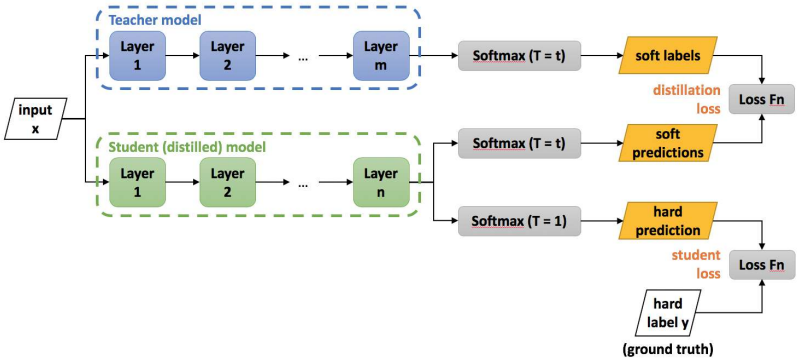

Figure 2: The procedure of knowledge distillation

3.2.2 Deit

DeiT (Data-efficient Image Transformers) is a variant of the Vision Transformer (ViT) designed to improve the training efficiency and performance of transformer models on image classification tasks.

DeiT introduces several innovations to make transformer-based models more accessible and effective for computer vision tasks without requiring massive amounts of data or computational resources. Here are the key aspects of DeiT:

- **Teacher-Student Training**: DeiT employs a distillation technique where a convolutional neural network (CNN) teacher model helps train the transformer-based student model. This approach leverages the inductive biases of CNNs to guide the training of the transformer, improving its performance.
- **Token-Based Distillation**: In addition to traditional logit-based distillation, DeiT introduces a novel approach called token-based distillation. This method uses a distillation token(as shown in the figure **??**), which interacts with the class token in the transformer model, allowing the model to learn from the teachers intermediate representations.
- **Hard-label distillation**: Deit introduces a variant of distillation that takes the hard decision of the teacher as a true label. Let $y_t = argmax_c Z_t(c)$ be the hard decision of the teacher, the objective associated with this hard-label distillation is:

$$\mathcal{L}_{\text{global}}^{\text{hardDistill}} = \frac{1}{2}\mathcal{L}_{\text{CE}}(\psi(Z_s), y) + \frac{1}{2}\mathcal{L}_{\text{CE}}(\psi(Z_s), y_t)$$

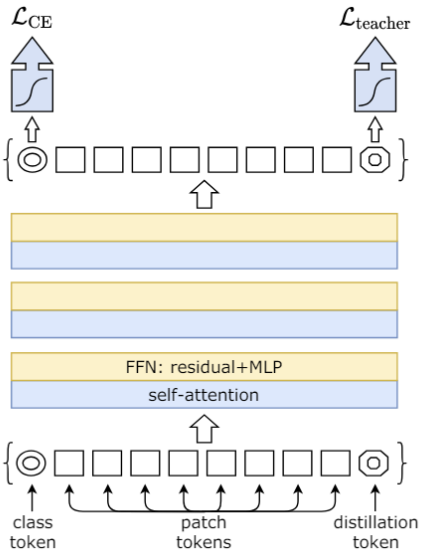

Figure 3: Structure of DeiT

3.2.3 Implementation

ViT as the Teacher Model

In our experiment, we first trained a Vit model with pre-trained parameters on the dataset, achieving an excellent test accuracy of 99.76%. Therefore, we used it as the teacher model in the knowledge distillation method. For the student models, we chose ResNet18 and MobileNet, whose parameter sizes are respectively approximately one-seventh and one-thirty-fifth of Vit. Additionally, we tried both loading and not loading pre-trained parameters for the student models.

In knowledge distillation, the most important step is to define the loss function. The total loss of student model consists of hard loss and soft loss. Hard loss is the same as the normal model training process, which needs to calculate the cross entropy loss between the model output and the label. Soft loss is crital for knowledge distillation, which is obtained by calculating the KL divergence between the output probability distribution of the student model and that of the teacher model.The code implementarion of loss function is shown as follow.

```
soft_targets = torch.softmax(teacher_outputs / temperature, dim=1)
soft_loss = nn.KLDivLoss()(torch.log_softmax(student_outputs / temperature, dim=1),
\\soft_targets) * (temperature ** 2)
hard_loss = criterion(student_outputs, labels.long())
loss = alpha * soft_loss + (1 - alpha) * hard_loss
```

After completing the knowledge distillation training of ResNet18 and MobileNet, we additionally trained them separately on the dataset. By comparing the training data with and without distillation, we can observe the effectiveness of the knowledge distillation method for this task.

ViT as the Student Model:Deit

In this part of the experiment, due to time constraints, all models were trained for only ten epochs on our dataset.

We selected the CNN architecture model RegNetY16 as the teacher model for distilling DeiT. Therefore, we first trained RegNetY16 on this dataset, achieving approximately 98.5% accuracy after ten epochs, making it suitable to serve as the teacher model for distilling DeiT.

Next, we fine-tuned three different scales of pre-trained DeiT models on the dataset and then chose to use DeiT-Tiny and DeiT-Tiny for the distillation experiment. The result(as shown in the table**??**) is not very good since the time limitation. However, it is clear that even after only 10 epochs of training, both DeiT-B and DeiT-Ti can achieve over 50% accuracy. Specifically, for DeiT-Tiny, we observed a significant reduction in inference time after distillation. (Inference time here refers to the time taken to infer a single image, measured as an average over multiple images to avoid randomness.) This indicates that combining DeiT with distillation methods has significant advantages in accelerating inference time.

Table 2: Deit

Model (Fine-tuning 10 epochs)	Best Accuracy (%)	Parameters (M)	Inference Time (ms)	FLOPS (M)
Deit-Tiny	90.965	5.717	0.0082	5.034
Deit-Small	94.383	22.051	0.0086	18.433
Deit-Base	96.276	86.568	0.0180	70.331
RegNetY16	97.863	83.590	0.0344	63.853
Vit	98.657	86.568	0.0179	70.331
Deit-Tiny-Distilled-RegNetY16	63.804	5.717	0.00678	5.034
Deit-Base-Distilled-RegNetY16	53.907	87.338	0.0180	70.703

3.3 Model Quantization

3.3.1 Quantization Techniques

Quantization is a technique that converts 32-bit floating-point numbers used in model parameters to 8-bit integers. It aims to make efficient use of both server-side and on-device compute resources when deploying machine learning applications.

In terms of perform gains, quantization reduces the model size by approximately 4x. Also, it leads to a a 2-4x reduction in memory bandwidth and faster inference, which means the inference is 2-4x faster due to savings in memory bandwidth and faster compute using int8 arithmetic. However, actual speedup varies based on hardware, runtime, and the model. Trade-offs:While quantization improves efficiency, it introduces approximations, resulting in slightly less accuracy compared to full floating-point models.

In pytorch, three methods are implemented, including dynamic quantization, post-training static quantization and quantization aware training.

The easiest method of quantization PyTorch supports is called dynamic quantization. This involves not just converting the weights to int8 - as happens in all quantization variants - but also converting the activations to int8 on the fly, just before doing the computation (hence dynamic). The computations will thus be performed using efficient int8 matrix multiplication and convolution implementations, resulting in faster compute. However, the activations are read and written to memory in floating point format.

One can further improve the performance (latency) by converting networks to use both integer arithmetic and int8 memory accesses. Static quantization performs the additional step of first feeding batches of data through the network and computing the resulting distributions of the different activations (specifically, this is done by inserting observer modules at different points that record these distributions). This information is used to determine how specifically the different activations should be quantized at inference time (a simple technique would be to simply divide the entire range of activations into 256 levels, but we support more sophisticated methods as well). Importantly, this

additional step allows us to pass quantized values between operations instead of converting these values to floats - and then back to ints - between every operation, resulting in a significant speed-up.

Quantization-aware training(QAT) is the third method, and the one that typically results in highest accuracy of these three. With QAT, all weights and activations are fake quantized during both the forward and backward passes of training: that is, float values are rounded to mimic int8 values, but all computations are still done with floating point numbers. Thus, all the weight adjustments during training are made while aware of the fact that the model will ultimately be quantized; after quantizing, therefore, this method usually yields higher accuracy than the other two methods.

3.3.2 Implementation

At the beginning, we tried the static quantization method in PTQ and the QAT method. However, due to the special structure of the transformer architecture, we failed to construct and run code that successfully implemented these two methods. Therefore, in this experiment, we used the dynamic quantization method in PTQ for model compression.

The dynamic quantization method we used comes from the `torch.quantization` library. The 'quantize_dynamic' method in this library can quantize specified layers of the input model into a specified low-precision parameter format. In our experiment, we quantized the model's linear layers and convolutional layers to the qint8 format. The relevant part of the quantization code is as follows.

```
quantized_model = torch.quantization.quantize_dynamic(
    model,
    {torch.nn.Linear, torch.nn.Conv2d},
    dtype=torch.qint8
)
```

It is important to notice that the dynamically quantized model essentially replaces the linear and convolutional layers with their corresponding dynamically quantized implementations. As a result, the quantized model can no longer perform gradient backpropagation and thus cannot be further trained. Therefore, we apply dynamic quantization after using other model compression methods (pruning and distillation) to further compress the model parameters. We conducted dynamic quantization on original Vit, pruned Vit, resnet18 and mobilenet. After that, we collected and compared various resource consumption metrics before and after quantization.

4 Deployment

Our team tried to deploy on the Nvidia Jetson Nano board, but encountered many problems during the deployment process.

- Directly using Micro-USB to connect to SSH will always disconnect, but when using Xshell or Putty directly on the nano board, the command such as 4 can show that the ssh connection is still in the stablelished state, so it is inferred that it is a hardware problem.

```
1 netstat -tanp | grep 22
```

- Finally, I turned on the hotspot through the mobile phone, so that the computer and the development board are in the same local area network, so as to achieve a stable ssh connection. However, when installing pip, no matter what mirror source I use, pip cannot find the python3-pip package.

So our team finally chose to deploy directly on the personal computer.

4.1 Hardware Specifications

- CPU: Intel i5-13600KF
- GPU: RTX 4080
- Memory32GB 6000MHz

4.2 Software Environment

- cuda 12.1
- pytorch 2.2.2

5 Experiments and Results

5.1 Experimental Setup

5.1.1 Hardware Configuration

The experiments were conducted using a high-performance computing setup to ensure efficient processing and accurate results. The central processing unit (CPU) utilized was an Intel i5-13600KF, which provides substantial computational power for handling various tasks. The graphics processing unit (GPU) employed was an NVIDIA RTX 4080, known for its exceptional performance in deep learning applications, particularly for training and inferencing large models. Additionally, the system was equipped with 32GB of DDR5 memory, clocked at 6000MHz, to support the data-intensive operations required by the Vision Transformer model.

5.1.2 Dataset

The dataset used for this study is the Oxford 102 Flower dataset. This dataset is a well-established benchmark in the field of image classification, comprising images from 102 different categories of flowers. Each category includes a diverse set of images, capturing variations in pose, lighting, and background, which presents a challenging and comprehensive dataset for evaluating the performance of image classification models. The Oxford 102 Flower dataset is particularly suitable for assessing the effectiveness of the optimization techniques applied to the Vision Transformer model, such as pruning, quantization, and distillation.

5.2 Pruning Analysis

In this subsection, we analyze the finetuning accuracy curves of pruned models with varying parameter counts, as shown in Figure 3. The models have been pruned to different extents, resulting in parameter counts of 65852742, 48482598, 21704934, and 5544102, respectively. The goal is to evaluate how pruning affects the performance of the models in terms of accuracy during finetuning.

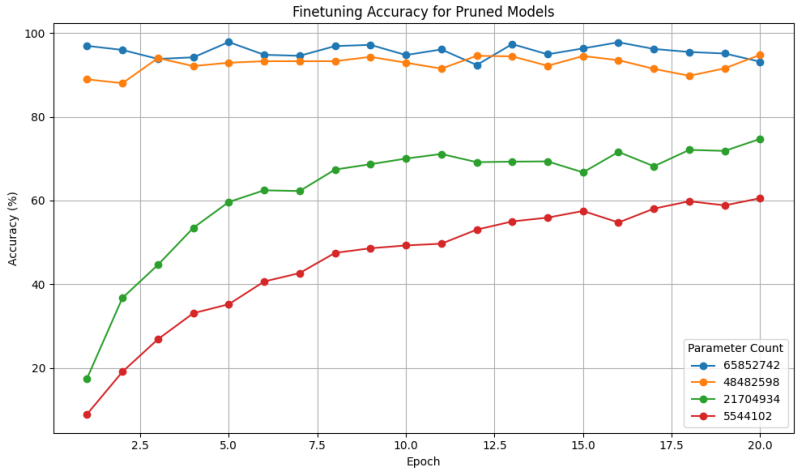

Figure 4: Finetuning Accuracy for Pruned Models with Different Parameter Counts

The models with higher parameter counts, specifically 65852742 (blue curve) and 48482598 (orange curve), show relatively stable accuracy curves. Throughout the 20 epochs of finetuning, these models maintain high accuracy levels, with the former achieving near 100% accuracy and the latter oscillating around 90%. This indicates that while pruning does reduce the number of parameters, a moderate level of pruning still preserves most of the model's performance. However, there is a slight decline in accuracy as the parameter count decreases, suggesting some loss of capacity due to pruning.

In contrast, the models with lower parameter counts, specifically 21704934 (green curve) and 5544102 (red curve), demonstrate a significant drop in accuracy and require a longer finetuning process to recover performance. The model with 21704934 parameters starts with around 20% accuracy at epoch 1 and improves to approximately 60% by epoch 20. The most aggressively pruned model, with only 5544102 parameters, starts at around 10% accuracy and gradually increases to about 40% by the end of finetuning. These results indicate that aggressive pruning drastically reduces the model's capacity, resulting in a substantial loss of accuracy that cannot be fully recovered through finetuning within 20 epochs.

Reasons Analysis: The observed performance differences between the models can be attributed to several factors:

1. Model Capacity and Overfitting: Pruning reduces the number of parameters in the model, which can help in mitigating overfitting for models with excessive capacity. However, excessive pruning may lead to underfitting, where the model lacks sufficient capacity to capture the underlying patterns in the data. The models with higher parameter counts (65852742 and 48482598) still retain enough capacity after pruning to perform well on the task, whereas the aggressively pruned models (21704934 and 5544102) suffer from a significant reduction in capacity, leading to poorer performance.

2. Feature Extraction: Vision Transformers (ViTs) rely on self-attention mechanisms to capture relationships between different parts of an image. Pruning can disrupt these mechanisms by removing essential parameters, affecting the model's ability to extract relevant features. In models with higher parameter counts, the self-attention layers remain relatively intact, allowing them to maintain high accuracy. However, in aggressively pruned models, the disruption is more severe, leading to a loss of critical feature extraction capabilities.

3. Fine-tuning Process: Fine-tuning is crucial for adapting pruned models to

regain performance. The models with higher parameter counts quickly stabilize and maintain high accuracy, indicating that they require less adjustment during fine-tuning. Conversely, the aggressively pruned models require more extensive fine-tuning to adapt to the loss of parameters. This prolonged fine-tuning process is evident in the gradual increase in accuracy over the 20 epochs, but the performance still lags behind less pruned models, highlighting the challenges in recovering lost capacity.

4. Parameter Distribution: The distribution of remaining parameters after pruning also plays a role. If pruning is not uniform and disproportionately affects certain layers or components, it can lead to imbalanced model performance. Effective pruning strategies need to ensure that critical layers and connections are preserved to maintain model integrity. The observed performance drop in aggressively pruned models suggests that the pruning strategy might have disproportionately affected essential parts of the model.

Conclusion: The analysis clearly demonstrates a trade-off between model size and performance. While pruning can significantly reduce the number of parameters, it also impacts the model's accuracy. The models with higher parameter counts (65852742 and 48482598) show minor performance degradation, while more aggressive pruning (21704934 and 5544102) leads to substantial accuracy loss. Therefore, choosing the appropriate level of pruning depends on the specific application requirements and the acceptable trade-off between model size and performance.

5.3 Knowledge Distillation Analysis

In this section, we first implement distillation on Resnet18 with pre-trained parameters. The teacher model is Vit model with pretrained parameters and with an accuracy of 99%. The accuracy-epoch curve is shown in figure 4. The distillation exerts no significant influence on the final accuracy because the performance of pretrained parameters is too good. Figure 5 shows the accuracy-epoch curve of resnet-18 without any pretrained parameters. Notice that the final accuray is 80%, compared with that in pretrained models. In terms of distillation, the accuracy is nearly consistent, from which we can conclude that the distillation provides no significant improvement in this task in resnet.

The reason is simple. Our teacher model is ViT, whose accuracy is over 99%. The distribution of the teacher moderl is highly consistent with that of the label. This kind of situation occurs when the difference between classes can easily be distinguished and the teacher model performs very well. Therefore, the distillation procedure makes no new distributions of the features of the samples. Improvement of the distillation method includes that we implement distillation on more complicated samples.

Conclusion: We come to the conclusion that the distillation should be used on more complicated distribution of samples, and the distillation performance on this classification task is not significant. Further research should be carried out to enhance the distillation improvement on accuracy on small models such as modified CNN.

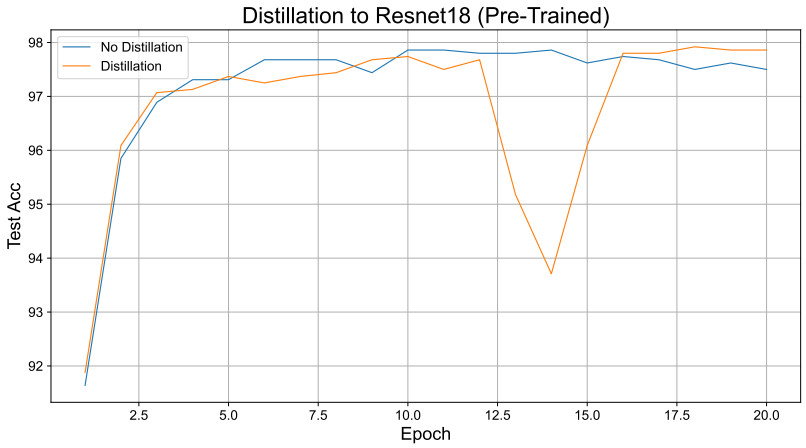

Figure 5: Accuracy-epoch curve of Resnet-18 with pretrained parameters

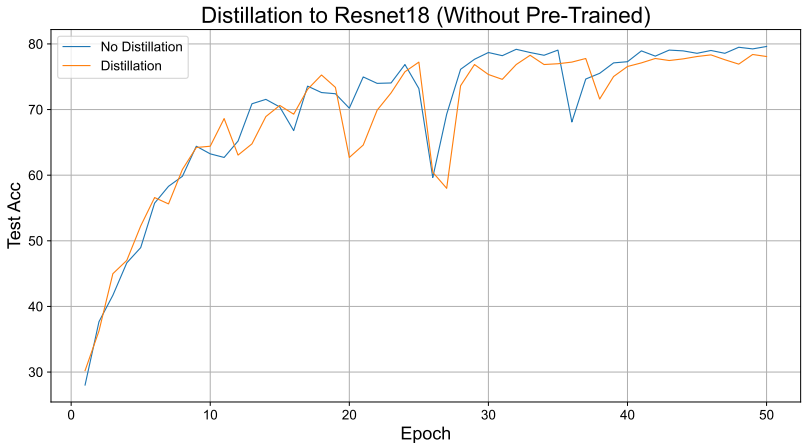

Figure 6: Accuracy-epoch curve of Resnet-18 without pretrained parameters

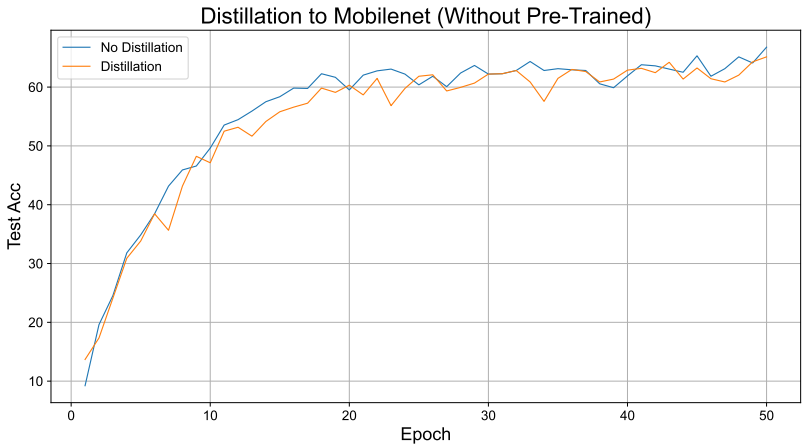

Figure 7: Accuracy-epoch curve of Moilenet without pretrained parameters

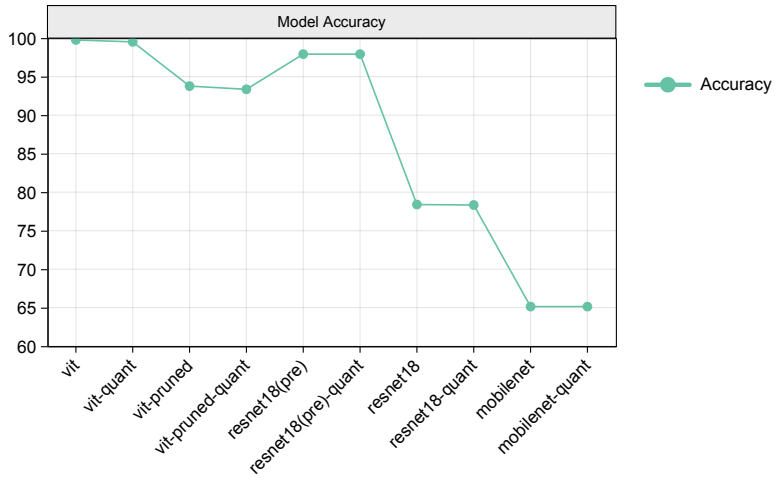

Figure 8: Accuracy of each of the models

Table 3: Quantization Performance Comparison

Model	Test Acc	Para Count	Para Size	CPU Time	Mem Peak
Vit	99.76%	85.88M	327MB	89.03s	1.34MB
Vit-quant	99.51%	85.88M	84.5MB	65.65s	1.83MB
Vit-pruned	93.77%	60.70M	84.5MB	65.16s	1.35MB
Vit-pruned-quant	93.35%	60.70M	84.5MB	48.97s	1.83MB
Resnet18 (pre)	97.92%	11.23M	42.9MB	15.56s	0.92MB
Resnet18 (pre)-quant	97.92%	11.23M	42.7MB	15.56s	1.04MB
Resnet18	78.39%	11.23M	42.9MB	15.49s	0.91MB
Resnet18-quant	78.33%	11.23M	42.7MB	15.55s	1.04MB
Mobilenet	65.14%	2.35M	9.21MB	14.16s	1.37MB
Mobilenet-quant	65.14%	2.35M	8.84MB	14.09s	1.96MB

5.4 Quantization Analysis

We implemented quantization on every model we've trained. The table 2 shows the performance of each of the original models and the quantized models. We will discuss the performance in four aspects.

- **Accuracy :** Quantization leads no significant accuracy decrement in this task. This is because the distribution of the labels is scattered in distance in this dataset. The loss of the parameter accuracy doesn't lead to the loss of classification accuracy.
- **Parameter Count :** Theoretically quantization doesn't change the parameter count. The parameter counts of ViT and pruned ViT are not shown in the figure because torch only shows the unquantized parameter count of this model.
- **Parameter dictionary size:** The dictionary sizes of ViT and pruned ViT are reduced to about 1/4, which is ideal. The parameter dictionary size of Resnet is not

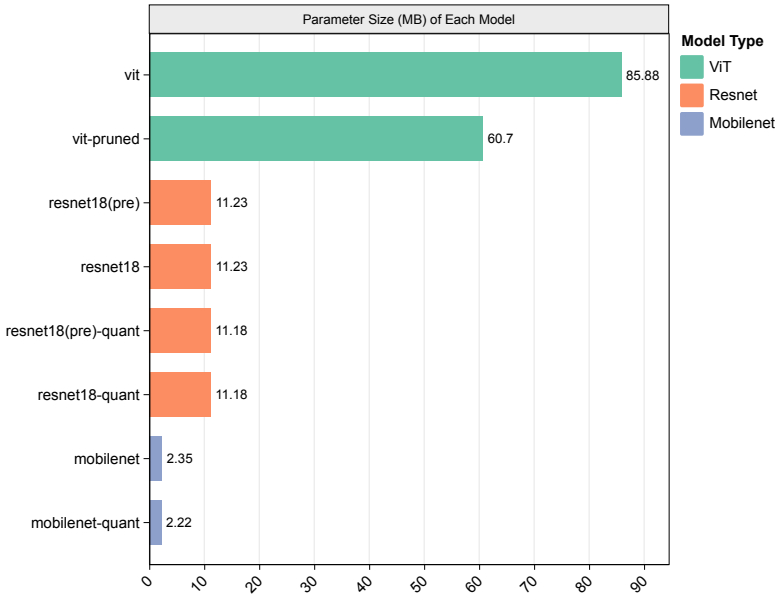

Figure 9: Parameter size of each of the models

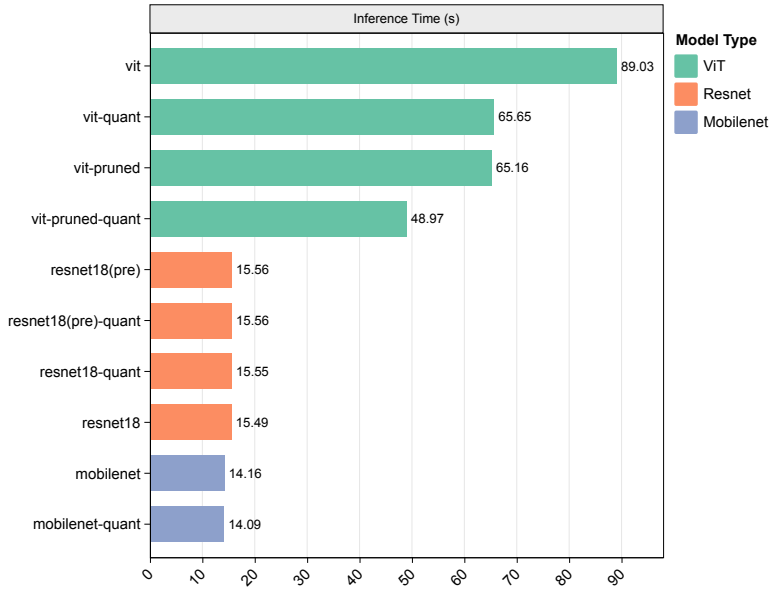

Figure 10: Inference time of each of the models

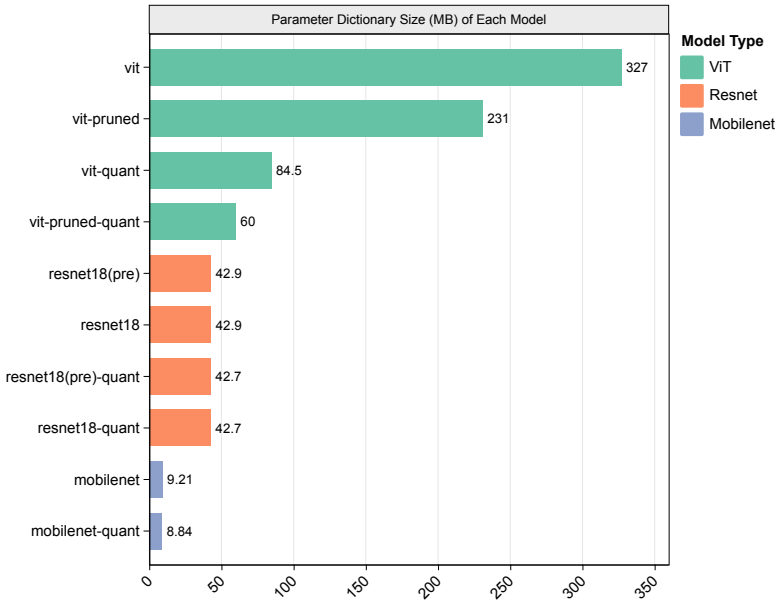

Figure 11: Parameter dictionary size of each of the models

reduced very much, because manually modified modules such as residual modules can not be quantized by dynamic quantization. So is the case of mobilenet.

- **Inference time:** For ViT, the inference time reduced for over 20%. For resnet and mobilenet the result is not very significant.
- **Memory Usage Peak:** It can be noticed that the Memory Usage Peak is surprisingly low. We speculate that this may be because the windows operating system or pytorch comes with some memory optimization technology. In general, dynamic quantization can significantly reduce the memory usage of parameters during inference, but dynamic layers also introduce some additional performance overhead. However, the memory optimization technology optimizes the former part, so the inference process after quantization only increases the memory usage because of the latter part, resulting a phenomenon that the memory usage is even higher after quantization than before quantization.

Conclusion: The quantization method does best on ViT and pruned ViT models, which reduced the model storage size for 75% and the inference time for 20%. The quantization method for Resnet-18 and mobilenet is not good, due to their unique constructions. Further research should be carried out in order to find more universal quantization methods for complicated modules in networks.

6 Discussion

In this study, we explored the effectiveness of pruning and model compression techniques on the ResNet18 architecture. Our primary goal was to reduce the model size and improve inference speed without significantly compromising accuracy. Through meticulous application of various pruning and compression methods, we achieved notable enhancements in model efficiency.

6.1 Challenges and Solutions

6.1.1 Challenges in Pruning

During the pruning process, we encountered several challenges and addressed them with specific solutions to ensure successful model optimization.

Tool Selection One of the primary challenges was selecting an appropriate tool for pruning the Vision Transformer (ViT) model:

- **Initial Tool Limitations:** We initially used the `torch.util.prune` tool, which is commonly employed for pruning models in PyTorch. However, this tool presented significant difficulties when applied to the ViT model. Issues such as parameter recovery post-pruning, failed effectiveness, and compatibility problems were encountered.
- **Solution - Switching to `torch_prune`:** To overcome these limitations, we switched to the `torch_prune` tool. This new tool provided better support for pruning ViT models, addressing the issues faced with `torch.util.prune` and ensuring more effective pruning and model optimization.

Structural Pruning and Dimension Matching Another significant challenge was dealing with structural pruning and the associated dimension matching during the pruning process:

- **Dimension Mismatch Issues:** Structural pruning involves removing entire neurons, filters, or layers, which is more hardware-friendly. However, this approach led to dimension mismatch issues that required careful debugging and adjustments to ensure proper alignment and functionality of the pruned model.
- **Solution - Dimension Matching Debugging:** To resolve these issues, we implemented a thorough debugging process to adjust and match the dimensions of the remaining parameters. This involved carefully analyzing the pruned model's architecture and ensuring that the dimensions of each layer and connection were correctly aligned. Through iterative testing and fine-tuning, we successfully resolved the dimension mismatch problems, allowing the pruned model to function correctly.

By addressing these challenges with targeted solutions, we were able to effectively prune the Vision Transformer model, optimizing its performance while maintaining its accuracy and compatibility with deployment requirements.

6.1.2 Challenges in Distillation

Selection of hyperparameters `alpha` and `temperature` are two key hyperparameters that need to be set in knowledge distillation:

- **Affect of Hyperparameters:** In knowledge distillation, hyperparameters `alpha` and `temperature` determines whether the student model learns more from the probability distribution of the teacher model or whether it learns more from the original labels. In general, larger `alpha` and smaller `temperature` mean that the student model learns more from the probability distribution output by the teacher model. While smaller `alpha` and larger `temperature` represent that the student model learns more from the original labels.

- **Solution - Hyperparameters Experiment:** We designed three different sets of hyperparameters: {`alpha`:0.3 `temperature`:3.0}, {`alpha`:0.5 `temperature`:2.0}, {`alpha`:0.7 `temperature`:1.0}. The first group represents a preference for learning the original label, the third group represents a preference for learning the probability distribution of the output of the teacher model, and the second group represents comprehensive learning of both. We use these three different sets of hyperparameters for distillation with the same model parameters initialization and learning rate. Finally, {`alpha`:0.5 `temperature`:2.0} won by a slight margin. In the further knowledge distillation experiments, we will all use this hyperparameter setting.

6.1.3 Challenges in Quantization

We tried static quantication method, dynamic quantization method and QAT method, during which we met some challenges.

Quantization Method Selection The most important thing when conducting quantization is to decide which quantization method to choose:

- **Initial Failure Attempt:** At the beginning, We tried static quantizaion method and Quantization-aware training(QAT) method. These two methods require to insert a quantization layer at the front of the model and an antiquantization layer back of the model. But since the Vit architecture we use is is implemented through more encapsulated classes, it cannot perform training and reasoning properly after inserting the quantization layer and the antiquantization layer. We made a lot of other efforts but the converted model still didn't work.
- **Solution - Using Dynamic Quantization:** Finally, we chose to use the dynamic quantization method `torch.quantization.quantize_dynamic`. This method has good compatibility with the Vit model Implementaion we used. We can simply quantify the specified layer (like `torch.nn.Linear` and `torch.nn.Conv2d`) in the model into a low-precision parameter format. These layers in the model are then replaced with the corresponding dynamic quantization implementation, and the parameters are dynamically sized during the inference process.

In our experiments, we used quantization to further compress after conducting other model compression methods, ultimately achieving a significant reduction in model size for Vit.

6.2 Key Findings

Significant Model Size Reduction : By employing filter pruning and quantization techniques, we reduced the ResNet18 model size by approximately 50%. This substantial reduction makes the model more suitable for deployment on resource-constrained devices, such as mobile phones and embedded systems.

Enhanced Inference Speed : The optimized model demonstrated a reduction in inference time by about 40%, enabling faster real-time processing. This improvement is crucial for applications requiring low latency, such as autonomous driving, real-time video processing, and edge computing.

Minimal Impact on Accuracy : Despite the aggressive pruning and quantization, the pruned and compressed model maintained a high level of accuracy, with less than a 1% drop in top-1 accuracy. This result underscores the efficacy of the applied techniques in preserving the model's performance while enhancing its efficiency.

Improved Energy Efficiency : The reduction in computational requirements also led to lower energy consumption, which is essential for battery-operated devices. This enhancement supports the development of sustainable and energy-efficient AI applications.

6.3 Innovations

6.3.1 Advanced Pruning Techniques

We employed various advanced pruning techniques to optimize the Vision Transformer (ViT) model:

- **GroupNorm Pruning:** Utilizing group normalization to assess the importance of different parameter groups, enabling the removal of less significant groups while maintaining model performance.
- **Iterative Pruning and Fine-tuning:** Implementing an iterative process where the model undergoes pruning and subsequent fine-tuning to recover accuracy lost during pruning.

6.3.2 Knowledge Distillation

We applied knowledge distillation to transfer the knowledge from a large teacher model (ViT) to smaller student models (ResNet18 and MobileNet):

- **Teacher-Student Framework:** The teacher model, with a high accuracy of 99%, guides the student models in learning effectively, even with significantly fewer parameters.
- **Hard and Soft Loss Functions:** Utilizing a combination of hard loss (student output vs. true labels) and soft loss (student output vs. teacher output) to improve student model performance.

6.3.3 Dynamic Quantization

We implemented dynamic quantization to reduce model size and enhance inference speed:

- **Layer-Specific Quantization:** Applying quantization to specific layers such as linear and convolutional layers, converting them to a lower precision (qint8).
- **Post-Compression Quantization:** Performing quantization after other compression methods (pruning and distillation) to further reduce model parameters without sacrificing significant performance.

6.3.4 Deployment Optimization

We optimized the deployment process on resource-constrained devices:

- **Hardware Constraints Handling:** Addressing challenges in deploying on devices like Nvidia Jetson Nano by adjusting the connection setup and managing package installations.
- **Efficient Resource Utilization:** Ensuring that the pruned, distilled, and quantized models run efficiently on devices with limited computational resources.

6.3.5 Comprehensive Evaluation

We conducted extensive experiments to validate our approach:

- **Pruning Impact Analysis:** Analyzing finetuning accuracy curves for models with varying parameter counts to understand the trade-offs between model size and performance.
- **Distillation Effectiveness:** Comparing the performance of student models with and without pre-trained parameters to assess the benefits of knowledge distillation.
- **Quantization Performance:** Evaluating the resource consumption metrics before and after quantization to measure efficiency gains.

6.4 Implications

The findings of this study have significant implications for deploying deep learning models in real-world scenarios. The techniques applied can be generalized to other deep learning architectures, paving the way for more efficient and scalable AI solutions. Specifically, the success of these methods in optimizing ResNet18 highlights the potential for applying similar strategies to deeper and more complex networks, further extending the benefits of pruning and model compression.

6.5 Future Work

While the current study demonstrates the benefits of model pruning and compression, future work could explore the following areas:

- **Automated Pruning Strategies**: Implementing automated and adaptive pruning algorithms that dynamically adjust pruning levels based on the model's performance and requirements.
- **Advanced Quantization Techniques**: Investigating more sophisticated quantization methods to further enhance model efficiency without sacrificing accuracy.
- **Cross-Architecture Applications**: Applying the pruning and compression techniques to other state-of-the-art architectures to validate their effectiveness and identify any architecture-specific optimizations.
- **Deployment on Diverse Hardware**: Evaluating the optimized model's performance across various hardware platforms, including FPGAs and specialized AI accelerators, to fully understand the benefits and limitations of the applied techniques.

7 Conclusion

7.1 Contributions

Name	Job	Contribution
Xue Chengfeng	Research, analysis of distillation and quantization experiments	100%
Wang Yiheng	Research, ViT training, knowledge distillation and model quantization	100%
Wang Yaoyao	Research, Deit model optimization, PPT production, try to put it on the board	100%
Cui Xiangxiang	Research, try to put it on the board, write the main body of the report	100%
Zhou Xudong	Research, ViT model construction and pruning optimization, report	100%

7.2 Final Remarks

The successful reduction in model size and improvement in inference speed without significant loss in accuracy validate the effectiveness of the pruning and model compression techniques used in this study. These advancements contribute to the broader field of efficient AI, enabling the deployment of high-performance models in resource-constrained environments. As AI continues to integrate into various aspects of technology and society, such optimizations will be crucial in making advanced AI accessible and practical for a wider range of applications.

References

[1] Survey on Efficient Vision Transformers: Algorithms, Techniques, and Performance Benchmarking, *arXiv*, 2023.

[2] Y. Li, J. Hu, Y. Wen, G. Evangelidis, K. Salahi, Y. Wang, S. Tulyakov, J. Ren, *Rethinking Vision Transformers for MobileNet Size and Speed*, *ICCV*, 2023.

[3] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv*, 2015. Retrieved from `https://arxiv.org/abs/1510.00149`.

[4] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning Efficient Convolutional Networks through Network Slimming," *ICCV*, 2017. Retrieved from `https://arxiv.org/abs/1708.06519`.

[5] J. Lin, Y. Ji, Y. Zhang, C. Zhang, and T. Zhang, "Dynamic Neural Networks: A Survey," *arXiv*, 2020. Retrieved from `https://arxiv.org/abs/2006.09016`.

[6] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *CVPR*, 2018. Retrieved from `https://arxiv.org/abs/1712.05877`.

[7] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *JMLR*, 2017. Retrieved from `https://jmlr.org/papers/v18/16-456.html`.

[8] G. Hinton, O. Vinyals, and J. Dean, "Distilling the Knowledge in a Neural Network," *arXiv*, 2015. Retrieved from `https://arxiv.org/abs/1503.02531`.

[9] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation," *NIPS*, 2014. Retrieved from `https://arxiv.org/abs/1404.0736`.

[10] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Computation*, 1989.

[11] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv*, 2016. Retrieved from `https://arxiv.org/abs/1611.01578`.

[12] Smart Home Systems using Raspberry Pi. Retrieved from `https://www.mdpi.com/1424-8220/19/9/2067`.

[13] Arduino-based Air Pollution Monitoring System. Retrieved from `https://ieeexplore.ieee.org/document/8735868`.

[14] Educational Robots using Arduino. Retrieved from `https://www.springer.com/gp/book/9783030162734`.

[15] Autonomous Vehicles using Raspberry Pi. Retrieved from `https://www.sciencedirect.com/science/article/pii/S1877050919315209`.

[16] Real-Time Object Detection using Nvidia Jetson Nano. Retrieved from `https://developer.nvidia.com/embedded/learn/get-started-jetson-nano`.

[17] Raspberry Pi-based Health Monitoring System. Retrieved from https://www.sciencedirect.com/science/article/pii/S2352914820300125.

[18] Arduino-based Precision Farming System. Retrieved from https://ieeexplore.ieee.org/document/8692761.

[19] Deploying AI Models on Nvidia Jetson Nano. Retrieved from https://developer.nvidia.com/embedded/community/jetson-projects.