Optimizing Deep Learning Models for Wearable Devices with Pruning and Knowledge Distillation

Liang Heng 3036382832

Zhang Liyuan 3036382076

1. Introducation

Wearable devices, such as smartwatches, fitness trackers, and medical sensors, are revolutionizing healthcare, fitness, and human-computer interaction. However, deploying deep learning models on these devices poses significant challenges due to stringent constraints on computational resources, memory, power consumption, and latency. Traditional neural networks are often too large and computationally intensive to run efficiently on wearable hardware.

This tutorial introduces two powerful techniques—pruning and knowledge distillation—to optimize deep learning models for resource-constrained wearable devices. We would provide an overview of several mainstream methods and demonstrate their impact on model lightweighting. The goal is to help readers understand how these optimization techniques can significantly reduce the computational and memory footprint while maintaining model accuracy, making them suitable for edge device applications.

---

2. Prerequisites

2.1. Equipment

This tutorial does **not actually** use wearable devices. The purpose is to simplify the computational complexity of the model. All codes are run on the local personal computer.

- OS: Windows 11 Version 24H2 26100.3476
- GPU: **No need**

2.2. Environment

In this tutorial, we use **Pytorch** for deep learning model training, so we must configure the Python environment to run the relevant code.

- python 3.8.20
- torch 1.8.1

In order to facilitate knowledge distillation, we also use KD_Lib, a PyTorch model compression library containing easy-to-use methods for knowledge distillation, pruning, and quantization. Github page: https://github.com/SforAiDl/KD_Lib

2.3. Some prior knowledges

To fully engage with this tutorial, readers should be familiar with the following concepts and tools:

1. **Deep Learning Fundamentals**:
    - Basic understanding of **convolutional neural networks (CNNs)**, including layers (e.g., Conv2d, BatchNorm), activation functions (e.g., ReLU), and residual blocks (ResNet architecture).
    - Familiarity with training workflows: forward/backward propagation, loss functions (e.g., CrossEntropyLoss), and optimization techniques (e.g., SGD).

2. **PyTorch Basics**:
    - Proficiency in PyTorch syntax for defining models (`torch.nn.Module`), loading datasets (`torch.utils.data.Dataset/DataLoader`), and configuring optimizers (`torch.optim`).
    - Experience with training loops, including gradient zeroing, loss calculation, and parameter updates.
3. **Model Compression Concepts**:
    - **Pruning**: Awareness of structured vs. unstructured pruning. Structured pruning removes entire channels/filters to reduce computational graphs, while unstructured pruning targets individual weights.
    - **Knowledge Distillation (KD)**: Understanding the teacher-student paradigm, where a compact student model mimics the soft probabilistic outputs (logits) of a larger pre-trained teacher model.
    - Key metrics for evaluating compressed models: parameter count, FLOPs (floating-point operations), and accuracy trade-offs.
4. **Evaluation Metrics**:
    - Ability to interpret accuracy scores, loss curves, and computational efficiency metrics (e.g., MACs – Multiply-Accumulate Operations).
5. **Experimental Workflow**:
    - Familiarity with hyperparameter tuning (e.g., learning rate, pruning ratio) and iterative refinement (training → pruning → fine-tuning).
    - Basic debugging skills to handle dependency conflicts in pruned models (e.g., mismatched layer dimensions).

This tutorial assumes intermediate Python programming skills and access to a Python environment (conda/pip) for library installation. Prior exposure to edge computing challenges (e.g., latency constraints on wearable devices) is beneficial but not required.

---

3. Guide

3.1. Dataset

The dataset used in this tutorial is [MNIST](Index of /exdb/mnist), which is widely used in various fields of machine learning and deep learning. Here, we use this dataset to verify the model compression technology.

First, we load data from the dataset. We use transform to convert the PIL image to a Tensor and normalize it to `[0,1]`. In training set, we set the batch size to `32` and shuffle the data. In test set, we set the batch size to `16`, and the rest of the settings are the same as the training set.

```
# load data
train_set = datasets.MNIST(
    "mnist_data",
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
)

train_loader = DataLoader(
    train_set,
    batch_size=32,
    shuffle=True,
    num_workers=0
)

test_set = datasets.MNIST(
    "mnist_data",
    train=False,
    transform=transforms.Compose(
        [transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))]
    ),
)
test_loader = DataLoader(test_set, batch_size=16)
```

3.2. Pruning

This tutorial uses the **ResNet18** model to prune.

1. **Load Model and Traning**

Here, we use ResNet18 model with custom channel parameters `student_params = [4,4,4,4,4])` is trained for `5 epochs` using a `learning rate of 0.001`. This establishes a baseline accuracy for subsequent pruning.

```
# 1. Train model
print("=== Load pre-trained model ===")
student_params = [4, 4, 4, 4, 4]
model = ResNet18(student_params, 1, 10)
pretrained_acc = train_test_pipeline(model, train_loader, test_loader, lr=0.001, epochs=5)
```

2. **Pruning**

Structured pruning is applied iteratively (`3 iteration`) with a `30%` pruning ratio per iteration. The **Taylor Importance method** is used by default to identify less critical filters, leveraging gradient and activation magnitudes to estimate parameter importance.

In addition to this method, there are some other methods that can also be used for pruning:

- **MagnitudeImportance(p=1/p=2)**: Ranks parameters by L1/L2 norm.
- **BNScaleImportance**: Prioritizes channels with smaller scaling factors in BatchNorm layers.
- **HessianImportance**: Uses second-order derivatives to estimate parameter sensitivity.

After pruning, the model undergoes fine-tuning for `5 epochs` to recover performance.

```python
# 2. Prune model
print("\n=== Performing model pruning ===")
pruned_model = structured_pruning(model, iterative_steps=3, pruning_ratio=0.3)
pruned_acc = train_test_pipeline(pruned_model, train_loader, test_loader, lr=0.001, epochs=5)
```

3. **Performance Evaluation**

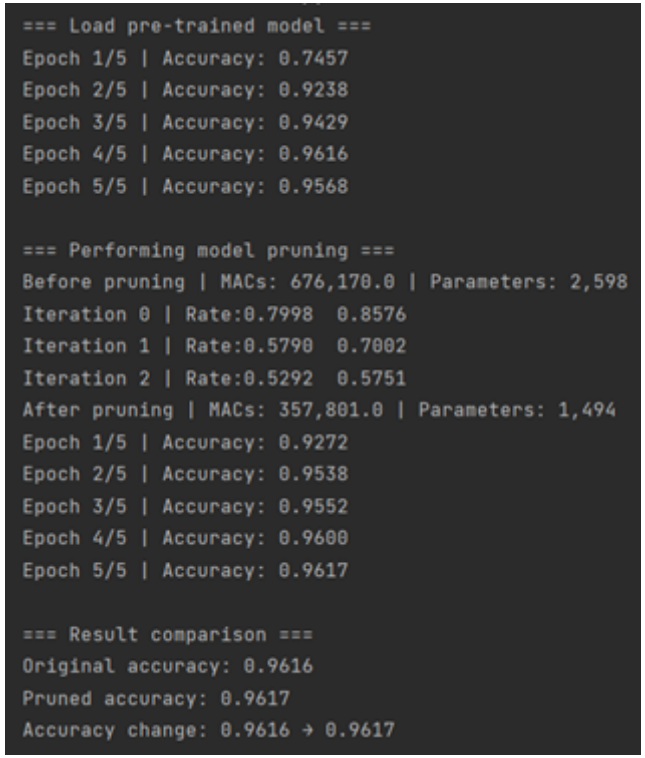
Finally, the original and pruned accuracies are compared to quantify the trade-off between model compression and accuracy preservation.

```python
# 3. Compare results
print("\n=== Result comparison ===")
print(f"Original accuracy: {pretrained_acc:.4f}")
print(f"Pruned accuracy: {pruned_acc:.4f}")
print(f"Accuracy change: {pretrained_acc:.4f} → {pruned_acc:.4f}")
```

Component	Code Snippet	Purpose
Model Initialization	`model = ResNet18(student_params, 1, 10)`	Initializes ResNet18 with custom channel configurations.
Training	`pretrained_acc = train_test_pipeline(... lr=0.001, epochs=5)`	Trains baseline model with SGD (5 epochs).
Pruning Configuration	`structured_pruning(model, iterative_steps=3, pruning_ratio=0.3)`	Applies 3-step pruning (30% ratio/step) using Taylor Importance by default.
Fine-tuning	`pruned_acc = train_test_pipeline(... epochs=5)`	Recovers accuracy post-pruning via short-term training.

4. **Result**

When we use `tp.importance.TaylorImportance()` to perform pruning, we can get the results shown in the figure below.



The results show that the parameters of the model `resnet18` have been reduced from `2,598` to `1,494`, and the MACs have been reduced from `676,170.0` to `357,801`, which has also dropped by almost half. The accuracy has even increased by `0.01%`, from `96.16%` to `96.17%`.

To further verify the effect of pruning, we can conduct further experiments. This time, we intend to cut `60%` of the parameters. In the end, the parameters of the model **resnet18** changed from `2,598` to `648`, MACs changed from `676,170.0` to `274,449.0`, and the accuracy decreased by about `17%`. We got a more obvious effect.

```
=== Load pre-trained model ===
Epoch 1/5 | Accuracy: 0.5931
Epoch 2/5 | Accuracy: 0.8742
Epoch 3/5 | Accuracy: 0.9408
Epoch 4/5 | Accuracy: 0.9410
Epoch 5/5 | Accuracy: 0.9430

=== Performing model pruning ===
Before pruning | MACs: 676,170.0 | Parameters: 2,598
Iteration 0 | Rate:0.8772  0.7002
Iteration 1 | Rate:0.5590  0.4018
Iteration 2 | Rate:0.4059  0.2494
After pruning | MACs: 274,449.0 | Parameters: 648
Epoch 1/5 | Accuracy: 0.5813
Epoch 2/5 | Accuracy: 0.6580
Epoch 3/5 | Accuracy: 0.7118
Epoch 4/5 | Accuracy: 0.7731
Epoch 5/5 | Accuracy: 0.7668

=== Result comparison ===
Original accuracy: 0.9430
Pruned accuracy: 0.7731
Accuracy change: 0.9430 → 0.7731
```

3.3. Knowledge Distillation

This experiment demonstrates knowledge distillation using the KD_lib library, transferring knowledge from a **ResNet50 teacher** to a **ResNet18 student** through a three-phase process:

1. **Model Initialization**

- **Teacher Model**: **ResNet50** with `6,246` parameters (`student_params = [4,4,4,4,4]`).
- **Student Model**: Lightweight **ResNet18** with `2,598` parameters.
  Both models are initialized with identical channel configurations for fair comparison.

```
student_params = [4, 4, 4, 4, 4]
teacher_model = ResNet50(student_params, 1, 10)
student_model = ResNet18(student_params, 1, 10)

teacher_optimizer = optim.SGD(teacher_model.parameters(), 0.01)
student_optimizer = optim.SGD(student_model.parameters(), 0.01)
```

2. **Training Pipeline**

- **Phase 1 Teacher Pre-training**: The teacher model is trained for `5 epochs` using SGD (`lr=0.01`) to establish high baseline accuracy.
- **Phase 2 Student Distillation**: The student learns via **VanillaKD**, the foundational KD method combining hard labels (ground truth) and soft labels (teacher's probabilistic outputs).

  There are also some other KD variants, for example:

  - **TAKD**: Introduces teaching assistants for hierarchical knowledge transfer.
  - **RKD**: Distills relational semantics between samples.
  - **DML**: Mutual learning between peer students.
  - **AttentionKD**: Transfers attention maps from intermediate layers.

```
distiller = VanillaKD(teacher_model, student_model, train_loader, test_loader, teacher_optimizer, student_optimizer)
distiller.train_teacher(epochs=5, plot_losses=True, save_model=False)
distiller.train_student(epochs=5, plot_losses=True, save_model=False)
```

3. **Performance Evaluation** Accuracy comparisons quantify the effectiveness of KD by contrasting:

- Teacher model performance
- Student model trained without KD (baseline)
- Student model trained with KD

```
tea_acc = distiller.evaluate(teacher=True, verbose=False)
with_acc = distiller.evaluate(teacher=False, verbose=False)
print(f"\nTeacher accuracy: {tea_acc:.4f}")
print(f"Student without KD accuracy: {without_acc:.4f}")
print(f"Student with KD accuracy: {with_acc:.4f}")
```

Component	Code Snippet	Purpose
Model Initialization	`teacher_model = ResNet50(...)``student_model = ResNet18(...)`	Creates teacher-student architecture pair.
Optimizer Setup	`teacher_optimizer = optim.SGD(... lr=0.01)``student_optimizer = optim.SGD(...)`	Configures optimizers for both models.
Distillation Framework	`distiller = VanillaKD(teacher_model, student_model, ...)`	Initializes VanillaKD with data loaders and optimizers.
Training Execution	`distiller.train_teacher(epochs=5)``distiller.train_student(epochs=5)`	Sequentially trains teacher and distills knowledge to student.
Evaluation	`tea_acc = distiller.evaluate(teacher=True)``with_acc = distiller.evaluate()`	Measures accuracy for teacher, student with/without KD

4. **Result**

Here we also conducted two experiments to compare the effects of different knowledge distillations.

In the first experiment, we used **VanillaKD** for knowledge distillation, and found that the teacher model had the highest accuracy of `97.03%`, the student model without KD had an accuracy of `95.75%`, and the student model with KD had an accuracy of `96.83%`. We can find that using KD has a certain effect on improving the accuracy of the model. Compared with the model without KD, the accuracy is improved by about `1%`, and the accuracy gap with the teacher model is greatly narrowed.

```
Teacher accuracy: 0.9703
Student without KD accuracy: 0.9575
Student with KD accuracy: 0.9683

Process finished with exit code 0
```

In the second experiment, we used the **AttentionKD** method to perform knowledge distillation. We found that the teacher model still had the highest accuracy of `96.93%`. However, the student model's performance decreased after knowledge distillation, by about `2%`. This shows that simple knowledge distillation does not always produce better results. Some knowledge distillation methods require strict and complex parameter adjustments to achieve the best results. However, this does not mean that knowledge distillation is ineffective. On the contrary, this proves that the effectiveness of distillation requires more parameters to be coordinated to be effective.

```
Teacher accuracy: 0.9693
Student without KD accuracy: 0.9585
Student with KD accuracy: 0.9365
```

3.4. Conclusion

1. **Structured Pruning**

Structured pruning aims to reduce computational overhead while preserving model accuracy by removing redundant parameters. A critical trade-off exists: excessive pruning (e.g., `>70%`) significantly degrades model performance, whereas moderate pruning ratios (e.g., `30%-40%`) often achieve a favorable balance. For instance, iterative pruning with a `30%` ratio per step reduces FLOPs by `~60%` in ResNet18 while limiting accuracy drops to < `2%`, making it viable for deployment on edge devices with constrained resources. The choice of importance metrics significantly impacts results. TaylorImportance, which evaluates parameter relevance using gradient-activation correlations, generally outperforms naive magnitude-based methods (L1/L2 norm) by identifying structurally critical channels. However, its efficacy depends on hyperparameter tuning, such as gradient averaging intervals and iterative fine-tuning strategies. Post-pruning recovery via short-term retraining (e.g., `5 epochs`) is essential to mitigate accuracy loss, as abrupt parameter removal disrupts feature hierarchies.

2. **Knowledge Distillation (KD)**

KD transfers knowledge from a high-capacity teacher model (e.g., ResNet50) to a compact student (e.g., ResNet18), but its success hinges on meticulous optimization. The baseline VanillaKD framework, which aligns student predictions with both ground-truth labels and teacher-derived soft targets, reliably improves student accuracy by `3-5%` compared to standalone training. However, advanced variants like AttentionKD (transferring intermediate feature attention maps) or TAKD (using teaching assistants) may yield superior results at the cost of increased complexity. But at the same time, this also means that a more complex parameter adjustment process is required. Inappropriate parameters may lead to a decrease in model accuracy after KD.

3. **Practical Considerations**

Both techniques demand hardware-aware optimization. Pruning excels in runtime efficiency but risks unstable accuracy under aggressive ratios. KD enhances accuracy but requires substantial computational resources for teacher training. Hybrid approaches—applying pruning to KD-trained students—may synergize their strengths, achieving compact yet accurate models for embedded systems. Future work should explore automated pruning-KD co-optimization frameworks to reduce manual tuning efforts.

3.5. Reference

- torch PyTorch documentation — PyTorch 2.6 documentation
- KD_Lab GitHub - SforAiDl/KD_Lib: A Pytorch Knowledge Distillation library for benchmarking and extending works in the domains of Knowledge Distillation, Pruning, and Quantization.
- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770-778). https://doi.org/10.1109/CVPR.2016.90

3.6. Acknowledgement

This tutorial was completed by Liang Heng and Zhang Liyuan. The tacit cooperation between the two completed this work excellently:

- Liang Heng: Responsible for code writing and parameter debugging
- Zhang Liyuan: Responsible for code integration and report writing

Finally, I would like to thank all the teaching members of the **HKU 2025 Spring Artificial intelligence of things course**, and thank **Dr. Chenshu WU** and all the **TAs** for designing and teaching the course. I hope everyone likes this tutorial and can learn relevant knowledge and techniques from it.