MDS 5122 — Deep Leaning
Homework 1

Zhouyixin

March 16, 2025

1. Environment Setting and Reproduction

I installed torch==2.5.1+cu121 and torchvision==0.20.1+cu121 by the following command.

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
```

I run the notebook on Kaggle with GPU T4*2. After about 45 minutes, the training of the example was finished with a loss of 0.209. The model performed an accuracy of 86.97% on CIFAR-10.

2. Part A

2.1 Introduction

Part A[1] realized an image classification task on CIFAR-10 with Deep learning models based on Pytorch. CIFAR-10 is a classical dataset in the computer vision, which contains more than 60,000 32*32 images of 10 classes. This experiment is aimed to explore the impacts of different DL techniques on the performances.

Trough a series of improvements, including network architecture modification, batch normalization, data augmentation, learning rate scheduler etc., the model finnally increased the test accuracy from 86.97% to 93.04% on CIFAR-10 2. Moreover, the best model also achieved a good performance of 99.71% on the MNIST dataset.

2.2 Baseline model

The baseline model employs a simple CNN architecture, comprising 5 convolutional layers and 4 fully connected layers. It is trained for 128 epochs using the Adam optimizer with weight decay regularization. For data augmentation, the training set is augmented with `RandomRotation`, `RandomHorizontalFlip`, and `RandomAffine` transformations, while the test set is normalized using the Normalize operation.

Regarding the network architecture, since the CIFAR-10 dataset consists of image data with three color channels, the input channel number is set to 3. The number of channels is progressively increased through the convolutional layers, following the sequence $3 \rightarrow 128 \rightarrow 256 \rightarrow 512 \rightarrow 512 \rightarrow 256$. Each convolutional layer is followed by a ReLU activation function. Additionally, dimensionality reduction is achieved via pooling layers, with a 2×2 max-pooling layer placed after the 1st, 2nd, and 5th convolutional layers for downsampling. The four fully connected layers have 512, 256, and 128 neurons, respectively, with the final layer outputting 10 classes.

[1]Codes and version logs: https://www.kaggle.com/code/yixinzhou2002/dl-hw1

After about 45 minutes, the training of the example was finished with a loss of 0.209. The model performed an accuracy of 86.97% on CIFAR-10.

The net is shown below:

```
net = nn.Sequential(
    nn.Conv2d(3, 128, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(256, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.Conv2d(512, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Flatten(),
    nn.Linear(256 * 4 * 4, 512), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(512, 256), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(256, 128), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(128, 10),
)
```

2.3 Variants

I sequentially experimented with FIVE variants and documented the models' performance on both the training and testing sets, with the test accuracy progressively improving (1). Due to the long runtime and minimal performance gains, the attempt to incorporate residual connections in Variant 3 was abandoned. Variant 4 continued to build upon the architecture of Variant 2.

Model	Conv	Linear	Batchnorm	epoch	Addintional Data Augmentation	lr scheduler
Baseline	5	4	×	128	×	constant
Variant1	5	4	√	128	×	constant
Variant2	7	4	√	64	×	constant
Variant3	7 ResBlock	4	√	64	×	constant
Variant4	7	4	√	64	√	constant
Variant5	7	4	√	64	√	Warm up + Cosine

Table 1: Model information

Model	Test acc	Time	Train loss	Test loss	Param
Baseline	86.97%	40m	-	-	7.29M
Variant1	89.31%	2h28m	0.033	0.472	7.29M
Variant2	89.60%	2h04m	0.042	0.456	127.06M
Variant3	92.04%	6h27m	0.070	0.335	134.87M
Variant4	92.04%	2h04m	0.215	0.291	127.06M
Variant5	93.04%	2h04m	0.154	0.249	127.06M

Table 2: Model performances on CFAR-10

2.3.1 Variant 1: BatchNorm & Dropout Ratio

Initially, based on the baseline model, this study considered employing normalization to mitigate internal covariate shift by adding `BatchNorm2d` and `BatchNorm1d` layers. This approach stabilizes the input distribution for each layer, thereby accelerating model convergence. Additionally, to preserve learning information, Dropout was removed from the convolutional layers, and the Dropout rate in the fully connected layers was reduced to **0.3**. The result was an accuracy increase to 89.31%.

From the learning curves (1), the training loss rapidly decreased in the first 20 epochs and then slowly declined after 40 epochs, eventually reaching a loss of 0.033. In contrast, the testing loss began to rise after 20 epochs, indicating model overfitting, with a final test loss of 0.472. However, the accuracy did not show significant changes. The discrepancy between loss and accuracy may be attributed to the fact that cross-entropy focuses on the model's predicted probability distribution, while accuracy is solely concerned with whether the highest-probability class is correct. Therefore, despite the improved accuracy, the model remains overfitted.

The net is shown below:

```
net = nn.Sequential(
    nn.Conv2d(3, 128, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2),nn.BatchNorm2d(128),
    nn.Conv2d(128, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2),nn.BatchNorm2d(256),
    nn.Conv2d(256, 512, 3, padding=1), nn.ReLU(inplace=True),nn.BatchNorm2d(512),
    nn.Conv2d(512, 512, 3, padding=1), nn.ReLU(inplace=True),nn.BatchNorm2d(512),
    nn.Conv2d(512, 256, 3, padding=1), nn.ReLU(inplace=True), nn.MaxPool2d(2),nn.BatchNorm2d(256),
    nn.Flatten(),
    nn.Linear(256 * 4 * 4, 512), nn.ReLU(inplace=True), nn.Dropout(0.3),nn.BatchNorm1d(512),
    nn.Linear(512, 256), nn.ReLU(inplace=True), nn.Dropout(0.3),nn.BatchNorm1d(256),
    nn.Linear(256, 128), nn.ReLU(inplace=True), nn.Dropout(0.3),nn.BatchNorm1d(128),
    nn.Linear(128, 10),
)
```

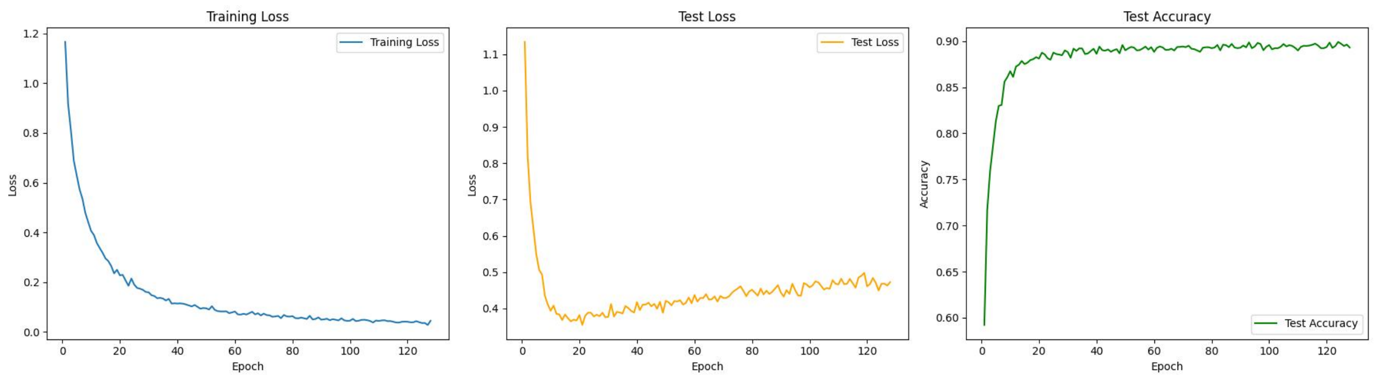

Figure 1: Variant 1: learning curve

2.3.2 Variant 2: Go Deeper and Wider

Variant 2 attempts to enhance the model's capacity to learn complex patterns by increasing the number of network layers and expanding the channel dimensions. Building upon Variant 1, this study increases the number of convolutional layers to 7 and widens the channel dimensions ($3 \to 256 \to 512 \to 1024 \to 1024 \to 2048 \to 2048$).

Additionally, `MaxPool` layers are added after the 1st, 2nd, and 7th convolutional layers, with normalization layers inserted between each layer. As a result, the number of model parameters increases from 7.29M to 127.06M.

The outcome is an accuracy improvement to 89.60%, and the learning curves (2) exhibit a shape similar to that of Variant 1.

The net is shown below:

```
net = nn.Sequential(
    nn.Conv2d(3, 256, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2), nn.BatchNorm2d(256),
    nn.Conv2d(256, 512, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2), nn.BatchNorm2d(512),
    nn.Conv2d(512, 1024, 3, padding=1), nn.ReLU(inplace=True), nn.BatchNorm2d(1024),
    nn.Conv2d(1024, 1024, 3, padding=1), nn.ReLU(inplace=True), nn.BatchNorm2d(1024),
    nn.Conv2d(1024, 2048, 3, padding=1), nn.ReLU(inplace=True), nn.BatchNorm2d(2048),
    nn.Conv2d(2048, 2048, 3, padding=1), nn.ReLU(inplace=True), nn.BatchNorm2d(2048),
    nn.Conv2d(2048, 1024, 3, padding=1), nn.ReLU(inplace=True),
    nn.MaxPool2d(2), nn.BatchNorm2d(1024),
    nn.Flatten(),
    nn.Linear(1024 * 4 * 4, 2048), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(2048),
    nn.Linear(2048, 1024), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(1024),
    nn.Linear(1024, 512), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(512),
    nn.Linear(512, 10),
    )
```

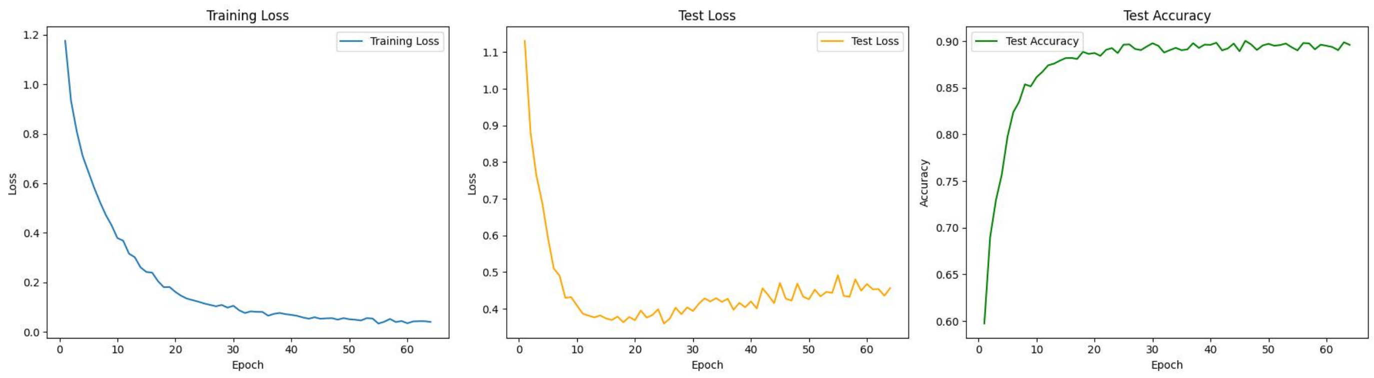

Figure 2: Variant 2: learning curve

2.3.3 Variant 3: Residual Connections

To address the issues of vanishing or exploding gradients, I initially considered incorporating residual connections. My first attempt involved directly implementing the ResNet34 architecture. However, this approach resulted in a training loss of 0.129 and a test accuracy of 84.33%, which was inferior to the baseline model. Moreover, the training process required 1 hour and 13 minutes, indicating that this architecture was not only less effective but also computationally expensive.

Given these results, I decided to integrate residual connections into Variant 2 instead. The architecture begins with a convolutional layer identical to that of Variant 2, featuring 256 convolutional kernels of size 3×3 with padding set to 1, followed by a ReLU activation function and BatchNorm2d. Subsequently, 7 residual blocks are employed, with the number of channels progressively increasing as follows: $256 \to 512 \to 512 \to 1024 \to 1024 \to 2048 \to 2048 \to 1024$. MaxPool2d layers are inserted after the 1st, 3rd, and 7th residual blocks to further reduce spatial dimensions. The fully connected layers remain consistent with those in Variant 2.

This modified architecture achieved a test accuracy of 92.04%. However, the learning curves (3) exhibited significant fluctuations, particularly in the test loss, which was highly unstable. This instability suggests that the model's performance is sensitive to the number of epochs. Additionally, the training time increased to 6 hours and 27 minutes, and the number of parameters reached 134.87M. Given these drawbacks, I concluded that this architecture was not suitable for further development. Instead, I continued to **build upon the structure of Variant 2**, which proved to be more effective and efficient.

The net is shown below:

```
net = nn.Sequential(
  nn.Conv2d(3, 256, 3, padding=1), nn.ReLU(inplace=True), nn.BatchNorm2d(256),
  ResBlock(256, 256),
  nn.MaxPool2d(2),

    ResBlock(256, 512),
    ResBlock(512, 512),
    nn.MaxPool2d(2),

    ResBlock(512, 1024),
    ResBlock(1024, 1024),

    ResBlock(1024, 2048),
    ResBlock(2048, 2048),

    ResBlock(2048, 1024),
    nn.MaxPool2d(2),

    nn.Flatten(),
    nn.Linear(1024 * 4 * 4, 2048), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(2048),
    nn.Linear(2048, 1024), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(1024),
    nn.Linear(1024, 512), nn.ReLU(inplace=True), nn.Dropout(0.3), nn.BatchNorm1d(512),
    nn.Linear(512, 10),
)
```

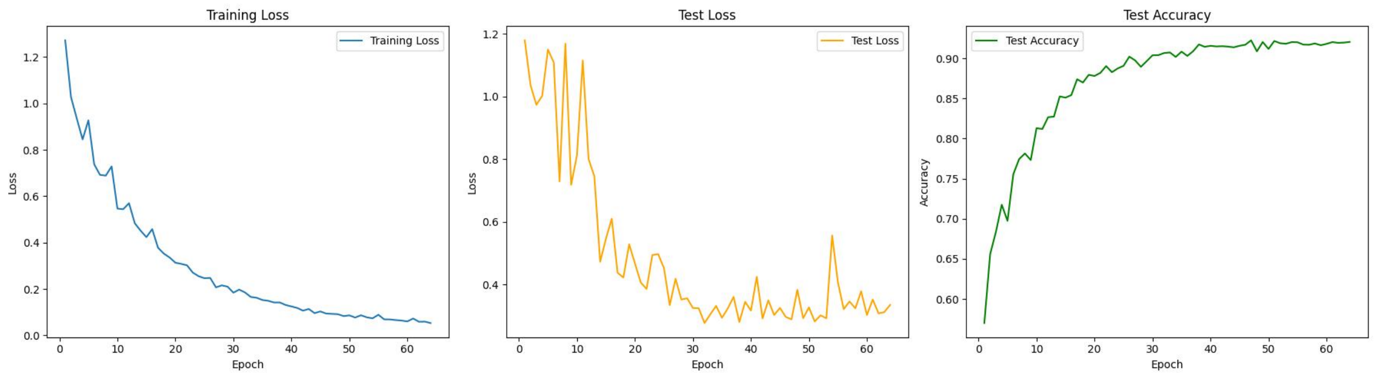

Figure 3: Variant 3: learning curve

2.3.4 Variant 4: Data augmentation

Building upon the wider and deeper CNN architecture of Variant 2, we further enhanced the model's generalization capability by applying more extensive data augmentation techniques to the images. Specifically, we introduced additional data augmentation methods such as `RandomCrop`, `ColorJitter`, and `RandomErasing` to the training set, while maintaining normalization as the sole preprocessing step for the test set. As a result, the model achieved a test accuracy of 92.04%, matching that of the ResNet architecture.

From the learning curves (4), it was observed that the training loss did not decrease rapidly; instead, it converged only after 50 epochs. In contrast, the test loss stabilized as early as after 20 epochs, and the accuracy curve exhibited a relatively smooth progression. This indicates that the model's generalization performance and stability were significantly improved through the use of data augmentation.

Therefore, compared with Variant 3, which employed residual connections, Variant 4 demonstrated that data augmentation is more effective in achieving better generalization and stability.

The net is shown below:

```
for data_type in ("train", "test"):
    is_train = data_type=="train"
    if is_train:
        transformation[data_type] = tv_transforms.Compose([
            tv_transforms.RandomCrop(32, padding=4, padding_mode='reflect'),
            tv_transforms.RandomHorizontalFlip(),
            tv_transforms.RandomRotation(degrees=15),
            tv_transforms.RandomAffine(degrees=0, translate=(0.1, 0.1), scale=(0.9, 1.1)),
            tv_transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
            tv_transforms.ToTensor(),
            tv_transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
            tv_transforms.RandomErasing(p=0.5, scale=(0.02, 0.33), ratio=(0.3, 3.3)),
        ])
    else:
        transformation[data_type] = tv_transforms.Compose([
            tv_transforms.ToTensor(),
            tv_transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
        ])
```

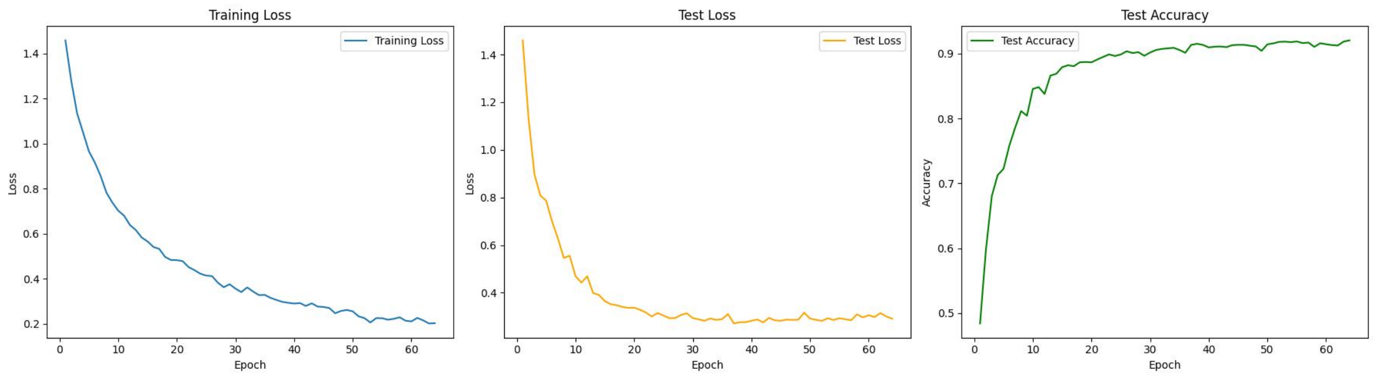

Figure 4: Variant 4: learning curve

2.3.5 Variant 5: Optimizer and learning rate scheduler

In the final stage of our experiments, we explored adjustments to the optimizer and learning rate schedule. Initially, we tested the AdamW optimizer on the baseline model with an increased weight decay (from 1e-6 to 0.01) and employed a

StepLR scheduler. However, this configuration resulted in a significant drop in accuracy to 83.82%. Subsequently, we replaced the StepLR with a constant learning rate followed by a cooldown phase, which further degraded the performance to an accuracy of 68.85%. These results indicated that the modifications to the optimizer and learning rate schedule were not conducive to improving model performance.

Given these outcomes, we reverted to the original Adam optimizer with the initial learning rate of 3e-4 and weight decay of 1e-6. Building on this foundation, we introduced a more sophisticated learning rate schedule comprising a warmup phase and a cosine annealing phase. Specifically, the learning rate linearly increased from the initial value of 3e-4 to the maximum learning rate of 3e-3 over the first 10% of epochs. This maximum learning rate was maintained for the subsequent 40% of epochs. Finally, during the last 50% of epochs, we employed a CosineAnnealingLR scheduler to gradually reduce the learning rate. The learning rate adjustment curve is illustrated in Figure 5.

With this optimized learning rate schedule, the model achieved a test accuracy of 93.04%. This result demonstrates that a carefully designed learning rate schedule, combined with the appropriate optimizer and regularization parameters, can significantly enhance the model's performance.

The learning rate scheduler settings are shown below:

```
def get_lr(epoch):
    if epoch < warmup_epochs:
        return initial_lr + (max_lr - initial_lr) * (epoch / warmup_epochs)
    return max_lr

scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
                T_max=num_epochs*0.5, eta_min=1e-6)

for epoch in range(num_epochs):
    if epoch <= num_epochs*0.5:
            current_lr = get_lr(epoch)
        else:
            scheduler.step()
            current_lr = optimizer.param_groups[0]['lr']
```

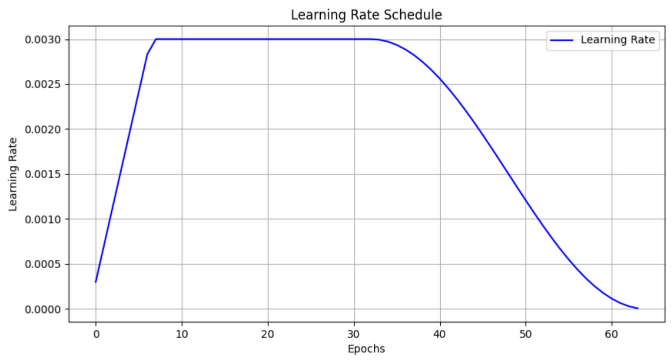

Figure 5: Variant 5: learning rate

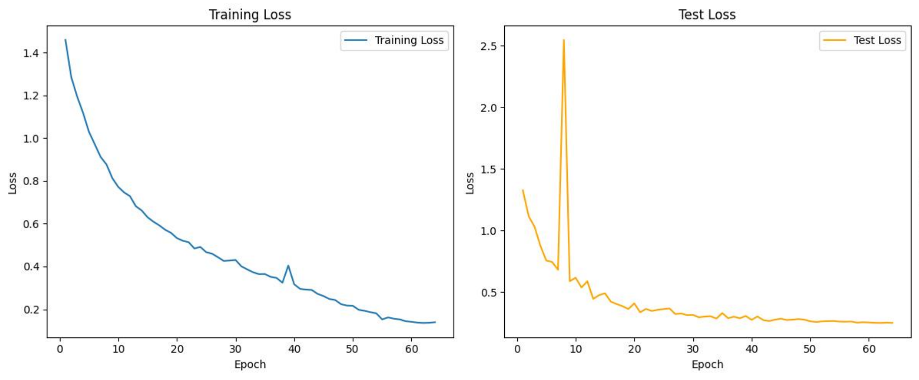

Figure 6: Variant 5: learning curve

2.4 Retrain on MNIST

Variant 5 was adopted as the final model and retrained on the MNIST dataset to evaluate its performance. Given that the MNIST dataset consists of single-channel grayscale images with a resolution of 28×28 pixels, several adjustments were made to the model settings:

- Data Augmentation: The size of the RandomCrop was changed from 32 to 28 to match the image dimensions of MNIST. The normalization parameters were set to empirically derived values of mean = 0.1307 and standard deviation = 0.3081. Additionally, the RandomHorizontalFlip transformation was removed, as flipping digits in MNIST can alter their semantic meaning (e.g., flipping a ”6” may resemble a ”9”).
- Network Architecture: The input channel number was set to 1, reflecting the grayscale nature of MNIST images. The model incorporated three MaxPool2d layers with a stride of 2, reducing the feature map size from 28 $\rightarrow$ 14 $\rightarrow$ 7 $\rightarrow$ 3. Consequently, the final feature map size was 3×3. Considering the simplicity of the MNIST dataset, the model was trained for only 32 epochs.

The final model achieved a test accuracy of 99.71%. However, the learning curves 7 were not entirely smooth. Although the loss values were low, they did not reach a plateau, indicating that further training could potentially improve the model’s performance.

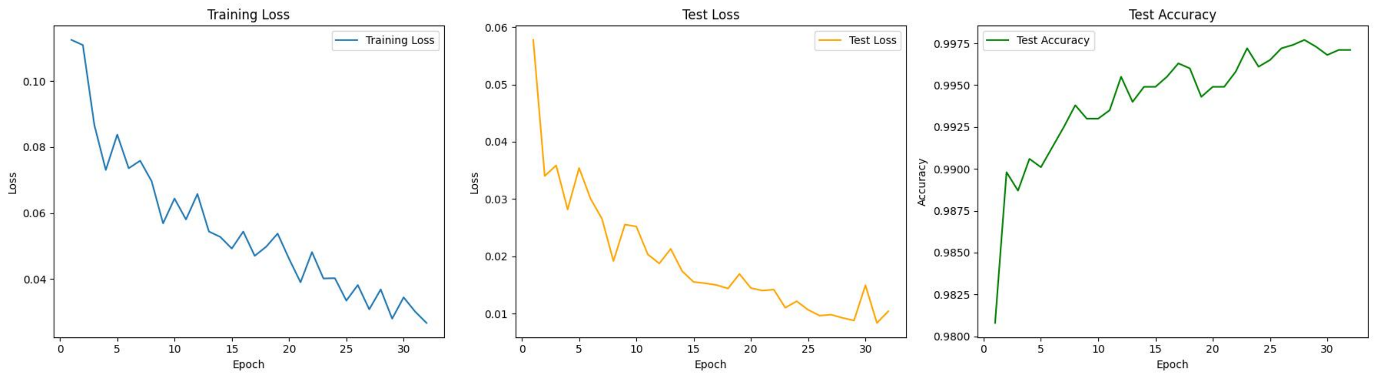

Figure 7: Variant 5 on MNIST: learning curve

2.5 Discussion and Conclusions

2.5.1 Analysis of Improvements

Normalization techniques, particularly Batch Normalization, have been found to significantly enhance model performance. Batch Normalization accelerates training convergence by stabilizing the input distribution for each layer, thereby reducing the internal covariate shift.

The depth and width of the network architecture play crucial roles in determining the model's capacity for feature extraction. Deeper networks generally offer stronger feature extraction capabilities, enabling the model to learn more complex and abstract representations of the input data. Similarly, wider networks, characterized by a larger number of channels, increase the diversity of features that can be learned, thereby potentially improving the model's performance. However, increasing the depth and width of the network also leads to a higher computational cost and a larger number of parameters. Therefore, it is essential to strike a balance between model capacity and computational efficiency to avoid overfitting and ensure practical usability.

Residual connections, while theoretically advantageous in facilitating the training of deeper networks, leads to a much higher computational cost in our experiments. Its great accuracy can be achieved by data augmentation and learning rate scheduler (see Variant 4& 5).

Data augmentation emerged as a critical factor in enhancing the model's generalization capability. By artificially expanding the training dataset through techniques such as random cropping, color jittering, and random erasing, data augmentation effectively reduces overfitting. This process introduces variability into the training data, enabling the model to learn more robust features that generalize better to unseen data. Consequently, data augmentation not only improves the model's performance on the test set but also contributes to a smoother learning curve during training.

Regarding training strategies, the learning rate schedule emerged as a crucial factor in achieving optimal performance. An adaptive learning rate schedule, such as the warmup followed by cosine annealing, proved to be highly effective in balancing convergence speed and stability.

2.5.2 Conclusions

Through systematic experimentation and optimization, I successfully achieved a classification accuracy of 93.04% on the CIFAR-10 dataset. The key improvements included the incorporation of Batch Normalization layers, which stabilized the training process and accelerated convergence; the deepening and widening of the network architecture, which enhanced feature extraction capabilities; the application of robust data augmentation techniques, which improved generalization; and the implementation of an adaptive learning rate schedule, which optimized training efficiency. The model's strong performance on the MNIST dataset further demonstrated the generalizability of its architecture.

3. Part B

3.1 Introduction

Part B[2] implemented the model from Part A using CuPy (code available in the attached `dl-hw1-B.ipynb`) and successfully ran it on the MNIST dataset. However, due to low computational efficiency, the number of Conv2d layers and epochs were reduced. As a result, training was affected by vanishing gradients, making it difficult to achieve the accuracy obtained in Part A.

[2]Codes and version logs: `https://www.kaggle.com/code/yixinzhou2002/dl-hw1-cupy-code`

3.2 Trial 1

Initially, the model's training was significantly slowed down by the MaxPool layer, which took 8-10 minutes per backward pass. Therefore, the MaxPool layers were removed. However, without downsampling, the model had a large number of parameters. Consequently, only 4 convolutional layers and 3 linear layers were used, with reduced channel numbers (64 $\rightarrow$ 128 $\rightarrow$ 128 $\rightarrow$ 64). The model was trained for 32 epochs. After 5 hours and 18 minutes of training, the training loss became stuck at 2.303 (8) from the second epoch onwards, resulting in a test accuracy of only 9.80%.

The network architecture is as follows:

```
def build_model():
    net = Sequential(
        Conv2d(1, 64, 3, padding=1), ReLU(), BatchNorm2d(64),
        Conv2d(64, 128, 3, padding=1), ReLU(), BatchNorm2d(128),
        Conv2d(128, 128, 3, padding=1), ReLU(), BatchNorm2d(128),
        Conv2d(128, 64, 3, padding=1), ReLU(), BatchNorm2d(64),

        Flatten(),

        Linear(64 * 28 * 28, 128), ReLU(), Dropout(0.3), BatchNorm1d(128),
        Linear(128, 64), ReLU(), Dropout(0.3), BatchNorm1d(64),
        Linear(64, 10),
    )

    return net
```

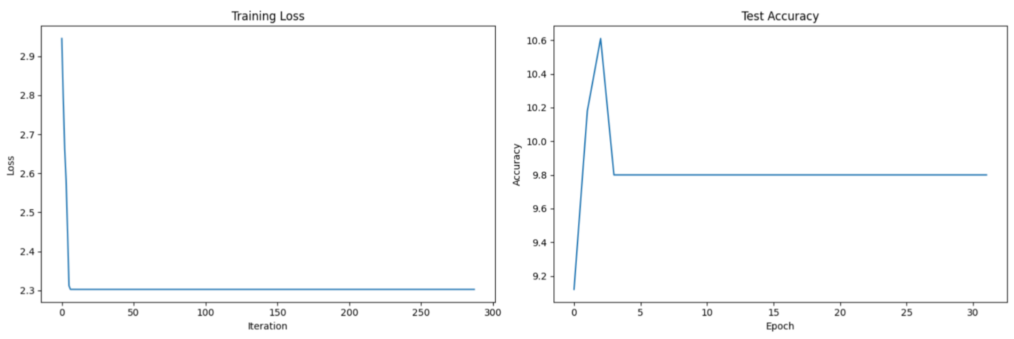

Figure 8: Trial 1 on MNIST: learning curve

3.3 Trial 2

Then, I improved the computation method of MaxPool2d.backward() by replacing the for loop with a list comprehension, which enabled the smooth use of structures with MaxPool. Consequently, I restored the channel width and the number of MaxPool layers to be the same as in Variant 5, but reduced one convolutional layer. Moreover, I increased the maximum learning rate by a factor of 10, hoping to mitigate the vanishing gradient problem. Due to the excessive runtime, I prematurely terminated the execution and exported the model file saved at the end of the eight epoch to plot the learning curve.[3] Unfortunately, after training for eight epochs, the model's training process still got stuck at a train loss of 2.303 (9), with the accuracy remaining at 9.80%. With a parameter count of 127 million, it took 70 minutes to run a single epoch.

[3]Code and dataset: https://www.kaggle.com/code/yixinzhou2002/cupy-results

The network architecture is as follows:

```
def build_model():
    net = Sequential(
        Conv2d(1, 256, 3, padding=1), ReLU(), MaxPool2d(2),BatchNorm2d(256),
        Conv2d(256, 512, 3, padding=1), ReLU(),MaxPool2d(2), BatchNorm2d(512),
        Conv2d(512, 1024, 3, padding=1), ReLU(), BatchNorm2d(1024),
        Conv2d(1024, 2048, 3, padding=1), ReLU(), BatchNorm2d(2048),
        Conv2d(2048, 1024, 3, padding=1), ReLU(), MaxPool2d(2),BatchNorm2d(1024),

        Flatten(),

        Linear(1024 * 3 * 3, 2048), ReLU(), Dropout(0.3), BatchNorm1d(2048),
        Linear(2048, 1024), ReLU(), Dropout(0.3), BatchNorm1d(1024),
        Linear(1024, 512), ReLU(), Dropout(0.3), BatchNorm1d(512),
        Linear(512, 10),
    )
return net
```

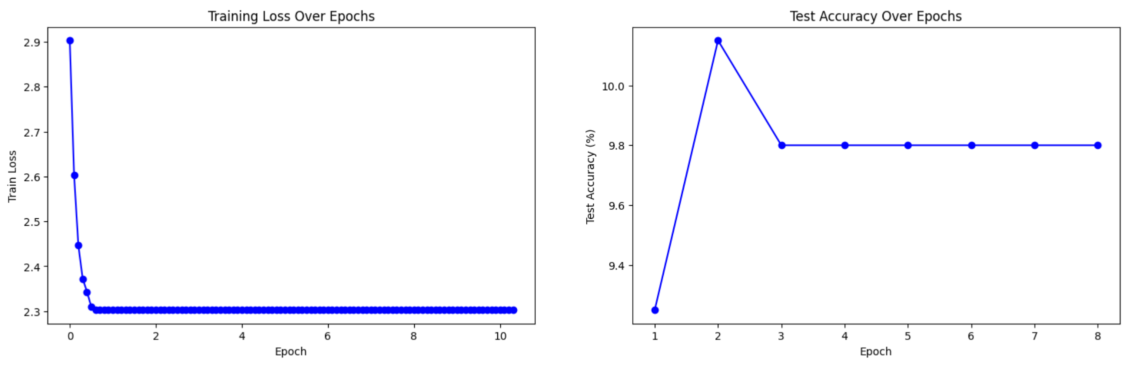

Figure 9: Trial 2 on MNIST: learning curve

3.4 Reflection

In Part B, I implemented the main functions of a deep learning network using Cupy. This process was extremely challenging but also highly rewarding. By repeatedly understanding the computational processes of each network structure and coding the forward and backward propagation steps, I also implemented other necessary functions such as optimizers, loss functions, and learning rate adjustments. I utilized object-oriented programming to implement classes and functions, involving techniques such as inheritance and composition. Additionally, I added log points to identify the reasons for long computation times and replaced for loops with list comprehensions to accelerate the backward propagation of MaxPool. Ultimately, my Cupy version was successfully executed. However, due to time constraints, I was unable to fully run the experiments in Part B and also encountered the problem of vanishing gradients.