DDA4220 Homework 1 Report

121090057

March 16, 2025

Part A. Train on Cifar-10: Results and Analysis

1. Baseline Performance

The default model involves baseline CNN and Adam optimizer. Reproduced accuracy obtained from the provide code is 87.30%.

2. Optimization Strategies and Accuracy Improvement

The following strategies were explored to improve accuracy:

a) Regularization methods (batch normalization)

b) Network depth and width modifications

c) Optimizer selection

d) Data augmentation

2.1 Results

Model	Optimizer	Data Augmentation	Accuracy
CNN (Baseline)	Adam	Original Setting	87.30%
CNN + BatchNorm	Adam	Original Setting	89.68%
Modified CNN	Adam	Original Setting	91.74%
Modified CNN	AdamW	Original Setting	91.58%
Modified CNN	SGD	Original Setting	91.76%
Modified CNN	Adam	Additional Augmentation	91.71%
Modified CNN	AdamW	Additional Augmentation	91.76%
Modified CNN	SGD	Additional Augmentation	90.89%
Complex CNN	Adam	Original Setting	93.27%

Table 1: Performance comparison of different CNN models, optimizers, and data augmentation strategies.

Note: Net architectures, parameter settings in different optimizers, different data augmentation strategies, and figures for training/testing results (less important) are illustrated in Appendix. For complex CNN, even though it achieves higher accuracy, it is excluded from Part B due to excessive runtime.

Figure 1 indicates that the modified CNN with SGD optimizer and original data augmentation yields the best accuracy.

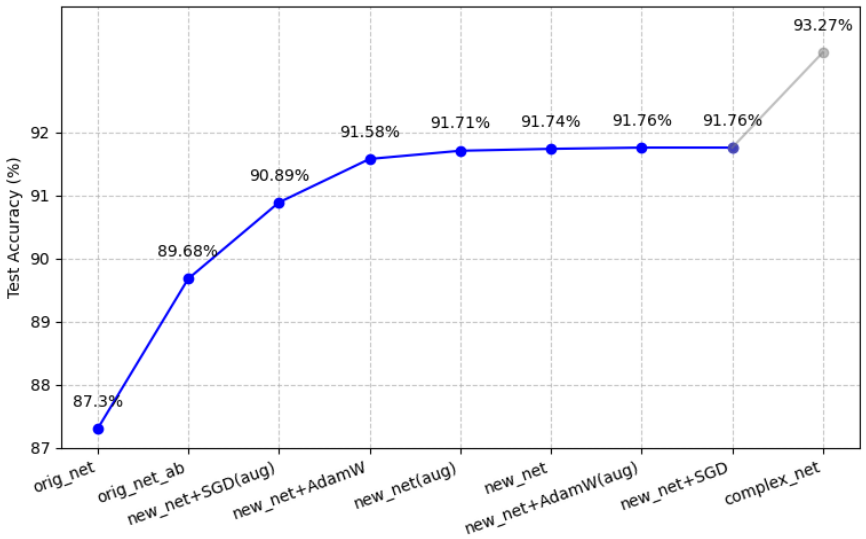

Figure 1: Test Accuracy

2.2 Analysis (what I've learned through attempts)

Consider different CNN structure:

a) The addition of Batch Normalization (`orig_net_ab`) improved test accuracy (Figure 2), indicating its effectiveness in stabilizing training and accelerating convergence, which is represented in Figure 3.

b) The width and the depth of CNN should be carefully constructed. For example, reducing a linear layer may increase accuracy (`new_net` vs. `orig_net`), while wider and deeper CNN may also boost accuracy (`complex_net` vs. `orig_net`).

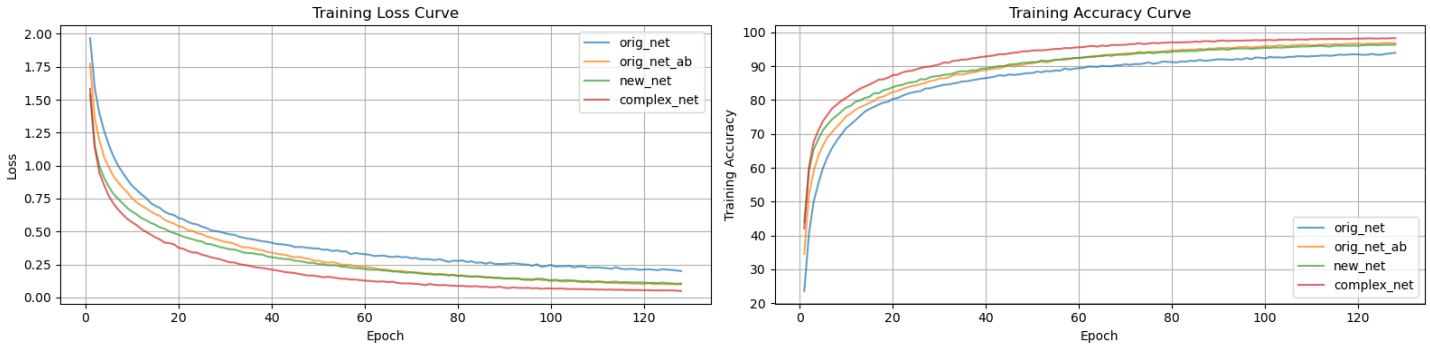

Figure 2: Training plot for Different CNN Architecture

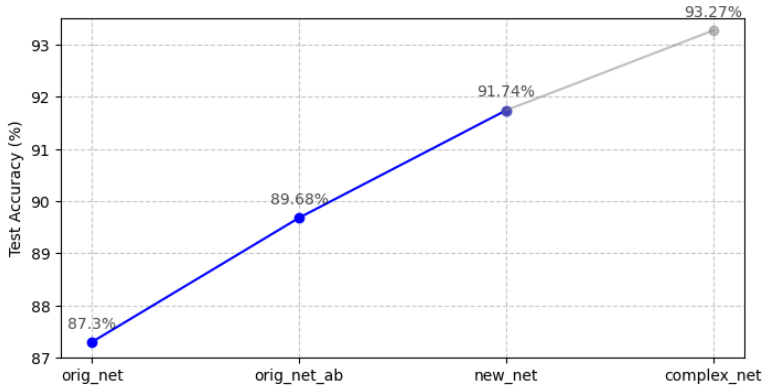

Figure 3: Test accuracy plot for Different CNN Architecture

Consider different optimizers and data augmentation methods:

a) In this training scheme, different optimizers have little effect on test accuracy (Figure 5), and CNN with SGD with momentum yields the best test accuracy. Figure 4 shows AdamW has faster convergence but may fluctuate, while Adam and SGD are steadier.

b) Figure 5 also shows data augmentation may not always help boost accuracy. Too many augmentations may not help image classification.

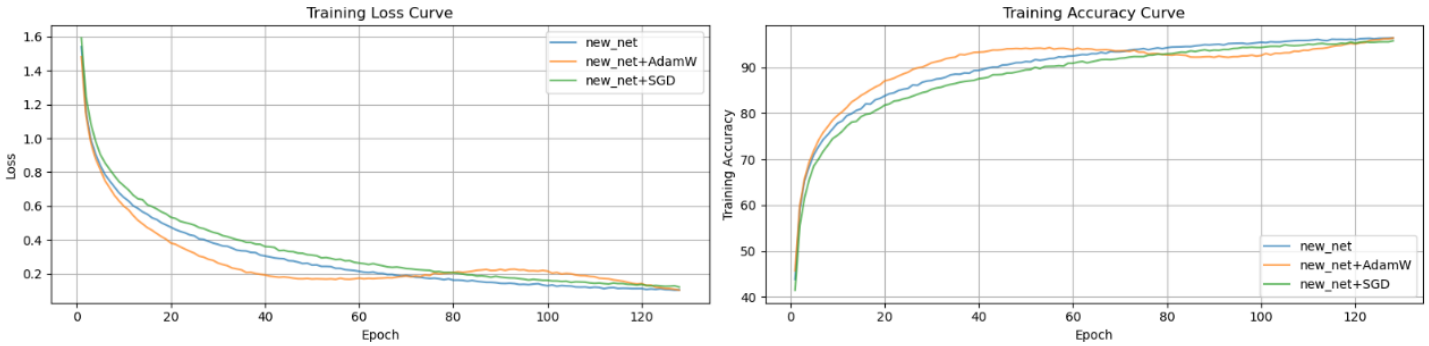

Figure 4: Training plot for Different Optimizers & Data Augmentation

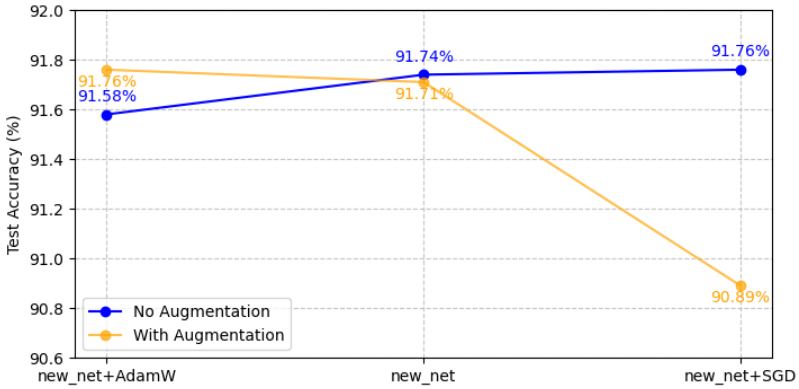

Figure 5: Test accuracy plot for Different Optimizers & Data Augmentation

3. Retrain on MNIST

Using a similar network architecture, SGD optimizer, and no additional data augmentation as the best-performing model in Section 2, but trained for only 2 epochs, the classification accuracy reaches **99.22%** on the MNIST dataset.

Part B. Build a Neural Network

1. For the implementation of forward and backward functions, I strictly followed the mathematical formulas provided in PyTorch website. See implementation details in the code. In addition to the requirements, I implemented padding in the `Conv2d`, `BatchNorm1d`, `Flatten`, and `MaxPool2d` to reproduce the previous network in my setting.
2. The reproduced accuracy: **96.87%**.
3. Comparison plot (Figure 6) of training losses and test accuracy between the reproduced version and Torch Comparison figure:

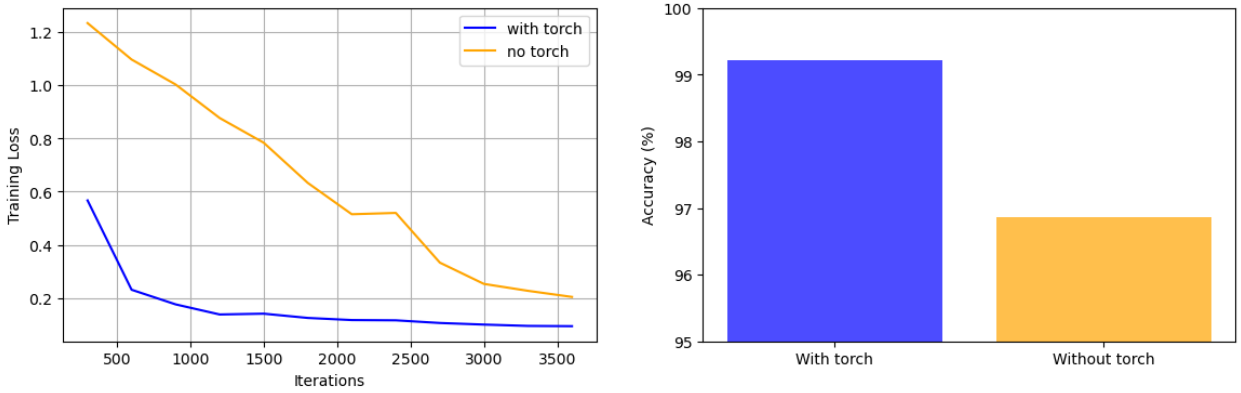

Figure 6: Comparison figure

4. **Analysis:** By comparing losses, using Torch resulted in lower losses over iterations, and Torch achieves higher accuracy. Torch requires significantly less time to train than the reproduced version. This may be because when implementing modules like Conv2d, I iteratively calculate the convolution window output, which results in heavy and long-time training process.

Appendix

Parameter	Adam	AdamW	SGD
Learning Rate (`lr`)	`3e-4`	`1e-3`	`1e-2`
Weight Decay	`1e-6`	`1e-3`	`1e-5`
Momentum	-	-	`0.9`

Table 2: Parameters in optimizers

Original Setting	Additional Augmentation
`RandomRotation(degrees=15)`	`RandomRotation(degrees=15)`
`RandomHorizontalFlip()`	`RandomHorizontalFlip()`
`RandomAffine(degrees=0, translate=(0.1, 0.1))`	`RandomAffine(degrees=10, translate=(0.1, 0.1))`
-	`RandomCrop(32, padding=2)`
-	`ColorJitter(brightness=0.2, contrast=0.2, hue=0.1)`
`ToTensor()`	`ToTensor()`
`Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])`	`Normalize(mean=[0.4914, 0.4822, 0.4465], std=[0.2023, 0.1994, 0.2010])`

Table 3: Data augmentation setting

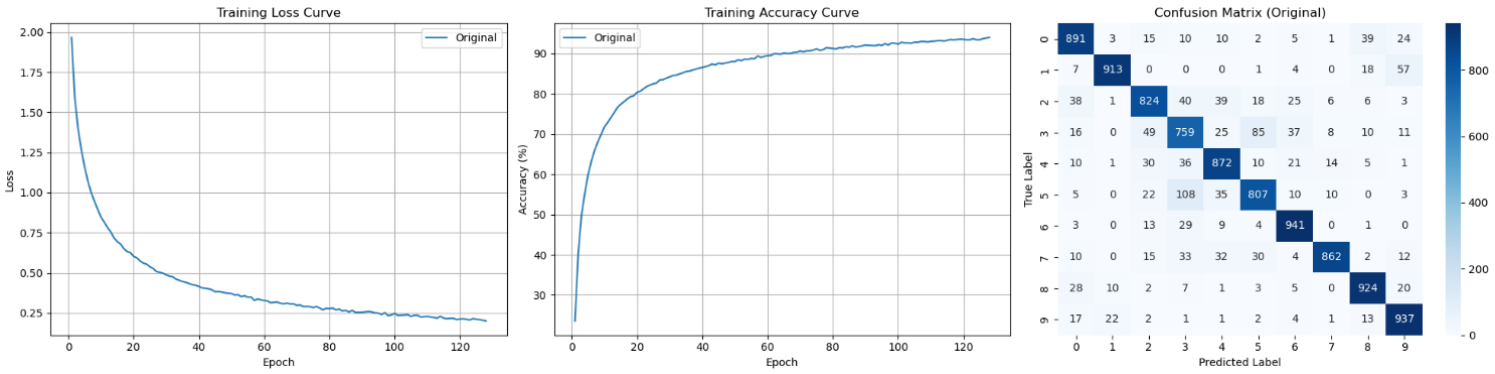

Figure 7: Training plot for baseline CNN

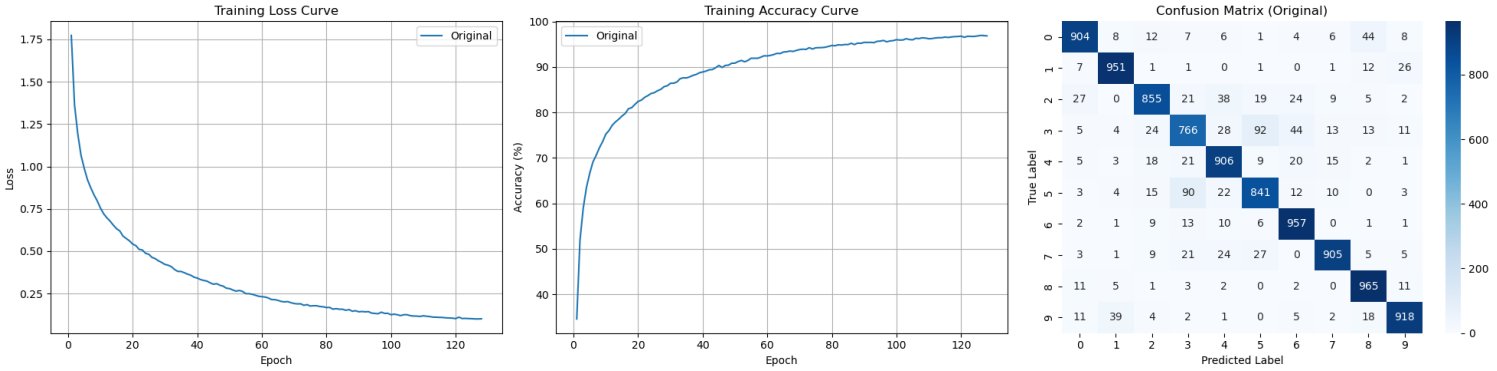

Figure 8: Training plot for baseline CNN with batchnorm layers

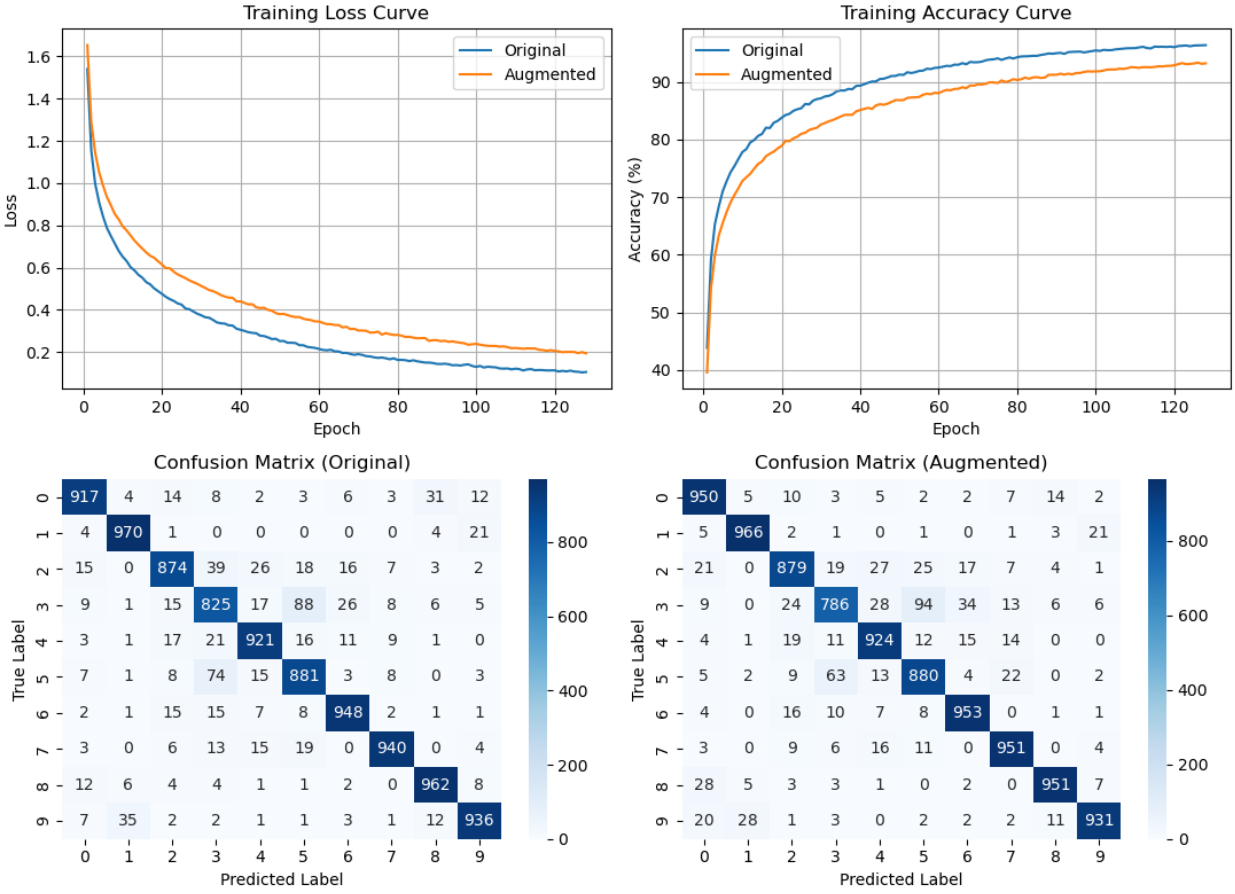

Figure 9: Training plot for modified net with Adam

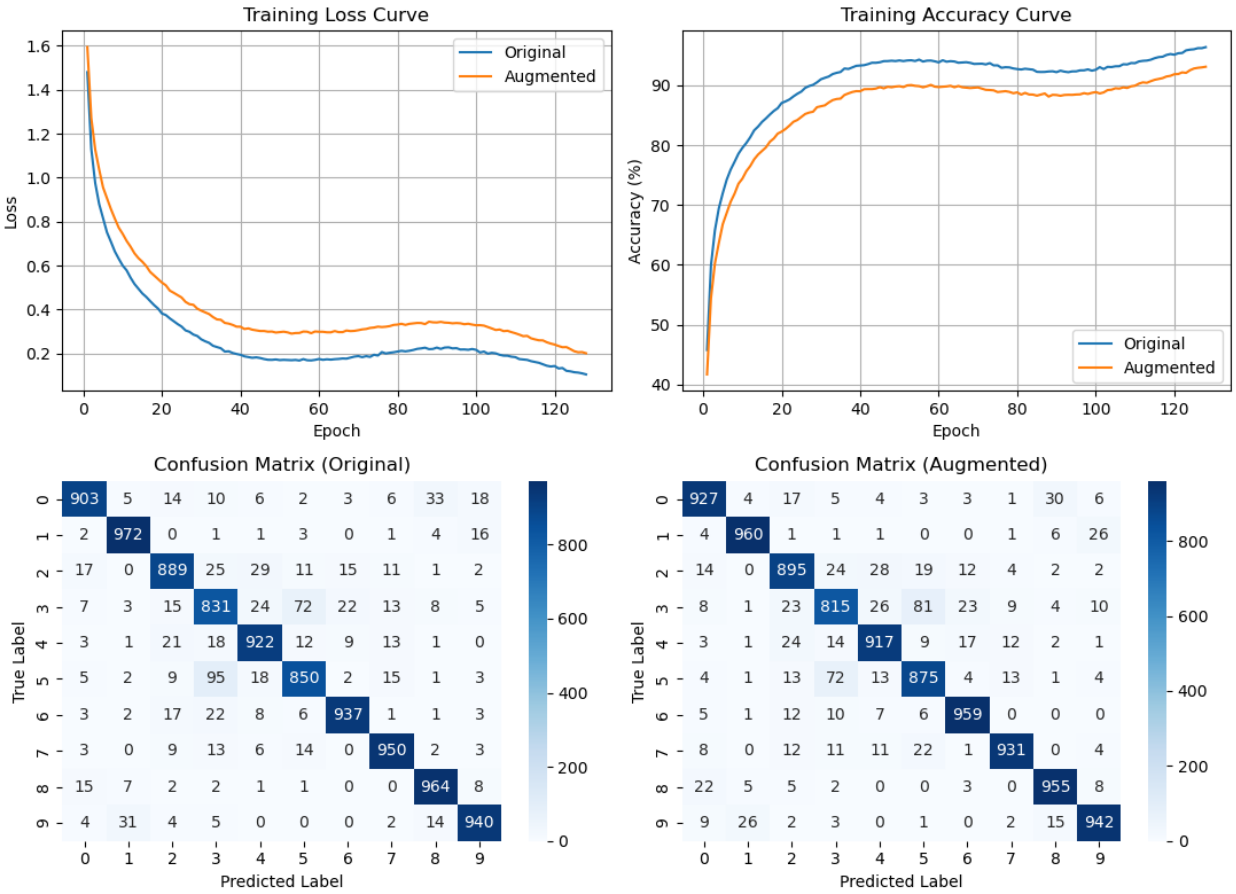

Figure 10: Training plot for modified net with AdamW

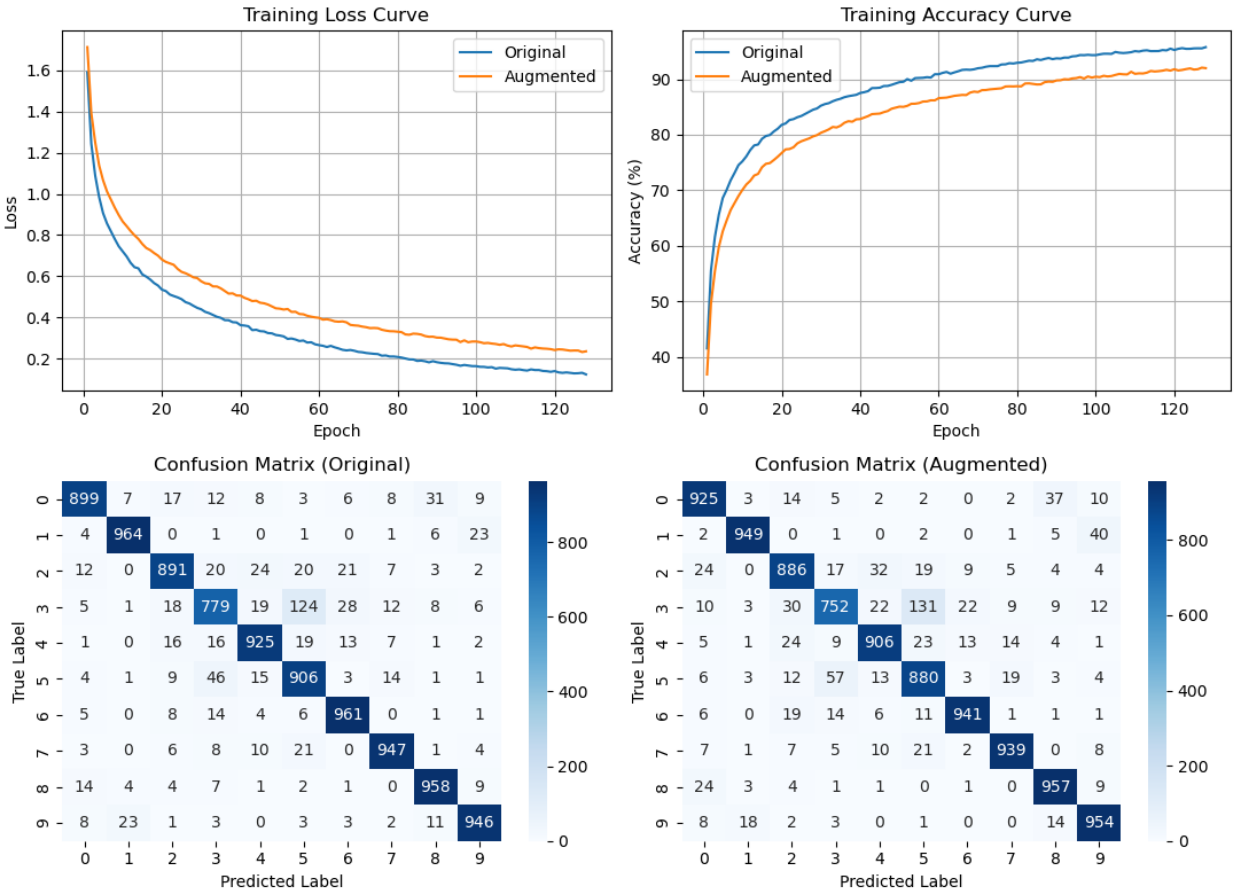

Figure 11: Training plot for modified net with SGD

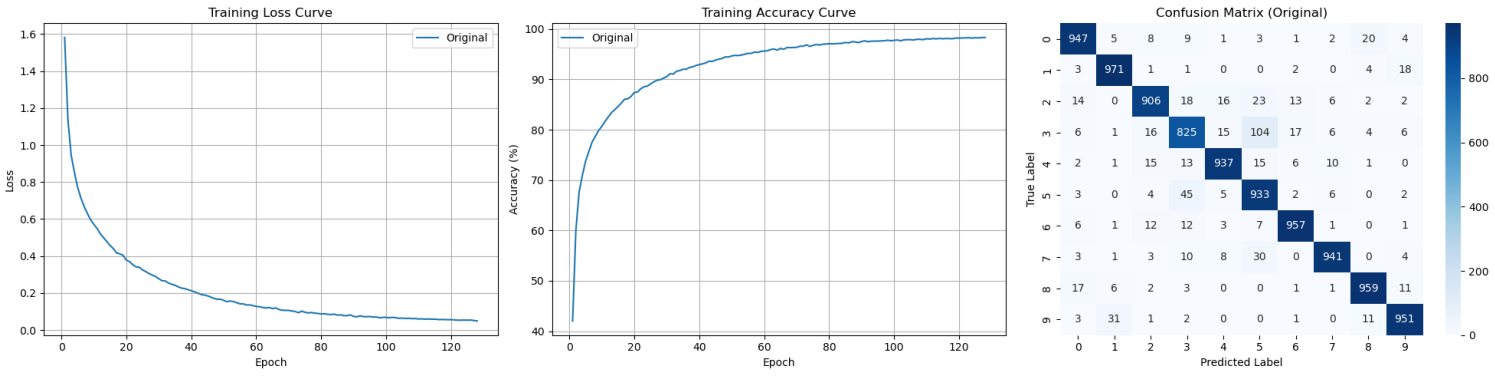

Figure 12: Training plot for complex CNN

```python
## orignal net
origin_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),
)

## orginal net + regularization (batchnorm layers)
ab_net = nn.Sequential(
    nn.Conv2d(3, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(256, 512, 3, padding=1), nn.BatchNorm2d(512), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.BatchNorm2d(512), nn.ReLU(inplace=True),
    nn.Conv2d(512, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Flatten(),
    nn.Linear(256 * 4 * 4, 512), nn.BatchNorm1d(512), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(512, 256), nn.BatchNorm1d(256), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(256, 128), nn.BatchNorm1d(128), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(128, 10),
)


## modified net
new_net = nn.Sequential(
    nn.Conv2d(3, 128, 3, padding=1), nn.BatchNorm2d(128), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(128, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(inplace=True),
    nn.Conv2d(256, 256, 3, padding=1), nn.BatchNorm2d(256), nn.ReLU(inplace=True), nn.MaxPool2d(2), nn.Dropout(0.3),
    nn.Conv2d(256, 512, 3, padding=1), nn.BatchNorm2d(512), nn.ReLU(inplace=True),
    nn.Conv2d(512, 512, 3, padding=1), nn.BatchNorm2d(512), nn.ReLU(inplace=True),nn.MaxPool2d(2), nn.Dropout(0.4),
    nn.Flatten(),
    nn.Linear(512 * 4 * 4, 512), nn.BatchNorm1d(512), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(512, 256), nn.BatchNorm1d(256), nn.ReLU(inplace=True), nn.Dropout(0.5),
    nn.Linear(256, 10),
)
```

Figure 13: CNN architecture