Deep Learning Spring 2025: CIFAR-10 Classification Project Report

1. Project Overview

This project implements a modified Residual Network (ResNet) architecture for CIFAR-10 image classification with fewer than 5 million parameters, as required by the Deep Learning Spring 2025 course competition.

1.1 Project Objectives

- Design a modified ResNet architecture with <5M parameters
- Train the model on CIFAR-10 dataset
- Achieve a test accuracy of at least 80%, targeting 90%
- Submit complete project report and code

1.2 CIFAR-10 Dataset

The CIFAR-10 dataset contains 60,000 32×32 color images across 10 classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck), divided into 50,000 training images and 10,000 test images.

2. Data Preparation and Augmentation

We divided the original CIFAR-10 training set using an 80:20 ratio:

- Training set: 40,000 images
- Validation set: 10,000 images

Data augmentation techniques implemented:

```
transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    transforms.RandomErasing(p=0.5, scale=(0.02, 0.2))
])
```

3. Model Architecture

3.1 Modified ResNet Architecture

Our ResNet variant consists of:

- Initial convolutional layer: 48 channels, 3×3 kernel
- Four stages of residual blocks with channel counts: 48 → 196 → 192 → 384
- Residual blocks: [2, 2, 2, 1], totaling 7 residual blocks

- 50% Dropout after global average pooling
- Final fully connected layer to 10 classes

Basic residual block structure:

```
class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3,
stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.relu2 = nn.ReLU(inplace=True)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=stride, bias=False),
                nn.BatchNorm2d(out_channels)
            )
```

3.2 Model Parameter Statistics

```
Total params: 4,662,098
Trainable params: 4,662,098
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 14.43
Params size (MB): 17.78
Estimated Total Size (MB): 32.22
```

Total model parameters: 4,662,098, below the 5 million parameter limit.

4. Training Strategy

4.1 Optimizer and Loss Function

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, weight_decay=5e-
4)
criterion = nn.CrossEntropyLoss()
```

4.2 Learning Rate Scheduling

```
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

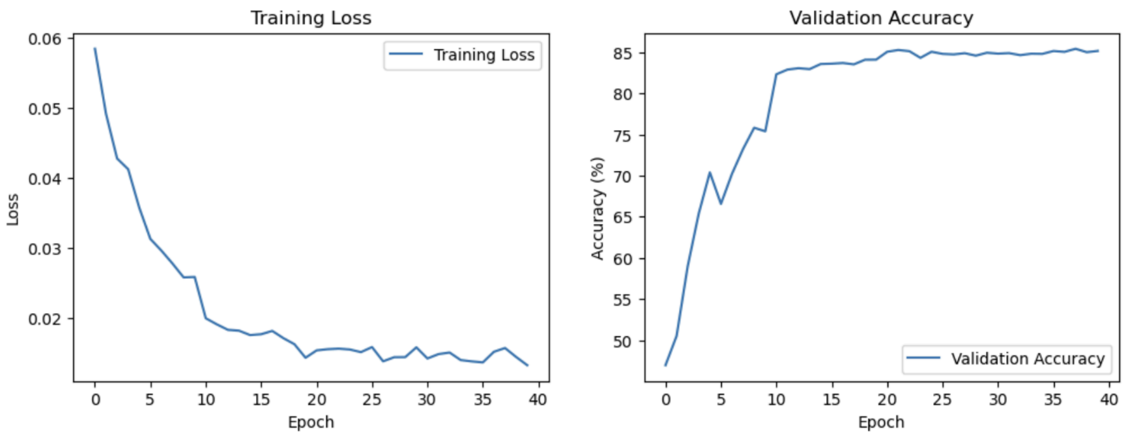
4.3 Training Process

Our training process included:

1. Full dataset iteration each epoch
2. Loss reporting every 100 batches
3. Validation after each epoch
4. Learning rate adjustment via scheduler
5. Saving the final model weights

5. Experimental Results

5.1 Training and Validation Curves



Training observations:

- Training loss decreased rapidly in initial epochs
- Validation accuracy stabilized around 85%
- Final validation accuracy: 85.16%

Final training outputs:

```
Epoch [40/40], Step [100/625], Loss: 0.362
Epoch [40/40], Step [200/625], Loss: 0.365
Epoch [40/40], Step [300/625], Loss: 0.364
Epoch [40/40], Step [400/625], Loss: 0.377
Epoch [40/40], Step [500/625], Loss: 0.379
Epoch [40/40], Step [600/625], Loss: 0.376
Validation Accuracy: 85.16%
```

6. Improvement Opportunities

Potential enhancements:

1. **Architecture Optimization**:

    - Test bottleneck residual blocks
    - Adjust channel distribution
    - Explore deeper network structures
2. **Training Strategies**:
    - Try Adam/AdamW optimizers
    - Implement cosine annealing schedules
    - Use test-time augmentation (TTA)

7. Conclusion

We successfully implemented a modified ResNet architecture with 4.66 million parameters, below the 5 million limit. The model achieved a validation accuracy of 85.16%, exceeding the 80% baseline target. Our approach demonstrates the effectiveness of residual networks even under parameter constraints, and suggests several promising directions for further improvement.