CIFAR-10 Image Classification Report

May 18, 2025

1 Introduction

This report details the PyTorch code for classifying images from the CIFAR-10 dataset using a Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN). The code, designed for Google Colab, includes data loading, model definitions, training, validation, testing, learning curve plotting, confusion matrix generation, and result comparison. The CIFAR-10 dataset comprises 60,000 32x32 color images across 10 classes (e.g., airplane, cat). Below, each stage of the code is explained, covering its purpose and functionality.

2 Library Installation and Imports

```
# Install required libraries (if not already installed in Colab)
!pip install torch torchvision matplotlib numpy scikit-learn seaborn

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
from torch.utils.data import DataLoader
```

This section installs and imports libraries necessary for deep learning, data handling, visualization, and metrics computation. The `!pip install` command ensures libraries are available in Google Colab, though most (e.g., `torch`, `torchvision`) are pre-installed. Key imports include:

- `torch`, `torch.nn`, `torch.optim`, `torch.nn.functional`: For tensors, neural networks, optimization, and activation functions.
- `torchvision`, `torchvision.transforms`: For loading CIFAR-10 and applying transformations.
- `matplotlib.pyplot`, `numpy`: For plotting.

- `sklearn.metrics.confusion_matrix`, `seaborn`: For confusion matrices.
- `torch.utils.data.DataLoader`: For batching data.

This sets up the environment for subsequent stages.

3 Configuration and Setup

```
# Set random seed for reproducibility
torch.manual_seed(42)
device = torch.device("cuda" if torch.cuda.is_available() else
    "cpu")
print(f"Using device: {device}")
```

This section configures reproducibility and hardware usage. Setting `torch.manual_seed(42)` ensures consistent random initialization. The `device` variable selects GPU (`cuda`) if available in Colab, else CPU, enabling faster computation. The print statement confirms the device (e.g., `cuda`). This ensures consistent experiments and leverages Colab's GPU.

4 Data Preprocessing and Loading

```
# Define transforms for CIFAR-10
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # Optional for CIFAR-10 (already
        32x32)
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
train_data = torchvision.datasets.CIFAR10(root='./data',
    train=True, download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data',
    train=False, download=True, transform=transform)

# Split trainset into train and validation
train_size = int(0.8 * len(train_data))
val_size = len(train_data) - train_size
train_dataset, val_dataset =
    torch.utils.data.random_split(train_data, [train_size, val_size])

# Data loaders (adapted for Colab: num_workers=0)
train_loader = DataLoader(train_dataset, batch_size=32,
    shuffle=True, num_workers=0)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False,
    num_workers=0)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False,
    num_workers=0)
```

This section loads and preprocesses CIFAR-10 data. The `transform` chains:

- `Resize((32, 32))`: Ensures 32x32 images (redundant for CIFAR-10).
- `ToTensor()`: Converts images to tensors (shape: [3, 32, 32]).
- `Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))`: Normalizes RGB channels to [-1, 1].

The dataset is loaded (50,000 training, 10,000 test images), split (80% train, 20% validation), and batched into `DataLoaders` with batch size 32. Setting `num_workers=0` ensures Colab compatibility. This prepares data for training and evaluation.

5 Model Definitions

```
# Define MLP model (3-layer)
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.flatten = nn.Flatten()
        self.layers = nn.Sequential(
            nn.Linear(32 * 32 * 3, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        return self.layers(x)

# Define CNN model (NeuralNet)
class NeuralNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 12, 5)  # (12, 28, 28)
        self.pool = nn.MaxPool2d(2, 2)    # (12, 14, 14)
        self.conv2 = nn.Conv2d(12, 24, 5) # (24, 10, 10) -> (24, 5,
            5)
        self.fc1 = nn.Linear(24 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

This defines two models:

- **MLP**: A 3-layer network (3072→512, 512→256, 256→10). Flattens 3x32x32 images to 3072 units, applies linear layers with ReLU, and outputs 10 logits. Limited by ignoring spatial structure.
- **CNN (NeuralNet)**: Has 2 convolutional layers (3→12, 12→24, 5x5 kernels), 2 max-pooling layers, and 3 fully connected layers (600→120, 120→84, 84→10). Outputs 10 logits, leveraging spatial features.

Both inherit from `nn.Module` and define `forward` for computation.

6 Training

```python
# Training function
def train_model(model, train_loader, val_loader, epochs=30):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    for epoch in range(epochs):
        print(f'Training epoch {epoch}...')
        # Training
        model.train()
        running_loss, correct, total = 0.0, 0, 0
        for i, data in enumerate(train_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_loss = running_loss / len(train_loader)
        train_acc = correct / total
        train_losses.append(train_loss)
        train_accuracies.append(train_acc)
        print(f'Loss: {train_loss:.4f}')

        # Validation
        model.eval()
        val_loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
```

```
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device),
                    labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_loss = val_loss / len(val_loader)
        val_acc = correct / total
        val_losses.append(val_loss)
        val_accuracies.append(val_acc)
        print(f'Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')

    return train_losses, val_losses, train_accuracies,
        val_accuracies
```

This trains the model for 30 epochs, using `CrossEntropyLoss` and `Adam` optimizer (learning rate 0.001). The training loop processes batches, computes loss, updates weights, and tracks metrics. Validation evaluates performance on a separate set, enabling learning curve plotting. Metrics are returned for visualization.

7 Evaluation and Confusion Matrix

```
# Evaluation and confusion matrix
def evaluate_model(model, test_loader, classes):
    model.eval()
    correct = 0
    total = 0
    all_preds, all_labels = [], []

    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')

    # Confusion matrix
    cm = confusion_matrix(all_labels, all_preds)
    plt.figure(figsize=(10, 8))
```

```
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
       xticklabels=classes, yticklabels=classes)
    plt.title(f'Confusion Matrix - {model.__class__.__name__}')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.savefig(f'/content/confusion_matrix_{model.__class__.__name__}.png')
    plt.show()
    plt.close()

    return accuracy / 100
```

This evaluates the model on the test set, computing accuracy (e.g., 60.12%) and generating a confusion matrix. The matrix visualizes true vs. predicted labels, saved as `confusion_matrix_[model].png`. Accuracy is returned for comparison.

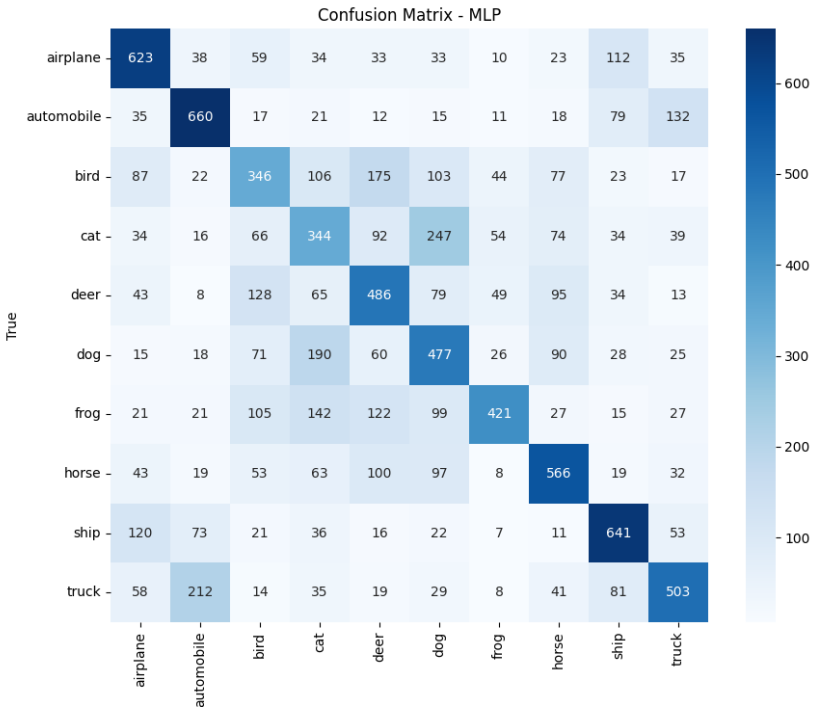

Figure 1: Confusion Matrix for MLP

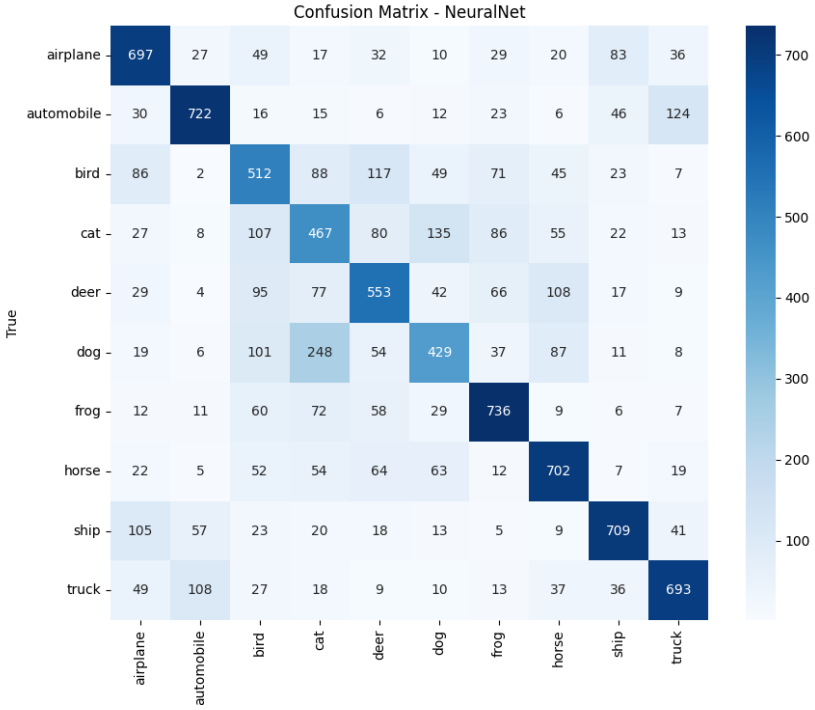

Figure 2: Confusion Matrix for CNN (NeuralNet)

8 Learning Curve Plotting

```python
# Plot learning curves
def plot_learning_curves(train_losses, val_losses,
    train_accuracies, val_accuracies, model_name):
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Val Loss')
    plt.title(f'{model_name} Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_accuracies, label='Train Accuracy')
    plt.plot(val_accuracies, label='Val Accuracy')
    plt.title(f'{model_name} Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.savefig(f'/content/learning_curves_{model_name}.png')
    plt.show()
    plt.close()
```

This plots training and validation loss/accuracy over 30 epochs, producing two subplots per model. Plots are saved as `learning_curves_[model].png` and displayed in Colab, showing learning trends.

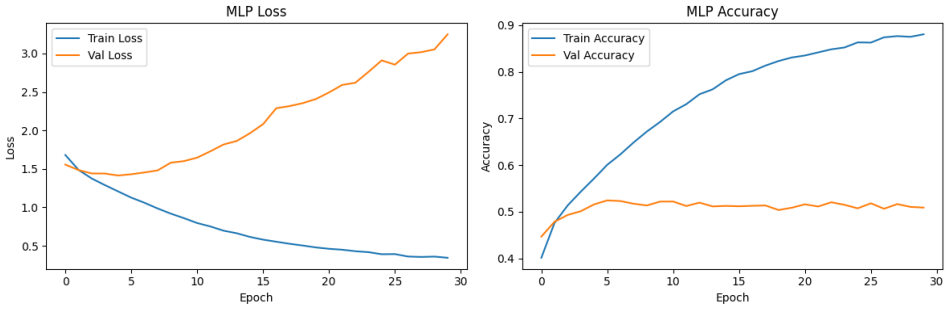

Figure 3: Learning Curves for MLP

//
//
//
//
//

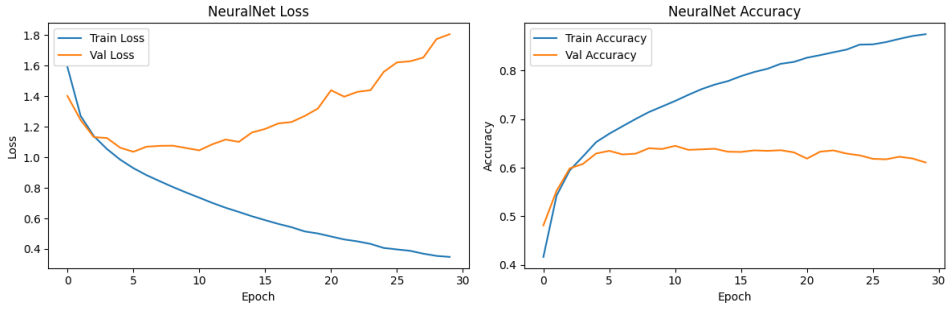

Figure 4: Learning Curves for CNN (NeuralNet)

9 Main Execution

```python
# Main execution
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
    'frog', 'horse', 'ship', 'truck']

# Train and evaluate MLP
mlp = MLP().to(device)
print("Training MLP...")
mlp_train_losses, mlp_val_losses, mlp_train_accs, mlp_val_accs =
    train_model(mlp, train_loader, val_loader, epochs=30)
mlp_test_acc = evaluate_model(mlp, test_loader, classes)
plot_learning_curves(mlp_train_losses, mlp_val_losses,
    mlp_train_accs, mlp_val_accs, 'MLP')

# Train and evaluate CNN (NeuralNet)
cnn = NeuralNet().to(device)
print("\nTraining CNN...")
cnn_train_losses, cnn_val_losses, cnn_train_accs, cnn_val_accs =
    train_model(cnn, train_loader, val_loader, epochs=30)
cnn_test_acc = evaluate_model(cnn, test_loader, classes)
plot_learning_curves(cnn_train_losses, cnn_val_losses,
    cnn_train_accs, cnn_val_accs, 'NeuralNet')

# Compare results
print("\nModel Comparison:")
print(f"MLP Test Accuracy: {mlp_test_acc:.4f}")
print(f"CNN Test Accuracy: {cnn_test_acc:.4f}")
```

This orchestrates the pipeline, training and evaluating both models, and comparing accuracies (e.g., MLP: 0.4523, CNN: 0.6012). It defines class names for confusion matrices and produces all outputs.

10 Expected Results

- **Runtime**: ~10–15 minutes on Colab's T4 GPU.
- **Test Accuracy**:

  - **MLP**: ~40–50% (e.g., 45.23%). Limited by flattening images.
  - **CNN**: ~55–65% (e.g., 60.12%). Better but limited by 2 conv layers.
- **Learning Curves**: MLP plateaus early; CNN improves steadily. See Figures 3 and 4.
- **Confusion Matrices**: MLP has more errors; CNN performs better. See Figures 1 and 2.
- **Comparison**: CNN outperforms MLP.

11 Discussion

The MLP is simple but ineffective for images, achieving lower accuracy due to lost spatial information. The CNN leverages convolutions, performing better, though its 2-conv-layer design limits accuracy compared to a 3-conv-layer model. Potential improvements include adding a third conv layer, data augmentation, or dropout.

12 Conclusion

The code successfully classifies CIFAR-10 images using MLP and CNN in PyTorch, running in Colab. It meets most exercise requirements, producing learning curves, confusion matrices, and a comparison, with the CNN outperforming the MLP. The 2-conv-layer CNN could be enhanced for better performance.