POSTS AND TELECOMMUNICATIONS
INSTITUTE OF TECHNOLOGY

REPORT

Course: Programming with Python

Student: Nguyen Dinh Quang Huy

Student ID: B23DCCE046

Class: D23CQCE04-B

Hanoi - 06/2025

Introduction

This report presents an image classification project using the CIFAR-10 dataset. The main objective is to build, evaluate, and compare two distinct neural network models: a basic Multi-Layer Perceptron (MLP) and a Convolutional Neural Network (CNN).

The entire implementation is developed using the PyTorch library. The following sections will detail the code, visualize the results through learning curves and confusion matrices, and provide a final discussion on the performance of both models.

I. Problem analysis

Perform image classification using the CIFAR-10 dataset: https://www.cs.toronto.edu/~kriz/cifar.html

Tasks:

- Build a basic MLP (Multi-Layer Perceptron) neural network with 3 layers.
- Build a Convolutional Neural Network (CNN) with 3 convolution layers.
- Perform image classification using both neural networks, including training, validation, and testing.
- Plot learning curves.
- Plot confusion matrix.
- Compare and discuss the results of the two neural networks.
- Use the PyTorch library.

II. Code

1. Ideal

- **Setup & Imports**
  + *import torch*: Main PyTorch library.
  + *import torch.nn as nn*: Module containing neural network layers (Conv2d, Linear, ReLU, MaxPool2d, CrossEntropyLoss).
  + *import torch.optim as optim*: Module containing optimization algorithms (Adam, SGD).
  + *import torchvision*: Provides popular datasets (like CIFAR-10), model architectures, and common image transformations.
  + *import torchvision.transforms as transforms*: Contains image preprocessing functions.
  + *from torch.utils.data import DataLoader, SubsetRandomSampler*: For loading data in batches and creating a validation set.

    + *import numpy as np*: For numerical operations, especially when handling indices.
    + *import matplotlib.pyplot as plt and import seaborn as sns*: For plotting graphs (learning curves, confusion matrix).
    + *from sklearn.metrics import confusion_matrix*: To calculate the confusion matrix.
- **Configuration & Hyperparameters:** Centralize tunable values in one place for easy experimentation and adjustment.
- **Data Loading & Preprocessing:** Prepare data in a format suitable for the model and split it into training, validation, and test sets.
- **Model Definition:** Create a Python class that inherits from *torch.nn.Module* to encapsulate the network architecture.
- **Loss Function and Optimizer:** Choose a suitable loss function for the multi-class classification task, select an optimization algorithm, and pass the model's parameters along with the learning rate.
- **Training Loop:** Iteratively feed data into the model, calculate the error (loss), and update the weights for the model to learn. Monitor performance on both the training and validation sets.
- **Testing Loop:** Evaluate the final performance of the model on the test dataset (data the model has never seen before).
- **Plotting:** Plot learning curves and plot the confusion matrix.

2. Detailed Description of Key Components

- **Device**

```
# 1. Device (CPU/GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

    + The code automatically selects the GPU if available; otherwise, it will use the CPU.

- **Hyperparameters**

```
# 2. Hyperparameters
num_epochs = 25
batch_size = 64
learning_rate = 0.001
validation_split = 0.1
random_seed = 42
```

+ *num_epochs*: The number of times the entire training dataset is iterated through.
+ *batch_size*: The number of data samples processed in one weight update iteration.
+ *learning_rate*: The learning speed of the model.
+ *validation_split*: The proportion of training data separated to be used as the validation set.
+ *random_seed*: Used to ensure the reproducibility of results.

- **Data Loading and Preprocessing**

```
# 3. Data Loading and Preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)

num_train = len(train_dataset)
indices = list(range(num_train))
split = int(np.floor(validation_split * num_train))

np.random.seed(random_seed)
np.random.shuffle(indices)

train_indices, val_indices = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)

train_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=train_sampler)
val_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=val_sampler)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

+ *transforms.ToTensor()*: Converts a PIL Image or NumPy ndarray (H x W x C) with a pixel range of [0, 255] into a PyTorch tensor (C×H×W) with a range of [0.0, 1.0].
+ *transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))*: Normalizes the tensor with the mean and standard deviation for each channel.
+ *SubsetRandomSampler*: Provides indices to randomly sample from the original dataset, typically used to create training and validation sets without altering the original data.

- **CNN Architecture**

```python
# 4. CNN Architecture
class CNN(nn.Module):
    def __init__(self, num_classes=10):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1)
        self.relu3 = nn.ReLU()
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.relu4 = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(512, num_classes)
    def forward(self, x):
        x = self.pool1(self.relu1(self.conv1(x)))
        x = self.pool2(self.relu2(self.conv2(x)))
        x = self.pool3(self.relu3(self.conv3(x)))
        x = x.view(-1, 128 * 4 * 4) # Flatten
        x = self.relu4(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

  + ***nn.Conv2d***: A 2D convolution layer.
    - *in_channels*: The number of input channels (e.g., 3 for an RGB image).
    - *out_channels*: The number of filters or output channels.
    - *kernel_size*: The size of the filter.
    - *padding=1*: Helps maintain the spatial dimensions of the image after convolution if *kernel_size=3* and *stride=1*.
  + ***nn.ReLU***: Rectified Linear Unit activation function, a common non-linear function.
  + ***nn.MaxPool2d***: LMax pooling layer, reduces the spatial dimensions of the feature map, helping to decrease the number of parameters and control overfitting.
    - kernel_size=2, stride=2: Reduces each dimension by half.

+ ***x.view(-1, 128 * 4 * 4)***: "Flattens" the output from the convolutional/pooling layers into a vector to be fed into a fully connected layer.. 128 * 4 * 4 is the size of the output after the *pool3* layer.
+ ***nn.Linear***: A fully connected layer.
+ ***nn.Dropout***: A dropout layer, a regularization technique to reduce overfitting by randomly "turning off" some neurons during training.

- **Loss Function and Optimizer**

```
# 5. Loss Function and Optimizer
model = CNN(num_classes=len(classes)).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

+ ***nn.CrossEntropyLoss***: Commonly used for multi-class classification problems.
+ It combines *LogSoftmax* and *NLLLoss* into a single class..
+ ***optim.Adam***: A popular optimization algorithm that often yields good results with minimal hyperparameter tuning.

- **Training Loop**

```python
# 6. Training Loop
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

print("Starting training...")
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct_train = 0
    total_train = 0

    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted_train = torch.max(outputs.data, 1)
        total_train += labels.size(0)
        correct_train += (predicted_train == labels).sum().item()

    epoch_train_loss = running_loss / len(train_loader)
    epoch_train_acc = 100 * correct_train / total_train
    train_losses.append(epoch_train_loss)
    train_accuracies.append(epoch_train_acc)

    # Validation
    model.eval()
    val_loss = 0.0
    correct_val = 0
    total_val = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            val_loss += loss.item()
            _, predicted_val = torch.max(outputs.data, 1)
            total_val += labels.size(0)
            correct_val += (predicted_val == labels).sum().item()

    epoch_val_loss = val_loss / len(val_loader)
    epoch_val_acc = 100 * correct_val / total_val
    val_losses.append(epoch_val_loss)
    val_accuracies.append(epoch_val_acc)

    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Train Loss: {epoch_train_loss:.4f}, Train Acc: {epoch_train_acc:.2f}%, '
          f'Val Loss: {epoch_val_loss:.4f}, Val Acc: {epoch_val_acc:.2f}%')

print("Finished Training")
```

+ *model.train()*: Sets the model to training mode (e.g., activates dropout, batch normalization updates running mean/variance).
+ *optimizer.zero_grad()*: Clears gradients from previous iterations.
+ *loss.backward()*: Computes the gradient of the loss with respect to model parameters.
+ *optimizer.step()*: Updates the model's weights based on the gradients.

- **Validation Loop**

```python
# 7. Validation Loop
model.eval()
all_labels = []
all_predictions = []
test_correct = 0
test_total = 0

with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)

        test_total += labels.size(0)
        test_correct += (predicted == labels).sum().item()

        all_labels.extend(labels.cpu().numpy())
        all_predictions.extend(predicted.cpu().numpy())

test_accuracy = 100 * test_correct / test_total
print(f'Accuracy of the network on the {len(test_dataset)} test images: {test_accuracy:.2f} %')
```

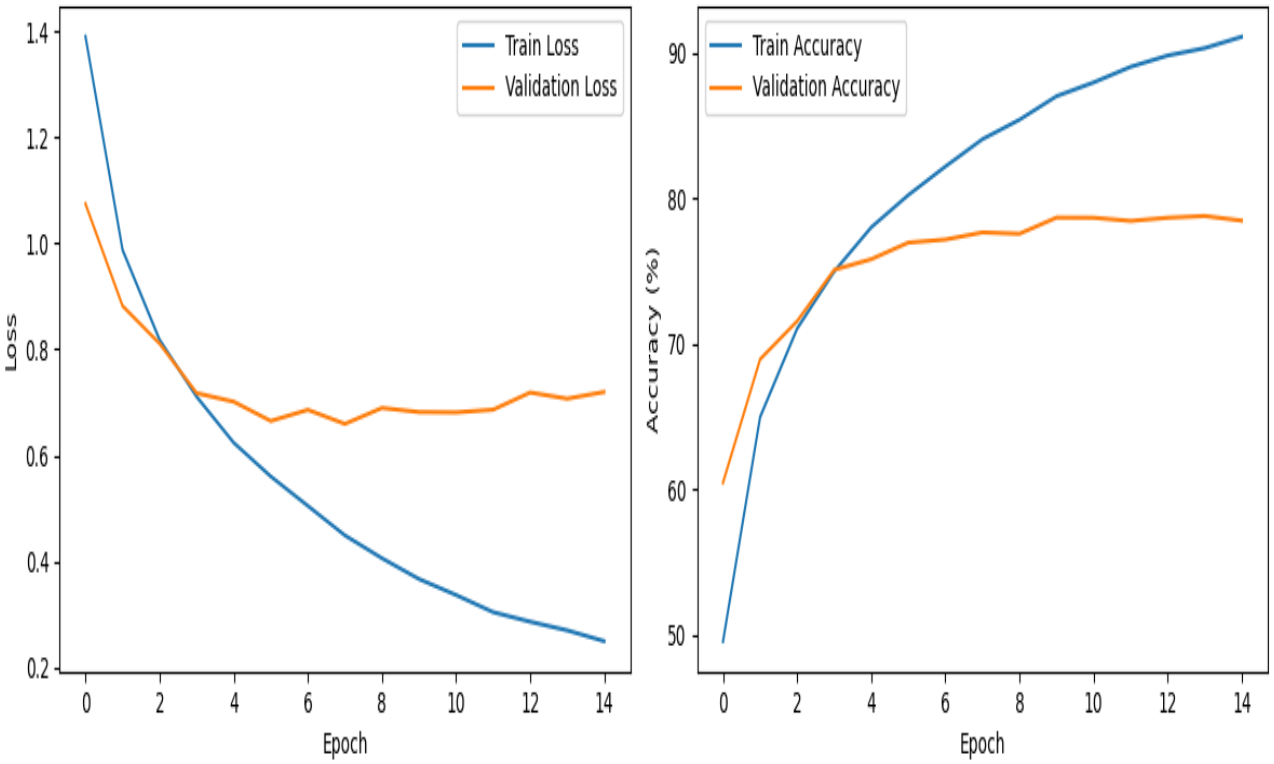
  + *model.eval()*: Sets the model to evaluation mode (e.g., deactivates dropout, batch normalization uses learned running mean/variance).
  + *with torch.no_grad()*: Disables gradient computation, saving memory and speeding up the process during evaluation or testing.
- **Plotting**
  + ***Learning Curves*:** Display the change in loss and accuracy on the training and validation sets across epochs.

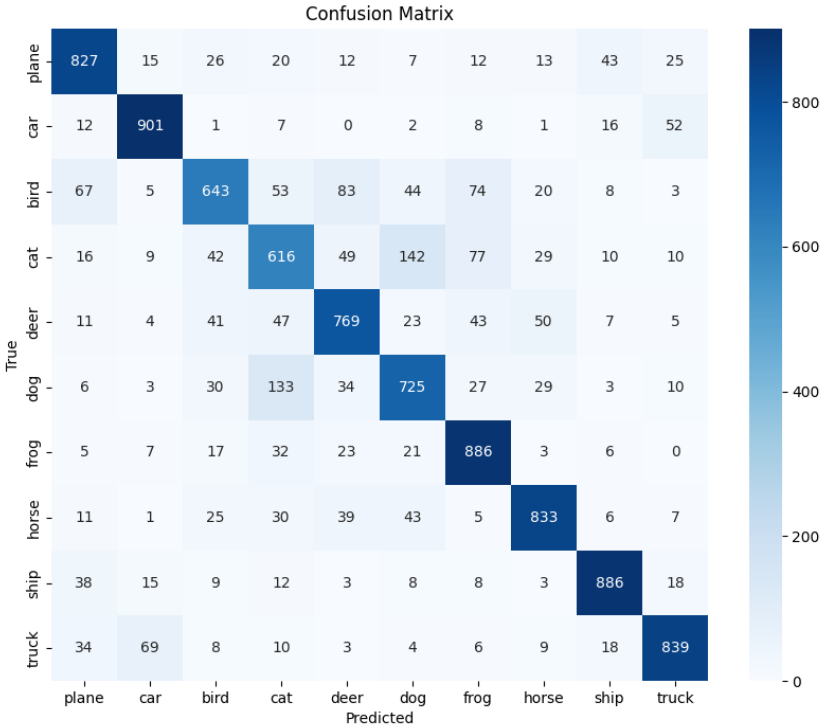
```python
# 8.1. Plot learning curves.
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Training Loss')
plt.plot(range(1, num_epochs + 1), val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Learning Curve - Loss')
plt.legend()
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label='Training Accuracy')
plt.plot(range(1, num_epochs + 1), val_accuracies, label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.title('Learning Curve - Accuracy')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.savefig('learning_curves.png')
plt.show()
```

+ ***Confusion Matrix*:** Shows the number of correct and incorrect predictions for each class. The main diagonal represents correct predictions.

```
# 8.2. Plot confusion matrix.
cm = confusion_matrix(all_labels, all_predictions)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.savefig('confusion_matrix.png')
plt.show()
```

III. Result


Train Loss
Validation Loss
Loss
Epoch
Train Accuracy
Validation Accuracy
Accuracy (%)
Epoch


Confusion Matrix
plane 827 15 26 20 12 7 12 13 43 25
car 12 901 1 7 0 2 8 1 16 52
bird 67 5 643 53 83 44 74 20 8 3
cat 16 9 42 616 49 142 77 29 10 10
deer 11 4 41 47 769 23 43 50 7 5
dog 6 3 30 133 34 725 27 29 3 10
frog 5 7 17 32 23 21 886 3 6 0
horse 11 1 25 30 39 43 5 833 6 7
ship 38 15 9 12 3 8 8 3 886 18
truck 34 69 8 10 3 4 6 9 18 839
True
plane car bird cat deer dog frog horse ship truck
Predicted
800
600
400
200
0