Deep Learning Project 2 Task 3: Training on Image Classification Based on ResNet

1. Task Overview:

This task focuses on end-to-end implementation of a deep learning pipeline to build, train, and evaluate a ResNet18 model for image classification on the CIFAR-10 dataset. It covers all critical stages of the workflow, from environment setup and data preparation to model definition, training, and performance assessment, ensuring a comprehensive approach to training a residual network.

The task begins with importing essential libraries, including PyTorch for model building, torchvision for dataset handling, and tools for data manipulation and visualization, to establish the technical foundation. It then proceeds to load the CIFAR-10 dataset, compute channel-wise mean and standard deviation for normalization, and define data augmentation (e.g., random flips, rotations) and preprocessing pipelines to enhance model generalization. DataLoaders are created to enable efficient batch processing of training and test data.

Next, the task defines core components of ResNet18: a `BottleNeck` residual block (with skip connections to mitigate vanishing gradients) and the full ResNet18 architecture, which stacks these blocks into hierarchical layers to extract features. The model is initialized, and training parameters (epochs, loss function, optimizer) are configured. Finally, the task implements a training loop with batch-wise forward/backward passes, an evaluation function to track accuracy and loss, and mechanisms like model checkpointing (saving the best-performing model) and early stopping (halting training on stagnation) to optimize performance. Throughout, metrics are tracked to monitor training dynamics and assess the model’ s generalization to unseen test data.

2. Task Objective

The task objective is to enable comprehensive mastery of end-to-end deep learning pipeline implementation, focusing on successfully building, training, and evaluating a ResNet18 model for CIFAR-10 image classification, while grasping both technical details and practical optimization strategies.

Objective Category	Sub - objective	Details

Ideological & Political Objectives	Ideological & Political Elements	Highlight the value of systematic pipeline design in deep learning, emphasizing the importance of rigorous technical implementation in advancing computer vision applications. Promote the awareness of resource-efficient model training and the significance of reproducible workflows in scientific and technical practices.
Learning Objectives	Knowledge Objectives	Enable learners to master key knowledge, including the structure and principles of ResNet18 (e.g., residual blocks, skip connections for mitigating vanishing gradients), CIFAR-10 dataset characteristics, data preprocessing techniques (channel-wise normalization, data augmentation like random flips/rotations), and model training essentials (loss functions, optimizers, training loops, evaluation metrics). Understand the role of components like DataLoaders, model checkpointing, and early stopping in the pipeline.
	Skill Objectives	Develop learners' practical skills in environment setup and library import (PyTorch, torchvision, data manipulation/visualization tools), data preparation (loading CIFAR-10, computing normalization stats, defining augmentation/preprocessing pipelines, creating DataLoaders), and ResNet18 implementation (defining BottleNeck blocks and full architecture). Cultivate abilities in configuring training parameters, implementing training/evaluation loops, and applying optimization mechanisms (checkpointing, early stopping) to enhance model performance.
	Literacy Objectives	Enhance learners' literacy in technical documentation application (e.g., referencing PyTorch/torchvision docs for library usage), data-driven analysis (interpreting training dynamics via tracked metrics), and clear presentation of pipeline results (explaining model performance on test data, justifying choices of augmentation/optimization strategies). Improve the ability to connect theoretical concepts (e.g., residual learning) with practical pipeline outcomes.

3. Task Implementation

3.1 Importing Libraries for Deep Learning and Data Processing

The following code imports essential Python libraries required for building and training deep learning models using **PyTorch**.

- `torch` **and** `torch.nn` : Core PyTorch library and neural network module for building layers, defining models, and handling tensors.
- `torchvision.datasets` **and** `torchvision.transforms` : Provides standard datasets (like CIFAR-10, MNIST) and common image preprocessing/transformation functions.
- `torch.utils.data.DataLoader` : Facilitates batching, shuffling, and parallel loading of datasets during training.
- `pandas` **and** `numpy` : For data manipulation, numerical operations, and handling tabular or array-based datasets.
- `sklearn.metrics.accuracy_score` : Computes accuracy to evaluate model
- `matplotlib.pyplot` : Used for visualization of results such as loss curves, accuracy trends, or image samples.

Code block

```
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import pandas as pd
import numpy as np
from torch import nn
import torch
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

This setup forms a robust foundation for any computer vision project, from data loading and preprocessing to model training, evaluation, and visualization.

3.2 Loading the CIFAR-10 Dataset

The following code loads the **CIFAR-10 dataset**, a widely used benchmark dataset in computer vision consisting of 60,000 32×32 color images across 10 classes (such as airplanes, cars, birds, etc.). The dataset is split into **50,000 training images** and **10,000 test images**.

Code block

```
train_dataset = datasets.CIFAR10(root='./dataset/', train=True, download=True, transform=transforms.ToTensor())
test_dataset = datasets.CIFAR10(root='./dataset/', train=False, download=True, transform=transforms.ToTensor())
```

- `root='./dataset/'` : Specifies the local directory where the dataset will be stored.
- `train=True/False` : Determines whether to load the training set (`True`) or the test set (`False`).
- `download=True` : Downloads the dataset from the internet if it’ s not already available locally.
- `transform=transforms.ToTensor()` : Converts images from PIL format to PyTorch tensors, scaling pixel values from `[0, 255]` to `[0.0, 1.0]` .

This setup ensures that the data is ready for model input, enabling seamless integration into training and evaluation pipelines.

3.3 Computing CIFAR-10 Dataset Normalization Statistics

Before training a neural network, it is important to normalize input data so that pixel values have zero mean and unit variance. The following code calculates the **channel-wise mean and standard deviation** for the CIFAR-10 training dataset.

Code block

```
train_loader = DataLoader(train_dataset, batch_size=len(train_dataset),
shuffle=False)
data = next(iter(train_loader))[0]
mean = data.mean(dim=[0,2,3])
std  = data.std(dim=[0,2,3])

print("Mean:", mean)
print("Std:", std)
```

- `DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=False)` : Loads the entire training dataset in a single batch for easy computation of global statistics.
- `next(iter(train_loader))[0]` : Extracts the images from the batch (ignoring labels).
- `data.mean(dim=[0,2,3])` : Computes the mean of each RGB channel across all images and spatial dimensions (height × width).
- `data.std(dim=[0,2,3])` : Computes the standard deviation of each channel similarly.
- The printed **mean** and **std** values are then used for normalizing the dataset during training, ensuring consistent input distributions for the neural network.

This step is crucial for **stabilizing and accelerating the training process** of deep learning models.

3.4 Data Augmentation and Normalization for Training and Testing

Data augmentation helps neural networks generalize better by creating variations of the input images, while normalization ensures consistent pixel distributions. The following code defines **augmentation and preprocessing pipelines** for both training and test datasets using `torchvision.transforms.Compose`.

Code block

```
# Augmentation

train_augment = transforms.Compose(
    [transforms.Resize((227, 227)),
     transforms.RandomHorizontalFlip(),
     transforms.RandomRotation(15),
     transforms.ToTensor(),
     transforms.Normalize(mean=mean, std=std)]
)

test_transform = transforms.Compose([
    transforms.Resize((227, 227)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

- `Resize((227, 227))` : Resizes all images to 227×227 pixels, which is compatible with AlexNet input requirements.
- `RandomHorizontalFlip()` : Randomly flips images horizontally to simulate variations.
- `RandomRotation(15)` : Randomly rotates images by ±15 degrees to increase robustness.
- `ToTensor()` : Converts images to PyTorch tensors and scales pixel values to [0,1].
- `Normalize(mean=mean, std=std)` : Standardizes each channel using the previously computed mean and standard deviation.

The **training augmentation** introduces randomness for better generalization, while the **test transform** applies only resizing and normalization to ensure consistent evaluation

3.5 Loading CIFAR-10 with Data Augmentation and Normalization

After defining augmentation and normalization pipelines, the CIFAR-10 dataset is reloaded with these transformations applied directly during data retrieval. This ensures that every image is automatically preprocessed before being fed into the model.

Code block

```
train_dataset = datasets.CIFAR10(root='./dataset/', train=True, download=True, transform=train_augment)
test_dataset = datasets.CIFAR10(root='./dataset/', train=False, download=True, transform=test_transform)
```

- `transform=train_augment` : Applies resizing, random horizontal flips, rotations, tensor conversion, and normalization to the training images.
- `transform=test_transform` : Applies resizing, tensor conversion, and normalization to the test images only, without random augmentations to maintain evaluation consistency.
- `download=True` : Ensures that the dataset is downloaded if not already available locally.
- `root='./dataset/'` : Specifies the directory where the dataset is stored.

This approach creates a seamless data pipeline where **training data is diversified** for robust learning, and **test data remains standardized** for fair evaluation.

3.6 Creating DataLoaders for Training and Testing

PyTorch `DataLoader` objects are used to efficiently load and batch datasets during model training and evaluation. The following code creates loaders for the CIFAR-10 training and test sets.

Code block

```
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

- `train_loader` :
  - `batch_size=64` → Loads 64 images per batch during training, which balances memory usage and gradient stability.
  - `shuffle=True` → Randomizes the order of images each epoch to prevent the model from learning spurious patterns based on data ordering.
- `test_loader` :
  - `batch_size=1000` → Loads 1000 test images per batch to reduce evaluation time while keeping memory usage manageable.
  - `shuffle=False` → Maintains a fixed order of images for consistent evaluation.

Using `DataLoader` allows **efficient mini-batch training**, automatic shuffling for better generalization, and convenient iteration over the dataset during training and testing loops.

3.7 BottleNeck Residual Block

The `BottleNeck` class implements a **residual block**, a fundamental building unit of ResNet architectures. Residual blocks help networks **train deeper models** by introducing skip connections that allow gradients to flow directly through the network.

Code block

```
class BottleNeck(nn.Sequential):
    def __init__(self, in_channel, out_channel, downsample=False):
        super(BottleNeck, self).__init__()
        self.stride = 2 if downsample else 1
        self.conv1 = nn.Conv2d(in_channels=in_channel,
out_channels=out_channel, kernel_size=3, stride=self.stride, padding=1)
        self.batch1 = nn.BatchNorm2d(out_channel)
        self.rel1 = nn.ReLU()
        self.conv2 = nn.Conv2d(in_channels=out_channel,
out_channels=out_channel, kernel_size=3, stride=1, padding=1)
        self.batch2 = nn.BatchNorm2d(out_channel)

        if downsample or in_channel != out_channel:
            self.conv = nn.Conv2d(in_channels=in_channel,
out_channels=out_channel, kernel_size=1, stride=self.stride, bias=False)
        else:
            self.conv = None

    def forward(self, x):
        residual = x
        if self.conv is not None:
            residual = self.conv(x)
        x = self.conv1(x)
        x = self.batch1(x)
        x = self.rel1(x)
        x = self.conv2(x)
        x = self.batch2(x)
        x = x + residual
        x = self.rel1(x)
        return x
```

- **Conv-BN-ReLU-Conv-BN**: Two consecutive convolutional layers with BatchNorm and ReLU activation.
- **Skip connection (`residual`)**: Adds input to output, helping gradients bypass layers and preventing vanishing gradients.
- **Downsample**: If the block changes the feature map size or number of channels, a 1×1 convolution adjusts the residual to match dimensions.

3.8 ResNet18 Model

The `ResNet18` class builds the **full 18-layer residual network** for CIFAR-10 (or similar datasets).

Code block

```
class ResNet18(nn.Module):
    def __init__(self, in_channel=3, classes=10):
        super(ResNet18, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(in_channels=in_channel, out_channels=64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        )
        self.layer_1 = nn.Sequential(
            BottleNeck(64,64, downsample=False),
            BottleNeck(64,64, downsample=False)
        )
        self.layer_2 = nn.Sequential(
            BottleNeck(64,128, downsample=True),
            BottleNeck(128,128, downsample=False)
        )
        self.layer_3 = nn.Sequential(
            BottleNeck(128,256, downsample=True),
            BottleNeck(256,256, downsample=False)
        )
        self.layer_4 = nn.Sequential(
            BottleNeck(256,512, downsample=True),
            BottleNeck(512,512, downsample=False)
        )
        self.avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.classifier = nn.Linear(512, classes)

    def forward(self, x):
        x = self.features(x)
        x = self.layer_1(x)
        x = self.layer_2(x)
        x = self.layer_3(x)
        x = self.layer_4(x)
        x = self.avg_pool(x)
        x = x.view(-1, 512)
        x = self.classifier(x)
        return x
```

- **Initial feature extraction**: Conv7×7 → BatchNorm → ReLU → MaxPool reduces spatial size.
- **Four residual layers (`layer_1` to `layer_4`)**: Each consists of 2 `BottleNeck` blocks, progressively increasing channel depth (64 → 128 → 256 → 512) and downsampling spatial dimensions.
- **Adaptive AvgPool2d(1,1)**: Reduces feature maps to 1×1 spatially regardless of input size.
- **Fully connected layer**: Maps 512 features to the number of classes (e.g., 10 for CIFAR-10).
- **Forward pass**: Sequentially applies layers, pools the features, flattens, and outputs logits for classification.

3.9 Initializing ResNet18 and Setting Training Parameters

After defining the ResNet18 model, it is instantiated and prepared for training on a specified computational device (`CPU` or `GPU`). Key training parameters such as the number of epochs, loss function, and optimizer are also configured.

Code block

```
model = ResNet18().to(device)

# Now set the parameters
epochs = 30
criterian = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

- `model = ResNet18().to(device)`: Creates an instance of ResNet18 and moves it to the selected device for efficient computation.
- `epochs = 30`: The model will make 30 complete passes over the training dataset.
- `criterian = nn.CrossEntropyLoss()`: Suitable for multi-class classification problems like CIFAR-10. Combines `LogSoftmax` and `NLLLoss` to compute the difference between predicted logits and true labels.
- `optimizer = torch.optim.Adam(model.parameters(), lr=0.001)`: Uses the Adam optimizer to update model weights. Adam combines the benefits of RMSProp and momentum, and `lr=0.001` is a common starting learning rate for deep networks.

This setup establishes the **training loop foundation**, enabling the model to learn patterns in the CIFAR-10 dataset efficiently.

Here's a detailed, structured explanation of your **training and evaluation loop** for the ResNet18 model, split into parts for clarity:

3.9.1 Evaluation Function (`evaluate`)

The `evaluate` function computes the **accuracy and average loss** of the model on a given dataset (typically the test set).

Code block

```
def evaluate(loader: DataLoader):
    test_true_label = []
    test_pred_label = []
    test_overall_loss = 0.0
    with torch.no_grad():
        for data, label in loader:
            data, label = data.to(device), label.to(device)
            test_prediction = model(data)
            test_loss = criterian(test_prediction, label)
            _, test_pred = torch.max(test_prediction, 1)
            test_true_label.append(label.cpu().numpy())
            test_pred_label.append(test_pred.cpu().numpy())
            test_overall_loss += test_loss.item()
        average_test_loss = test_overall_loss / len(loader)
        ty_true = np.concatenate(test_true_label)
        ty_pred = np.concatenate(test_pred_label)
        testing_final_accuracy = accuracy_score(ty_true, ty_pred)
        return testing_final_accuracy, average_test_loss
```

Key points:

- **`torch.no_grad()`** : Disables gradient computation during evaluation to save memory.
- **`torch.max(test_prediction, 1)`** : Finds the predicted class index for each sample.
- Loss and labels are collected batch-wise and then aggregated.
- Returns **accuracy** and **average loss** for the dataset.

3.9.2 Training Function (`train`)

The `train` function performs **multi-epoch training** with optional **early stopping** and model checkpointing.

Code block

```
def train(training_loader, testing_loader, test_eval: bool = True, patience:
int = 5):
    benchmark_testing_accuracy = 0.0
    counter = 0
    train_plot_accuracy, test_plot_accuracy = [], []
    train_plot_loss, test_plot_loss = [], []
```

- `benchmark_testing_accuracy` : Tracks the best testing accuracy for saving the model.
- `counter` : Counts epochs without improvement for early stopping.
- Lists store **accuracy and loss per epoch** for plotting.

3.9.3 Epoch Loop

Code block

```
    for epoch in range(epochs):
        model.train()
        train_loss = 0.0
        true_label, pred_label = [], []
```

- Sets the model to **training mode** (`model.train()`) to enable Dropout and BatchNorm behavior.
- Initializes variables to accumulate **batch losses and predictions**.

3.9.4 Batch Training Step

Code block

```
        for data, label in training_loader:
            data, label = data.to(device), label.to(device)
            model.zero_grad()
            train_pred = model(data)
            training_loss = criterian(train_pred, label)
            training_loss.backward()
            optimizer.step()
            _, prediction = torch.max(train_pred, 1)
            true_label.append(label.cpu().numpy())
            pred_label.append(prediction.cpu().detach().numpy())
            train_loss += training_loss.item()
```

- **Forward pass:** `train_pred = model(data)`
- **Loss computation:** `criterian(train_pred, label)`
- **Backpropagation:** `.backward()` and `.step()` update weights.
- Predictions and labels are stored for calculating training accuracy.

3.9.5 Epoch Metrics

Code block

```
        average_loss = train_loss / len(training_loader)
        y_true = np.concatenate(true_label)
        y_pred = np.concatenate(pred_label)
        train_accuracy = accuracy_score(y_true, y_pred)
```

- Computes **average training loss** for the epoch.
- Computes **training accuracy** across all batches.

3.9.6 Evaluation and Logging

Code block

```
        testing_final_accuracy, average_test_loss = evaluate(testing_loader)
        train_plot_accuracy.append(train_accuracy)
        test_plot_accuracy.append(testing_final_accuracy)
        train_plot_loss.append(average_loss)
        test_plot_loss.append(average_test_loss)

        print(f'Epoch {epoch + 1}: Training Loss: {average_loss}, Training Accuracy: {train_accuracy}, Testing Loss: {average_test_loss}, Testing Accuracy: {testing_final_accuracy}')
```

- Runs `evaluate()` on test data each epoch.
- Logs **training and testing metrics** for visualization and monitoring

3.9.7 Model Checkpointing and Early Stopping

Code block

```
        if testing_final_accuracy > benchmark_testing_accuracy:
            torch.save(model.state_dict(), './model_checkpoint/alex_net/alex_net.pth')
            benchmark_testing_accuracy = testing_final_accuracy
            counter = 0
            print(f"Model saved at accuracy of: {testing_final_accuracy}")
        else:
            counter += 1
        if counter > patience:
            print("Model stopping due to no improvement in testing accuracy")
            return train_plot_accuracy, test_plot_accuracy, train_plot_loss, test_plot_loss
```

- **Checkpointing:** Saves model weights when test accuracy improves.

- **Early stopping:** Stops training if no improvement is observed for `patience` consecutive epochs.

3.9.8 Return Training History

Code block

```
    return train_plot_accuracy, test_plot_accuracy, train_plot_loss,
test_plot_loss
```

- Returns **epoch-wise metrics** for plotting learning curves.

3.9.9 Running the Training

Code block

```
train_plot_accuracy, test_plot_accuracy, train_plot_loss, test_plot_loss =
train(train_loader, test_loader)
```

- Starts the full training process for ResNet18 using CIFAR-10.
- Produces metrics that can be visualized to monitor training progress.