Deep Learning Task 4: Training on Artificial Neural Networks Based on MNIST Handwritten Digit Recognition

1. Task Overview

The Basic Technology Verification Task, Practical Training on Handwritten Digit Image Classification Based on Deep Learning, focuses on the implementation and evaluation of fundamental deep learning techniques within the context of image recognition. Specifically, the task centers on the classification of handwritten digits, a classical and widely utilized benchmark in the field of computer vision and machine learning. The objective of this task is to facilitate a comprehensive understanding of the end-to-end deep learning pipeline, including data preprocessing, model architecture selection, training strategies, performance evaluation, and result interpretation.

In addition, the task provides an opportunity to understand the practical implications of model hyperparameters, overfitting and underfitting scenarios, and the importance of validation techniques. Through this hands-on implementation, the task enables the consolidation of theoretical knowledge with practical skill, thereby laying the groundwork for more advanced applications of deep learning in image processing tasks.

2. Task Objective

This table outlines the structured learning objectives for implementing and evaluating deep learning techniques in handwritten digit classification. The task emphasizes mastering the end - to - end deep learning pipeline using the MNIST dataset, fostering both technical proficiency and theoretical understanding in image recognition.

Objective Category	Sub - objective	Details
Ideological & Political Objectives	Ideological & Political Elements	
Learning Objectives	Knowledge Objectives	
	Skill Objectives	
	Literacy Objectives	

Caption: Table presenting the learning objectives for handwritten digit classification practical training, covering ideological cultivation, technical knowledge, practical skills, and collaborative literacy to reinforce deep learning proficiency in image recognition tasks.

Task Implementation

Importing Essential Libraries for Deep Learning and Model Evaluation

The following code snippet imports the fundamental libraries required for building, training, and evaluating deep learning models. **PyTorch** is used as the deep learning framework, while **Torchvision** provides datasets and transformations specifically for computer vision tasks. **NumPy** is included for numerical operations, and **Matplotlib** assists with visualizations. Additionally, **Scikit-learn's metrics** are used to compute performance measures such as precision, recall, F1-score, and accuracy, which are crucial for evaluating classification models.

Code block

```
import torch
from torch import nn

from torch.utils.data import DataLoader # loads data in batches
from torchvision import datasets # load MNIST
import torchvision.transforms as T # transformers for computer vision
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import precision_score, recall_score, f1_score,
accuracy_score
```

- `import torch` → Imports PyTorch, the main deep learning framework.
- `from torch import nn` → Gives access to neural network building blocks (e.g., layers, activations, loss functions).
- `from torch.utils.data import DataLoader` → Enables efficient loading of data in mini-batches, essential for training.
- `from torchvision import datasets` → Provides access to standard datasets such as MNIST, CIFAR-10, etc.
- `import torchvision.transforms as T` → Handles preprocessing and transformations (e.g., normalization, resizing, tensor conversion) for images.
- `import numpy as np` → Used for numerical computations and array manipulations.
- `import matplotlib.pyplot as plt` → Useful for plotting graphs and visualizing data or training progress.
- `from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score` → Imports evaluation metrics to assess the quality of trained models.

Defining Image Transformations for Preprocessing

Before training a deep learning model, input images need to be preprocessed into a form that the model can effectively work with. In this snippet, we define a transformation pipeline using **`torchvision.transforms.Compose`**, which ensures that each image is converted into a tensor and normalized to a suitable range. Normalization helps stabilize and speed up training by ensuring consistent input distribution across the dataset.

Code block

```
my_transform = T.Compose([
    T.ToTensor(),
    T.Normalize((0.5,), (0.5,))
])
```

- `T.Compose([...])` → Combines multiple transformations into a single pipeline.
- `T.ToTensor()` → Converts an image (e.g., from PIL or NumPy) into a PyTorch tensor with pixel values scaled between 0 and 1.
- `T.Normalize((0.5,), (0.5,))` → Normalizes the image tensor so that pixel values are rescaled to the range **[-1, 1]**.
  - The first tuple `(0.5,)` represents the mean.
  - The second tuple `(0.5,)` represents the standard deviation.

- Since this dataset (e.g., MNIST) is grayscale, a single channel value is used.

This transformation ensures that all images are in a consistent format, making training more efficient and improving convergence.

Loading the MNIST Dataset with Applied Transformations

This snippet demonstrates how to load the **MNIST dataset**, one of the most widely used benchmark datasets for handwritten digit recognition. Using `torchvision.datasets.MNIST`, the code separates the dataset into training and testing subsets and applies the preprocessing transformations defined earlier (`my_transform`). By specifying the root directory, the dataset is either loaded from disk or downloaded if not already available.

Code block

```
# Loading Dataset
train_dataset = datasets.MNIST(root='./data', train=True, download=False, transform=my_transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=False, transform=my_transform)
```

- `datasets.MNIST(...)` → Loads the MNIST dataset from the specified `root` directory.
- `train=True/False` → Determines whether to load the training set (`True`) or the test set (`False`).
- `download=False` → If set to `True`, PyTorch will download the dataset from the internet if it's not already present.
- `transform=my_transform` → Applies the previously defined transformations to each image (conversion to tensor and normalization).

By preparing the dataset this way, the data is ready to be fed into a neural network for training and evaluation.

Creating DataLoaders for Efficient Mini-Batch Training

In deep learning, feeding the entire dataset to a model at once is inefficient and often infeasible due to memory constraints. `DataLoader` in PyTorch provides an easy way to iterate over datasets in **mini-batches**, allowing for faster training and evaluation. This snippet wraps the previously loaded MNIST datasets into DataLoaders with specified batch sizes and shuffling behavior.

Code block

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

- `DataLoader(dataset=..., batch_size=..., shuffle=...)` → Prepares the dataset to be loaded in batches.
- `train_dataset` → Dataset used for training the model.
- `test_dataset` → Dataset used for evaluation.
- `batch_size=64` → Number of samples per batch during training. Smaller batches can stabilize training but may increase training time.
- `batch_size=1000` → Larger batch size during testing since no backpropagation occurs, making it more efficient.
- `shuffle=True` → Randomly shuffles the training data each epoch to improve model generalization.
- `shuffle=False` → Keeps the test data in a fixed order for consistent evaluation results.

Using DataLoaders in this way ensures efficient memory usage and proper randomization for better model performance.

Flattening MNIST Images for Fully Connected Networks

Neural networks often require input data to be in a **1-dimensional vector** rather than a 2D image. This snippet demonstrates how to **flatten a single MNIST image** (28×28 pixels) into a 1D tensor of size 784, which is commonly used when feeding images into **fully connected (dense) layers**.

Code block

```
# Flatten the image to [784]
train_dataset[0][0].view(1, -1).shape
```

- `train_dataset[0]` → Retrieves the first sample from the training dataset.
- `[0]` → Selects the image tensor (the second element `[1]` would be the label).
- `.view(1, -1)` → Reshapes the tensor into a 2D shape with 1 row and as many columns as needed (`-1` automatically calculates this value, here 28×28 = 784).
- `.shape` → Returns the shape of the resulting tensor, which will be `[1, 784]` .

Flattening is essential for models that expect 1D input vectors rather than 2D image matrices, such as **multilayer perceptrons (MLPs)**., so it’ s easier to understand how 28×28 becomes a

784-element vector. Do you want me to do that?

Defining a Fully Connected Neural Network (ANN) for MNIST

This snippet defines a **fully connected artificial neural network (ANN)** using PyTorch's `nn.Module`. The network is designed to classify MNIST digits, taking flattened 28×28 images as input and producing 10 output scores corresponding to the digit classes (0–9). The model consists of three linear layers with ReLU activations applied to the first two layers to introduce non-linearity.

Code block

```
class ANNModel(nn.Module):
    def __init__(self):
        super(ANNModel, self).__init__()
        self.fc1 = nn.Linear(28*28, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Explanation of each part:

- `class ANNModel(nn.Module)` → Defines a custom neural network class inheriting from PyTorch's base module.
- `init(self)` → Constructor that initializes the network layers.
- `self.fc1 = nn.Linear(28*28, 512)` → First fully connected layer transforming the 784-dimensional input into 512 neurons.
- `self.fc2 = nn.Linear(512, 128)` → Second layer reducing 512 features to 128 neurons.
- `self.fc3 = nn.Linear(128, 10)` → Output layer producing 10 scores for the 10 digit classes.
- `forward(self, x)` → Defines the forward pass of the network.
  - `x.view(x.size(0), -1)` → Flattens each image in the batch to a 1D vector.
  - `torch.relu(self.fc1(x))` → Applies linear transformation followed by ReLU activation.

    - `torch.relu(self.fc2(x))` → Second layer with ReLU.
    - `self.fc3(x)` → Output layer producing raw class scores (logits).
- `return x` → Returns the output logits for further processing (e.g., applying softmax during loss calculation).

This architecture is simple yet effective for MNIST digit classification, combining **fully connected layers** with **non-linear activation functions** to learn complex patterns in the data.

Setting Up the Model, Loss Function, and Optimizer

After defining the neural network, the next step is to **initialize the model**, move it to the appropriate device (GPU or CPU), define the **loss function**, and configure the **optimizer** for training. This snippet sets up a typical workflow for training an ANN on MNIST using PyTorch.

Code block

```
model = ANNModel().to("cuda")
criterian = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
epochs = 10
device = "cuda"
```

Explanation of each part:

- `model = ANNModel().to("cuda")` → Instantiates the neural network and moves it to the GPU for faster computation.
- `criterian = nn.CrossEntropyLoss()` → Defines the loss function for multi-class classification. Cross-entropy combines **softmax activation** and **negative log-likelihood**, making it suitable for digit classification.
- `optimizer = torch.optim.Adam(model.parameters(), lr=0.001)` → Sets up the Adam optimizer to update model weights, with a learning rate of 0.001. Adam is widely used due to its adaptive learning rate and momentum.
- `epochs = 10` → Specifies the number of full passes over the training dataset.
- `device = "cuda"` → Stores the device type for consistent use throughout training and evaluation.

This setup ensures the model is ready for training on GPU, with a proper loss function and optimizer to update its parameters efficiently.

Evaluating the Model on the Test Dataset

After training, it is crucial to evaluate the model’ s performance on unseen data. This function computes the **average test loss** and **accuracy** over the entire test dataset. It disables gradient computation for efficiency, collects predictions and true labels, and calculates metrics using **Scikit-learn**.

Code block

```
def evaluate(test_loader):
    with torch.no_grad():
        test_loss_overall = 0.0
        true_label = []
        pred_label = []

        for data, label in test_loader:
            data, label = data.to(device), label.to(device)
            test_pred = model(data)
            test_loss = criterian(test_pred, label)
            test_loss_overall += test_loss.item()

            _, test_prediction = torch.max(test_pred, 1)
            true_label.append(label.cpu().numpy())
            pred_label.append(test_prediction.cpu().numpy())

        average_test_loss = test_loss_overall / len(test_loader)
        ty_true = np.concatenate(true_label)
        ty_pred = np.concatenate(pred_label)

        test_accuracy = accuracy_score(ty_true, ty_pred)
        print(f'')
        return average_test_loss, test_accuracy
```

Explanation of each part:

- `with torch.no_grad():` → Disables gradient computation to reduce memory usage and speed up evaluation.
- `test_loss_overall = 0.0` → Initializes a variable to accumulate the total test loss.
- `true_label, pred_label = []` → Lists to store ground truth labels and model predictions.
- `for data, label in test_loader:` → Iterates over test batches.
- `data, label = data.to(device), label.to(device)` → Moves the batch to GPU (or CPU).
- `test_pred = model(data)` → Performs a forward pass to obtain predictions (logits).

- `test_loss = criterian(test_pred, label)` → Computes the batch loss using cross-entropy.
- `test_loss_overall += test_loss.item()` → Accumulates total loss over all
- `_, test_prediction = torch.max(test_pred, 1)` → Converts logits to predicted class indices.
- `true_label.append(label.cpu().numpy())` → Moves labels to CPU and stores them as NumPy arrays.
- `pred_label.append(test_prediction.cpu().numpy())` → Stores predictions similarly.
- `average_test_loss = test_loss_overall / len(test_loader)` → Computes mean loss across all batches.
- `ty_true = np.concatenate(true_label)` → Concatenates all true labels into a single array.
- `ty_pred = np.concatenate(pred_label)` → Concatenates all predictions.
- `test_accuracy = accuracy_score(ty_true, ty_pred)` → Computes overall test accuracy using Scikit-learn.
- `return average_test_loss, test_accuracy` → Returns average loss and accuracy for reporting.

This function provides a **comprehensive evaluation** of the model, allowing for performance monitoring on unseen data during or after training.

Executing Model Training and Saving the Best Checkpoint

This snippet runs the **training loop** for the specified number of epochs and automatically saves the **best-performing model** based on test accuracy. By specifying a file path, the trained model can be reloaded later for inference or further training.

Code block

```
path = r'D:\BOOK\Deep_learning_book\ANN\Models\MNIST.pt'
train(epochs=epochs, train_data_loader=train_loader, path=path)
```

Explanation of each part:

- `path = r'D:\BOOK\Deep_learning_book\ANN\Models\MNIST.pt'` → Defines the file path where the **best model checkpoint** will be saved. The raw string `r''` ensures backslashes are interpreted correctly in Windows paths.

- `train(epochs=epochs, train_data_loader=train_loader, path=path)` → Calls the `train` function defined earlier:
  - Trains the ANN for the specified number of epochs.
  - Monitors **training and test performance** after each epoch.
  - Saves the model whenever the test accuracy improves, ensuring the best version is retained.

This approach ensures reproducibility, allows model evaluation without retraining, and provides a **persistent checkpoint** for later use in deployment or experimentation.