

Assignment-1-Part-2-2

April 25, 2025

1 Assignment 1 Part 2.2

1.1 Import necessary modules

```
[1]: import torch
import torch.nn as nn
import os
import mlflow
from datetime import datetime
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import time
```

1.2 Layer class

```
[2]: class Layer:
    def __init__(self, neurons_size: int, inputs_size: int, activation: str = '
↳ 'none',
                leaky_slope: float = 0.01, softmax_dim: int = 0):
        """
        Initialize the layer with random weights and biases

        Parameters:
            neurons_size: int
                The number of neurons in the layer
            inputs_size: int
                The number of inputs to the layer
            activation: str
                The activation function to use
            leaky_slope: float
                The slope of the leaky relu activation function
            softmax_dim: int
                The dimension of the softmax activation function
        """

        # initialize weights and biases with random values
```

```

        # use the He initialization method to initialize the weights if the
        ↪ activation function is relu or leaky relu else use the Xavier initialization
        ↪ method
        if activation == 'relu' or activation == 'leaky_relu':
            self._weights: torch.Tensor = torch.randn(
                neurons_size, inputs_size) * torch.sqrt(2 / torch.
            ↪ tensor(inputs_size, dtype=torch.float32))
        else:
            self._weights: torch.Tensor = torch.randn(
                neurons_size, inputs_size) * torch.sqrt(1 / torch.
            ↪ tensor(inputs_size, dtype=torch.float32))
            self._biases: torch.Tensor = torch.zeros(neurons_size)
            self._activation: str = activation
            self._leaky_slope: float = leaky_slope
            self._softmax_dim: int = softmax_dim

            self._activation_function: dict[str, callable] = {
                'sigmoid': nn.Sigmoid(),
                'tanh': nn.Tanh(),
                'relu': nn.ReLU(),
                'leaky_relu': nn.LeakyReLU(negative_slope=self._leaky_slope),
                'softmax': nn.Softmax(dim=self._softmax_dim),
                'none': nn.Identity()
            }

    def set_weights(self, weights: torch.Tensor):
        """
        Set the weights of the layer

        Parameters:
            weights: torch.Tensor
                The weights to set
        """
        self._weights = weights

    def get_weights(self) -> torch.Tensor:
        """
        Get the weights of the layer

        Returns:
            torch.Tensor
                The weights of the layer
        """
        return self._weights

    def set_biases(self, biases: torch.Tensor):
        """

```

```

        Set the biases of the layer

Parameters:
    biases: torch.Tensor
        The biases to set
    """
    self._biases = biases

def get_biases(self) -> torch.Tensor:
    """
    Get the biases of the layer

Returns:
    torch.Tensor
        The biases of the layer
    """
    return self._biases

def forward(self, inputs: torch.Tensor) -> torch.Tensor:
    """
    Forward pass

Parameters:
    inputs: torch.Tensor
        The inputs to the layer
    activation: str
        The activation function to use
Returns:
    torch.Tensor
        The outputs of the layer
    """
    # calculate the sum of the inputs multiplied by the weights and add the
    ↪biases
    if inputs.dim() == 2:
        sum: torch.Tensor = torch.matmul(
            inputs, self._weights.t()) + self._biases
    else:
        sum: torch.Tensor = torch.matmul(
            inputs, self._weights) + self._biases

    if self._activation in self._activation_function:
        return self._activation_function[self._activation](sum)
    else:
        raise ValueError(
            f"Activation function {self._activation} not found")

```

1.3 Custom Layer class

```
[3]: class CustomLayer(nn.Module):
    def __init__(self, neurons_size: int, inputs_size: int, activation: str = 'none',
        ↪leaky_slope: float = 0.01, softmax_dim: int = 0):
        super(CustomLayer, self).__init__()

        '''
        Initialize the custom layer with the given neurons size, inputs size,
        activation, leaky slope, and softmax dimension
        '''

        # initialize the custom layer
        self.layer = Layer(neurons_size, inputs_size,
            activation, leaky_slope, softmax_dim)

        # register the weights and biases as parameters to PyTorch
        self.weights = nn.Parameter(self.layer.get_weights())
        self.biases = nn.Parameter(self.layer.get_biases())

        # point the layer parameters to the PyTorch parameters
        self.layer.set_weights(self.weights)
        self.layer.set_biases(self.biases)

        # store the activation function parameters
        self.activation = activation
        self.leaky_slope = leaky_slope
        self.softmax_dim = softmax_dim

    def forward(self, inputs: torch.Tensor) -> torch.Tensor:

        # use custom layer forward method
        return self.layer.forward(inputs)
```

1.4 Perceptron class

```
[4]: class Perceptron(nn.Module):
    def __init__(self, input_size: int, hidden_size: int, output_size: int,
        ↪hidden_activation: str = 'relu',
        ↪output_activation: str = 'none', leaky_slope: float = 0.01,
        ↪softmax_dim: int = 0):
        super(Perceptron, self).__init__()

        '''
```

```

        Initialize the perceptron with the given input size, hidden size,
        ↪output size,
        hidden activation, output activation, leaky slope, and softmax dimension

Parameters:
    input_size: int
        The size of the input layer
    hidden_size: int
        The size of the hidden layer
    output_size: int
        The size of the output layer
    hidden_activation: str
        The activation function to use for the hidden layer
    output_activation: str
        The activation function to use for the output layer
    leaky_slope: float
        The slope of the leaky relu activation function
    softmax_dim: int
        The dimension of the softmax activation function
'''

# create a list of layer sizes including the input and output sizes
layer_sizes: list[int] = [input_size] + hidden_size + [output_size]
layers: list[CustomLayer] = []

# create hidden layers
for i in range(len(layer_sizes) - 2):
    layers.append(CustomLayer(
        layer_sizes[i+1], layer_sizes[i], hidden_activation,
        ↪leaky_slope, softmax_dim))

# create output layer
layers.append(CustomLayer(
    layer_sizes[-1], layer_sizes[-2], output_activation, leaky_slope,
    ↪softmax_dim))

# store the layers
self.layers = nn.ModuleList(layers)

def forward(self, inputs: torch.Tensor) -> torch.Tensor:
    '''
    Forward pass

Parameters:
    inputs: torch.Tensor
        The inputs to the perceptron

```

```

Returns:
    torch.Tensor
    The outputs of the perceptron
'''

# process the input through the layers
x = inputs
for layer in self.layers:
    x = layer(x)
return x

```

1.5 Trainer class: Added performance metrics

```

[5]: class Trainer:
    def __init__(self,
                 model,
                 criterion,
                 optimizer,
                 train_loader,
                 val_loader=None,
                 test_loader=None,
                 device=None,
                 scheduler=None,
                 checkpoint_dir='./checkpoints',
                 experiment_name=None):
        """
        Initialize the trainer with model, criterion, optimizer, and data loaders

        Parameters:
            model: nn.Module
                The model to train
            criterion: loss function
                The loss function to use
            optimizer: torch.optim
                The optimizer to use
            train_loader: DataLoader
                The data loader for training data
            val_loader: DataLoader
                The data loader for validation data
            test_loader: DataLoader
                The data loader for test data
            device: torch.device
                The device to use for training
            scheduler: torch.optim.lr_scheduler
                Learning rate scheduler (optional)
            checkpoint_dir: str

```

```

        Directory to save checkpoints
        experiment_name: str
        Name of the experiment for MLflow tracking
        """
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.train_loader = train_loader
        self.val_loader = val_loader if val_loader is not None else test_loader
        self.test_loader = test_loader
        self.device = device if device is not None else torch.device(
            'cuda' if torch.cuda.is_available() else 'cpu')
        self.scheduler = scheduler
        self.checkpoint_dir = checkpoint_dir
        self.experiment_name = experiment_name
        self.input_size = None # will be set during first forward pass

        # create checkpoint directory if it doesn't exist
        os.makedirs(checkpoint_dir, exist_ok=True)

        # move model to device
        self.model.to(self.device)

        # initialize training metrics
        self.best_accuracy = 0.0
        self.best_loss = float('inf')
        self.history = {
            'train_loss': [],
            'val_loss': [],
            'val_accuracy': [],
            'learning_rates': [],
            'epoch_times': [] # Add tracking of per-epoch times
        }

        # initialize performance tracking
        self.total_training_time = 0
        self.epoch_start_time = 0
        self.total_epochs_completed = 0

def train_epoch(self, epoch, num_epochs):
    """
    Train the model for one epoch
    """
    # start timing the epoch
    self.epoch_start_time = time.time()

    self.model.train()

```

```

running_loss = 0.0

for i, (inputs, labels) in enumerate(self.train_loader):
    # get input size from first batch if not set
    if self.input_size is None and hasattr(inputs, 'view'):
        self.input_size = inputs.view(inputs.size(0), -1).size(1)

    # prepare inputs
    if hasattr(inputs, 'view'):
        # for image data, flatten if needed
        inputs = inputs.view(inputs.size(0), -1).to(self.device)
    else:
        inputs = inputs.to(self.device)

    labels = labels.to(self.device)

    # forward pass
    outputs = self.model(inputs)
    loss = self.criterion(outputs, labels)

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

    # update statistics
    running_loss += loss.detach().item()

    # print progress
    if (i+1) % 100 == 0:
        print(
            f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{len(self.
↵train_loader)}], Loss: {loss.detach().item():.4f}')

    # calculate average loss for the epoch
    epoch_loss = running_loss / len(self.train_loader)
    self.history['train_loss'].append(epoch_loss)

    # track learning rate
    current_lr = self.optimizer.param_groups[0]['lr']
    self.history['learning_rates'].append(current_lr)

    # calculate and store epoch time
    epoch_time = time.time() - self.epoch_start_time
    self.history['epoch_times'].append(epoch_time)
    self.total_training_time += epoch_time
    self.total_epochs_completed += 1

```



```

        # print epoch time
        print(f'Epoch time: {epoch_time:.2f} seconds')

    return epoch_loss

def validate(self, epoch=None):
    """
    Validate the model on validation data
    """
    if self.val_loader is None:
        return None, None

    self.model.eval()
    running_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in self.val_loader:
            # prepare inputs
            if hasattr(inputs, 'view'):
                # for image data
                inputs = inputs.view(inputs.size(0), -1).to(self.device)
            else:
                inputs = inputs.to(self.device)

            labels = labels.to(self.device)

            # forward pass
            outputs = self.model(inputs)
            loss = self.criterion(outputs, labels)

            # update statistics
            running_loss += loss.item()

            # calculate accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # calculate average loss and accuracy
    val_loss = running_loss / len(self.val_loader)
    val_accuracy = 100 * correct / total

    if epoch is not None:
        self.history['val_loss'].append(val_loss)

```

```

        self.history['val_accuracy'].append(val_accuracy)

    return val_loss, val_accuracy

def test(self):
    """
    Test the model on test data
    """
    if self.test_loader is None:
        return None, None

    return self.validate() # reuse validation code for testing

def save_checkpoint(self, epoch, train_loss, val_loss=None,
    ↪ val_accuracy=None, is_best=False):
    """
    Save a checkpoint of the model
    """
    checkpoint = {
        'epoch': epoch + 1,
        'model_state_dict': self.model.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        'loss': train_loss
    }

    if val_loss is not None:
        checkpoint['val_loss'] = val_loss

    if val_accuracy is not None:
        checkpoint['val_accuracy'] = val_accuracy

    if self.scheduler is not None:
        checkpoint['scheduler_state_dict'] = self.scheduler.state_dict()

    # save regular checkpoint
    checkpoint_path = os.path.join(
        self.checkpoint_dir, f'checkpoint_epoch_{epoch+1}.pt')
    torch.save(checkpoint, checkpoint_path)

    # save as best model if applicable
    if is_best:
        best_model_path = os.path.join(
            self.checkpoint_dir, 'best_model.pt')
        torch.save(checkpoint, best_model_path)
        print(f'New best model saved with accuracy: {val_accuracy:.2f}%')

    return checkpoint_path

```

```

def load_checkpoint(self, checkpoint_path):
    """
    Load a checkpoint
    """
    print(f"Loading checkpoint from {checkpoint_path}")
    checkpoint = torch.load(checkpoint_path, map_location=self.device)

    self.model.load_state_dict(checkpoint['model_state_dict'])
    self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

    if 'scheduler_state_dict' in checkpoint and self.scheduler is not None:
        self.scheduler.load_state_dict(checkpoint['scheduler_state_dict'])

    start_epoch = checkpoint['epoch']
    loss = checkpoint.get('loss', 0)
    val_loss = checkpoint.get('val_loss', 0)
    val_accuracy = checkpoint.get('val_accuracy', 0)

    print(
        f"Checkpoint loaded - Epoch: {start_epoch}, Loss: {loss:.4f}, Val_
↳ Loss: {val_loss:.4f}, Val Accuracy: {val_accuracy:.2f}%"
    )

    return start_epoch

def train(self, num_epochs, start_epoch=0, log_to_mlflow=True,
↳ early_stopping_patience=None):
    """
    Train the model for multiple epochs

    Parameters:
        num_epochs: int
            Number of epochs to train for
        start_epoch: int
            Starting epoch (useful when resuming training)
        log_to_mlflow: bool
            Whether to log metrics to MLflow
        early_stopping_patience: int
            Number of epochs to wait for improvement before stopping
    """
    # start timer for total training run
    total_run_start_time = time.time()

    # reset performance tracking for a new training run
    if start_epoch == 0:
        self.total_training_time = 0
        self.total_epochs_completed = 0

```

```

# set mlflow experiment if provided
if log_to_mlflow and self.experiment_name:
    mlflow.set_experiment(self.experiment_name)

# initialize early stopping variables
if early_stopping_patience is not None:
    early_stopping_counter = 0
    best_val_loss = float('inf')

# start mlflow run if applicable
run_context = mlflow.start_run(
    run_name=f"{self.model.__class__.__name__}_{datetime.now().
↳strftime('%Y%m%d_%H%M%S')}" if log_to_mlflow else DummyContextManager()

# store previous learning rate to detect changes
prev_lr = self.optimizer.param_groups[0]['lr']

with run_context:
    # log parameters if using mlflow
    if log_to_mlflow:
        params = {
            "optimizer": self.optimizer.__class__.__name__,
            "learning_rate": self.optimizer.param_groups[0]['lr'],
            "num_epochs": num_epochs
        }
        # add model architecture params if available
        if hasattr(self.model, 'input_size'):
            params["input_size"] = self.model.input_size
        mlflow.log_params(params)

print("Starting training...")
for epoch in range(start_epoch, num_epochs):
    # train for one epoch
    print('')
    train_loss = self.train_epoch(epoch, num_epochs)
    print(
        f'Epoch [{epoch+1}/{num_epochs}], Loss: {train_loss:.4f}')

    # validate
    val_loss, val_accuracy = self.validate(epoch)

    # early stopping check for overfitting
    if early_stopping_patience is not None and val_loss is not None:
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            early_stopping_counter = 0

```

```

        else:
            early_stopping_counter += 1
            print(
                f'EarlyStopping counter: {early_stopping_counter}␣
↪out of {early_stopping_patience}')

            if early_stopping_counter >= early_stopping_patience:
                print(
                    f'Early stopping triggered after epoch␣
↪{epoch+1}')

                break

    # log metrics
    if log_to_mlflow:
        mlflow.log_metric("train_loss", train_loss, step=epoch)
        mlflow.log_metric("epoch_time", self.
↪history['epoch_times'][-1], step=epoch)
        if val_loss is not None:
            mlflow.log_metric("val_loss", val_loss, step=epoch)
        if val_accuracy is not None:
            mlflow.log_metric(
                "val_accuracy", val_accuracy, step=epoch)

    # print validation results
    if val_loss is not None and val_accuracy is not None:
        print(
            f'Validation - Epoch [{epoch+1}/{num_epochs}], Loss:␣
↪{val_loss:.4f}, Accuracy: {val_accuracy:.2f}%' )

    # check if this is the best model so far
    is_best = False
    if val_accuracy is not None and val_accuracy > self.
↪best_accuracy:
        self.best_accuracy = val_accuracy
        is_best = True

    # save checkpoint
    self.save_checkpoint(
        epoch, train_loss, val_loss, val_accuracy, is_best)

    # update learning rate if scheduler is provided
    if self.scheduler is not None:
        if isinstance(self.scheduler, torch.optim.lr_scheduler.
↪ReduceLRonPlateau):
            self.scheduler.step(val_loss)
        else:

```

```

        self.scheduler.step()

        # check if learning rate changed
        current_lr = self.optimizer.param_groups[0]['lr']
        if current_lr != prev_lr:
            print(
                f"Learning rate changed from {prev_lr:.6f} to_
↪{current_lr:.6f}")
            prev_lr = current_lr

        # calculate total run time
        total_run_time = time.time() - total_run_start_time

        print("Training complete!")

        # print performance summary
        print("\nTraining Performance Summary:")
        print(f"Total training time: {total_run_time:.2f} seconds_
↪({total_run_time/60:.2f} minutes)")
        print(f"Total epochs completed: {self.total_epochs_completed}")
        print(f"Average time per epoch: {self.total_training_time/self.
↪total_epochs_completed:.2f} seconds")
        print(f"Fastest epoch: {min(self.history['epoch_times']):.2f}_
↪seconds")
        print(f"Slowest epoch: {max(self.history['epoch_times']):.2f}_
↪seconds")

        # log performance metrics to MLflow
        if log_to_mlflow:
            mlflow.log_metric("total_training_time", total_run_time)
            mlflow.log_metric("avg_epoch_time", self.total_training_time/
↪self.total_epochs_completed)
            mlflow.log_metric("fastest_epoch_time", min(self.
↪history['epoch_times']))
            mlflow.log_metric("slowest_epoch_time", max(self.
↪history['epoch_times']))

        # final evaluation
        print("\nFinal Evaluation:")
        test_loss, test_accuracy = self.test()
        if test_loss is not None and test_accuracy is not None:
            print(f'Final Test Loss: {test_loss:.4f}')
            print(f'Final Test Accuracy: {test_accuracy:.2f}%')

        # log final metrics

```

```

        if log_to_mlflow:
            mlflow.log_metric("test_loss", test_loss)
            mlflow.log_metric("test_accuracy", test_accuracy)

        # save the final model
        model_timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        final_model_path = os.path.join(
            self.checkpoint_dir, f"final_model_{model_timestamp}.pt")
        torch.save(self.model.state_dict(), final_model_path)
        print(f"Final model saved as {final_model_path}\n")

        # log the model with mlflow
        if log_to_mlflow and self.input_size is not None:

            model_to_log = self.model.to('cpu')

            # create an input example
            sample_input = torch.rand(1, self.input_size).numpy()

            # Create custom pip requirements
            pip_requirements = [
                f"torch=={torch.__version__}",
                "torchvision",
                "mlflow"
            ]

            # log the model
            mlflow.pytorch.log_model(
                model_to_log,
                "model",
                input_example=sample_input,
                pip_requirements=pip_requirements
            )

            # move the model back to the original device
            self.model = self.model.to(self.device)

        # plot and save training curves
        self.plot_training_curves()

        return self.history

def plot_training_curves(self):
    """
    plot and save the training curves
    """
    if len(self.history['train_loss']) == 0:

```

```

        return

    # create plots directory
    plots_dir = os.path.join(self.checkpoint_dir, 'plots')
    os.makedirs(plots_dir, exist_ok=True)

    # plot training and validation loss
    plt.figure(figsize=(10, 6))
    epochs = range(1, len(self.history['train_loss']) + 1)
    plt.plot(epochs, self.history['train_loss'],
             'b-', label='Training Loss')

    if len(self.history['val_loss']) > 0:
        plt.plot(epochs, self.history['val_loss'],
                 'r-', label='Validation Loss')

    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.savefig(os.path.join(plots_dir, 'loss_curves.png')) # save the plot
↳ as a png file
    plt.tight_layout()
    plt.show()

    # plot validation accuracy if available
    if len(self.history['val_accuracy']) > 0:
        plt.figure(figsize=(10, 6))
        plt.plot(epochs, self.history['val_accuracy'], 'g-')
        plt.title('Validation Accuracy')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy (%)')
        plt.grid(True)
        plt.savefig(os.path.join(plots_dir, 'accuracy_curve.png')) # save
↳ the plot as a png file
        plt.tight_layout()
        plt.show()

    # plot epoch times
    if len(self.history['epoch_times']) > 0:
        plt.figure(figsize=(10, 6))
        plt.plot(epochs, self.history['epoch_times'], 'm-')
        plt.title('Epoch Training Times')
        plt.xlabel('Epoch')
        plt.ylabel('Time (seconds)')
        plt.grid(True)

```



```

        plt.savefig(os.path.join(plots_dir, 'epoch_times.png')) # save the
        ↪plot as a png file
        plt.tight_layout()
        plt.show()

# helper class for context management when not using mlflow
class DummyContextManager:
    def __enter__(self):
        return None

    def __exit__(self, *args):
        pass

```

1.6 Runner

```

[6]: # set random seed for reproducibility
torch.manual_seed(42)

# check for gpu availability
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using device: {device}\n')

# define transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# load mnist dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=True,
    download=True,
    transform=transform
)

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

# create data loaders
train_loader = DataLoader(
    train_dataset, batch_size=64, shuffle=True, pin_memory=True)

```

```

test_loader = DataLoader(test_dataset, batch_size=64,
                          shuffle=False, pin_memory=True)

# create model
input_size = 28 * 28 # MNIST images are 28x28 pixels
hidden_size = [128] # one hidden layer
output_size = 10 # 10 classes (digits 0-9)
model = Perceptron(
    input_size=input_size,
    hidden_size=hidden_size,
    output_size=output_size,
    softmax_dim=-1
)

# define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# create a learning rate scheduler for overfitting
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer,
    mode='min',
    factor=0.1,
    patience=3,
)

# create our trainer
trainer = Trainer(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    train_loader=train_loader,
    test_loader=test_loader, # using test set as validation set
    device=device,
    scheduler=scheduler,
    checkpoint_dir='./checkpoints',
    experiment_name="MNIST_Perceptron_Model"
)

# train the model
history = trainer.train(
    num_epochs=10,
    log_to_mlflow=True,
    early_stopping_patience=5 # enable early stopping for overfitting
)

# uncomment to resume training from a checkpoint

```

```
# don't forget to change the checkpoint path
# checkpoint_path = './checkpoints/checkpoint_epoch_2.pt'
# start_epoch = trainer.load_checkpoint(checkpoint_path)
# trainer.train(num_epochs=10, start_epoch=start_epoch)

# uncomment to use the best model for inference
# trainer.load_checkpoint('./checkpoints/best_model.pt')
```

Using device: cuda

Starting training...

```
Epoch [1/10], Step [100/938], Loss: 0.4322
Epoch [1/10], Step [200/938], Loss: 0.3024
Epoch [1/10], Step [300/938], Loss: 0.3595
Epoch [1/10], Step [400/938], Loss: 0.2423
Epoch [1/10], Step [500/938], Loss: 0.2881
Epoch [1/10], Step [600/938], Loss: 0.3294
Epoch [1/10], Step [700/938], Loss: 0.3723
Epoch [1/10], Step [800/938], Loss: 0.2187
Epoch [1/10], Step [900/938], Loss: 0.1286
Epoch time: 22.75 seconds
Epoch [1/10], Loss: 0.3607
Validation - Epoch [1/10], Loss: 0.2217, Accuracy: 93.28%
New best model saved with accuracy: 93.28%
```

```
Epoch [2/10], Step [100/938], Loss: 0.4463
Epoch [2/10], Step [200/938], Loss: 0.1599
Epoch [2/10], Step [300/938], Loss: 0.3721
Epoch [2/10], Step [400/938], Loss: 0.2644
Epoch [2/10], Step [500/938], Loss: 0.2724
Epoch [2/10], Step [600/938], Loss: 0.2156
Epoch [2/10], Step [700/938], Loss: 0.1613
Epoch [2/10], Step [800/938], Loss: 0.1414
Epoch [2/10], Step [900/938], Loss: 0.1897
Epoch time: 22.44 seconds
Epoch [2/10], Loss: 0.1879
Validation - Epoch [2/10], Loss: 0.1493, Accuracy: 95.53%
New best model saved with accuracy: 95.53%
```

```
Epoch [3/10], Step [100/938], Loss: 0.1784
Epoch [3/10], Step [200/938], Loss: 0.1165
Epoch [3/10], Step [300/938], Loss: 0.1515
Epoch [3/10], Step [400/938], Loss: 0.2542
Epoch [3/10], Step [500/938], Loss: 0.0659
Epoch [3/10], Step [600/938], Loss: 0.1272
Epoch [3/10], Step [700/938], Loss: 0.3140
Epoch [3/10], Step [800/938], Loss: 0.1646
```

Epoch [3/10], Step [900/938], Loss: 0.0224
Epoch time: 22.56 seconds
Epoch [3/10], Loss: 0.1383
Validation - Epoch [3/10], Loss: 0.1256, Accuracy: 96.14%
New best model saved with accuracy: 96.14%

Epoch [4/10], Step [100/938], Loss: 0.3256
Epoch [4/10], Step [200/938], Loss: 0.0810
Epoch [4/10], Step [300/938], Loss: 0.0935
Epoch [4/10], Step [400/938], Loss: 0.1180
Epoch [4/10], Step [500/938], Loss: 0.1764
Epoch [4/10], Step [600/938], Loss: 0.2124
Epoch [4/10], Step [700/938], Loss: 0.0451
Epoch [4/10], Step [800/938], Loss: 0.1489
Epoch [4/10], Step [900/938], Loss: 0.0503
Epoch time: 22.44 seconds
Epoch [4/10], Loss: 0.1098
Validation - Epoch [4/10], Loss: 0.1031, Accuracy: 96.82%
New best model saved with accuracy: 96.82%

Epoch [5/10], Step [100/938], Loss: 0.0236
Epoch [5/10], Step [200/938], Loss: 0.1259
Epoch [5/10], Step [300/938], Loss: 0.0292
Epoch [5/10], Step [400/938], Loss: 0.1083
Epoch [5/10], Step [500/938], Loss: 0.0232
Epoch [5/10], Step [600/938], Loss: 0.0444
Epoch [5/10], Step [700/938], Loss: 0.0331
Epoch [5/10], Step [800/938], Loss: 0.0711
Epoch [5/10], Step [900/938], Loss: 0.1074
Epoch time: 22.60 seconds
Epoch [5/10], Loss: 0.0960
EarlyStopping counter: 1 out of 5
Validation - Epoch [5/10], Loss: 0.1063, Accuracy: 96.51%

Epoch [6/10], Step [100/938], Loss: 0.0274
Epoch [6/10], Step [200/938], Loss: 0.0241
Epoch [6/10], Step [300/938], Loss: 0.0546
Epoch [6/10], Step [400/938], Loss: 0.1778
Epoch [6/10], Step [500/938], Loss: 0.0370
Epoch [6/10], Step [600/938], Loss: 0.0572
Epoch [6/10], Step [700/938], Loss: 0.0574
Epoch [6/10], Step [800/938], Loss: 0.1473
Epoch [6/10], Step [900/938], Loss: 0.0581
Epoch time: 22.53 seconds
Epoch [6/10], Loss: 0.0862
Validation - Epoch [6/10], Loss: 0.0915, Accuracy: 97.08%
New best model saved with accuracy: 97.08%

Epoch [7/10], Step [100/938], Loss: 0.1411
Epoch [7/10], Step [200/938], Loss: 0.0363
Epoch [7/10], Step [300/938], Loss: 0.0203
Epoch [7/10], Step [400/938], Loss: 0.1467
Epoch [7/10], Step [500/938], Loss: 0.0771
Epoch [7/10], Step [600/938], Loss: 0.0529
Epoch [7/10], Step [700/938], Loss: 0.0938
Epoch [7/10], Step [800/938], Loss: 0.0370
Epoch [7/10], Step [900/938], Loss: 0.0660
Epoch time: 22.47 seconds
Epoch [7/10], Loss: 0.0731
EarlyStopping counter: 1 out of 5
Validation - Epoch [7/10], Loss: 0.0983, Accuracy: 96.92%

Epoch [8/10], Step [100/938], Loss: 0.2122
Epoch [8/10], Step [200/938], Loss: 0.0411
Epoch [8/10], Step [300/938], Loss: 0.0945
Epoch [8/10], Step [400/938], Loss: 0.0278
Epoch [8/10], Step [500/938], Loss: 0.0855
Epoch [8/10], Step [600/938], Loss: 0.0477
Epoch [8/10], Step [700/938], Loss: 0.0758
Epoch [8/10], Step [800/938], Loss: 0.0171
Epoch [8/10], Step [900/938], Loss: 0.1193
Epoch time: 22.61 seconds
Epoch [8/10], Loss: 0.0685
EarlyStopping counter: 2 out of 5
Validation - Epoch [8/10], Loss: 0.0955, Accuracy: 97.20%
New best model saved with accuracy: 97.20%

Epoch [9/10], Step [100/938], Loss: 0.0374
Epoch [9/10], Step [200/938], Loss: 0.0376
Epoch [9/10], Step [300/938], Loss: 0.0417
Epoch [9/10], Step [400/938], Loss: 0.0091
Epoch [9/10], Step [500/938], Loss: 0.0976
Epoch [9/10], Step [600/938], Loss: 0.0440
Epoch [9/10], Step [700/938], Loss: 0.0942
Epoch [9/10], Step [800/938], Loss: 0.0112
Epoch [9/10], Step [900/938], Loss: 0.1761
Epoch time: 22.82 seconds
Epoch [9/10], Loss: 0.0621
EarlyStopping counter: 3 out of 5
Validation - Epoch [9/10], Loss: 0.0996, Accuracy: 96.87%

Epoch [10/10], Step [100/938], Loss: 0.0189
Epoch [10/10], Step [200/938], Loss: 0.0427
Epoch [10/10], Step [300/938], Loss: 0.0367
Epoch [10/10], Step [400/938], Loss: 0.0460
Epoch [10/10], Step [500/938], Loss: 0.1362

Epoch [10/10], Step [600/938], Loss: 0.0072
Epoch [10/10], Step [700/938], Loss: 0.0275
Epoch [10/10], Step [800/938], Loss: 0.0069
Epoch [10/10], Step [900/938], Loss: 0.0329
Epoch time: 22.29 seconds
Epoch [10/10], Loss: 0.0547
EarlyStopping counter: 4 out of 5
Validation - Epoch [10/10], Loss: 0.0969, Accuracy: 97.12%
Learning rate changed from 0.001000 to 0.000100
Training complete!

Training Performance Summary:
Total training time: 261.04 seconds (4.35 minutes)
Total epochs completed: 10
Average time per epoch: 22.55 seconds
Fastest epoch: 22.29 seconds
Slowest epoch: 22.82 seconds

Final Evaluation:
Final Test Loss: 0.0969
Final Test Accuracy: 97.12%
Final model saved as ./checkpoints\final_model_20250424_223502.pt



