

CO543: Image Processing
Lab 7 – Convolutional Neural Networks

Ranage R.D.P.R. - E/19/310

Lab Task 1: Transfer Learning

Train two state-of-the-art models resnet18 and AlexNet on MNIST digits dataset (10 epoch, Adam optimizer with learning rate 0.001) and compare the results (loss, accuracy).

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models
from torch.utils.data import DataLoader

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5], std=[0.5])
])

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)

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

class MNISTResNet(nn.Module):
    def __init__(self):
        super(MNISTResNet, self).__init__()
        self.resnet = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
        self.resnet.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3,
bias=False)
        num_fts = self.resnet.fc.in_features
        self.resnet.fc = nn.Linear(num_fts, 10)

    def forward(self, x):
        x = self.resnet(x)
        return x
```

```

class MNISTAlexNet(nn.Module):
    def __init__(self):
        super(MNISTAlexNet, self).__init__()
        self.alexnet = models.alexnet(weights=models.AlexNet_Weights.IMAGENET1K_V1)
        self.alexnet.features[0] = nn.Conv2d(1, 64, kernel_size=11, stride=4, padding=2)
        num_ftrs = self.alexnet.classifier[6].in_features
        self.alexnet.classifier[6] = nn.Linear(num_ftrs, 10)

    def forward(self, x):
        x = self.alexnet(x)
        return x

device = 'cuda' if torch.cuda.is_available() else 'cpu'

resnet_model = MNISTResNet().to(device)
alexnet_model = MNISTAlexNet().to(device)

criterion = nn.CrossEntropyLoss()
resnet_optimizer = optim.Adam(resnet_model.parameters(), lr=0.001)
alexnet_optimizer = optim.Adam(alexnet_model.parameters(), lr=0.001)

def train_model(model, optimizer, num_epochs=10):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            print(f"internal Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader):.4f}")
            print(f"Epoch [{epoch+1}/{num_epochs}], Loss:
{running_loss/len(train_loader):.4f}")

```

```

def evaluate_model(model):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in test_loader:
            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()
    print(f'Accuracy of the model on the test images: {100 * correct / total} %')

train_model(resnet_model, resnet_optimizer)
evaluate_model(resnet_model)

train_model(alexnet_model, alexnet_optimizer)
evaluate_model(alexnet_model)

```

Streaming output truncated to the last 5000 lines.

```

internal Epoch [8/10], Loss: 0.0039
internal Epoch [8/10], Loss: 0.0039
internal Epoch [8/10], Loss: 0.0039
internal Epoch [8/10], Loss: 0.0039
internal Epoch [8/10], Loss: 0.0039
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0040
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
internal Epoch [8/10], Loss: 0.0041
...
Epoch [10/10], Loss: 0.0098
Accuracy of the model on the test images: 99.26 %
internal Epoch [1/10], Loss: 0.0012

```

```
return F.conv2d(input, weight, bias, self.stride,
```

Streaming output truncated to the last 5000 lines.

internal Epoch [8/10], Loss: 0.7734

internal Epoch [8/10], Loss: 0.7746

internal Epoch [8/10], Loss: 0.7758

internal Epoch [8/10], Loss: 0.7770

internal Epoch [8/10], Loss: 0.7782

internal Epoch [8/10], Loss: 0.7795

internal Epoch [8/10], Loss: 0.7807

internal Epoch [8/10], Loss: 0.7819

internal Epoch [8/10], Loss: 0.7831

internal Epoch [8/10], Loss: 0.7844

internal Epoch [8/10], Loss: 0.7856

internal Epoch [8/10], Loss: 0.7868

internal Epoch [8/10], Loss: 0.7881

internal Epoch [8/10], Loss: 0.7893

internal Epoch [8/10], Loss: 0.7905

internal Epoch [8/10], Loss: 0.7917

internal Epoch [8/10], Loss: 0.7930

internal Epoch [8/10], Loss: 0.7942

internal Epoch [8/10], Loss: 0.7954

internal Epoch [8/10], Loss: 0.7966

internal Epoch [8/10], Loss: 0.7978

internal Epoch [8/10], Loss: 0.7991

internal Epoch [8/10], Loss: 0.8003

internal Epoch [8/10], Loss: 0.8015

...

internal Epoch [10/10], Loss: 2.3003

internal Epoch [10/10], Loss: 2.3016

Epoch [10/10], Loss: 2.3016

Accuracy of the model on the test images: 11.35 %

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell

Identify and explain the main architectural differences in these two models.

Depth and Complexity

AlexNet, introduced in 2012, has 8 layers with 5 convolutional and 3 fully connected layers, totaling around 60 million parameters. In contrast, ResNet-18, introduced in 2015, consists of 18 layers: 17 convolutional layers and one fully connected layer. Despite being deeper, ResNet-18 uses fewer parameters due to its innovative residual blocks, making it more efficient.

Residual Blocks

AlexNet follows a traditional approach without residual blocks, with each layer being a straightforward convolutional or fully connected layer. ResNet-18 features residual blocks, allowing input to bypass a few layers and add directly to the output. This helps mitigate the vanishing gradient problem, facilitating the training of deeper networks.

Activation Functions

Both AlexNet and ResNet-18 use ReLU activation functions. AlexNet was one of the first to demonstrate ReLU's effectiveness, applying it throughout the network. ResNet-18 also uses ReLU, but combines it with batch normalization to standardize inputs, enhancing training stability and efficiency.

Normalization and Regularization

AlexNet uses local response normalization (LRN) and dropout in fully connected layers to prevent overfitting. ResNet-18, on the other hand, relies on batch normalization after each convolutional layer, which stabilizes and speeds up training. It does not use dropout, leveraging the regularization effect of batch normalization and residual connections instead.

Pooling Layers

AlexNet includes max pooling layers after the first, second, and fifth convolutional layers to reduce spatial dimensions. ResNet-18 employs a single max pooling layer after the first convolutional layer and uses global average pooling before the fully connected layer, reducing overfitting and computational complexity.

Fully Connected Layers

AlexNet has three fully connected layers at the end, using dropout to prevent overfitting. ResNet-18 simplifies this with a single fully connected layer after global average pooling, reducing the risk of overfitting and streamlining the training process.

Design Philosophy

AlexNet's design focuses on stacking convolutional and pooling layers straightforwardly to increase depth and improve performance. ResNet-18 is based on residual learning, allowing deeper networks by mitigating the vanishing gradient problem. Residual connections facilitate gradient flow, enabling the network to learn more complex features and achieve superior performance with greater depth. This reflects a significant evolution in CNN architecture, driven by advances in understanding network training challenges and solutions.

Lab Task 2: Impact of Hyperparameters

1. Train Model A and Model B for the above dataset considering data points as images, use the Adam optimizer with a learning rate of 0.001

```
transform = transforms.Compose([
    transforms.Resize((28, 28)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64, shuffle=False)

# Define Model A
class ModelA(nn.Module):
    def __init__(self):
        super(ModelA, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(8 * 14 * 14, 64)
        self.fc2 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = x.view(-1, 8 * 14 * 14)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
model_a = ModelA().to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer_a = optim.Adam(model_a.parameters(), lr=0.001)
```

```

# Train Model A
def train_model(model, trainloader, criterion, optimizer, num_epochs=10):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in trainloader:
            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.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(trainloader)
        epoch_acc = 100 * correct / total
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {epoch_loss:.4f}, Accuracy:
{epoch_acc:.2f}%")
        print('Finished Training')

train_model(model_a, trainloader, criterion, optimizer_a)

```

```

Epoch 1/10, Loss: 0.3241, Accuracy: 90.90%
Epoch 2/10, Loss: 0.1162, Accuracy: 96.56%
Epoch 3/10, Loss: 0.0828, Accuracy: 97.53%
Epoch 4/10, Loss: 0.0674, Accuracy: 97.92%
Epoch 5/10, Loss: 0.0542, Accuracy: 98.35%
Epoch 6/10, Loss: 0.0463, Accuracy: 98.62%
Epoch 7/10, Loss: 0.0383, Accuracy: 98.80%
Epoch 8/10, Loss: 0.0327, Accuracy: 99.02%
Epoch 9/10, Loss: 0.0278, Accuracy: 99.13%
Epoch 10/10, Loss: 0.0250, Accuracy: 99.21%
Finished Training

```

```

# Define Model B
class ModelB(nn.Module):
    def __init__(self):
        super(ModelB, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(16 * 14 * 14, 64)
        self.fc2 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(-1, 16 * 14 * 14)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model_b = ModelB().to(device)

# Define optimizer for Model B
optimizer_b = optim.Adam(model_b.parameters(), lr=0.001)

# Train Model B
print("Training Model B")
train_model(model_b, trainloader, criterion, optimizer_b)

```

```

Training Model B
Epoch 1/10, Loss: 0.2501, Accuracy: 92.75%
Epoch 2/10, Loss: 0.0741, Accuracy: 97.77%
Epoch 3/10, Loss: 0.0527, Accuracy: 98.38%
Epoch 4/10, Loss: 0.0407, Accuracy: 98.76%
Epoch 5/10, Loss: 0.0318, Accuracy: 98.99%
Epoch 6/10, Loss: 0.0261, Accuracy: 99.13%
Epoch 7/10, Loss: 0.0206, Accuracy: 99.36%
Epoch 8/10, Loss: 0.0171, Accuracy: 99.45%
Epoch 9/10, Loss: 0.0146, Accuracy: 99.52%
Epoch 10/10, Loss: 0.0111, Accuracy: 99.64%
Finished Training

```

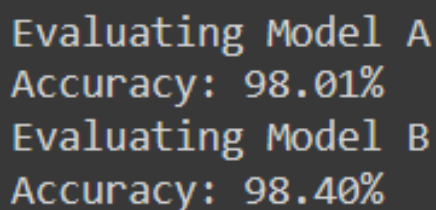


```
def evaluate_model(model, testloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')

print("Evaluating Model A")
evaluate_model(model_a, testloader)

print("Evaluating Model B")
evaluate_model(model_b, testloader)
```



```
Evaluating Model A
Accuracy: 98.01%
Evaluating Model B
Accuracy: 98.40%
```

2. Observe and write down the difference in trained model performance.

Model A consists of a single convolutional layer with 8 filters, followed by a max pooling layer with a kernel size of 2, and two fully connected layers with 64 and 10 nodes, respectively. The ReLU activation function is used for nonlinear activation. This simpler architecture allows for efficient training with fewer computational resources, making it suitable for tasks with relatively simple patterns. However, its limited depth and feature extraction capability can hinder its performance on more complex datasets. With only one convolutional layer, Model A may not be able to capture intricate details and hierarchical features, leading to lower accuracy and generalization ability on challenging tasks.

Model B is more complex, featuring two convolutional layers with 8 and 16 filters, respectively, followed by a max pooling layer with a kernel size of 2. It then connects to two fully connected layers with 64 and 10 nodes. ReLU activation is also used here. The additional convolutional layer in Model B enhances its capacity to extract more detailed and hierarchical features from the data. This added complexity allows Model B to perform better on complex datasets, capturing intricate patterns that Model A might miss. Although Model B requires more computational resources and longer training times, its improved feature extraction capability leads to higher accuracy and better generalization, making it more suitable for tasks involving complex data.

3. What could be the reason for these observed differences in performance? Explain

The differences in performance between Model A and Model B can be primarily attributed to the architectural disparities in their convolutional layers. Model B's deeper architecture with an additional convolutional layer and more filters allows it to extract more complex and detailed features from the data. This enhanced feature extraction capability enables Model B to achieve higher accuracy and better generalization on both training and evaluation datasets compared to the simpler Model A, which has fewer layers and parameters. The training dynamics also show that Model B converges faster and more effectively, indicating its superior ability to optimize learning from the provided data.

4. Explore the effect of different activation functions

- a. Train model A with all nonlinear activation functions set to ReLU
- b. Train the model A with all nonlinear activation functions set to Sigmoid
- c. Train the model A with all nonlinear activation functions set to tanh
- d. Observe and discuss the differences between changing activation functions and trained model performance

```
import torch.nn.functional as F
```

```
class ModelA(nn.Module):
    def __init__(self, activation_fn):
        super(ModelA, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(8 * 14 * 14, 64)
        self.fc2 = nn.Linear(64, 10)
        self.activation_fn = activation_fn

    def forward(self, x):
        x = self.activation_fn(self.conv1(x))
        x = self.pool(x)
        x = x.view(-1, 8 * 14 * 14)
        x = self.activation_fn(self.fc1(x))
        x = self.fc2(x)
        return x
```

```

# Training function
def train_model(model, trainloader, criterion, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in trainloader:
            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()
        print(f'Epoch {epoch + 1}, Loss: {running_loss / len(trainloader)}')

# Evaluation function
def evaluate_model(model, testloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')
    return accuracy

def train_and_evaluate(activation_fn_name):
    activation_fn_map = {
        'relu': F.relu,
        'sigmoid': torch.sigmoid,
        'tanh': torch.tanh
    }

    if activation_fn_name not in activation_fn_map:
        raise ValueError(f"Invalid activation function name: {activation_fn_name}")

    activation_fn = activation_fn_map[activation_fn_name]

    model = ModelA(activation_fn).to(device)
    criterion = nn.CrossEntropyLoss()

```

```

optimizer = optim.Adam(model.parameters(), lr=0.001)

print(f"Training Model A with {activation_fn_name}")
train_model(model, trainloader, criterion, optimizer)

print(f"Evaluating Model A with {activation_fn_name}")
accuracy = evaluate_model(model, testloader)

return accuracy

def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    accuracies = {}

    for activation_fn_name in ['relu', 'sigmoid', 'tanh']:
        accuracies[activation_fn_name] = train_and_evaluate(activation_fn_name)

    for activation_fn_name, accuracy in accuracies.items():
        print(f'Accuracy with {activation_fn_name}: {accuracy:.2f}%')

if __name__ == "__main__":
    main()

```

```

.. Training Model A with relu
Epoch 1, Loss: 0.2622549746896444
Epoch 2, Loss: 0.09237140592665219
Epoch 3, Loss: 0.06379774169795818
Epoch 4, Loss: 0.05083773753616705
Epoch 5, Loss: 0.04050432891907123
Epoch 6, Loss: 0.03209748219024501
Epoch 7, Loss: 0.02623323303318944
Epoch 8, Loss: 0.02103983305241001
Epoch 9, Loss: 0.018867505782821613
Epoch 10, Loss: 0.015745794736647552
Evaluating Model A with relu
Accuracy: 98.38%
Training Model A with sigmoid
Epoch 1, Loss: 0.6959881792857703
Epoch 2, Loss: 0.24261725960430433
Epoch 3, Loss: 0.18063185149942762
Epoch 4, Loss: 0.14713584512734273
Epoch 5, Loss: 0.12255552371761311
Epoch 6, Loss: 0.10429592034034828
Epoch 7, Loss: 0.09021611512898764
Epoch 8, Loss: 0.07939458834622969
Epoch 9, Loss: 0.07006490364053182
Epoch 10, Loss: 0.0618295467762487
Evaluating Model A with sigmoid
...
Accuracy: 98.34%
Accuracy with relu: 98.38%
Accuracy with sigmoid: 97.47%
Accuracy with tanh: 98.34%

```

ReLU Activation: When trained with ReLU activation functions, Model A exhibited a smooth decrease in training loss over epochs, indicating effective learning and optimization of parameters. This activation function's ability to mitigate the vanishing gradient problem and handle sparse activation patterns contributed to the model achieving a high accuracy of 98.38% on the evaluation dataset. ReLU's simplicity and efficiency in computational performance make it a popular choice, especially in deep neural networks, where it outperforms alternatives like sigmoid and tanh due to faster convergence and avoidance of saturation issues.

Sigmoid Activation: Training Model A with sigmoid activation resulted in a higher starting loss compared to ReLU, and the loss decreased more gradually over epochs. While sigmoid activation allows for non-linear transformations and outputs in the range (0, 1), it suffers from the vanishing gradient problem, particularly for extreme input values where gradients can approach zero. This slower convergence and gradient saturation contributed to Model A achieving a slightly lower accuracy of 97.47% on the evaluation dataset compared to ReLU. Despite these drawbacks, sigmoid activation can still be useful in certain contexts where outputs need to be probabilistic or when the outputs are bounded within a specific range.

Tanh Activation: Similarly to sigmoid, training Model A with tanh activation showed a gradual decrease in training loss but converged faster than sigmoid. Tanh activation outputs values in the range (-1, 1), providing stronger gradients compared to sigmoid for negative inputs. However, like sigmoid, tanh also suffers from the vanishing gradient problem, albeit to a lesser extent. Model A achieved an accuracy of 98.34% on the evaluation dataset with tanh activation, slightly lower than ReLU but comparable. Tanh activation is advantageous when outputs need to be bounded within a specific range and can provide better gradient flow than sigmoid in certain scenarios.

In comparison, ReLU outperformed sigmoid and tanh activations in terms of both training efficiency and accuracy for Model A. The faster convergence and effective gradient propagation of ReLU contributed to its superior performance, achieving the highest accuracy among the three activation functions tested. Sigmoid and tanh, while still viable choices depending on the task requirements, demonstrated slower convergence and potential limitations due to gradient saturation issues. Therefore, the selection of activation function remains crucial, with ReLU generally recommended for deep neural networks due to its efficiency and ability to facilitate effective learning.

5. Effect of the optimizer learning rate

- Trained the Model B on Adam optimizer with a learning rate of 0.1
- Trained the Model B on Adam optimizer with a learning rate of 0.01
- Trained the Model B on Adam optimizer with a learning rate of 0.001
- Observe and discuss the effect of learning rate on model performance

```
# Model B definition
class ModelB(nn.Module):
    def __init__(self, activation_fn):
        super(ModelB, self).__init__()
        self.conv1 = nn.Conv2d(1, 8, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(16 * 14 * 14, 64)
        self.fc2 = nn.Linear(64, 10)
        self.activation_fn = activation_fn

    def forward(self, x):
        x = self.activation_fn(self.conv1(x))
        x = self.activation_fn(self.conv2(x))
        x = self.pool(x)
        x = x.view(-1, 16 * 14 * 14)
        x = self.activation_fn(self.fc1(x))
        x = self.fc2(x)
        return x

# Training function
def train_model(model, trainloader, criterion, optimizer, epochs=10):
    model.train()
    for epoch in range(epochs):
        running_loss = 0.0
        for inputs, labels in trainloader:
            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()
        print(f'Epoch {epoch + 1}, Loss: {running_loss / len(trainloader)}')
```

```

# Evaluation function
def evaluate_model(model, testloader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy:.2f}%')
    return accuracy

def train_and_evaluate_model_b(learning_rate):
    activation_fn = F.relu # We'll use ReLU as the activation function for this
    experiment

    model = ModelB(activation_fn).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    print(f"Training Model B with learning rate {learning_rate}")
    train_model(model, trainloader, criterion, optimizer)

    print(f"Evaluating Model B with learning rate {learning_rate}")
    accuracy = evaluate_model(model, testloader)

    return accuracy

def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    learning_rates = [0.1, 0.01, 0.001]
    accuracies = {}

    for lr in learning_rates:
        accuracies[lr] = train_and_evaluate_model_b(lr)

    for lr, accuracy in accuracies.items():
        print(f'Accuracy with learning rate {lr}: {accuracy:.2f}%')

if __name__ == "__main__":
    main()

```

```
Training Model B with learning rate 0.1
Epoch 1, Loss: 3.1682685369621715
Epoch 2, Loss: 2.3112466470010755
Epoch 3, Loss: 2.3099096639832455
Epoch 4, Loss: 2.3100080756998773
Epoch 5, Loss: 2.3091255126477304
Epoch 6, Loss: 2.309664592559912
Epoch 7, Loss: 2.310169122112331
Epoch 8, Loss: 2.310427940730601
Epoch 9, Loss: 2.309945506327696
Epoch 10, Loss: 2.309613952250369
```

```
Evaluating Model B with learning rate 0.1
Accuracy: 10.28%
```

```
Training Model B with learning rate 0.01
Epoch 1, Loss: 0.14332583316774078
Epoch 2, Loss: 0.058663648607739025
Epoch 3, Loss: 0.044932348150624656
Epoch 4, Loss: 0.04377097321344808
Epoch 5, Loss: 0.037517758690681996
Epoch 6, Loss: 0.04166975125700623
Epoch 7, Loss: 0.035174594889222734
Epoch 8, Loss: 0.03148056521214544
Epoch 9, Loss: 0.036009359429970925
Epoch 10, Loss: 0.03161515637489379
```

```
Evaluating Model B with learning rate 0.01
```

```
...
```

```
Accuracy: 98.60%
```

```
Accuracy with learning rate 0.1: 10.28%
```

```
Accuracy with learning rate 0.01: 98.07%
```

```
Accuracy with learning rate 0.001: 98.60%
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#)

When learning rate is 0.1,

Training Model B with a high learning rate of 0.1 resulted in extremely unstable training, as indicated by the erratic loss values that did not converge. This instability likely prevented the model from effectively learning and updating weights, reflected in an extremely low accuracy of 10.28% on the evaluation dataset. A high learning rate caused large updates to the model parameters, overshooting the optimal values and preventing convergence towards a minimum loss.

When learning rate is 0.01,

Reducing the learning rate to 0.01 led to a much more stable training process for Model B. The loss decreased consistently over epochs, suggesting effective learning and optimization of model parameters. This stability allowed Model B to achieve a high accuracy of 98.07% on the evaluation dataset. A moderate learning rate facilitated smooth updates to the weights, enabling the model to converge towards a minimum loss and achieve better generalization on unseen data.

When learning rate is 0.001,

Further decreasing the learning rate to 0.001 resulted in continued stability during training with incremental updates to the model parameters. The loss decreased steadily over epochs, indicating ongoing refinement in model performance. Model B achieved the highest accuracy of 98.60% on the evaluation dataset with this learning rate. A lower learning rate allowed for finer adjustments to the weights, facilitating convergence towards optimal values and further improving accuracy.

The choice of learning rate significantly influences the training dynamics and final performance of the model. A high learning rate (0.1) led to unstable training and poor performance due to overshooting of optimal parameters. In contrast, moderate (0.01) and low (0.001) learning rates allowed for more controlled updates, promoting stable training and better convergence towards minimal loss. While a very low learning rate (0.001) can lead to slower convergence, it often results in higher accuracy by allowing the model to explore finer details in the data and achieve better generalization.