24Fall Advanced Control for Robotics

miniProject: Train a MLP model based on MNIST dataset

Name: Siyuan Wang SID: 12443028

Prepare MNIST Dataset

In [1]:
```
import torch
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter


if torch.cuda.is_available():
    print("CUDA is available. Number of CUDA devices:", torch.cuda.device_count())
    for device_id in range(torch.cuda.device_count()):
        print("Device ID:", device_id, "Device name:", torch.cuda.get_device_name(d
else:
    print("CUDA is not available. Using CPU instead.")
```

```
CUDA is available. Number of CUDA devices: 2
Device ID: 0 Device name: NVIDIA GeForce RTX 4090
Device ID: 1 Device name: NVIDIA GeForce RTX 4090
```

In [2]:
```
device = torch.device("cuda:1")
device
```

Out[2]:
```
device(type='cuda', index=1)
```

In [3]:
```
# Load the MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=6

test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False, num_workers=
```

In [4]:
```
for batch_idx, data in enumerate(train_loader):
    # Check data is on which device
    if data[0].is_cuda:
        print(f"Batch {batch_idx} is on CUDA (GPU)")
    else:
        print(f"Batch {batch_idx} is on CPU")
    # Only check the first batch
    break
```

```
Batch 0 is on CPU
```

In [20]:
```
import matplotlib.pyplot as plt
# Function to show images
def show_images(images, labels):
    # Create a grid of 8x8 images
    images, labels = images.to("cpu"), labels.to("cpu")
    fig, axes = plt.subplots(8, 8, figsize=(8, 8))
    axes = axes.flatten()

    # Loop through each image and display it
    for img, ax, label in zip(images, axes, labels):
        img = img.numpy().squeeze()  # Convert to numpy and remove unnecessary dime
        ax.imshow(img, cmap='gray')
        ax.set_title(f'Label: {label}')
        ax.axis('off')

    plt.tight_layout()
    plt.show()

# Get a batch of images and labels
data_iter = iter(train_loader)
images, labels = next(data_iter)

# Show the first 64 images and labels
show_images(images[:64], labels[:64])
```

Construct a MLP (with dropouts), Criterion & Optimizer

In [6]:
```
import torch.nn as nn

class SimpleMLP(nn.Module):
    def __init__(self, dropout_prob=0.2):
        super().__init__()
        self.fc1 = nn.Linear(28*28, 256)
        self.fc2 = nn.Linear(256, 64)
        self.fc3 = nn.Linear(64, 10)
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, x):
        x = x.view(-1, 28*28) # Flatten the input
        x = self.dropout(self.fc1(x))
        x = torch.relu(x)
        x = self.dropout(self.fc2(x))
        x = torch.relu(x)
```

```
        x = self.fc3(x)
        return x

my_model = SimpleMLP().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(my_model.parameters(), lr=0.001, momentum=0.9, dampening=0.1,
```

Define utility functions

In [7]:
```
# Declare tensorboard SummaryWriter
writer = SummaryWriter('./log')
```

In [8]:
```
def train(model, train_loader, criterion, optimizer, epochs=1, test_every_n_epochs=
    model.train()  # Set the nn.module model to "train" mode
    for epoch_0 in range(epochs):
        training_loss = 0.0
        epoch = epoch_0 + 1
        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()
            training_loss += loss.item()
        print('Epoch {}, Loss {}'.format(epoch, training_loss/len(train_loader)))
        writer.add_scalar('training_loss', training_loss, epoch)
        if test_every_n_epochs != 0 and epoch %test_every_n_epochs == 0:
            test(model=model, test_loader=test_loader, clean_test=False, epoch=epoc
```

In [9]:
```
def test(model, test_loader, clean_test=True, epoch=None):
    model.eval()  # This Function is NEEDED to Call before Testing!!!
    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, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')
    if clean_test == False:
        writer.add_scalar('testing_accuracy', accuracy, epoch)
    else:
        show_images(images[:64], predicted[:64])
```

In [15]:
```
def save_model(model):
    import datetime
    now = datetime.datetime.now()
    torch.save(model, "my_simpleMLP_model_{}_{}_{}_{}_{}_{}".format(now.year, now.m
```

Train, then Test

- Train this simpleMLP for 120 epoches

In [11]:
```
train(model=my_model, train_loader=train_loader, criterion=criterion, optimizer=opt
```

```
Epoch 1, Loss 1.2745358663390693
Epoch 2, Loss 0.5306923572919262
Epoch 3, Loss 0.4218244421869707
Epoch 4, Loss 0.3625748970631216
Epoch 5, Loss 0.3224547675280556
Epoch 6, Loss 0.2946550121455431
Epoch 7, Loss 0.26887119128537584
Epoch 8, Loss 0.2518350677465452
Epoch 9, Loss 0.23188821183465946
Epoch 10, Loss 0.21739036338860546
Accuracy: 95.0%
Epoch 11, Loss 0.15345677713563702
Epoch 12, Loss 0.14101509250867278
Epoch 13, Loss 0.13155750380848835
Epoch 14, Loss 0.12319043572189839
Epoch 15, Loss 0.11610287907066694
Epoch 16, Loss 0.1098599271612492
Epoch 17, Loss 0.1039449668485425
Epoch 18, Loss 0.09854012502949121
Epoch 19, Loss 0.09401631361937948
Epoch 20, Loss 0.08951483455115258
Accuracy: 96.94%
Epoch 21, Loss 0.0856638228270545
Epoch 22, Loss 0.08195361445870386
Epoch 23, Loss 0.07850473792627771
Epoch 24, Loss 0.07558456829179135
Epoch 25, Loss 0.07255330646454271
Epoch 26, Loss 0.06973504316685264
Epoch 27, Loss 0.06681222720061744
Epoch 28, Loss 0.06452211366096602
Epoch 29, Loss 0.06240249512943505
Epoch 30, Loss 0.060391329032783984
Accuracy: 97.51%
Epoch 31, Loss 0.05758447894778873
Epoch 32, Loss 0.05599646323046752
Epoch 33, Loss 0.054339879721617586
Epoch 34, Loss 0.05240819791405361
Epoch 35, Loss 0.050846304181021956
Epoch 36, Loss 0.04940502497224586
Epoch 37, Loss 0.047559944079019256
Epoch 38, Loss 0.04637284032893238
Epoch 39, Loss 0.04495803004146607
Epoch 40, Loss 0.043995254451154805
Accuracy: 97.77%
Epoch 41, Loss 0.04242001689656743
Epoch 42, Loss 0.04097334823972269
Epoch 43, Loss 0.04005363428377425
Epoch 44, Loss 0.038865463056319605
Epoch 45, Loss 0.037266074954075364
Epoch 46, Loss 0.03642100310757327
Epoch 47, Loss 0.035674459484618296
Epoch 48, Loss 0.034791985805184125
Epoch 49, Loss 0.0339335270984563
Epoch 50, Loss 0.032988683119523465
Accuracy: 97.75%
Epoch 51, Loss 0.03227342386642642
```

```
Epoch 52, Loss 0.031002049476577483
Epoch 53, Loss 0.030344579851866435
Epoch 54, Loss 0.02958311679820691
Epoch 55, Loss 0.029181425374295158
Epoch 56, Loss 0.02800323098025688
Epoch 57, Loss 0.027463990745188266
Epoch 58, Loss 0.027036524202271518
Epoch 59, Loss 0.02625029459263065
Epoch 60, Loss 0.025748705011613762
Accuracy: 97.96%
Epoch 61, Loss 0.025120113608933715
Epoch 62, Loss 0.02440382184216014
Epoch 63, Loss 0.02401380323152592
Epoch 64, Loss 0.02324461374917864
Epoch 65, Loss 0.02291446145443218
Epoch 66, Loss 0.022436097074401324
Epoch 67, Loss 0.021725203108806004
Epoch 68, Loss 0.02149205542440568
Epoch 69, Loss 0.02113578587384231
Epoch 70, Loss 0.020572820931289797
Accuracy: 97.99%
Epoch 71, Loss 0.02025578220423037
Epoch 72, Loss 0.019744785890024878
Epoch 73, Loss 0.01939574130294499
Epoch 74, Loss 0.019006176270357868
Epoch 75, Loss 0.018822770811300148
Epoch 76, Loss 0.01847276234078898
Epoch 77, Loss 0.018003349055050574
Epoch 78, Loss 0.017842258906800117
Epoch 79, Loss 0.017438452087912653
Epoch 80, Loss 0.01699026785334195
Accuracy: 98.0%
Epoch 81, Loss 0.01678298638976201
Epoch 82, Loss 0.016483253539492215
Epoch 83, Loss 0.016282963694563744
Epoch 84, Loss 0.016158642978328013
Epoch 85, Loss 0.01564987084809949
Epoch 86, Loss 0.015614491004346889
Epoch 87, Loss 0.015375503083951136
Epoch 88, Loss 0.015156840221700208
Epoch 89, Loss 0.014796252039929868
Epoch 90, Loss 0.014754040783811321
Accuracy: 98.06%
Epoch 91, Loss 0.014497358603889484
Epoch 92, Loss 0.014394512675059186
Epoch 93, Loss 0.013977682221058343
Epoch 94, Loss 0.013999916700103174
Epoch 95, Loss 0.013791705434422083
Epoch 96, Loss 0.013692079453981284
Epoch 97, Loss 0.013315273711690183
Epoch 98, Loss 0.013371427839464033
Epoch 99, Loss 0.013226612472036945
Epoch 100, Loss 0.013085783697661758
Accuracy: 98.0%
Epoch 101, Loss 0.01271000111002479
Epoch 102, Loss 0.01273333241296525
```

```
Epoch 103, Loss 0.012559231153184445
Epoch 104, Loss 0.012377213659524513
Epoch 105, Loss 0.012436068485374811
Epoch 106, Loss 0.012112650568961903
Epoch 107, Loss 0.012059779445849744
Epoch 108, Loss 0.011974496287735564
Epoch 109, Loss 0.011874931598596497
Epoch 110, Loss 0.011706056189102762
Accuracy: 98.15%
Epoch 111, Loss 0.011594086784426607
Epoch 112, Loss 0.011567216901940658
Epoch 113, Loss 0.011348633448467222
Epoch 114, Loss 0.01131533745138634
Epoch 115, Loss 0.011260213191881141
Epoch 116, Loss 0.011093041959972278
Epoch 117, Loss 0.011085009362835532
Epoch 118, Loss 0.010964535615607493
Epoch 119, Loss 0.010847478428471294
Epoch 120, Loss 0.01081454559401331
Accuracy: 98.09%
Epoch 121, Loss 0.010735512503595558
Epoch 122, Loss 0.010687517224878533
Epoch 123, Loss 0.01055954208772288
Epoch 124, Loss 0.010501546505615234
Epoch 125, Loss 0.010379721942136306
Epoch 126, Loss 0.01037178071824385
Epoch 127, Loss 0.010256064620411603
Epoch 128, Loss 0.010199596998525034
Epoch 129, Loss 0.010187893623433041
Epoch 130, Loss 0.010117735604131059
Accuracy: 98.01%
Epoch 131, Loss 0.00996940541724021
Epoch 132, Loss 0.01006573664680022
Epoch 133, Loss 0.009963432686894374
Epoch 134, Loss 0.009856528590750029
Epoch 135, Loss 0.00982483230271329
Epoch 136, Loss 0.009826498444632454
Epoch 137, Loss 0.009764125738948234
Epoch 138, Loss 0.009661411296880083
Epoch 139, Loss 0.009588021300618909
Epoch 140, Loss 0.009559761832601257
Accuracy: 98.16%
Epoch 141, Loss 0.009508923099843908
Epoch 142, Loss 0.009581976658369261
Epoch 143, Loss 0.009450852872430485
Epoch 144, Loss 0.009436063242948123
Epoch 145, Loss 0.00937065602551256
Epoch 146, Loss 0.009313561480744545
Epoch 147, Loss 0.009263883612907009
Epoch 148, Loss 0.009370472146518854
Epoch 149, Loss 0.009251679632383814
Epoch 150, Loss 0.009229945472199847
Accuracy: 98.11%
```

Save the trained model

In [16]:
```
save_model(my_model)
```

Through observing the logged tensorboard data, the training loss is still decreasing, while the testing accuracy stays around.

Perhaps the limit of this model has been reached, we encountered an overfitting!

- Test this model, after training for 150 epochs

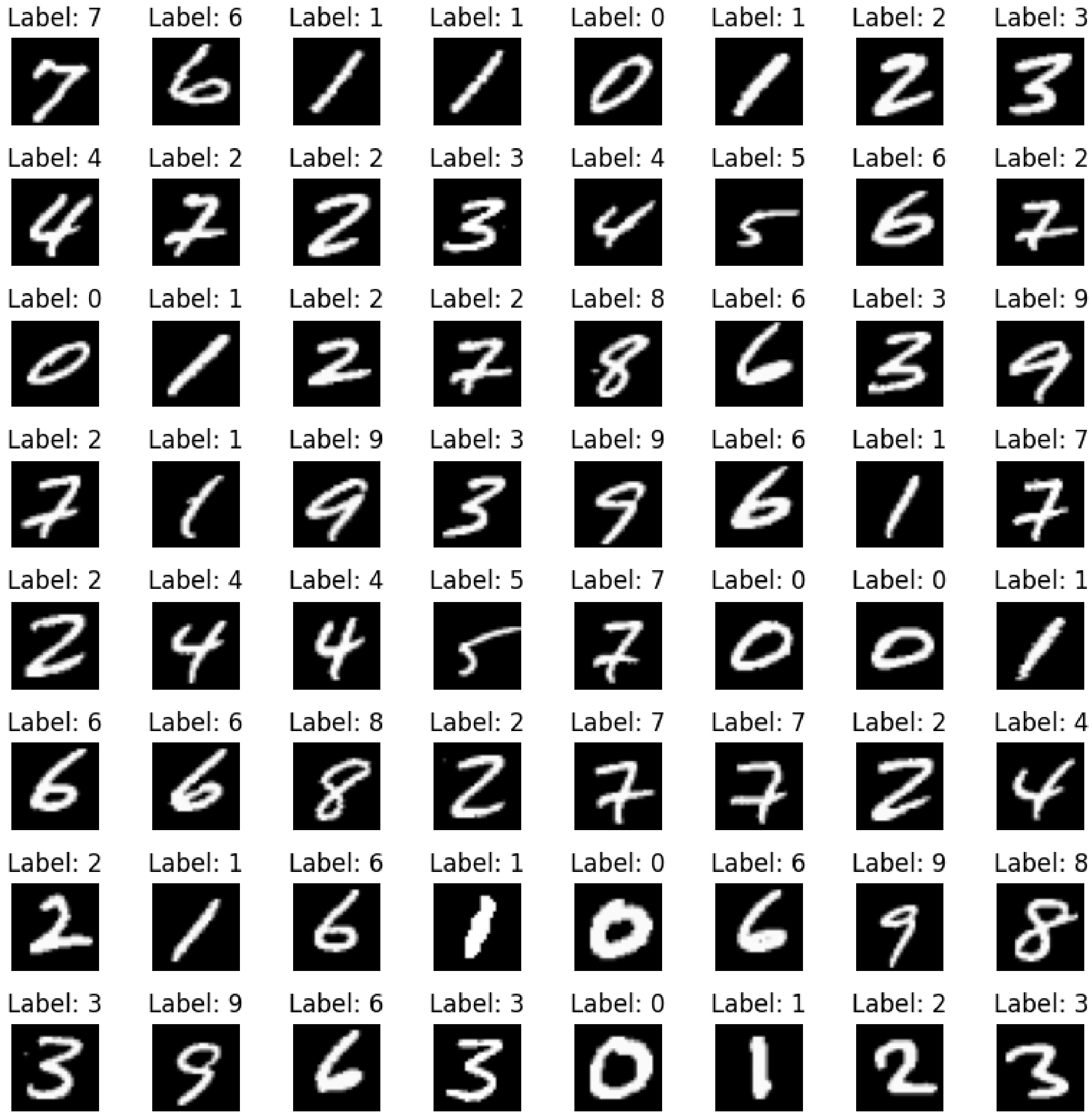
In [18]:
```
model_path = 'my_simpleMLP_model_2024_9_29_16_6_41_.pth'
loaded_model = torch.load(model_path, map_location=device)
loaded_model.to(device)
```

```
/tmp/ipykernel_2127915/3600639568.py:2: FutureWarning: You are using `torch.load` wi
th `weights_only=False` (the current default value), which uses the default pickle m
odule implicitly. It is possible to construct malicious pickle data which will execu
te arbitrary code during unpickling (See https://github.com/pytorch/pytorch/blob/mai
n/SECURITY.md#untrusted-models for more details). In a future release, the default v
alue for `weights_only` will be flipped to `True`. This limits the functions that co
uld be executed during unpickling. Arbitrary objects will no longer be allowed to be
loaded via this mode unless they are explicitly allowlisted by the user via `torch.s
erialization.add_safe_globals`. We recommend you start setting `weights_only=True` f
or any use case where you don't have full control of the loaded file. Please open an
issue on GitHub for any issues related to this experimental feature.
  loaded_model = torch.load(model_path, map_location=device)
```

Out[18]:
```
SimpleMLP(
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [21]:
```
test(model=my_model, test_loader=test_loader)
```

```
Accuracy: 98.11%
```


Label: 7
Label: 6
Label: 1
Label: 1
Label: 0
Label: 1
Label: 2
Label: 3
Label: 4
Label: 2
Label: 2
Label: 3
Label: 4
Label: 5
Label: 6
Label: 2
Label: 0
Label: 1
Label: 2
Label: 2
Label: 8
Label: 6
Label: 3
Label: 9
Label: 2
Label: 1
Label: 9
Label: 3
Label: 9
Label: 6
Label: 1
Label: 7
Label: 2
Label: 4
Label: 4
Label: 5
Label: 7
Label: 0
Label: 0
Label: 1
Label: 6
Label: 6
Label: 8
Label: 2
Label: 7
Label: 7
Label: 2
Label: 4
Label: 2
Label: 1
Label: 6
Label: 1
Label: 0
Label: 6
Label: 9
Label: 8
Label: 3
Label: 9
Label: 6
Label: 3
Label: 0
Label: 1
Label: 2
Label: 3