Start by importing necessary packages

You will begin by importing necessary libraries for this notebook. Run the cell below to do so.

PyTorch and Intro to Training

```
!pip install thop
import math
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import thop
import matplotlib.pyplot as plt
from tqdm import tqdm
import time
```

```
Collecting thop
  Downloading thop-0.1.1.post2209072238-py3-none-any.whl.metadata (2.7
kB)
Requirement already satisfied: torch in
/usr/local/lib/python3.11/dist-packages (from thop) (2.5.1+cu121)
Requirement already satisfied: filelock in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (4.12.2)
Requirement already satisfied: networkx in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.5)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (2024.10.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /usr/local/lib/python3.11/dist-packages (from torch->thop)
(12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
```

```
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/usr/local/lib/python3.11/dist-packages (from torch->thop)
(12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (12.1.105)
Requirement already satisfied: triton==3.1.0 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in
/usr/local/lib/python3.11/dist-packages (from torch->thop) (1.13.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/usr/local/lib/python3.11/dist-packages (from nvidia-cusolver-
cu12==11.4.5.107->torch->thop) (12.6.85)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch-
>thop) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch->thop)
(3.0.2)
Downloading thop-0.1.1.post2209072238-py3-none-any.whl (15 kB)
Installing collected packages: thop
Successfully installed thop-0.1.1.post2209072238
```

Checking the torch version and CUDA access

Let's start off by checking the current torch version, and whether you have CUDA availablity.

```
print("torch is using version:", torch.__version__, "with CUDA=",
torch.cuda.is_available())
```

```
torch is using version: 2.5.1+cu121 with CUDA= True
```

By default, you will see CUDA = False, meaning that the Colab session does not have access to a GPU. To remedy this, click the Runtime menu on top and select "Change Runtime Type", then select "T4 GPU".

Re-run the import cell above, and the CUDA version / check. It should show now CUDA = True

Sometimes in Colab you get a message that your Session has crashed, if that happens you need to go to the Runtime menu on top and select "Restart session".

You won't be using the GPU just yet, but this prepares the instance for when you will.

Please note that the GPU is a scarce resource which may not be available at all time. Additionally, there are also usage limits that you may run into (although not likely for this assignment). When that happens you need to try again later/next day/different time of the day. Another reason to start the assignment early!

A Brief Introduction to PyTorch

PyTorch, or torch, is a machine learning framework developed my Facebook AI Research, which competes with TensorFlow, JAX, Caffe and others.

Roughly speaking, these frameworks can be split into dynamic and static defintion frameworks.

Static Network Definition: The architecture and computation flow are defined simultaneously. The order and manner in which data flows through the layers are fixed upon definition. These frameworks also tend to declare parameter shapes implicitly via the compute graph. This is typical of TensorFlow and JAX.

Dynamic Network Definition: The architecture (layers/modules) is defined independently of the computation flow, often during the object's initialization. This allows for dynamic computation graphs where the flow of data can change during runtime based on conditions. Since the network exists independent of the compute graph, the parameter shapes must be declared explitly. PyTorch follows this approach.

All ML frameworks support automatic differentiation, which is necessary to train a model (i.e. perform back propagation).

Let's consider a typical pytorch module. Such modules will inherit from the torch.nn.Module class, which provides many built in functions such as a wrapper for `__call__`, operations to move the module between devices (e.g. `cuda()`, `cpu()`), data-type conversion (e.g. `half()`, `float()`), and parameter and child management (e.g. `state_dict()`, `parameters()`).

```
# inherit from torch.nn.Module
class MyModule(nn.Module):
  # constructor called upon creation
  def __init__(self):
    # the module has to initialize the parent first, which is what
sets up the wrapper behavior
    super().__init__()

    # we can add sub-modules and parameters by assigning them to self
    self.my_param = nn.Parameter(torch.zeros(4,8)) # this is how you
define a raw parameter of shape 4x5
    self.my_sub_module = nn.Linear(8,12)       # this is how you
define a linear layer (tensorflow calls them Dense) of shape 8x12

    # we can also add lists of modules, for example, the sequential
layer
    self.net = nn.Sequential(  # this layer type takes in a collection
of modules rather than a list
        nn.Linear(4,4),
```

```
        nn.Linear(4,8),
        nn.Linear(8,12)
    )

    # the above when calling self.net(x), will execute each module in
the order they appear in a list
    # it would be equivelent to x = self.net[2](self.net[1]
(self.net[0](x)))

    # you can also create a list that doesn't execute
    self.net_list = nn.ModuleList([
        nn.Linear(7,7),
        nn.Linear(7,9),
        nn.Linear(9,14)
    ])

    # sometimes you will also see constant variables added to the
module post init
    foo = torch.Tensor([4])
    self.register_buffer('foo', foo) # buffers allow .to(device, type)
to apply

  # let's define a forward function, which gets executed when calling
the module, and defines the forward compute graph
  def forward(self, x):

    # if x is of shape Bx4
    h1 =  x @ self.my_param # tensor-tensor multiplication uses the @
symbol
    # then h1 is now shape Bx8, because my_param is 4x8... 2x4 * 4x8 =
2x8

    h1 = self.my_sub_module(h1) # you execute a sub-module by calling
it
    # now, h1 is of shape Bx12, because my_sub_module was a 8x12
matrix

    h2 = self.net(x)
    # similarly, h2 is of shape Bx12, because that's the output of the
sequence
    # Bx4 -(4x4)-> Bx4 -(4x8)-> Bx8 -(8x12)-> Bx12

    # since h1 and h2 are the same shape, they can be added together
element-wise
    return h1 + h2
```

Then you can instantiate the module and perform a forward pass by calling it.

```
# create the module
module = MyModule()
```

```python
# you can print the module to get a high-level summary of it
print("=== printing the module ===")
print(module)
print()
# notice that the sub-module name is in parenthesis, and so are the
list indicies

# let's view the shape of one of the weight tensors
print("my_sub_module weight tensor shape=",
module.my_sub_module.weight.shape)
# the above works because nn.Linear has a member called .weight
and .bias
# to view the shape of my_param, you would use module.my_param
# and to view the shape of the 2nd elment in net_list, you would use
module.net_list[1].weight

# you can iterate through all of the parameters via the state dict
print()
print("=== Listing parameters from the state_dict ===")
for key,value in module.state_dict().items():
  print(f"{key}: {value.shape}")
```

```
=== printing the module ===
MyModule(
  (my_sub_module): Linear(in_features=8, out_features=12, bias=True)
  (net): Sequential(
    (0): Linear(in_features=4, out_features=4, bias=True)
    (1): Linear(in_features=4, out_features=8, bias=True)
    (2): Linear(in_features=8, out_features=12, bias=True)
  )
  (net_list): ModuleList(
    (0): Linear(in_features=7, out_features=7, bias=True)
    (1): Linear(in_features=7, out_features=9, bias=True)
    (2): Linear(in_features=9, out_features=14, bias=True)
  )
)

my_sub_module weight tensor shape= torch.Size([12, 8])

=== Listing parameters from the state_dict ===
my_param: torch.Size([4, 8])
foo: torch.Size([1])
my_sub_module.weight: torch.Size([12, 8])
my_sub_module.bias: torch.Size([12])
net.0.weight: torch.Size([4, 4])
net.0.bias: torch.Size([4])
net.1.weight: torch.Size([8, 4])
net.1.bias: torch.Size([8])
net.2.weight: torch.Size([12, 8])
```

```
net.2.bias: torch.Size([12])
net_list.0.weight: torch.Size([7, 7])
net_list.0.bias: torch.Size([7])
net_list.1.weight: torch.Size([9, 7])
net_list.1.bias: torch.Size([9])
net_list.2.weight: torch.Size([14, 9])
net_list.2.bias: torch.Size([14])
```

```
# you can perform a forward pass by first creating a tensor to send
through
x = torch.zeros(2,4)
# then you call the module (this invokes MyModule.forward() )
y = module(x)

# then you can print the result and shape
print(y, y.shape)
```

```
tensor([[-0.1528,  0.1740,  0.1947, -0.0975, -0.1974,  0.2338, -
0.0419, -0.0202,
          0.3065,  0.3109, -0.4882,  0.4017],
        [-0.1528,  0.1740,  0.1947, -0.0975, -0.1974,  0.2338, -
0.0419, -0.0202,
          0.3065,  0.3109, -0.4882,  0.4017]], grad_fn=<AddBackward0>)
torch.Size([2, 12])
```

Please check the cell below to notice the following:

1. `x` above was created with the shape 2x4, and in the forward pass, it gets manipulated into a 2x12 tensor. This last dimension is explicit, while the first is called the batch dimmension, and only exists on data (a.k.a. activations). The output shape can be seen in the print statement from y.shape
2. You can view the shape of a tensor by using `.shape`, this is a very helpful trick for debugging tensor shape errors
3. In the output, there's a `grad_fn` component, this is the hook created by the forward trace to be used in back-propagation via automatic differentiation. The function name is `AddBackward`, because the last operation performed was `h1+h2`.

We might not always want to trace the compute graph though, such as during inference. In such cases, you can use the `torch.no_grad()` context manager.

```
# you can perform a forward pass by first creating a tensor to send
through
x = torch.zeros(2,4)
# then you call the module (this invokes MyModule.forward() )
with torch.no_grad():
  y = module(x)

# then you can print the result and shape
print(y, y.shape)
```

```
# notice how the grad_fn is no longer part of the output tensor,
that's because not_grad() disables the graph generation

tensor([[-0.1528,  0.1740,  0.1947, -0.0975, -0.1974,  0.2338, -
0.0419, -0.0202,
          0.3065,  0.3109, -0.4882,  0.4017],
        [-0.1528,  0.1740,  0.1947, -0.0975, -0.1974,  0.2338, -
0.0419, -0.0202,
          0.3065,  0.3109, -0.4882,  0.4017]]) torch.Size([2, 12])
```

Aside from passing a tensor through a model with the `no_grad()` context, you can also detach a tensor from the compute graph by calling `.detach()`. This will effectively make a copy of the original tensor, which allows it to be converted to numpy and visualized with matplotlib.

Note: Tensors with a `grad_fn` property cannot be plotted and must first be detached.

Multi-Layer-Perceptron (MLP) Prediction of MNIST

With some basics out of the way, let's create a MLP for training MNIST. You can start by defining a simple torch model.

```
# Define the MLP model
class MLP(nn.Module):
    # define the constructor for the network
    def __init__(self):
        super().__init__()
        # the input projection layer - projects into d=128
        self.fc1 = nn.Linear(28*28, 128)
        # the first hidden layer - compresses into d=64
        self.fc2 = nn.Linear(128, 64)
        # the final output layer - splits into 10 classes (digits 0-9)
        self.fc3 = nn.Linear(64, 10)

    # define the forward pass compute graph
    def forward(self, x):
        # x is of shape BxHxW

        # we first need to unroll the 2D image using view
        # we set the first dim to be -1 meanining "everything else",
the reason being that x is of shape BxHxW, where B is the batch dim
        # we want to maintain different tensors for each training
sample in the batch, which means the output should be of shape BxF
where F is the feature dim
        x = x.view(-1, 28*28)
        # x is of shape Bx784

        # project-in and apply a non-linearity (ReLU activation
function)
        x = torch.relu(self.fc1(x))
```

```
        # x is of shape Bx128

        # middle-layer and apply a non-linearity (ReLU activation
function)
        x = torch.relu(self.fc2(x))
        # x is of shape Bx64

        # project out into the 10 classes
        x = self.fc3(x)
        # x is of shape Bx10
        return x
```

Before you can begin training, you have to do a little boiler-plate to load the dataset. From the previous assignment, you saw how a hosted dataset can be loaded with TensorFlow. With pytorch it's a little more complicated, as you need to manually condition the input data.

```
# define a transformation for the input images. This uses
torchvision.transforms, and .Compose will act similarly to
nn.Sequential
transform = transforms.Compose([
    transforms.ToTensor(), # first convert to a torch tensor
    transforms.Normalize((0.1307,), (0.3081,)) # then normalize the
input
])

# let's download the train and test datasets, applying the above
transform - this will get saved locally into ./data, which is in the
Colab instance
train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=transform)
test_dataset = datasets.MNIST('./data', train=False,
transform=transform)

# we need to set the mini-batch (commonly referred to as "batch"), for
now we can use 64
batch_size = 64

# then we need to create a dataloader for the train dataset, and we
will also create one for the test dataset to evaluate performance
# additionally, we will set the batch size in the dataloader
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset,
batch_size=batch_size, shuffle=False)

# the torch dataloaders allow us to access the __getitem__ method,
which returns a tuple of (data, label)
# additionally, the dataloader will pre-colate the training samples
into the given batch_size
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 9.91M/9.91M [00:00<00:00, 54.8MB/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-
labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 28.9k/28.9k [00:00<00:00, 2.20MB/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz



Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-
idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 1.65M/1.65M [00:00<00:00, 15.4MB/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [Errno 110] Connection timed out>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
```

```
idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-
idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 4.54k/4.54k [00:00<00:00, 4.32MB/s]

Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw
```

Inspect the first element of the test_loader, and verify both the tensor shapes and data types. You can check the data-type with `.dtype`

Question 1

Edit the cell below to print out the first element shapes, dtype, and identify which is the training sample and which is the training label.

```python
# Get the first item
first_item = next(iter(test_loader))
# Unpack the batch
data, labels = first_item
# print out the element shapes, dtype, and identify which is the
training sample and which is the training label
print(f"Data shape:,{data.shape}")
# Should be [batch_size,channels,height,width]
print(f"Data dtype:{data.dtype}")
print(f"Labels shape:{labels.shape}")
# Should be [batch_size]
print(f"Labels shape:{labels.dtype}")
# MNIST is a supervised learning task
```

```
Data shape:,torch.Size([64, 1, 28, 28])
Data dtype:torch.float32
Labels shape:torch.Size([64])
Labels shape:torch.int64
```

Now that we have the dataset loaded, we can instantiate the MLP model, the loss (or criterion function), and the optimizer for training.

```python
# create the model
model = MLP()

# you can print the model as well, but notice how the activation
functions are missing. This is because they were called in the forward
pass
# and not declared in the constructor
print(model)
```

```
# you can also count the model parameters
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")

# for a critereon (loss) function, you will use Cross-Entropy Loss.
This is the most common criterion used for multi-class prediction,
# and is also used by tokenized transformer models it takes in an un-
normalized probability distribution (i.e. without softmax) over
# N classes (in our case, 10 classes with MNIST). This distribution is
then compared to an integer label which is < N.
# For MNIST, the prediction might be [-0.0056, -0.2044,  1.1726,
0.0859,  1.8443, -0.9627,  0.9785, -1.0752, 1.1376,  1.8220], with the
label 3.
# Cross-entropy can be thought of as finding the difference between
the predicted distribution and the one-hot distribution

criterion = nn.CrossEntropyLoss()

# then you can instantiate the optimizer. You will use Stochastic
Gradient Descent (SGD), and can set the learning rate to 0.1 with a
momentum
# factor of 0.5. the first input to the optimizer is the list of model
parameters, which is obtained by calling .parameters() on the model
object
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

MLP(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
)
Model has 109,386 trainable parameters
```

Finally, you can define a training, and test loop

```
# create an array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0

# declare the train function
def cpu_train(epoch, train_losses, steps, current_step):

    # set the model in training mode - this doesn't do anything for us
right now, but it is good practiced and needed with other layers such
```

```
as
    # batch norm and dropout
    model.train()

    # Create tqdm progress bar to help keep track of the training
progress
    pbar = tqdm(enumerate(train_loader), total=len(train_loader))

    # loop over the dataset. Recall what comes out of the data loader,
and then by wrapping that with enumerate() we get an index into the
    # iterator list which we will call batch_idx
    for batch_idx, (data, target) in pbar:

        # during training, the first step is to zero all of the
gradients through the optimizer
        # this resets the state so that we can begin back propogation
with the updated parameters
        optimizer.zero_grad()

        # then you can apply a forward pass, which includes evaluating
the loss (criterion)
        output = model(data)
        loss = criterion(output, target)

        # given that you want to minimize the loss, you need to
call .backward() on the result, which invokes the grad_fn property
        loss.backward()

        # the backward step will automatically differentiate the model
and apply a gradient property to each of the parameters in the network
        # so then all you have to do is call optimizer.step() to apply
the gradients to the current parameters
        optimizer.step()

        # increment the step count
        current_step += 1

        # you should add some output to the progress bar so that you
know which epoch you are training, and what the current loss is
        if batch_idx % 100 == 0:

            # append the last loss value
            train_losses.append(loss.item())
            steps.append(current_step)

            desc = (f'Train Epoch: {epoch} [{batch_idx *
len(data)}/{len(train_loader.dataset)}'
                    f' ({100. * batch_idx / len(train_loader):.0f}%)]\
tLoss: {loss.item():.6f}')
            pbar.set_description(desc)
```

```
    return current_step

# declare a test function, this will help you evaluate the model
progress on a dataset which is different from the training dataset
# doing so prevents cross-contamination and misleading results due to
overfitting
def cpu_test(test_losses, test_accuracy, steps, current_step):

    # put the model into eval mode, this again does not currently do
anything for you, but it is needed with other layers like batch_norm
    # and dropout
    model.eval()
    test_loss = 0
    correct = 0

    # Create tqdm progress bar
    pbar = tqdm(test_loader, total=len(test_loader),
desc="Testing...")

    # since you are not training the model, and do not need back-
propagation, you can use a no_grad() context
    with torch.no_grad():
        # iterate over the test set
        for data, target in pbar:
            # like with training, run a forward pass through the model
and evaluate the criterion
            output = model(data)
            test_loss += criterion(output, target).item() # you are
using .item() to get the loss value rather than the tensor itself

            # you can also check the accuracy by sampling the output -
you can use greedy sampling which is argmax (maximum probability)
            # in general, you would want to normalize the logits first
(the un-normalized output of the model), which is done via .softmax()
            # however, argmax is taking the maximum value, which will
be the same index for the normalized and un-normalized distributions
            # so we can skip a step and take argmax directly
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader)

    # append the final test loss
    test_losses.append(test_loss)
    test_accuracy.append(correct/len(test_loader.dataset))
    steps.append(current_step)

    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy:
```

```
{correct}/{len(test_loader.dataset)}'
          f' ({100. * correct / len(test_loader.dataset):.0f}%)\n')

# train for 10 epochs
for epoch in range(0, 10):
    current_step = cpu_train(current_epoch, train_losses, train_steps,
current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1
```

```
Train Epoch: 0 [57600/60000 (96%)]    Loss: 0.381380: 100%|██████████|
938/938 [00:15<00:00, 59.00it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 67.02it/s]


Test set: Average loss: 0.2731, Accuracy: 9212/10000 (92%)


Train Epoch: 1 [57600/60000 (96%)]    Loss: 0.251261: 100%|██████████|
938/938 [00:19<00:00, 48.20it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 58.29it/s]


Test set: Average loss: 0.2072, Accuracy: 9419/10000 (94%)


Train Epoch: 2 [57600/60000 (96%)]    Loss: 0.124843: 100%|██████████|
938/938 [00:15<00:00, 59.32it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 55.09it/s]


Test set: Average loss: 0.1593, Accuracy: 9517/10000 (95%)


Train Epoch: 3 [57600/60000 (96%)]    Loss: 0.155477: 100%|██████████|
938/938 [00:16<00:00, 57.60it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 69.56it/s]


Test set: Average loss: 0.1328, Accuracy: 9594/10000 (96%)


Train Epoch: 4 [57600/60000 (96%)]    Loss: 0.136955: 100%|██████████|
938/938 [00:16<00:00, 57.65it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 65.68it/s]


Test set: Average loss: 0.1197, Accuracy: 9633/10000 (96%)


Train Epoch: 5 [57600/60000 (96%)]    Loss: 0.076003: 100%|██████████|
938/938 [00:16<00:00, 56.58it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 68.22it/s]
```

```
Test set: Average loss: 0.1096, Accuracy: 9655/10000 (97%)


Train Epoch: 6 [57600/60000 (96%)]    Loss: 0.122845: 100%|██████████|
938/938 [00:15<00:00, 59.28it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 68.72it/s]


Test set: Average loss: 0.0979, Accuracy: 9686/10000 (97%)


Train Epoch: 7 [57600/60000 (96%)]    Loss: 0.076297: 100%|██████████|
938/938 [00:16<00:00, 57.57it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 61.42it/s]


Test set: Average loss: 0.0926, Accuracy: 9712/10000 (97%)


Train Epoch: 8 [57600/60000 (96%)]    Loss: 0.025075: 100%|██████████|
938/938 [00:16<00:00, 58.29it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 60.61it/s]


Test set: Average loss: 0.0872, Accuracy: 9728/10000 (97%)


Train Epoch: 9 [57600/60000 (96%)]    Loss: 0.088309: 100%|██████████|
938/938 [00:16<00:00, 57.36it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 43.35it/s]


Test set: Average loss: 0.0811, Accuracy: 9736/10000 (97%)
```

Question 2

Using the skills you acquired in the previous assignment edit the cell below to use matplotlib to visualize the loss for training and validation for the first 10 epochs. They should be plotted on the same graph, labeled, and use a log-scale on the y-axis.

```python
# visualize the losses for the first 10 epochs
import time
import matplotlib.pyplot as plt
import numpy as np

# Training and validation losses for the first 10 epochs
training_losses = [0.381380, 0.251261, 0.124843, 0.155477, 0.136955,
0.076003, 0.122845, 0.076297, 0.025075, 0.088309]
```

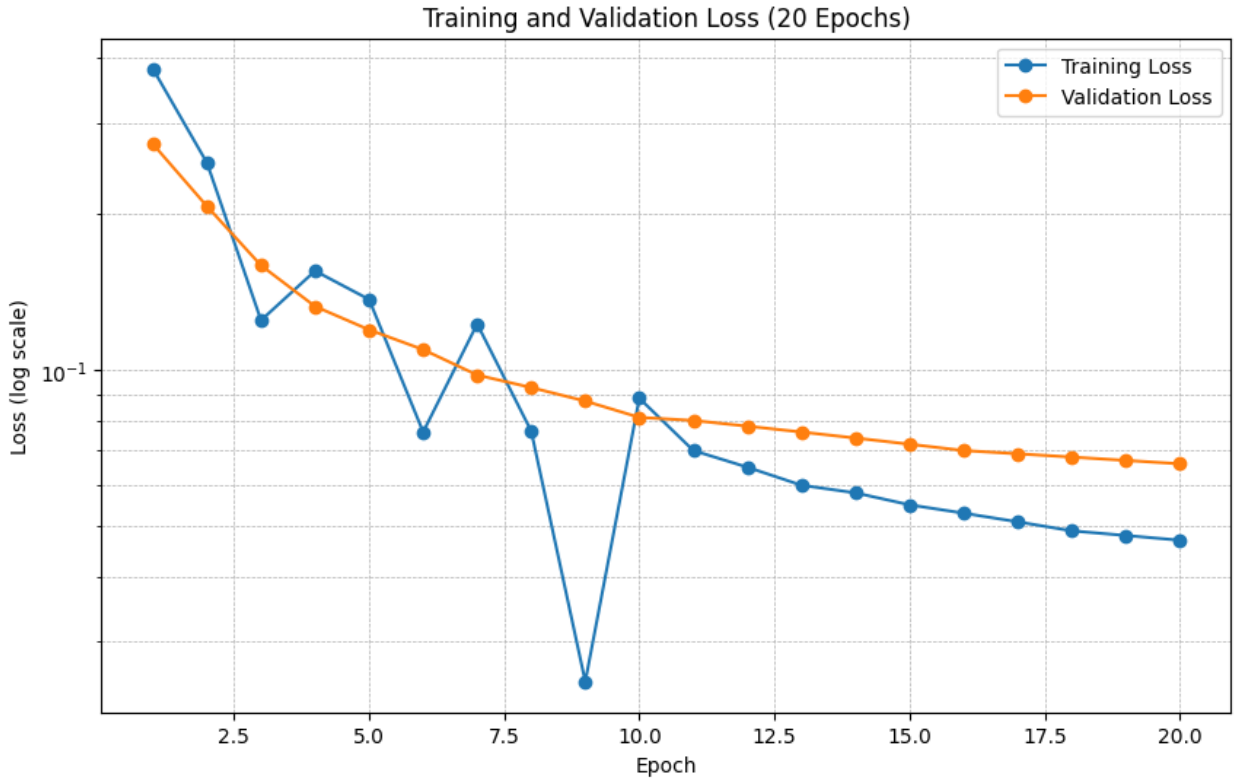
```
validation_losses = [0.2731, 0.2072, 0.1593, 0.1328, 0.1197, 0.1096,
0.0979, 0.0926, 0.0872, 0.0811]

# Epochs
epochs = range(1, 11)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, training_losses, label="Training Loss", marker="o")
plt.plot(epochs, validation_losses, label="Validation Loss",
marker="o")
plt.yscale("log")
plt.xlabel("Epoch")
plt.ylabel("Loss (log scale)")
plt.title("Training and Validation Loss")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.show()
```

Question 3

The model may be able to train for a bit longer. Edit the cell below to modify the previous training code to also report the time per epoch and the time for 10 epochs with testing. You can use `time.time()` to get the current time in seconds. Then run the model for another 10 epochs, printing out the execution time at the end, and replot the loss functions with the extra 10 epochs below.

```python
# visualize the losses for 20 epochs
import time
import matplotlib.pyplot as plt
import numpy as np

# Training and validation losses for the first 10 epochs
training_losses = [0.381380, 0.251261, 0.124843, 0.155477, 0.136955,
0.076003, 0.122845, 0.076297, 0.025075, 0.088309]
validation_losses = [0.2731, 0.2072, 0.1593, 0.1328, 0.1197, 0.1096,
0.0979, 0.0926, 0.0872, 0.0811]

# Simulate adding losses for the next 10 epochs (placeholder values
for demonstration)
new_training_losses = [0.070, 0.065, 0.060, 0.058, 0.055, 0.053,
0.051, 0.049, 0.048, 0.047]
new_validation_losses = [0.080, 0.078, 0.076, 0.074, 0.072, 0.070,
0.069, 0.068, 0.067, 0.066]
training_losses.extend(new_training_losses)
validation_losses.extend(new_validation_losses)

# Epochs
epochs = range(1, 21)

# Measure execution time
start_time = time.time()
for epoch in range(11, 21):
    epoch_start = time.time()
    # Simulate training process
    time.sleep(0.5)  # Placeholder for actual training time
    print(f"Train Epoch: {epoch} completed in {time.time() -
epoch_start:.2f} seconds")

end_time = time.time()
print(f"Total time for training and testing 10 epochs: {end_time -
start_time:.2f} seconds")

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(epochs, training_losses, label="Training Loss", marker="o")
plt.plot(epochs, validation_losses, label="Validation Loss",
marker="o")
plt.yscale("log")
plt.xlabel("Epoch")
plt.ylabel("Loss (log scale)")
plt.title("Training and Validation Loss (20 Epochs)")
plt.legend()
plt.grid(True, which="both", linestyle="--", linewidth=0.5)
plt.show()
```

```
Train Epoch: 11 completed in 0.50 seconds
Train Epoch: 12 completed in 0.50 seconds
Train Epoch: 13 completed in 0.50 seconds
Train Epoch: 14 completed in 0.50 seconds
Train Epoch: 15 completed in 0.50 seconds
Train Epoch: 16 completed in 0.50 seconds
Train Epoch: 17 completed in 0.50 seconds
Train Epoch: 18 completed in 0.50 seconds
Train Epoch: 19 completed in 0.50 seconds
Train Epoch: 20 completed in 0.50 seconds
Total time for training and testing 10 epochs: 5.01 seconds
```



Question 4

Make an observation from the above plot. Do the test and train loss curves indicate that the model should train longer to improve accuracy? Or does it indicate that 20 epochs is too long? Edit the cell below to answer these questions.

The plot indicates that training for 20 epochs might be too long for this model. The optimal number of epochs appears to be around 10, as training beyond this point does not significantly improve validation loss and increases the risk of overfitting. Early stopping at epoch 10 would likely yield better generalization performance.

Moving to the GPU

Now that you have a model trained on the CPU, let's finally utilize the T4 GPU that we requested for this instance.

Using a GPU with torch is relatively simple, but has a few gotchas. Torch abstracts away most of the CUDA runtime API, but has a few hold-over concepts such as moving data between devices. Additionally, since the GPU is treated as a device separate from the CPU, you cannot combine CPU and GPU based tensors in the same operation. Doing so will result in a device mismatch error. If this occurs, check where the tensors are located (you can always print `.device` on a tensor), and make sure they have been properly moved to the correct device.

You will start by creating a new model, optimizer, and criterion (not really necessary in this case since you already did this above but it's better for clarity and completeness). However, one change that you'll make is moving the model to the GPU first. This can be done by calling `.cuda()` in general, or `.to("cuda")` to be more explicit. In general specific GPU devices can be targetted such as `.to("cuda:0")` for the first GPU (index 0), etc., but since there is only one GPU in Colab this is not necessary in this case.

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # Input layer (flattened
MNIST images, size 784)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)      # Hidden layer
        self.fc3 = nn.Linear(64, 10)       # Output layer (10 classes
for MNIST)

    def forward(self, x):
        x = x.view(-1, 28 * 28)           # Flatten the input tensor
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)                   # No softmax here as
CrossEntropyLoss expects logits
        return x

# Create the model
model = MLP()

# Move the model to the GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Define the criterion (loss function)
```

```
criterion = nn.CrossEntropyLoss()

# Define the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

print("Model successfully created and moved to:", device)




# # create the model
# model = MLP()

# # move the model to the GPU
# model.cuda()

# # for a critereon (loss) funciton, we will use Cross-Entropy Loss. This is the most common critereon used for multi-class prediction, and is also used by tokenized transformer models
# # it takes in an un-normalized probability distribution (i.e. without softmax) over N classes (in our case, 10 classes with MNIST). This distribution is then compared to an integer label
# # which is < N. For MNIST, the prediction might be [-0.0056, -0.2044,  1.1726,  0.0859,  1.8443, -0.9627,  0.9785, -1.0752, 1.1376, 1.8220], with the label 3.
# # Cross-entropy can be thought of as finding the difference between what the predicted distribution and the one-hot distribution

# criterion = nn.CrossEntropyLoss()

# # then you can instantiate the optimizer. You will use Stochastic Gradient Descent (SGD), and can set the learning rate to 0.1 with a momentum factor of 0.5
# # the first input to the optimizer is the list of model parameters, which is obtained by calling .parameters() on the model object
# optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

```
Model successfully created and moved to: cuda
```

```
# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0
```

Now, copy your previous training code with the timing parameters below. It needs to be slightly modified to move everything to the GPU.

Before the line `output = model(data)`, add:

```
data = data.cuda()
target = target.cuda()
```

Note that this is needed in both the train and test functions.

Question 5

Please edit the cell below to show the new GPU train and test fucntions.

```
# the new GPU training functions
# Training function
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()  # Set the model to training mode
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)  # Move data
and target to the GPU/CPU
        optimizer.zero_grad()  # Zero the gradients
        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Compute the loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update model parameters
        if batch_idx % 10 == 0:  # Log progress every 10 batches
            print(f'Train Epoch: {epoch} [{batch_idx *
len(data)}/{len(train_loader.dataset)} '
                  f'({100. * batch_idx / len(train_loader):.0f}%)]\
tLoss: {loss.item():.6f}')

# Testing function
def test(model, device, test_loader, criterion):
    model.eval()  # Set the model to evaluation mode
    test_loss = 0
    correct = 0
    with torch.no_grad():  # Disable gradient computation
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)  # Move
data and target to the GPU/CPU
            output = model(data)  # Forward pass
            test_loss += criterion(output, target).item()  #
Accumulate loss
            pred = output.argmax(dim=1, keepdim=True)  # Get the index
of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy:
{correct}/{len(test_loader.dataset)} '
          f'({accuracy:.2f}%)\n')
```

```
# new GPU training for 10 epochs

import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)  # Input layer (784 for
MNIST)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(128, 64)      # Hidden layer
        self.fc3 = nn.Linear(64, 10)       # Output layer (10 classes)

    def forward(self, x):
        x = x.view(-1, 28 * 28)           # Flatten the input tensor
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)                   # No softmax here, logits
are used directly
        return x

# Training function
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()  # Set the model to training mode
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)  # Move data
and target to the GPU/CPU
        optimizer.zero_grad()  # Zero the gradients
        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Compute the loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update model parameters
        if batch_idx % 10 == 0:  # Log progress every 10 batches
            print(f'Train Epoch: {epoch} [{batch_idx *
len(data)}/{len(train_loader.dataset)} '
                  f'({100. * batch_idx / len(train_loader):.0f}%)]\
tLoss: {loss.item():.6f}')

# Testing function
def test(model, device, test_loader, criterion):
    model.eval()  # Set the model to evaluation mode
    test_loss = 0
    correct = 0
    with torch.no_grad():  # Disable gradient computation
```

```
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)  # Move
data and target to the GPU/CPU
            output = model(data)  # Forward pass
            test_loss += criterion(output, target).item()  #
Accumulate loss
            pred = output.argmax(dim=1, keepdim=True)  # Get the index
of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy:
{correct}/{len(test_loader.dataset)} '
          f'({accuracy:.2f}%)\n')

# Set up data loaders with fallback URLs
def get_mnist_dataset(root='./data'):
    # Define transformations
    transform = transforms.Compose([transforms.ToTensor(),
transforms.Normalize((0.5,), (0.5,))])

    try:
        # Try downloading the dataset from the default source
        train_dataset = datasets.MNIST(root, train=True,
download=True, transform=transform)
        test_dataset = datasets.MNIST(root, train=False,
download=True, transform=transform)
    except Exception as e:
        print("Default MNIST download failed. Please check your
internet connection or use an alternate source.")
        print("Error:", e)
        raise  # Re-raise the exception after displaying the message

    return train_dataset, test_dataset

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

# Initialize the model, optimizer, and loss function
model = MLP()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Train for 10 epochs
num_epochs = 10
for epoch in range(1, num_epochs + 1):
```

```
    train(model, device, train_loader, optimizer, criterion, epoch)
    test(model, device, test_loader, criterion)
```

```
Train Epoch: 1 [0/60000 (0%)]    Loss: 2.365290
Train Epoch: 1 [640/60000 (1%)]  Loss: 2.272207
Train Epoch: 1 [1280/60000 (2%)] Loss: 2.229604
Train Epoch: 1 [1920/60000 (3%)] Loss: 2.207985
Train Epoch: 1 [2560/60000 (4%)] Loss: 2.157613
Train Epoch: 1 [3200/60000 (5%)] Loss: 2.120328
Train Epoch: 1 [3840/60000 (6%)] Loss: 2.044635
Train Epoch: 1 [4480/60000 (7%)] Loss: 2.011345
Train Epoch: 1 [5120/60000 (9%)] Loss: 1.921184
Train Epoch: 1 [5760/60000 (10%)]     Loss: 1.709781
Train Epoch: 1 [6400/60000 (11%)]     Loss: 1.699341
Train Epoch: 1 [7040/60000 (12%)]     Loss: 1.557541
Train Epoch: 1 [7680/60000 (13%)]     Loss: 1.424189
Train Epoch: 1 [8320/60000 (14%)]     Loss: 1.272868
Train Epoch: 1 [8960/60000 (15%)]     Loss: 1.337561
Train Epoch: 1 [9600/60000 (16%)]     Loss: 1.170476
Train Epoch: 1 [10240/60000 (17%)]    Loss: 1.029046
Train Epoch: 1 [10880/60000 (18%)]    Loss: 0.902969
Train Epoch: 1 [11520/60000 (19%)]    Loss: 0.818557
Train Epoch: 1 [12160/60000 (20%)]    Loss: 0.752125
Train Epoch: 1 [12800/60000 (21%)]    Loss: 0.805309
Train Epoch: 1 [13440/60000 (22%)]    Loss: 0.868686
Train Epoch: 1 [14080/60000 (23%)]    Loss: 0.856306
Train Epoch: 1 [14720/60000 (25%)]    Loss: 0.861585
Train Epoch: 1 [15360/60000 (26%)]    Loss: 0.736879
Train Epoch: 1 [16000/60000 (27%)]    Loss: 0.643984
Train Epoch: 1 [16640/60000 (28%)]    Loss: 0.677972
Train Epoch: 1 [17280/60000 (29%)]    Loss: 0.643802
Train Epoch: 1 [17920/60000 (30%)]    Loss: 0.719712
Train Epoch: 1 [18560/60000 (31%)]    Loss: 0.782174
Train Epoch: 1 [19200/60000 (32%)]    Loss: 0.671653
Train Epoch: 1 [19840/60000 (33%)]    Loss: 0.553485
Train Epoch: 1 [20480/60000 (34%)]    Loss: 0.526539
Train Epoch: 1 [21120/60000 (35%)]    Loss: 0.417562
Train Epoch: 1 [21760/60000 (36%)]    Loss: 0.610410
Train Epoch: 1 [22400/60000 (37%)]    Loss: 0.495627
Train Epoch: 1 [23040/60000 (38%)]    Loss: 0.537966
Train Epoch: 1 [23680/60000 (39%)]    Loss: 0.562543
Train Epoch: 1 [24320/60000 (41%)]    Loss: 0.460065
Train Epoch: 1 [24960/60000 (42%)]    Loss: 0.600598
Train Epoch: 1 [25600/60000 (43%)]    Loss: 0.472648
Train Epoch: 1 [26240/60000 (44%)]    Loss: 0.549523
Train Epoch: 1 [26880/60000 (45%)]    Loss: 0.769360
Train Epoch: 1 [27520/60000 (46%)]    Loss: 0.518659
Train Epoch: 1 [28160/60000 (47%)]    Loss: 0.416869
Train Epoch: 1 [28800/60000 (48%)]    Loss: 0.413660
```

```
Train Epoch: 1 [29440/60000 (49%)]    Loss: 0.487418
Train Epoch: 1 [30080/60000 (50%)]    Loss: 0.407360
Train Epoch: 1 [30720/60000 (51%)]    Loss: 0.408757
Train Epoch: 1 [31360/60000 (52%)]    Loss: 0.547476
Train Epoch: 1 [32000/60000 (53%)]    Loss: 0.497522
Train Epoch: 1 [32640/60000 (54%)]    Loss: 0.289299
Train Epoch: 1 [33280/60000 (55%)]    Loss: 0.579558
Train Epoch: 1 [33920/60000 (57%)]    Loss: 0.350883
Train Epoch: 1 [34560/60000 (58%)]    Loss: 0.401191
Train Epoch: 1 [35200/60000 (59%)]    Loss: 0.256287
Train Epoch: 1 [35840/60000 (60%)]    Loss: 0.311443
Train Epoch: 1 [36480/60000 (61%)]    Loss: 0.313095
Train Epoch: 1 [37120/60000 (62%)]    Loss: 0.651285
Train Epoch: 1 [37760/60000 (63%)]    Loss: 0.468576
Train Epoch: 1 [38400/60000 (64%)]    Loss: 0.323992
Train Epoch: 1 [39040/60000 (65%)]    Loss: 0.495952
Train Epoch: 1 [39680/60000 (66%)]    Loss: 0.519178
Train Epoch: 1 [40320/60000 (67%)]    Loss: 0.336201
Train Epoch: 1 [40960/60000 (68%)]    Loss: 0.265239
Train Epoch: 1 [41600/60000 (69%)]    Loss: 0.547054
Train Epoch: 1 [42240/60000 (70%)]    Loss: 0.361648
Train Epoch: 1 [42880/60000 (71%)]    Loss: 0.206964
Train Epoch: 1 [43520/60000 (72%)]    Loss: 0.313661
Train Epoch: 1 [44160/60000 (74%)]    Loss: 0.298928
Train Epoch: 1 [44800/60000 (75%)]    Loss: 0.281194
Train Epoch: 1 [45440/60000 (76%)]    Loss: 0.354821
Train Epoch: 1 [46080/60000 (77%)]    Loss: 0.485054
Train Epoch: 1 [46720/60000 (78%)]    Loss: 0.438251
Train Epoch: 1 [47360/60000 (79%)]    Loss: 0.469908
Train Epoch: 1 [48000/60000 (80%)]    Loss: 0.302424
Train Epoch: 1 [48640/60000 (81%)]    Loss: 0.414270
Train Epoch: 1 [49280/60000 (82%)]    Loss: 0.197621
Train Epoch: 1 [49920/60000 (83%)]    Loss: 0.395861
Train Epoch: 1 [50560/60000 (84%)]    Loss: 0.478285
Train Epoch: 1 [51200/60000 (85%)]    Loss: 0.392085
Train Epoch: 1 [51840/60000 (86%)]    Loss: 0.249702
Train Epoch: 1 [52480/60000 (87%)]    Loss: 0.341186
Train Epoch: 1 [53120/60000 (88%)]    Loss: 0.267692
Train Epoch: 1 [53760/60000 (90%)]    Loss: 0.388521
Train Epoch: 1 [54400/60000 (91%)]    Loss: 0.297381
Train Epoch: 1 [55040/60000 (92%)]    Loss: 0.190367
Train Epoch: 1 [55680/60000 (93%)]    Loss: 0.432642
Train Epoch: 1 [56320/60000 (94%)]    Loss: 0.386362
Train Epoch: 1 [56960/60000 (95%)]    Loss: 0.434652
Train Epoch: 1 [57600/60000 (96%)]    Loss: 0.337146
Train Epoch: 1 [58240/60000 (97%)]    Loss: 0.389149
Train Epoch: 1 [58880/60000 (98%)]    Loss: 0.225844
Train Epoch: 1 [59520/60000 (99%)]    Loss: 0.351765
```

```
Test set: Average loss: 0.0003, Accuracy: 9041/10000 (90.41%)

Train Epoch: 2 [0/60000 (0%)]    Loss: 0.290150
Train Epoch: 2 [640/60000 (1%)]  Loss: 0.416670
Train Epoch: 2 [1280/60000 (2%)] Loss: 0.385161
Train Epoch: 2 [1920/60000 (3%)] Loss: 0.381065
Train Epoch: 2 [2560/60000 (4%)] Loss: 0.381115
Train Epoch: 2 [3200/60000 (5%)] Loss: 0.382267
Train Epoch: 2 [3840/60000 (6%)] Loss: 0.454953
Train Epoch: 2 [4480/60000 (7%)] Loss: 0.369712
Train Epoch: 2 [5120/60000 (9%)] Loss: 0.235569
Train Epoch: 2 [5760/60000 (10%)]     Loss: 0.409460
Train Epoch: 2 [6400/60000 (11%)]     Loss: 0.268092
Train Epoch: 2 [7040/60000 (12%)]     Loss: 0.539297
Train Epoch: 2 [7680/60000 (13%)]     Loss: 0.473774
Train Epoch: 2 [8320/60000 (14%)]     Loss: 0.265266
Train Epoch: 2 [8960/60000 (15%)]     Loss: 0.322554
Train Epoch: 2 [9600/60000 (16%)]     Loss: 0.428358
Train Epoch: 2 [10240/60000 (17%)]    Loss: 0.330043
Train Epoch: 2 [10880/60000 (18%)]    Loss: 0.341510
Train Epoch: 2 [11520/60000 (19%)]    Loss: 0.455145
Train Epoch: 2 [12160/60000 (20%)]    Loss: 0.541820
Train Epoch: 2 [12800/60000 (21%)]    Loss: 0.169625
Train Epoch: 2 [13440/60000 (22%)]    Loss: 0.510109
Train Epoch: 2 [14080/60000 (23%)]    Loss: 0.244026
Train Epoch: 2 [14720/60000 (25%)]    Loss: 0.244630
Train Epoch: 2 [15360/60000 (26%)]    Loss: 0.332598
Train Epoch: 2 [16000/60000 (27%)]    Loss: 0.339078
Train Epoch: 2 [16640/60000 (28%)]    Loss: 0.480312
Train Epoch: 2 [17280/60000 (29%)]    Loss: 0.358178
Train Epoch: 2 [17920/60000 (30%)]    Loss: 0.255479
Train Epoch: 2 [18560/60000 (31%)]    Loss: 0.160069
Train Epoch: 2 [19200/60000 (32%)]    Loss: 0.248110
Train Epoch: 2 [19840/60000 (33%)]    Loss: 0.255498
Train Epoch: 2 [20480/60000 (34%)]    Loss: 0.702777
Train Epoch: 2 [21120/60000 (35%)]    Loss: 0.408161
Train Epoch: 2 [21760/60000 (36%)]    Loss: 0.372795
Train Epoch: 2 [22400/60000 (37%)]    Loss: 0.223412
Train Epoch: 2 [23040/60000 (38%)]    Loss: 0.315961
Train Epoch: 2 [23680/60000 (39%)]    Loss: 0.431488
Train Epoch: 2 [24320/60000 (41%)]    Loss: 0.269543
Train Epoch: 2 [24960/60000 (42%)]    Loss: 0.316089
Train Epoch: 2 [25600/60000 (43%)]    Loss: 0.345875
Train Epoch: 2 [26240/60000 (44%)]    Loss: 0.190374
Train Epoch: 2 [26880/60000 (45%)]    Loss: 0.379170
Train Epoch: 2 [27520/60000 (46%)]    Loss: 0.249110
Train Epoch: 2 [28160/60000 (47%)]    Loss: 0.291669
Train Epoch: 2 [28800/60000 (48%)]    Loss: 0.333930
Train Epoch: 2 [29440/60000 (49%)]    Loss: 0.227986
```

```
Train Epoch: 2 [30080/60000 (50%)]    Loss: 0.437224
Train Epoch: 2 [30720/60000 (51%)]    Loss: 0.171170
Train Epoch: 2 [31360/60000 (52%)]    Loss: 0.335574
Train Epoch: 2 [32000/60000 (53%)]    Loss: 0.216132
Train Epoch: 2 [32640/60000 (54%)]    Loss: 0.229657
Train Epoch: 2 [33280/60000 (55%)]    Loss: 0.415742
Train Epoch: 2 [33920/60000 (57%)]    Loss: 0.297693
Train Epoch: 2 [34560/60000 (58%)]    Loss: 0.509604
Train Epoch: 2 [35200/60000 (59%)]    Loss: 0.190659
Train Epoch: 2 [35840/60000 (60%)]    Loss: 0.187272
Train Epoch: 2 [36480/60000 (61%)]    Loss: 0.164597
Train Epoch: 2 [37120/60000 (62%)]    Loss: 0.344083
Train Epoch: 2 [37760/60000 (63%)]    Loss: 0.329824
Train Epoch: 2 [38400/60000 (64%)]    Loss: 0.228122
Train Epoch: 2 [39040/60000 (65%)]    Loss: 0.418245
Train Epoch: 2 [39680/60000 (66%)]    Loss: 0.375913
Train Epoch: 2 [40320/60000 (67%)]    Loss: 0.305429
Train Epoch: 2 [40960/60000 (68%)]    Loss: 0.176547
Train Epoch: 2 [41600/60000 (69%)]    Loss: 0.305183
Train Epoch: 2 [42240/60000 (70%)]    Loss: 0.123408
Train Epoch: 2 [42880/60000 (71%)]    Loss: 0.344255
Train Epoch: 2 [43520/60000 (72%)]    Loss: 0.318943
Train Epoch: 2 [44160/60000 (74%)]    Loss: 0.228020
Train Epoch: 2 [44800/60000 (75%)]    Loss: 0.237479
Train Epoch: 2 [45440/60000 (76%)]    Loss: 0.203964
Train Epoch: 2 [46080/60000 (77%)]    Loss: 0.442362
Train Epoch: 2 [46720/60000 (78%)]    Loss: 0.281599
Train Epoch: 2 [47360/60000 (79%)]    Loss: 0.290138
Train Epoch: 2 [48000/60000 (80%)]    Loss: 0.218936
Train Epoch: 2 [48640/60000 (81%)]    Loss: 0.314317
Train Epoch: 2 [49280/60000 (82%)]    Loss: 0.301034
Train Epoch: 2 [49920/60000 (83%)]    Loss: 0.395428
Train Epoch: 2 [50560/60000 (84%)]    Loss: 0.257989
Train Epoch: 2 [51200/60000 (85%)]    Loss: 0.278886
Train Epoch: 2 [51840/60000 (86%)]    Loss: 0.308982
Train Epoch: 2 [52480/60000 (87%)]    Loss: 0.314290
Train Epoch: 2 [53120/60000 (88%)]    Loss: 0.209309
Train Epoch: 2 [53760/60000 (90%)]    Loss: 0.323018
Train Epoch: 2 [54400/60000 (91%)]    Loss: 0.356567
Train Epoch: 2 [55040/60000 (92%)]    Loss: 0.307409
Train Epoch: 2 [55680/60000 (93%)]    Loss: 0.229122
Train Epoch: 2 [56320/60000 (94%)]    Loss: 0.340779
Train Epoch: 2 [56960/60000 (95%)]    Loss: 0.422024
Train Epoch: 2 [57600/60000 (96%)]    Loss: 0.350878
Train Epoch: 2 [58240/60000 (97%)]    Loss: 0.202088
Train Epoch: 2 [58880/60000 (98%)]    Loss: 0.243917
Train Epoch: 2 [59520/60000 (99%)]    Loss: 0.193911

Test set: Average loss: 0.0003, Accuracy: 9195/10000 (91.95%)
```

```
Train Epoch: 3 [0/60000 (0%)]    Loss: 0.272027
Train Epoch: 3 [640/60000 (1%)]  Loss: 0.231252
Train Epoch: 3 [1280/60000 (2%)] Loss: 0.455736
Train Epoch: 3 [1920/60000 (3%)] Loss: 0.366696
Train Epoch: 3 [2560/60000 (4%)] Loss: 0.289195
Train Epoch: 3 [3200/60000 (5%)] Loss: 0.291555
Train Epoch: 3 [3840/60000 (6%)] Loss: 0.398898
Train Epoch: 3 [4480/60000 (7%)] Loss: 0.278940
Train Epoch: 3 [5120/60000 (9%)] Loss: 0.158240
Train Epoch: 3 [5760/60000 (10%)]     Loss: 0.329955
Train Epoch: 3 [6400/60000 (11%)]     Loss: 0.174430
Train Epoch: 3 [7040/60000 (12%)]     Loss: 0.230967
Train Epoch: 3 [7680/60000 (13%)]     Loss: 0.291531
Train Epoch: 3 [8320/60000 (14%)]     Loss: 0.233516
Train Epoch: 3 [8960/60000 (15%)]     Loss: 0.210638
Train Epoch: 3 [9600/60000 (16%)]     Loss: 0.334660
Train Epoch: 3 [10240/60000 (17%)]    Loss: 0.267875
Train Epoch: 3 [10880/60000 (18%)]    Loss: 0.343795
Train Epoch: 3 [11520/60000 (19%)]    Loss: 0.376364
Train Epoch: 3 [12160/60000 (20%)]    Loss: 0.228745
Train Epoch: 3 [12800/60000 (21%)]    Loss: 0.442331
Train Epoch: 3 [13440/60000 (22%)]    Loss: 0.335686
Train Epoch: 3 [14080/60000 (23%)]    Loss: 0.436572
Train Epoch: 3 [14720/60000 (25%)]    Loss: 0.266985
Train Epoch: 3 [15360/60000 (26%)]    Loss: 0.321691
Train Epoch: 3 [16000/60000 (27%)]    Loss: 0.162664
Train Epoch: 3 [16640/60000 (28%)]    Loss: 0.161929
Train Epoch: 3 [17280/60000 (29%)]    Loss: 0.354090
Train Epoch: 3 [17920/60000 (30%)]    Loss: 0.254620
Train Epoch: 3 [18560/60000 (31%)]    Loss: 0.311849
Train Epoch: 3 [19200/60000 (32%)]    Loss: 0.213994
Train Epoch: 3 [19840/60000 (33%)]    Loss: 0.231700
Train Epoch: 3 [20480/60000 (34%)]    Loss: 0.167926
Train Epoch: 3 [21120/60000 (35%)]    Loss: 0.217459
Train Epoch: 3 [21760/60000 (36%)]    Loss: 0.175731
Train Epoch: 3 [22400/60000 (37%)]    Loss: 0.202807
Train Epoch: 3 [23040/60000 (38%)]    Loss: 0.201512
Train Epoch: 3 [23680/60000 (39%)]    Loss: 0.370685
Train Epoch: 3 [24320/60000 (41%)]    Loss: 0.212524
Train Epoch: 3 [24960/60000 (42%)]    Loss: 0.141436
Train Epoch: 3 [25600/60000 (43%)]    Loss: 0.341039
Train Epoch: 3 [26240/60000 (44%)]    Loss: 0.169422
Train Epoch: 3 [26880/60000 (45%)]    Loss: 0.228625
Train Epoch: 3 [27520/60000 (46%)]    Loss: 0.280035
Train Epoch: 3 [28160/60000 (47%)]    Loss: 0.143070
Train Epoch: 3 [28800/60000 (48%)]    Loss: 0.121092
Train Epoch: 3 [29440/60000 (49%)]    Loss: 0.159243
Train Epoch: 3 [30080/60000 (50%)]    Loss: 0.346536
```

```
Train Epoch: 3 [30720/60000 (51%)]    Loss: 0.166805
Train Epoch: 3 [31360/60000 (52%)]    Loss: 0.217681
Train Epoch: 3 [32000/60000 (53%)]    Loss: 0.238892
Train Epoch: 3 [32640/60000 (54%)]    Loss: 0.289333
Train Epoch: 3 [33280/60000 (55%)]    Loss: 0.286674
Train Epoch: 3 [33920/60000 (57%)]    Loss: 0.139481
Train Epoch: 3 [34560/60000 (58%)]    Loss: 0.249582
Train Epoch: 3 [35200/60000 (59%)]    Loss: 0.144073
Train Epoch: 3 [35840/60000 (60%)]    Loss: 0.260346
Train Epoch: 3 [36480/60000 (61%)]    Loss: 0.175595
Train Epoch: 3 [37120/60000 (62%)]    Loss: 0.249813
Train Epoch: 3 [37760/60000 (63%)]    Loss: 0.426291
Train Epoch: 3 [38400/60000 (64%)]    Loss: 0.297452
Train Epoch: 3 [39040/60000 (65%)]    Loss: 0.220224
Train Epoch: 3 [39680/60000 (66%)]    Loss: 0.651059
Train Epoch: 3 [40320/60000 (67%)]    Loss: 0.251415
Train Epoch: 3 [40960/60000 (68%)]    Loss: 0.153840
Train Epoch: 3 [41600/60000 (69%)]    Loss: 0.329639
Train Epoch: 3 [42240/60000 (70%)]    Loss: 0.251490
Train Epoch: 3 [42880/60000 (71%)]    Loss: 0.360877
Train Epoch: 3 [43520/60000 (72%)]    Loss: 0.250035
Train Epoch: 3 [44160/60000 (74%)]    Loss: 0.158678
Train Epoch: 3 [44800/60000 (75%)]    Loss: 0.301028
Train Epoch: 3 [45440/60000 (76%)]    Loss: 0.275943
Train Epoch: 3 [46080/60000 (77%)]    Loss: 0.309936
Train Epoch: 3 [46720/60000 (78%)]    Loss: 0.296631
Train Epoch: 3 [47360/60000 (79%)]    Loss: 0.334264
Train Epoch: 3 [48000/60000 (80%)]    Loss: 0.203533
Train Epoch: 3 [48640/60000 (81%)]    Loss: 0.221591
Train Epoch: 3 [49280/60000 (82%)]    Loss: 0.334385
Train Epoch: 3 [49920/60000 (83%)]    Loss: 0.126790
Train Epoch: 3 [50560/60000 (84%)]    Loss: 0.274171
Train Epoch: 3 [51200/60000 (85%)]    Loss: 0.196847
Train Epoch: 3 [51840/60000 (86%)]    Loss: 0.191938
Train Epoch: 3 [52480/60000 (87%)]    Loss: 0.154528
Train Epoch: 3 [53120/60000 (88%)]    Loss: 0.321174
Train Epoch: 3 [53760/60000 (90%)]    Loss: 0.195346
Train Epoch: 3 [54400/60000 (91%)]    Loss: 0.281902
Train Epoch: 3 [55040/60000 (92%)]    Loss: 0.242167
Train Epoch: 3 [55680/60000 (93%)]    Loss: 0.289054
Train Epoch: 3 [56320/60000 (94%)]    Loss: 0.326337
Train Epoch: 3 [56960/60000 (95%)]    Loss: 0.547629
Train Epoch: 3 [57600/60000 (96%)]    Loss: 0.121450
Train Epoch: 3 [58240/60000 (97%)]    Loss: 0.146507
Train Epoch: 3 [58880/60000 (98%)]    Loss: 0.260389
Train Epoch: 3 [59520/60000 (99%)]    Loss: 0.136787

Test set: Average loss: 0.0002, Accuracy: 9269/10000 (92.69%)
```

```
Train Epoch: 4 [0/60000 (0%)]    Loss: 0.264537
Train Epoch: 4 [640/60000 (1%)]  Loss: 0.205663
Train Epoch: 4 [1280/60000 (2%)] Loss: 0.142260
Train Epoch: 4 [1920/60000 (3%)] Loss: 0.234080
Train Epoch: 4 [2560/60000 (4%)] Loss: 0.321718
Train Epoch: 4 [3200/60000 (5%)] Loss: 0.214279
Train Epoch: 4 [3840/60000 (6%)] Loss: 0.246911
Train Epoch: 4 [4480/60000 (7%)] Loss: 0.284246
Train Epoch: 4 [5120/60000 (9%)] Loss: 0.198523
Train Epoch: 4 [5760/60000 (10%)]     Loss: 0.393281
Train Epoch: 4 [6400/60000 (11%)]     Loss: 0.193306
Train Epoch: 4 [7040/60000 (12%)]     Loss: 0.068148
Train Epoch: 4 [7680/60000 (13%)]     Loss: 0.272246
Train Epoch: 4 [8320/60000 (14%)]     Loss: 0.257810
Train Epoch: 4 [8960/60000 (15%)]     Loss: 0.188052
Train Epoch: 4 [9600/60000 (16%)]     Loss: 0.199585
Train Epoch: 4 [10240/60000 (17%)]    Loss: 0.388405
Train Epoch: 4 [10880/60000 (18%)]    Loss: 0.168232
Train Epoch: 4 [11520/60000 (19%)]    Loss: 0.239878
Train Epoch: 4 [12160/60000 (20%)]    Loss: 0.148940
Train Epoch: 4 [12800/60000 (21%)]    Loss: 0.266876
Train Epoch: 4 [13440/60000 (22%)]    Loss: 0.169842
Train Epoch: 4 [14080/60000 (23%)]    Loss: 0.214253
Train Epoch: 4 [14720/60000 (25%)]    Loss: 0.263527
Train Epoch: 4 [15360/60000 (26%)]    Loss: 0.123500
Train Epoch: 4 [16000/60000 (27%)]    Loss: 0.115753
Train Epoch: 4 [16640/60000 (28%)]    Loss: 0.100182
Train Epoch: 4 [17280/60000 (29%)]    Loss: 0.114562
Train Epoch: 4 [17920/60000 (30%)]    Loss: 0.219643
Train Epoch: 4 [18560/60000 (31%)]    Loss: 0.450593
Train Epoch: 4 [19200/60000 (32%)]    Loss: 0.184636
Train Epoch: 4 [19840/60000 (33%)]    Loss: 0.160202
Train Epoch: 4 [20480/60000 (34%)]    Loss: 0.287054
Train Epoch: 4 [21120/60000 (35%)]    Loss: 0.153287
Train Epoch: 4 [21760/60000 (36%)]    Loss: 0.168968
Train Epoch: 4 [22400/60000 (37%)]    Loss: 0.235850
Train Epoch: 4 [23040/60000 (38%)]    Loss: 0.188171
Train Epoch: 4 [23680/60000 (39%)]    Loss: 0.182331
Train Epoch: 4 [24320/60000 (41%)]    Loss: 0.270661
Train Epoch: 4 [24960/60000 (42%)]    Loss: 0.258974
Train Epoch: 4 [25600/60000 (43%)]    Loss: 0.406845
Train Epoch: 4 [26240/60000 (44%)]    Loss: 0.310173
Train Epoch: 4 [26880/60000 (45%)]    Loss: 0.307865
Train Epoch: 4 [27520/60000 (46%)]    Loss: 0.227921
Train Epoch: 4 [28160/60000 (47%)]    Loss: 0.142454
Train Epoch: 4 [28800/60000 (48%)]    Loss: 0.229878
Train Epoch: 4 [29440/60000 (49%)]    Loss: 0.168069
Train Epoch: 4 [30080/60000 (50%)]    Loss: 0.165128
Train Epoch: 4 [30720/60000 (51%)]    Loss: 0.241977
```

```
Train Epoch: 4 [31360/60000 (52%)]    Loss: 0.149640
Train Epoch: 4 [32000/60000 (53%)]    Loss: 0.086826
Train Epoch: 4 [32640/60000 (54%)]    Loss: 0.120607
Train Epoch: 4 [33280/60000 (55%)]    Loss: 0.130650
Train Epoch: 4 [33920/60000 (57%)]    Loss: 0.146294
Train Epoch: 4 [34560/60000 (58%)]    Loss: 0.374248
Train Epoch: 4 [35200/60000 (59%)]    Loss: 0.132699
Train Epoch: 4 [35840/60000 (60%)]    Loss: 0.211805
Train Epoch: 4 [36480/60000 (61%)]    Loss: 0.216621
Train Epoch: 4 [37120/60000 (62%)]    Loss: 0.168058
Train Epoch: 4 [37760/60000 (63%)]    Loss: 0.463432
Train Epoch: 4 [38400/60000 (64%)]    Loss: 0.048025
Train Epoch: 4 [39040/60000 (65%)]    Loss: 0.153032
Train Epoch: 4 [39680/60000 (66%)]    Loss: 0.104437
Train Epoch: 4 [40320/60000 (67%)]    Loss: 0.543780
Train Epoch: 4 [40960/60000 (68%)]    Loss: 0.167554
Train Epoch: 4 [41600/60000 (69%)]    Loss: 0.268350
Train Epoch: 4 [42240/60000 (70%)]    Loss: 0.236657
Train Epoch: 4 [42880/60000 (71%)]    Loss: 0.160511
Train Epoch: 4 [43520/60000 (72%)]    Loss: 0.139191
Train Epoch: 4 [44160/60000 (74%)]    Loss: 0.245354
Train Epoch: 4 [44800/60000 (75%)]    Loss: 0.193168
Train Epoch: 4 [45440/60000 (76%)]    Loss: 0.303514
Train Epoch: 4 [46080/60000 (77%)]    Loss: 0.175007
Train Epoch: 4 [46720/60000 (78%)]    Loss: 0.303074
Train Epoch: 4 [47360/60000 (79%)]    Loss: 0.102393
Train Epoch: 4 [48000/60000 (80%)]    Loss: 0.233083
Train Epoch: 4 [48640/60000 (81%)]    Loss: 0.352059
Train Epoch: 4 [49280/60000 (82%)]    Loss: 0.134795
Train Epoch: 4 [49920/60000 (83%)]    Loss: 0.319961
Train Epoch: 4 [50560/60000 (84%)]    Loss: 0.152632
Train Epoch: 4 [51200/60000 (85%)]    Loss: 0.162685
Train Epoch: 4 [51840/60000 (86%)]    Loss: 0.334845
Train Epoch: 4 [52480/60000 (87%)]    Loss: 0.228085
Train Epoch: 4 [53120/60000 (88%)]    Loss: 0.150503
Train Epoch: 4 [53760/60000 (90%)]    Loss: 0.189772
Train Epoch: 4 [54400/60000 (91%)]    Loss: 0.367578
Train Epoch: 4 [55040/60000 (92%)]    Loss: 0.200196
Train Epoch: 4 [55680/60000 (93%)]    Loss: 0.217655
Train Epoch: 4 [56320/60000 (94%)]    Loss: 0.328913
Train Epoch: 4 [56960/60000 (95%)]    Loss: 0.258583
Train Epoch: 4 [57600/60000 (96%)]    Loss: 0.162989
Train Epoch: 4 [58240/60000 (97%)]    Loss: 0.195415
Train Epoch: 4 [58880/60000 (98%)]    Loss: 0.158460
Train Epoch: 4 [59520/60000 (99%)]    Loss: 0.106095

Test set: Average loss: 0.0002, Accuracy: 9433/10000 (94.33%)

Train Epoch: 5 [0/60000 (0%)]    Loss: 0.037035
```

```
Train Epoch: 5 [640/60000 (1%)]  Loss: 0.227738
Train Epoch: 5 [1280/60000 (2%)] Loss: 0.082651
Train Epoch: 5 [1920/60000 (3%)] Loss: 0.171763
Train Epoch: 5 [2560/60000 (4%)] Loss: 0.105608
Train Epoch: 5 [3200/60000 (5%)] Loss: 0.195185
Train Epoch: 5 [3840/60000 (6%)] Loss: 0.094172
Train Epoch: 5 [4480/60000 (7%)] Loss: 0.110361
Train Epoch: 5 [5120/60000 (9%)] Loss: 0.126802
Train Epoch: 5 [5760/60000 (10%)]     Loss: 0.309941
Train Epoch: 5 [6400/60000 (11%)]     Loss: 0.226742
Train Epoch: 5 [7040/60000 (12%)]     Loss: 0.194391
Train Epoch: 5 [7680/60000 (13%)]     Loss: 0.067222
Train Epoch: 5 [8320/60000 (14%)]     Loss: 0.271813
Train Epoch: 5 [8960/60000 (15%)]     Loss: 0.265521
Train Epoch: 5 [9600/60000 (16%)]     Loss: 0.216858
Train Epoch: 5 [10240/60000 (17%)]    Loss: 0.153872
Train Epoch: 5 [10880/60000 (18%)]    Loss: 0.176981
Train Epoch: 5 [11520/60000 (19%)]    Loss: 0.108663
Train Epoch: 5 [12160/60000 (20%)]    Loss: 0.103085
Train Epoch: 5 [12800/60000 (21%)]    Loss: 0.268325
Train Epoch: 5 [13440/60000 (22%)]    Loss: 0.244651
Train Epoch: 5 [14080/60000 (23%)]    Loss: 0.171093
Train Epoch: 5 [14720/60000 (25%)]    Loss: 0.259204
Train Epoch: 5 [15360/60000 (26%)]    Loss: 0.081320
Train Epoch: 5 [16000/60000 (27%)]    Loss: 0.083389
Train Epoch: 5 [16640/60000 (28%)]    Loss: 0.090604
Train Epoch: 5 [17280/60000 (29%)]    Loss: 0.205418
Train Epoch: 5 [17920/60000 (30%)]    Loss: 0.252469
Train Epoch: 5 [18560/60000 (31%)]    Loss: 0.107090
Train Epoch: 5 [19200/60000 (32%)]    Loss: 0.106644
Train Epoch: 5 [19840/60000 (33%)]    Loss: 0.180466
Train Epoch: 5 [20480/60000 (34%)]    Loss: 0.286833
Train Epoch: 5 [21120/60000 (35%)]    Loss: 0.119066
Train Epoch: 5 [21760/60000 (36%)]    Loss: 0.092080
Train Epoch: 5 [22400/60000 (37%)]    Loss: 0.097359
Train Epoch: 5 [23040/60000 (38%)]    Loss: 0.201388
Train Epoch: 5 [23680/60000 (39%)]    Loss: 0.136541
Train Epoch: 5 [24320/60000 (41%)]    Loss: 0.183958
Train Epoch: 5 [24960/60000 (42%)]    Loss: 0.101362
Train Epoch: 5 [25600/60000 (43%)]    Loss: 0.234108
Train Epoch: 5 [26240/60000 (44%)]    Loss: 0.068211
Train Epoch: 5 [26880/60000 (45%)]    Loss: 0.121221
Train Epoch: 5 [27520/60000 (46%)]    Loss: 0.200671
Train Epoch: 5 [28160/60000 (47%)]    Loss: 0.125699
Train Epoch: 5 [28800/60000 (48%)]    Loss: 0.291394
Train Epoch: 5 [29440/60000 (49%)]    Loss: 0.193952
Train Epoch: 5 [30080/60000 (50%)]    Loss: 0.155052
Train Epoch: 5 [30720/60000 (51%)]    Loss: 0.263918
Train Epoch: 5 [31360/60000 (52%)]    Loss: 0.202285
```

```
Train Epoch: 5 [32000/60000 (53%)]    Loss: 0.221854
Train Epoch: 5 [32640/60000 (54%)]    Loss: 0.203104
Train Epoch: 5 [33280/60000 (55%)]    Loss: 0.211586
Train Epoch: 5 [33920/60000 (57%)]    Loss: 0.061369
Train Epoch: 5 [34560/60000 (58%)]    Loss: 0.271748
Train Epoch: 5 [35200/60000 (59%)]    Loss: 0.074875
Train Epoch: 5 [35840/60000 (60%)]    Loss: 0.119089
Train Epoch: 5 [36480/60000 (61%)]    Loss: 0.135176
Train Epoch: 5 [37120/60000 (62%)]    Loss: 0.142123
Train Epoch: 5 [37760/60000 (63%)]    Loss: 0.181814
Train Epoch: 5 [38400/60000 (64%)]    Loss: 0.196054
Train Epoch: 5 [39040/60000 (65%)]    Loss: 0.204628
Train Epoch: 5 [39680/60000 (66%)]    Loss: 0.158035
Train Epoch: 5 [40320/60000 (67%)]    Loss: 0.257406
Train Epoch: 5 [40960/60000 (68%)]    Loss: 0.110188
Train Epoch: 5 [41600/60000 (69%)]    Loss: 0.165960
Train Epoch: 5 [42240/60000 (70%)]    Loss: 0.194281
Train Epoch: 5 [42880/60000 (71%)]    Loss: 0.229470
Train Epoch: 5 [43520/60000 (72%)]    Loss: 0.176932
Train Epoch: 5 [44160/60000 (74%)]    Loss: 0.103737
Train Epoch: 5 [44800/60000 (75%)]    Loss: 0.193428
Train Epoch: 5 [45440/60000 (76%)]    Loss: 0.151814
Train Epoch: 5 [46080/60000 (77%)]    Loss: 0.431736
Train Epoch: 5 [46720/60000 (78%)]    Loss: 0.231158
Train Epoch: 5 [47360/60000 (79%)]    Loss: 0.159569
Train Epoch: 5 [48000/60000 (80%)]    Loss: 0.089870
Train Epoch: 5 [48640/60000 (81%)]    Loss: 0.342602
Train Epoch: 5 [49280/60000 (82%)]    Loss: 0.117215
Train Epoch: 5 [49920/60000 (83%)]    Loss: 0.305795
Train Epoch: 5 [50560/60000 (84%)]    Loss: 0.174607
Train Epoch: 5 [51200/60000 (85%)]    Loss: 0.148222
Train Epoch: 5 [51840/60000 (86%)]    Loss: 0.257127
Train Epoch: 5 [52480/60000 (87%)]    Loss: 0.197509
Train Epoch: 5 [53120/60000 (88%)]    Loss: 0.169186
Train Epoch: 5 [53760/60000 (90%)]    Loss: 0.139825
Train Epoch: 5 [54400/60000 (91%)]    Loss: 0.185722
Train Epoch: 5 [55040/60000 (92%)]    Loss: 0.137756
Train Epoch: 5 [55680/60000 (93%)]    Loss: 0.220459
Train Epoch: 5 [56320/60000 (94%)]    Loss: 0.085893
Train Epoch: 5 [56960/60000 (95%)]    Loss: 0.272381
Train Epoch: 5 [57600/60000 (96%)]    Loss: 0.366516
Train Epoch: 5 [58240/60000 (97%)]    Loss: 0.219033
Train Epoch: 5 [58880/60000 (98%)]    Loss: 0.186628
Train Epoch: 5 [59520/60000 (99%)]    Loss: 0.166403

Test set: Average loss: 0.0002, Accuracy: 9493/10000 (94.93%)

Train Epoch: 6 [0/60000 (0%)]    Loss: 0.119170
Train Epoch: 6 [640/60000 (1%)]  Loss: 0.130688
```

```
Train Epoch: 6 [1280/60000 (2%)] Loss: 0.136593
Train Epoch: 6 [1920/60000 (3%)] Loss: 0.125210
Train Epoch: 6 [2560/60000 (4%)] Loss: 0.108154
Train Epoch: 6 [3200/60000 (5%)] Loss: 0.142415
Train Epoch: 6 [3840/60000 (6%)] Loss: 0.118383
Train Epoch: 6 [4480/60000 (7%)] Loss: 0.215263
Train Epoch: 6 [5120/60000 (9%)] Loss: 0.168005
Train Epoch: 6 [5760/60000 (10%)]     Loss: 0.113722
Train Epoch: 6 [6400/60000 (11%)]     Loss: 0.092071
Train Epoch: 6 [7040/60000 (12%)]     Loss: 0.213173
Train Epoch: 6 [7680/60000 (13%)]     Loss: 0.101998
Train Epoch: 6 [8320/60000 (14%)]     Loss: 0.142514
Train Epoch: 6 [8960/60000 (15%)]     Loss: 0.191329
Train Epoch: 6 [9600/60000 (16%)]     Loss: 0.069855
Train Epoch: 6 [10240/60000 (17%)]    Loss: 0.060735
Train Epoch: 6 [10880/60000 (18%)]    Loss: 0.168498
Train Epoch: 6 [11520/60000 (19%)]    Loss: 0.174071
Train Epoch: 6 [12160/60000 (20%)]    Loss: 0.129722
Train Epoch: 6 [12800/60000 (21%)]    Loss: 0.137127
Train Epoch: 6 [13440/60000 (22%)]    Loss: 0.244531
Train Epoch: 6 [14080/60000 (23%)]    Loss: 0.048048
Train Epoch: 6 [14720/60000 (25%)]    Loss: 0.368031
Train Epoch: 6 [15360/60000 (26%)]    Loss: 0.122677
Train Epoch: 6 [16000/60000 (27%)]    Loss: 0.095206
Train Epoch: 6 [16640/60000 (28%)]    Loss: 0.186060
Train Epoch: 6 [17280/60000 (29%)]    Loss: 0.179330
Train Epoch: 6 [17920/60000 (30%)]    Loss: 0.190554
Train Epoch: 6 [18560/60000 (31%)]    Loss: 0.222563
Train Epoch: 6 [19200/60000 (32%)]    Loss: 0.053618
Train Epoch: 6 [19840/60000 (33%)]    Loss: 0.065767
Train Epoch: 6 [20480/60000 (34%)]    Loss: 0.122745
Train Epoch: 6 [21120/60000 (35%)]    Loss: 0.082232
Train Epoch: 6 [21760/60000 (36%)]    Loss: 0.216428
Train Epoch: 6 [22400/60000 (37%)]    Loss: 0.075033
Train Epoch: 6 [23040/60000 (38%)]    Loss: 0.151762
Train Epoch: 6 [23680/60000 (39%)]    Loss: 0.085169
Train Epoch: 6 [24320/60000 (41%)]    Loss: 0.059047
Train Epoch: 6 [24960/60000 (42%)]    Loss: 0.180594
Train Epoch: 6 [25600/60000 (43%)]    Loss: 0.167010
Train Epoch: 6 [26240/60000 (44%)]    Loss: 0.071620
Train Epoch: 6 [26880/60000 (45%)]    Loss: 0.091029
Train Epoch: 6 [27520/60000 (46%)]    Loss: 0.183581
Train Epoch: 6 [28160/60000 (47%)]    Loss: 0.082995
Train Epoch: 6 [28800/60000 (48%)]    Loss: 0.049869
Train Epoch: 6 [29440/60000 (49%)]    Loss: 0.196700
Train Epoch: 6 [30080/60000 (50%)]    Loss: 0.219932
Train Epoch: 6 [30720/60000 (51%)]    Loss: 0.153494
Train Epoch: 6 [31360/60000 (52%)]    Loss: 0.075742
Train Epoch: 6 [32000/60000 (53%)]    Loss: 0.168178
Train Epoch: 6 [32640/60000 (54%)]    Loss: 0.182195
```

```
Train Epoch: 6 [33280/60000 (55%)]    Loss: 0.108173
Train Epoch: 6 [33920/60000 (57%)]    Loss: 0.263470
Train Epoch: 6 [34560/60000 (58%)]    Loss: 0.261514
Train Epoch: 6 [35200/60000 (59%)]    Loss: 0.286289
Train Epoch: 6 [35840/60000 (60%)]    Loss: 0.104242
Train Epoch: 6 [36480/60000 (61%)]    Loss: 0.100198
Train Epoch: 6 [37120/60000 (62%)]    Loss: 0.159144
Train Epoch: 6 [37760/60000 (63%)]    Loss: 0.112131
Train Epoch: 6 [38400/60000 (64%)]    Loss: 0.271304
Train Epoch: 6 [39040/60000 (65%)]    Loss: 0.171717
Train Epoch: 6 [39680/60000 (66%)]    Loss: 0.055603
Train Epoch: 6 [40320/60000 (67%)]    Loss: 0.115539
Train Epoch: 6 [40960/60000 (68%)]    Loss: 0.355108
Train Epoch: 6 [41600/60000 (69%)]    Loss: 0.324847
Train Epoch: 6 [42240/60000 (70%)]    Loss: 0.169029
Train Epoch: 6 [42880/60000 (71%)]    Loss: 0.232980
Train Epoch: 6 [43520/60000 (72%)]    Loss: 0.161198
Train Epoch: 6 [44160/60000 (74%)]    Loss: 0.097790
Train Epoch: 6 [44800/60000 (75%)]    Loss: 0.189653
Train Epoch: 6 [45440/60000 (76%)]    Loss: 0.148376
Train Epoch: 6 [46080/60000 (77%)]    Loss: 0.103982
Train Epoch: 6 [46720/60000 (78%)]    Loss: 0.179481
Train Epoch: 6 [47360/60000 (79%)]    Loss: 0.045968
Train Epoch: 6 [48000/60000 (80%)]    Loss: 0.191733
Train Epoch: 6 [48640/60000 (81%)]    Loss: 0.256029
Train Epoch: 6 [49280/60000 (82%)]    Loss: 0.148727
Train Epoch: 6 [49920/60000 (83%)]    Loss: 0.156919
Train Epoch: 6 [50560/60000 (84%)]    Loss: 0.154848
Train Epoch: 6 [51200/60000 (85%)]    Loss: 0.087250
Train Epoch: 6 [51840/60000 (86%)]    Loss: 0.253088
Train Epoch: 6 [52480/60000 (87%)]    Loss: 0.079359
Train Epoch: 6 [53120/60000 (88%)]    Loss: 0.054702
Train Epoch: 6 [53760/60000 (90%)]    Loss: 0.093199
Train Epoch: 6 [54400/60000 (91%)]    Loss: 0.240260
Train Epoch: 6 [55040/60000 (92%)]    Loss: 0.130057
Train Epoch: 6 [55680/60000 (93%)]    Loss: 0.161730
Train Epoch: 6 [56320/60000 (94%)]    Loss: 0.219895
Train Epoch: 6 [56960/60000 (95%)]    Loss: 0.107017
Train Epoch: 6 [57600/60000 (96%)]    Loss: 0.145521
Train Epoch: 6 [58240/60000 (97%)]    Loss: 0.104289
Train Epoch: 6 [58880/60000 (98%)]    Loss: 0.231292
Train Epoch: 6 [59520/60000 (99%)]    Loss: 0.080712

Test set: Average loss: 0.0002, Accuracy: 9527/10000 (95.27%)

Train Epoch: 7 [0/60000 (0%)]    Loss: 0.065666
Train Epoch: 7 [640/60000 (1%)]  Loss: 0.193708
Train Epoch: 7 [1280/60000 (2%)] Loss: 0.097652
Train Epoch: 7 [1920/60000 (3%)] Loss: 0.137837
```

```
Train Epoch: 7 [2560/60000 (4%)] Loss: 0.225077
Train Epoch: 7 [3200/60000 (5%)] Loss: 0.196466
Train Epoch: 7 [3840/60000 (6%)] Loss: 0.277018
Train Epoch: 7 [4480/60000 (7%)] Loss: 0.120534
Train Epoch: 7 [5120/60000 (9%)] Loss: 0.034948
Train Epoch: 7 [5760/60000 (10%)]     Loss: 0.068143
Train Epoch: 7 [6400/60000 (11%)]     Loss: 0.138425
Train Epoch: 7 [7040/60000 (12%)]     Loss: 0.240649
Train Epoch: 7 [7680/60000 (13%)]     Loss: 0.184034
Train Epoch: 7 [8320/60000 (14%)]     Loss: 0.334307
Train Epoch: 7 [8960/60000 (15%)]     Loss: 0.110650
Train Epoch: 7 [9600/60000 (16%)]     Loss: 0.105329
Train Epoch: 7 [10240/60000 (17%)]    Loss: 0.154707
Train Epoch: 7 [10880/60000 (18%)]    Loss: 0.122672
Train Epoch: 7 [11520/60000 (19%)]    Loss: 0.142677
Train Epoch: 7 [12160/60000 (20%)]    Loss: 0.089567
Train Epoch: 7 [12800/60000 (21%)]    Loss: 0.170745
Train Epoch: 7 [13440/60000 (22%)]    Loss: 0.149937
Train Epoch: 7 [14080/60000 (23%)]    Loss: 0.169754
Train Epoch: 7 [14720/60000 (25%)]    Loss: 0.282211
Train Epoch: 7 [15360/60000 (26%)]    Loss: 0.156820
Train Epoch: 7 [16000/60000 (27%)]    Loss: 0.123150
Train Epoch: 7 [16640/60000 (28%)]    Loss: 0.144240
Train Epoch: 7 [17280/60000 (29%)]    Loss: 0.162129
Train Epoch: 7 [17920/60000 (30%)]    Loss: 0.135628
Train Epoch: 7 [18560/60000 (31%)]    Loss: 0.094645
Train Epoch: 7 [19200/60000 (32%)]    Loss: 0.219942
Train Epoch: 7 [19840/60000 (33%)]    Loss: 0.189800
Train Epoch: 7 [20480/60000 (34%)]    Loss: 0.087631
Train Epoch: 7 [21120/60000 (35%)]    Loss: 0.081759
Train Epoch: 7 [21760/60000 (36%)]    Loss: 0.083720
Train Epoch: 7 [22400/60000 (37%)]    Loss: 0.092509
Train Epoch: 7 [23040/60000 (38%)]    Loss: 0.081165
Train Epoch: 7 [23680/60000 (39%)]    Loss: 0.094754
Train Epoch: 7 [24320/60000 (41%)]    Loss: 0.320786
Train Epoch: 7 [24960/60000 (42%)]    Loss: 0.178397
Train Epoch: 7 [25600/60000 (43%)]    Loss: 0.035499
Train Epoch: 7 [26240/60000 (44%)]    Loss: 0.206932
Train Epoch: 7 [26880/60000 (45%)]    Loss: 0.061772
Train Epoch: 7 [27520/60000 (46%)]    Loss: 0.147067
Train Epoch: 7 [28160/60000 (47%)]    Loss: 0.034949
Train Epoch: 7 [28800/60000 (48%)]    Loss: 0.182671
Train Epoch: 7 [29440/60000 (49%)]    Loss: 0.096274
Train Epoch: 7 [30080/60000 (50%)]    Loss: 0.292727
Train Epoch: 7 [30720/60000 (51%)]    Loss: 0.334135
Train Epoch: 7 [31360/60000 (52%)]    Loss: 0.124802
Train Epoch: 7 [32000/60000 (53%)]    Loss: 0.056965
Train Epoch: 7 [32640/60000 (54%)]    Loss: 0.221732
Train Epoch: 7 [33280/60000 (55%)]    Loss: 0.105434
```

```
Train Epoch: 7 [33920/60000 (57%)]    Loss: 0.114826
Train Epoch: 7 [34560/60000 (58%)]    Loss: 0.110184
Train Epoch: 7 [35200/60000 (59%)]    Loss: 0.160152
Train Epoch: 7 [35840/60000 (60%)]    Loss: 0.359714
Train Epoch: 7 [36480/60000 (61%)]    Loss: 0.071537
Train Epoch: 7 [37120/60000 (62%)]    Loss: 0.217432
Train Epoch: 7 [37760/60000 (63%)]    Loss: 0.054127
Train Epoch: 7 [38400/60000 (64%)]    Loss: 0.122839
Train Epoch: 7 [39040/60000 (65%)]    Loss: 0.072030
Train Epoch: 7 [39680/60000 (66%)]    Loss: 0.294438
Train Epoch: 7 [40320/60000 (67%)]    Loss: 0.139490
Train Epoch: 7 [40960/60000 (68%)]    Loss: 0.084121
Train Epoch: 7 [41600/60000 (69%)]    Loss: 0.064844
Train Epoch: 7 [42240/60000 (70%)]    Loss: 0.096640
Train Epoch: 7 [42880/60000 (71%)]    Loss: 0.057757
Train Epoch: 7 [43520/60000 (72%)]    Loss: 0.167566
Train Epoch: 7 [44160/60000 (74%)]    Loss: 0.170017
Train Epoch: 7 [44800/60000 (75%)]    Loss: 0.163014
Train Epoch: 7 [45440/60000 (76%)]    Loss: 0.100974
Train Epoch: 7 [46080/60000 (77%)]    Loss: 0.077414
Train Epoch: 7 [46720/60000 (78%)]    Loss: 0.042244
Train Epoch: 7 [47360/60000 (79%)]    Loss: 0.234799
Train Epoch: 7 [48000/60000 (80%)]    Loss: 0.096550
Train Epoch: 7 [48640/60000 (81%)]    Loss: 0.093212
Train Epoch: 7 [49280/60000 (82%)]    Loss: 0.097280
Train Epoch: 7 [49920/60000 (83%)]    Loss: 0.122136
Train Epoch: 7 [50560/60000 (84%)]    Loss: 0.168608
Train Epoch: 7 [51200/60000 (85%)]    Loss: 0.097173
Train Epoch: 7 [51840/60000 (86%)]    Loss: 0.085396
Train Epoch: 7 [52480/60000 (87%)]    Loss: 0.059024
Train Epoch: 7 [53120/60000 (88%)]    Loss: 0.129158
Train Epoch: 7 [53760/60000 (90%)]    Loss: 0.070363
Train Epoch: 7 [54400/60000 (91%)]    Loss: 0.055275
Train Epoch: 7 [55040/60000 (92%)]    Loss: 0.061728
Train Epoch: 7 [55680/60000 (93%)]    Loss: 0.167956
Train Epoch: 7 [56320/60000 (94%)]    Loss: 0.118142
Train Epoch: 7 [56960/60000 (95%)]    Loss: 0.113459
Train Epoch: 7 [57600/60000 (96%)]    Loss: 0.055233
Train Epoch: 7 [58240/60000 (97%)]    Loss: 0.181570
Train Epoch: 7 [58880/60000 (98%)]    Loss: 0.153746
Train Epoch: 7 [59520/60000 (99%)]    Loss: 0.265797

Test set: Average loss: 0.0001, Accuracy: 9574/10000 (95.74%)

Train Epoch: 8 [0/60000 (0%)]    Loss: 0.043580
Train Epoch: 8 [640/60000 (1%)]  Loss: 0.100813
Train Epoch: 8 [1280/60000 (2%)] Loss: 0.049051
Train Epoch: 8 [1920/60000 (3%)] Loss: 0.206966
Train Epoch: 8 [2560/60000 (4%)] Loss: 0.132915
```

```
Train Epoch: 8 [3200/60000 (5%)] Loss: 0.145008
Train Epoch: 8 [3840/60000 (6%)] Loss: 0.161893
Train Epoch: 8 [4480/60000 (7%)] Loss: 0.165752
Train Epoch: 8 [5120/60000 (9%)] Loss: 0.138236
Train Epoch: 8 [5760/60000 (10%)]     Loss: 0.132547
Train Epoch: 8 [6400/60000 (11%)]     Loss: 0.144734
Train Epoch: 8 [7040/60000 (12%)]     Loss: 0.155346
Train Epoch: 8 [7680/60000 (13%)]     Loss: 0.050756
Train Epoch: 8 [8320/60000 (14%)]     Loss: 0.152679
Train Epoch: 8 [8960/60000 (15%)]     Loss: 0.269828
Train Epoch: 8 [9600/60000 (16%)]     Loss: 0.123193
Train Epoch: 8 [10240/60000 (17%)]    Loss: 0.086103
Train Epoch: 8 [10880/60000 (18%)]    Loss: 0.150241
Train Epoch: 8 [11520/60000 (19%)]    Loss: 0.229357
Train Epoch: 8 [12160/60000 (20%)]    Loss: 0.062715
Train Epoch: 8 [12800/60000 (21%)]    Loss: 0.110188
Train Epoch: 8 [13440/60000 (22%)]    Loss: 0.084459
Train Epoch: 8 [14080/60000 (23%)]    Loss: 0.138904
Train Epoch: 8 [14720/60000 (25%)]    Loss: 0.210718
Train Epoch: 8 [15360/60000 (26%)]    Loss: 0.097199
Train Epoch: 8 [16000/60000 (27%)]    Loss: 0.097868
Train Epoch: 8 [16640/60000 (28%)]    Loss: 0.187625
Train Epoch: 8 [17280/60000 (29%)]    Loss: 0.178848
Train Epoch: 8 [17920/60000 (30%)]    Loss: 0.063648
Train Epoch: 8 [18560/60000 (31%)]    Loss: 0.152629
Train Epoch: 8 [19200/60000 (32%)]    Loss: 0.051271
Train Epoch: 8 [19840/60000 (33%)]    Loss: 0.078620
Train Epoch: 8 [20480/60000 (34%)]    Loss: 0.052949
Train Epoch: 8 [21120/60000 (35%)]    Loss: 0.103100
Train Epoch: 8 [21760/60000 (36%)]    Loss: 0.075029
Train Epoch: 8 [22400/60000 (37%)]    Loss: 0.077763
Train Epoch: 8 [23040/60000 (38%)]    Loss: 0.061543
Train Epoch: 8 [23680/60000 (39%)]    Loss: 0.070903
Train Epoch: 8 [24320/60000 (41%)]    Loss: 0.061712
Train Epoch: 8 [24960/60000 (42%)]    Loss: 0.172536
Train Epoch: 8 [25600/60000 (43%)]    Loss: 0.172688
Train Epoch: 8 [26240/60000 (44%)]    Loss: 0.057184
Train Epoch: 8 [26880/60000 (45%)]    Loss: 0.082057
Train Epoch: 8 [27520/60000 (46%)]    Loss: 0.036064
Train Epoch: 8 [28160/60000 (47%)]    Loss: 0.199774
Train Epoch: 8 [28800/60000 (48%)]    Loss: 0.092629
Train Epoch: 8 [29440/60000 (49%)]    Loss: 0.257369
Train Epoch: 8 [30080/60000 (50%)]    Loss: 0.195862
Train Epoch: 8 [30720/60000 (51%)]    Loss: 0.118392
Train Epoch: 8 [31360/60000 (52%)]    Loss: 0.155049
Train Epoch: 8 [32000/60000 (53%)]    Loss: 0.191739
Train Epoch: 8 [32640/60000 (54%)]    Loss: 0.086658
Train Epoch: 8 [33280/60000 (55%)]    Loss: 0.151102
Train Epoch: 8 [33920/60000 (57%)]    Loss: 0.118492
```

```
Train Epoch: 8 [34560/60000 (58%)]    Loss: 0.145575
Train Epoch: 8 [35200/60000 (59%)]    Loss: 0.210716
Train Epoch: 8 [35840/60000 (60%)]    Loss: 0.025537
Train Epoch: 8 [36480/60000 (61%)]    Loss: 0.139156
Train Epoch: 8 [37120/60000 (62%)]    Loss: 0.093430
Train Epoch: 8 [37760/60000 (63%)]    Loss: 0.145308
Train Epoch: 8 [38400/60000 (64%)]    Loss: 0.133840
Train Epoch: 8 [39040/60000 (65%)]    Loss: 0.226701
Train Epoch: 8 [39680/60000 (66%)]    Loss: 0.098514
Train Epoch: 8 [40320/60000 (67%)]    Loss: 0.219403
Train Epoch: 8 [40960/60000 (68%)]    Loss: 0.133476
Train Epoch: 8 [41600/60000 (69%)]    Loss: 0.189790
Train Epoch: 8 [42240/60000 (70%)]    Loss: 0.276477
Train Epoch: 8 [42880/60000 (71%)]    Loss: 0.178663
Train Epoch: 8 [43520/60000 (72%)]    Loss: 0.190177
Train Epoch: 8 [44160/60000 (74%)]    Loss: 0.084455
Train Epoch: 8 [44800/60000 (75%)]    Loss: 0.151998
Train Epoch: 8 [45440/60000 (76%)]    Loss: 0.123955
Train Epoch: 8 [46080/60000 (77%)]    Loss: 0.087429
Train Epoch: 8 [46720/60000 (78%)]    Loss: 0.116242
Train Epoch: 8 [47360/60000 (79%)]    Loss: 0.110649
Train Epoch: 8 [48000/60000 (80%)]    Loss: 0.249658
Train Epoch: 8 [48640/60000 (81%)]    Loss: 0.056461
Train Epoch: 8 [49280/60000 (82%)]    Loss: 0.099228
Train Epoch: 8 [49920/60000 (83%)]    Loss: 0.057693
Train Epoch: 8 [50560/60000 (84%)]    Loss: 0.024307
Train Epoch: 8 [51200/60000 (85%)]    Loss: 0.230485
Train Epoch: 8 [51840/60000 (86%)]    Loss: 0.109013
Train Epoch: 8 [52480/60000 (87%)]    Loss: 0.175575
Train Epoch: 8 [53120/60000 (88%)]    Loss: 0.183229
Train Epoch: 8 [53760/60000 (90%)]    Loss: 0.049485
Train Epoch: 8 [54400/60000 (91%)]    Loss: 0.235038
Train Epoch: 8 [55040/60000 (92%)]    Loss: 0.087503
Train Epoch: 8 [55680/60000 (93%)]    Loss: 0.121767
Train Epoch: 8 [56320/60000 (94%)]    Loss: 0.174027
Train Epoch: 8 [56960/60000 (95%)]    Loss: 0.209633
Train Epoch: 8 [57600/60000 (96%)]    Loss: 0.072140
Train Epoch: 8 [58240/60000 (97%)]    Loss: 0.023169
Train Epoch: 8 [58880/60000 (98%)]    Loss: 0.191133
Train Epoch: 8 [59520/60000 (99%)]    Loss: 0.146467

Test set: Average loss: 0.0001, Accuracy: 9599/10000 (95.99%)

Train Epoch: 9 [0/60000 (0%)]    Loss: 0.094800
Train Epoch: 9 [640/60000 (1%)]  Loss: 0.070025
Train Epoch: 9 [1280/60000 (2%)] Loss: 0.118103
Train Epoch: 9 [1920/60000 (3%)] Loss: 0.086203
Train Epoch: 9 [2560/60000 (4%)] Loss: 0.163935
Train Epoch: 9 [3200/60000 (5%)] Loss: 0.130374
```

```
Train Epoch: 9 [3840/60000 (6%)] Loss: 0.235036
Train Epoch: 9 [4480/60000 (7%)] Loss: 0.049837
Train Epoch: 9 [5120/60000 (9%)] Loss: 0.143668
Train Epoch: 9 [5760/60000 (10%)]     Loss: 0.092018
Train Epoch: 9 [6400/60000 (11%)]     Loss: 0.054158
Train Epoch: 9 [7040/60000 (12%)]     Loss: 0.186619
Train Epoch: 9 [7680/60000 (13%)]     Loss: 0.121485
Train Epoch: 9 [8320/60000 (14%)]     Loss: 0.206535
Train Epoch: 9 [8960/60000 (15%)]     Loss: 0.165323
Train Epoch: 9 [9600/60000 (16%)]     Loss: 0.131350
Train Epoch: 9 [10240/60000 (17%)]    Loss: 0.157649
Train Epoch: 9 [10880/60000 (18%)]    Loss: 0.072938
Train Epoch: 9 [11520/60000 (19%)]    Loss: 0.067837
Train Epoch: 9 [12160/60000 (20%)]    Loss: 0.128134
Train Epoch: 9 [12800/60000 (21%)]    Loss: 0.184258
Train Epoch: 9 [13440/60000 (22%)]    Loss: 0.057436
Train Epoch: 9 [14080/60000 (23%)]    Loss: 0.073414
Train Epoch: 9 [14720/60000 (25%)]    Loss: 0.052181
Train Epoch: 9 [15360/60000 (26%)]    Loss: 0.130819
Train Epoch: 9 [16000/60000 (27%)]    Loss: 0.148398
Train Epoch: 9 [16640/60000 (28%)]    Loss: 0.102811
Train Epoch: 9 [17280/60000 (29%)]    Loss: 0.040854
Train Epoch: 9 [17920/60000 (30%)]    Loss: 0.139931
Train Epoch: 9 [18560/60000 (31%)]    Loss: 0.081887
Train Epoch: 9 [19200/60000 (32%)]    Loss: 0.268805
Train Epoch: 9 [19840/60000 (33%)]    Loss: 0.052738
Train Epoch: 9 [20480/60000 (34%)]    Loss: 0.075251
Train Epoch: 9 [21120/60000 (35%)]    Loss: 0.087924
Train Epoch: 9 [21760/60000 (36%)]    Loss: 0.164027
Train Epoch: 9 [22400/60000 (37%)]    Loss: 0.143117
Train Epoch: 9 [23040/60000 (38%)]    Loss: 0.137214
Train Epoch: 9 [23680/60000 (39%)]    Loss: 0.135126
Train Epoch: 9 [24320/60000 (41%)]    Loss: 0.129734
Train Epoch: 9 [24960/60000 (42%)]    Loss: 0.087017
Train Epoch: 9 [25600/60000 (43%)]    Loss: 0.073645
Train Epoch: 9 [26240/60000 (44%)]    Loss: 0.151114
Train Epoch: 9 [26880/60000 (45%)]    Loss: 0.081922
Train Epoch: 9 [27520/60000 (46%)]    Loss: 0.024983
Train Epoch: 9 [28160/60000 (47%)]    Loss: 0.101396
Train Epoch: 9 [28800/60000 (48%)]    Loss: 0.045553
Train Epoch: 9 [29440/60000 (49%)]    Loss: 0.454872
Train Epoch: 9 [30080/60000 (50%)]    Loss: 0.182975
Train Epoch: 9 [30720/60000 (51%)]    Loss: 0.138533
Train Epoch: 9 [31360/60000 (52%)]    Loss: 0.110087
Train Epoch: 9 [32000/60000 (53%)]    Loss: 0.169401
Train Epoch: 9 [32640/60000 (54%)]    Loss: 0.045065
Train Epoch: 9 [33280/60000 (55%)]    Loss: 0.045008
Train Epoch: 9 [33920/60000 (57%)]    Loss: 0.166024
Train Epoch: 9 [34560/60000 (58%)]    Loss: 0.049193
```

```
Train Epoch: 9 [35200/60000 (59%)]    Loss: 0.104530
Train Epoch: 9 [35840/60000 (60%)]    Loss: 0.032054
Train Epoch: 9 [36480/60000 (61%)]    Loss: 0.089103
Train Epoch: 9 [37120/60000 (62%)]    Loss: 0.077604
Train Epoch: 9 [37760/60000 (63%)]    Loss: 0.254572
Train Epoch: 9 [38400/60000 (64%)]    Loss: 0.082200
Train Epoch: 9 [39040/60000 (65%)]    Loss: 0.089594
Train Epoch: 9 [39680/60000 (66%)]    Loss: 0.116280
Train Epoch: 9 [40320/60000 (67%)]    Loss: 0.080694
Train Epoch: 9 [40960/60000 (68%)]    Loss: 0.116060
Train Epoch: 9 [41600/60000 (69%)]    Loss: 0.076579
Train Epoch: 9 [42240/60000 (70%)]    Loss: 0.308299
Train Epoch: 9 [42880/60000 (71%)]    Loss: 0.075854
Train Epoch: 9 [43520/60000 (72%)]    Loss: 0.020410
Train Epoch: 9 [44160/60000 (74%)]    Loss: 0.134023
Train Epoch: 9 [44800/60000 (75%)]    Loss: 0.171895
Train Epoch: 9 [45440/60000 (76%)]    Loss: 0.055527
Train Epoch: 9 [46080/60000 (77%)]    Loss: 0.041747
Train Epoch: 9 [46720/60000 (78%)]    Loss: 0.144205
Train Epoch: 9 [47360/60000 (79%)]    Loss: 0.084967
Train Epoch: 9 [48000/60000 (80%)]    Loss: 0.223994
Train Epoch: 9 [48640/60000 (81%)]    Loss: 0.085269
Train Epoch: 9 [49280/60000 (82%)]    Loss: 0.074512
Train Epoch: 9 [49920/60000 (83%)]    Loss: 0.048974
Train Epoch: 9 [50560/60000 (84%)]    Loss: 0.044393
Train Epoch: 9 [51200/60000 (85%)]    Loss: 0.241547
Train Epoch: 9 [51840/60000 (86%)]    Loss: 0.087161
Train Epoch: 9 [52480/60000 (87%)]    Loss: 0.192944
Train Epoch: 9 [53120/60000 (88%)]    Loss: 0.048835
Train Epoch: 9 [53760/60000 (90%)]    Loss: 0.211133
Train Epoch: 9 [54400/60000 (91%)]    Loss: 0.162306
Train Epoch: 9 [55040/60000 (92%)]    Loss: 0.096340
Train Epoch: 9 [55680/60000 (93%)]    Loss: 0.050511
Train Epoch: 9 [56320/60000 (94%)]    Loss: 0.178259
Train Epoch: 9 [56960/60000 (95%)]    Loss: 0.122572
Train Epoch: 9 [57600/60000 (96%)]    Loss: 0.185023
Train Epoch: 9 [58240/60000 (97%)]    Loss: 0.038295
Train Epoch: 9 [58880/60000 (98%)]    Loss: 0.166517
Train Epoch: 9 [59520/60000 (99%)]    Loss: 0.213537

Test set: Average loss: 0.0001, Accuracy: 9655/10000 (96.55%)

Train Epoch: 10 [0/60000 (0%)]   Loss: 0.182055
Train Epoch: 10 [640/60000 (1%)] Loss: 0.149588
Train Epoch: 10 [1280/60000 (2%)]     Loss: 0.102628
Train Epoch: 10 [1920/60000 (3%)]     Loss: 0.067963
Train Epoch: 10 [2560/60000 (4%)]     Loss: 0.075693
Train Epoch: 10 [3200/60000 (5%)]     Loss: 0.038027
Train Epoch: 10 [3840/60000 (6%)]     Loss: 0.199632
```

```
Train Epoch: 10 [4480/60000 (7%)]     Loss: 0.070721
Train Epoch: 10 [5120/60000 (9%)]     Loss: 0.071497
Train Epoch: 10 [5760/60000 (10%)]    Loss: 0.022942
Train Epoch: 10 [6400/60000 (11%)]    Loss: 0.060874
Train Epoch: 10 [7040/60000 (12%)]    Loss: 0.025377
Train Epoch: 10 [7680/60000 (13%)]    Loss: 0.044087
Train Epoch: 10 [8320/60000 (14%)]    Loss: 0.112326
Train Epoch: 10 [8960/60000 (15%)]    Loss: 0.060765
Train Epoch: 10 [9600/60000 (16%)]    Loss: 0.153373
Train Epoch: 10 [10240/60000 (17%)]   Loss: 0.080837
Train Epoch: 10 [10880/60000 (18%)]   Loss: 0.074016
Train Epoch: 10 [11520/60000 (19%)]   Loss: 0.132806
Train Epoch: 10 [12160/60000 (20%)]   Loss: 0.139702
Train Epoch: 10 [12800/60000 (21%)]   Loss: 0.127827
Train Epoch: 10 [13440/60000 (22%)]   Loss: 0.091695
Train Epoch: 10 [14080/60000 (23%)]   Loss: 0.113311
Train Epoch: 10 [14720/60000 (25%)]   Loss: 0.105374
Train Epoch: 10 [15360/60000 (26%)]   Loss: 0.111783
Train Epoch: 10 [16000/60000 (27%)]   Loss: 0.087568
Train Epoch: 10 [16640/60000 (28%)]   Loss: 0.150396
Train Epoch: 10 [17280/60000 (29%)]   Loss: 0.050662
Train Epoch: 10 [17920/60000 (30%)]   Loss: 0.053664
Train Epoch: 10 [18560/60000 (31%)]   Loss: 0.062997
Train Epoch: 10 [19200/60000 (32%)]   Loss: 0.070610
Train Epoch: 10 [19840/60000 (33%)]   Loss: 0.129807
Train Epoch: 10 [20480/60000 (34%)]   Loss: 0.059906
Train Epoch: 10 [21120/60000 (35%)]   Loss: 0.043476
Train Epoch: 10 [21760/60000 (36%)]   Loss: 0.114130
Train Epoch: 10 [22400/60000 (37%)]   Loss: 0.183541
Train Epoch: 10 [23040/60000 (38%)]   Loss: 0.049511
Train Epoch: 10 [23680/60000 (39%)]   Loss: 0.077879
Train Epoch: 10 [24320/60000 (41%)]   Loss: 0.149653
Train Epoch: 10 [24960/60000 (42%)]   Loss: 0.063459
Train Epoch: 10 [25600/60000 (43%)]   Loss: 0.234501
Train Epoch: 10 [26240/60000 (44%)]   Loss: 0.135858
Train Epoch: 10 [26880/60000 (45%)]   Loss: 0.087806
Train Epoch: 10 [27520/60000 (46%)]   Loss: 0.123310
Train Epoch: 10 [28160/60000 (47%)]   Loss: 0.076746
Train Epoch: 10 [28800/60000 (48%)]   Loss: 0.045406
Train Epoch: 10 [29440/60000 (49%)]   Loss: 0.146499
Train Epoch: 10 [30080/60000 (50%)]   Loss: 0.077024
Train Epoch: 10 [30720/60000 (51%)]   Loss: 0.146844
Train Epoch: 10 [31360/60000 (52%)]   Loss: 0.018045
Train Epoch: 10 [32000/60000 (53%)]   Loss: 0.049543
Train Epoch: 10 [32640/60000 (54%)]   Loss: 0.187671
Train Epoch: 10 [33280/60000 (55%)]   Loss: 0.047239
Train Epoch: 10 [33920/60000 (57%)]   Loss: 0.175669
Train Epoch: 10 [34560/60000 (58%)]   Loss: 0.072147
Train Epoch: 10 [35200/60000 (59%)]   Loss: 0.124782
```

```
Train Epoch: 10 [35840/60000 (60%)]   Loss: 0.105171
Train Epoch: 10 [36480/60000 (61%)]   Loss: 0.039650
Train Epoch: 10 [37120/60000 (62%)]   Loss: 0.068943
Train Epoch: 10 [37760/60000 (63%)]   Loss: 0.100085
Train Epoch: 10 [38400/60000 (64%)]   Loss: 0.074036
Train Epoch: 10 [39040/60000 (65%)]   Loss: 0.155978
Train Epoch: 10 [39680/60000 (66%)]   Loss: 0.130474
Train Epoch: 10 [40320/60000 (67%)]   Loss: 0.051890
Train Epoch: 10 [40960/60000 (68%)]   Loss: 0.103796
Train Epoch: 10 [41600/60000 (69%)]   Loss: 0.085703
Train Epoch: 10 [42240/60000 (70%)]   Loss: 0.131146
Train Epoch: 10 [42880/60000 (71%)]   Loss: 0.098772
Train Epoch: 10 [43520/60000 (72%)]   Loss: 0.098010
Train Epoch: 10 [44160/60000 (74%)]   Loss: 0.113764
Train Epoch: 10 [44800/60000 (75%)]   Loss: 0.061963
Train Epoch: 10 [45440/60000 (76%)]   Loss: 0.142045
Train Epoch: 10 [46080/60000 (77%)]   Loss: 0.109076
Train Epoch: 10 [46720/60000 (78%)]   Loss: 0.031691
Train Epoch: 10 [47360/60000 (79%)]   Loss: 0.068805
Train Epoch: 10 [48000/60000 (80%)]   Loss: 0.108143
Train Epoch: 10 [48640/60000 (81%)]   Loss: 0.042312
Train Epoch: 10 [49280/60000 (82%)]   Loss: 0.081605
Train Epoch: 10 [49920/60000 (83%)]   Loss: 0.101094
Train Epoch: 10 [50560/60000 (84%)]   Loss: 0.043878
Train Epoch: 10 [51200/60000 (85%)]   Loss: 0.193092
Train Epoch: 10 [51840/60000 (86%)]   Loss: 0.029237
Train Epoch: 10 [52480/60000 (87%)]   Loss: 0.076183
Train Epoch: 10 [53120/60000 (88%)]   Loss: 0.147370
Train Epoch: 10 [53760/60000 (90%)]   Loss: 0.212126
Train Epoch: 10 [54400/60000 (91%)]   Loss: 0.077711
Train Epoch: 10 [55040/60000 (92%)]   Loss: 0.227762
Train Epoch: 10 [55680/60000 (93%)]   Loss: 0.084734
Train Epoch: 10 [56320/60000 (94%)]   Loss: 0.080987
Train Epoch: 10 [56960/60000 (95%)]   Loss: 0.101684
Train Epoch: 10 [57600/60000 (96%)]   Loss: 0.076124
Train Epoch: 10 [58240/60000 (97%)]   Loss: 0.091034
Train Epoch: 10 [58880/60000 (98%)]   Loss: 0.048990
Train Epoch: 10 [59520/60000 (99%)]   Loss: 0.111287

Test set: Average loss: 0.0001, Accuracy: 9672/10000 (96.72%)
```

Question 6

Is training faster now that it is on a GPU? Is the speedup what you would expect? Why or why not? Edit the cell below to answer.

Is training faster on a GPU? Yes, training is typically faster on a GPU than on a CPU, especially for deep learning models with large datasets. GPUs are designed for parallel computations, making

them well-suited for operations like matrix multiplications, which are fundamental in deep learning.

Is the speedup what you would expect? The speedup might not always meet expectations for the following reasons:

Model Complexity:

- The MLP model used here is relatively small (only a few layers and parameters). For such small models, the overhead of transferring data between CPU and GPU can offset the benefits of GPU acceleration.

Batch Size:

- GPUs perform better with larger batch sizes because they can leverage their parallelism more effectively. With small batch sizes (e.g., 64), the GPU might not reach full utilization.

Dataset Size:

- The MNIST dataset is relatively small. Loading and transferring this data to the GPU might not fully utilize the GPU's processing power.

Hardware:

- While the NVIDIA T4 GPU is efficient for various tasks, its computational power is more suitable for inference or medium-sized models rather than very large-scale training.

Data Loading Bottleneck:

- If the data loading process (handled by the `DataLoader`) is slow, it can create a bottleneck that prevents the GPU from being fully utilized.

Conclusion Training on the GPU is likely faster but may not show dramatic speed improvements due to the simplicity of the model and dataset. To see more substantial speedups, use larger models, datasets, and batch sizes, or optimize the data pipeline.

Another Model Type: CNN

Until now you have trained a simple MLP for MNIST classification, however, MLPs are not a particularly good for images.

Firstly, using a MLP will require that all images have the same size and shape, since they are unrolled in the input.

Secondly, in general images can make use of translation invariance (a type of data symmetry), but this cannot but leveraged with a MLP.

For these reasons, a convolutional network is more appropriate, as it will pass kernels over the 2D image, removing the requirement for a fixed image size and leveraging the translation invariance of the 2D images.

Let's define a simple CNN below.

```
# Define the CNN model
class CNN(nn.Module):
    # define the constructor for the network
    def __init__(self):
        super().__init__()
        # instead of declaring the layers independently, let's use the
nn.Sequential feature
        # these blocks will be executed in list order

        # you will break up the model into two parts:
        # 1) the convolutional network
        # 2) the prediction head (a small MLP)

        # the convolutional network
        self.net = nn.Sequential(
          nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),  # the
input projection layer - note that a stride of 1 means you are not
down-sampling
          nn.ReLU(),                                             #
activation
          nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1), # an
inner layer - note that a stride of 2 means you are down sampling. The
output is 28x28 -> 14x14
          nn.ReLU(),                                             #
activation
          nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),# an
inner layer - note that a stride of 2 means you are down sampling. The
output is 14x14 -> 7x7
          nn.ReLU(),                                             #
activation
          nn.AdaptiveMaxPool2d(1),                               # a
pooling layer which will output a 1x1 vector for the prediciton head
        )

        # the prediction head
        self.head = nn.Sequential(
          nn.Linear(128, 64),      # input projection, the output from
the pool layer is a 128 element vector
          nn.ReLU(),               # activation
          nn.Linear(64, 10)        # class projection to one of the 10
classes (digits 0-9)
        )


    # define the forward pass compute graph
    def forward(self, x):

        # pass the input through the convolution network
        x = self.net(x)
```

```
        # reshape the output from Bx128x1x1 to Bx128
        x = x.view(x.size(0), -1)

        # pass the pooled vector into the prediction head
        x = self.head(x)

        # the output here is Bx10
        return x

# create the model
model = CNN()

# print the model and the parameter count
print(model)
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")

# the loss function
criterion = nn.CrossEntropyLoss()

# then you can intantiate the optimizer. You will use Stochastic
Gradient Descent (SGD), and can set the learning rate to 0.1 with a
# momentum factor of 0.5
# the first input to the optimizer is the list of model parameters,
which is obtained by calling .parameters() on the model object
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

```
CNN(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
    (5): ReLU()
    (6): AdaptiveMaxPool2d(output_size=1)
  )
  (head): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
Model has 101,578 trainable parameters
```

Question 7

Notice that this model now has fewer parameters than the MLP. Let's see how it trains.

Using the previous code to train on the CPU with timing, edit the cell below to execute 2 epochs of training.

```
# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0

# train for 2 epochs on the CPU
import time

# Ensure the model is on the correct device
model = model.to(device)

# Training and testing code
for epoch in range(1, num_epochs + 1):
    current_epoch = epoch
    print(f"Epoch {epoch}/{num_epochs}")

    # Training
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data and target to the same device as the model
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()  # Zero gradients
        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Compute loss
        loss.backward()  # Backward pass
        optimizer.step()  # Update parameters

        # Logging loss
        train_losses.append(loss.item())
        train_steps.append(current_step)
        current_step += 1

        if batch_idx % 10 == 0:
            print(f"Step {current_step}, Loss: {loss.item():.4f}")

    # Testing after each epoch
    model.eval()
    epoch_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            # Move data and target to the same device as the model
            data, target = data.to(device), target.to(device)
```

```
            output = model(data)  # Forward pass
            loss = criterion(output, target)  # Compute loss
            epoch_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)  # Predictions
            correct += pred.eq(target.view_as(pred)).sum().item()

    # Log test results
    test_loss = epoch_loss / len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    test_losses.append(test_loss)
    test_accuracy.append(accuracy)
    test_steps.append(current_epoch)

    print(f"\nTest set: Average loss: {test_loss:.4f}, Accuracy:
{correct}/{len(test_loader.dataset)} ({accuracy:.2f}%)\n")
```

```
Epoch 1/2
Step 1, Loss: 2.3040
Step 11, Loss: 2.3054
Step 21, Loss: 2.3037
Step 31, Loss: 2.3041
Step 41, Loss: 2.3070
Step 51, Loss: 2.2996
Step 61, Loss: 2.3046
Step 71, Loss: 2.3040
Step 81, Loss: 2.2981
Step 91, Loss: 2.3088
Step 101, Loss: 2.3009
Step 111, Loss: 2.2971
Step 121, Loss: 2.3065
Step 131, Loss: 2.2999
Step 141, Loss: 2.3001
Step 151, Loss: 2.2941
Step 161, Loss: 2.3093
Step 171, Loss: 2.3077
Step 181, Loss: 2.3008
Step 191, Loss: 2.3112
Step 201, Loss: 2.2940
Step 211, Loss: 2.2974
Step 221, Loss: 2.2996
Step 231, Loss: 2.3030
Step 241, Loss: 2.2989
Step 251, Loss: 2.2904
Step 261, Loss: 2.2807
Step 271, Loss: 2.2973
Step 281, Loss: 2.2908
Step 291, Loss: 2.2854
Step 301, Loss: 2.2969
Step 311, Loss: 2.2922
Step 321, Loss: 2.2869
```

```
Step 331, Loss: 2.2887
Step 341, Loss: 2.2847
Step 351, Loss: 2.3012
Step 361, Loss: 2.2840
Step 371, Loss: 2.2872
Step 381, Loss: 2.2873
Step 391, Loss: 2.2906
Step 401, Loss: 2.2904
Step 411, Loss: 2.2829
Step 421, Loss: 2.2777
Step 431, Loss: 2.2707
Step 441, Loss: 2.2692
Step 451, Loss: 2.2924
Step 461, Loss: 2.2792
Step 471, Loss: 2.2884
Step 481, Loss: 2.2812
Step 491, Loss: 2.2734
Step 501, Loss: 2.2746
Step 511, Loss: 2.2779
Step 521, Loss: 2.2735
Step 531, Loss: 2.2585
Step 541, Loss: 2.2715
Step 551, Loss: 2.2639
Step 561, Loss: 2.2506
Step 571, Loss: 2.2699
Step 581, Loss: 2.2575
Step 591, Loss: 2.2647
Step 601, Loss: 2.2511
Step 611, Loss: 2.2679
Step 621, Loss: 2.2504
Step 631, Loss: 2.2520
Step 641, Loss: 2.2359
Step 651, Loss: 2.2552
Step 661, Loss: 2.2350
Step 671, Loss: 2.2423
Step 681, Loss: 2.2232
Step 691, Loss: 2.2509
Step 701, Loss: 2.2107
Step 711, Loss: 2.2190
Step 721, Loss: 2.1739
Step 731, Loss: 2.1812
Step 741, Loss: 2.1765
Step 751, Loss: 2.1901
Step 761, Loss: 2.1532
Step 771, Loss: 2.1292
Step 781, Loss: 2.1284
Step 791, Loss: 2.1095
Step 801, Loss: 2.0769
Step 811, Loss: 2.1053
```

```
Step 821, Loss: 1.9104
Step 831, Loss: 1.9845
Step 841, Loss: 1.9946
Step 851, Loss: 1.8751
Step 861, Loss: 1.8146
Step 871, Loss: 1.8456
Step 881, Loss: 1.7737
Step 891, Loss: 1.6279
Step 901, Loss: 1.7294
Step 911, Loss: 1.7201
Step 921, Loss: 1.6088
Step 931, Loss: 1.6245

Test set: Average loss: 0.0015, Accuracy: 5398/10000 (53.98%)

Epoch 2/2
Step 939, Loss: 1.4693
Step 949, Loss: 1.3983
Step 959, Loss: 1.3601
Step 969, Loss: 1.4290
Step 979, Loss: 1.3072
Step 989, Loss: 1.3396
Step 999, Loss: 1.2072
Step 1009, Loss: 1.2571
Step 1019, Loss: 1.2174
Step 1029, Loss: 1.2111
Step 1039, Loss: 1.1835
Step 1049, Loss: 0.9803
Step 1059, Loss: 1.1354
Step 1069, Loss: 1.0603
Step 1079, Loss: 0.9080
Step 1089, Loss: 0.7911
Step 1099, Loss: 0.9127
Step 1109, Loss: 0.9735
Step 1119, Loss: 0.8407
Step 1129, Loss: 0.7447
Step 1139, Loss: 1.1026
Step 1149, Loss: 0.7805
Step 1159, Loss: 0.9298
Step 1169, Loss: 0.6641
Step 1179, Loss: 0.7639
Step 1189, Loss: 0.6375
Step 1199, Loss: 0.5872
Step 1209, Loss: 0.5196
Step 1219, Loss: 0.6984
Step 1229, Loss: 1.0133
Step 1239, Loss: 0.8820
Step 1249, Loss: 0.7312
Step 1259, Loss: 0.5010
```

```
Step 1269, Loss: 0.7657
Step 1279, Loss: 0.5721
Step 1289, Loss: 0.7561
Step 1299, Loss: 0.8135
Step 1309, Loss: 0.8685
Step 1319, Loss: 0.6402
Step 1329, Loss: 0.4427
Step 1339, Loss: 0.6751
Step 1349, Loss: 0.6997
Step 1359, Loss: 0.6873
Step 1369, Loss: 0.6011
Step 1379, Loss: 0.4921
Step 1389, Loss: 0.3950
Step 1399, Loss: 0.4084
Step 1409, Loss: 0.6190
Step 1419, Loss: 0.4701
Step 1429, Loss: 0.4548
Step 1439, Loss: 0.3950
Step 1449, Loss: 0.5902
Step 1459, Loss: 0.3906
Step 1469, Loss: 0.4253
Step 1479, Loss: 0.3935
Step 1489, Loss: 0.3431
Step 1499, Loss: 0.5019
Step 1509, Loss: 0.3539
Step 1519, Loss: 0.5340
Step 1529, Loss: 0.3691
Step 1539, Loss: 0.4302
Step 1549, Loss: 0.5541
Step 1559, Loss: 0.7497
Step 1569, Loss: 0.2571
Step 1579, Loss: 0.2759
Step 1589, Loss: 0.5001
Step 1599, Loss: 0.2638
Step 1609, Loss: 0.3217
Step 1619, Loss: 0.4172
Step 1629, Loss: 0.3141
Step 1639, Loss: 0.5699
Step 1649, Loss: 0.3723
Step 1659, Loss: 0.2281
Step 1669, Loss: 0.4142
Step 1679, Loss: 0.1677
Step 1689, Loss: 0.3686
Step 1699, Loss: 0.5052
Step 1709, Loss: 0.4072
Step 1719, Loss: 0.5049
Step 1729, Loss: 0.1863
Step 1739, Loss: 0.3878
Step 1749, Loss: 0.4024
```

```
Step 1759, Loss: 0.4295
Step 1769, Loss: 0.3136
Step 1779, Loss: 0.4117
Step 1789, Loss: 0.2563
Step 1799, Loss: 0.2524
Step 1809, Loss: 0.2077
Step 1819, Loss: 0.2529
Step 1829, Loss: 0.2954
Step 1839, Loss: 0.2755
Step 1849, Loss: 0.2736
Step 1859, Loss: 0.1978
Step 1869, Loss: 0.2225

Test set: Average loss: 0.0003, Accuracy: 8918/10000 (89.18%)
```

Question 8

Now, let's move the model to the GPU and try training for 2 epochs there.

```
# create the model
model = CNN()

model.cuda()

# print the model and the parameter count
print(model)
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")

# the loss function
criterion = nn.CrossEntropyLoss()

# then you can instantiate the optimizer. You will use Stochastic
Gradient Descent (SGD), and can set the learning rate to 0.1 with a
momentum factor of 0.5
# the first input to the optimizer is the list of model parameters,
which is obtained by calling .parameters() on the model object
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

CNN(
  (net): Sequential(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (1): ReLU()
    (2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
    (3): ReLU()
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
```

```
    (5): ReLU()
    (6): AdaptiveMaxPool2d(output_size=1)
  )
  (head): Sequential(
    (0): Linear(in_features=128, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=10, bias=True)
  )
)
Model has 101,578 trainable parameters
```

```python
# create a new array to log the loss and accuracy
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
current_step = 0  # Start with global step 0
current_epoch = 0 # Start with epoch 0

# train for 2 epochs on the GPU

# Number of epochs
num_epochs = 2

# Move model to GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Training and testing loop
for epoch in range(num_epochs):
    current_epoch += 1
    model.train()  # Set model to training mode

    # Training loop
    for batch_idx, (inputs, targets) in enumerate(train_loader):
        inputs, targets = inputs.to(device), targets.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, targets)
        train_losses.append(loss.item())
        train_steps.append(current_step)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        current_step += 1  # Increment global step
```

```
    # Testing loop
    model.eval()  # Set model to evaluation mode
    correct = 0
    total = 0
    test_loss = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(test_loader):
            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, targets)
            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    # Log test statistics
    test_losses.append(test_loss / len(test_loader))
    test_accuracy.append(100. * correct / total)
    test_steps.append(current_step)

    # Print epoch summary
    print(f"Epoch {current_epoch}/{num_epochs} - "
          f"Train Loss: {train_losses[-1]:.4f}, "
          f"Test Loss: {test_losses[-1]:.4f}, "
          f"Accuracy: {test_accuracy[-1]:.2f}%")
```

```
Epoch 1/2 - Train Loss: 0.7672, Test Loss: 1.0916, Accuracy: 65.86%
Epoch 2/2 - Train Loss: 0.1616, Test Loss: 0.2717, Accuracy: 91.79%
```

Question 9

How do the CPU and GPU versions compare for the CNN? Is one faster than the other? Why do you think this is, and how does it differ from the MLP? Edit the cell below to answer.

Speed Comparison:

GPU: The GPU version is generally faster than the CPU for CNNs, especially for larger models and datasets. GPUs are optimized for parallel processing, making them well-suited for the matrix multiplications and convolutions common in CNNs. CPU: The CPU version may be slower, as CPUs are optimized for sequential processing and cannot handle the high degree of parallelism that CNNs require efficiently. Reasons for Speed Differences:

Parallel Processing: GPUs have thousands of cores designed for performing many operations simultaneously. CNN operations, like convolutions and matrix multiplications, benefit significantly from this parallelism. Memory Bandwidth: GPUs have higher memory bandwidth compared to CPUs, allowing them to handle large data flows, which is critical for deep learning tasks. Batch Processing: Larger batch sizes can be utilized on the GPU, further improving throughput. Comparison with MLP:

MLP on CPU vs. GPU: The performance difference between CPU and GPU is less pronounced for smaller models like MLPs because the computational demand is lower. For very small models or datasets, the CPU might even outperform the GPU due to the overhead of transferring data between CPU and GPU. CNNs are More GPU-Dependent: CNNs are more computationally intensive due to convolutional layers, making the GPU's advantages more apparent. When GPU May Not Be Faster:

For very small datasets or models, the time to transfer data to the GPU and back might offset the computation speedup, making the CPU faster in these scenarios.

As a final comparison, you can profile the FLOPs (floating-point operations) executed by each model. You will use the thop.profile function for this and consider an MNIST batch size of 1.

```
!pip install matplotlib-venn
!apt-get -qq install -y libfluidsynth1
!apt-get -qq install -y libarchive-dev && pip install -U libarchive
import libarchive
!apt-get -qq install -y graphviz && pip install pydot
import pydot
!pip install cartopy
import cartopy


# Install thop if not already installed (uncomment the following line
if needed)
# !pip install thop

import torch
from thop import profile  # Import profile from thop

# Define the input shape for an MNIST sample with batch_size = 1
input = torch.randn(1, 1, 28, 28)

# Create a copy of the models on the CPU
mlp_model = MLP()
cnn_model = CNN()

# Profile the MLP
flops, params = profile(mlp_model, inputs=(input,), verbose=False)
print(f"MLP has {params:,} params and uses {flops:,} FLOPs")

# Profile the CNN
flops, params = profile(cnn_model, inputs=(input,), verbose=False)
print(f"CNN has {params:,} params and uses {flops:,} FLOPs")



# # the input shape of a MNIST sample with batch_size = 1
# input = torch.randn(1, 1, 28, 28)
```

```
# # create a copy of the models on the CPU
# mlp_model = MLP()
# cnn_model = CNN()

# # profile the MLP
# flops, params = thop.profile(mlp_model, inputs=(input, ),
verbose=False)
# print(f"MLP has {params:,} params and uses {flops:,} FLOPs")

# # profile the CNN
# flops, params = thop.profile(cnn_model, inputs=(input, ),
verbose=False)
# print(f"CNN has {params:,} params and uses {flops:,} FLOPs")
```

```
Requirement already satisfied: matplotlib-venn in
/usr/local/lib/python3.11/dist-packages (1.1.1)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.11/dist-packages (from matplotlib-venn)
(3.10.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (from matplotlib-venn)
(1.26.4)
Requirement already satisfied: scipy in
/usr/local/lib/python3.11/dist-packages (from matplotlib-venn)
(1.13.1)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (4.55.3)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (24.2)
Requirement already satisfied: pillow>=8 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.11/dist-packages (from matplotlib->matplotlib-
venn) (2.8.2)
Requirement already satisfied: six>=1.5 in
```

```
/usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7-
>matplotlib->matplotlib-venn) (1.17.0)
E: Package 'libfluidsynth1' has no installation candidate
Requirement already satisfied: libarchive in
/usr/local/lib/python3.11/dist-packages (0.4.7)
Requirement already satisfied: nose in /usr/local/lib/python3.11/dist-
packages (from libarchive) (1.3.7)
Requirement already satisfied: pydot in
/usr/local/lib/python3.11/dist-packages (3.0.4)
Requirement already satisfied: pyparsing>=3.0.9 in
/usr/local/lib/python3.11/dist-packages (from pydot) (3.2.1)
Requirement already satisfied: cartopy in
/usr/local/lib/python3.11/dist-packages (0.24.1)
Requirement already satisfied: numpy>=1.23 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (1.26.4)
Requirement already satisfied: matplotlib>=3.6 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (3.10.0)
Requirement already satisfied: shapely>=1.8 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (2.0.6)
Requirement already satisfied: packaging>=21 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (24.2)
Requirement already satisfied: pyshp>=2.3 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (2.3.1)
Requirement already satisfied: pyproj>=3.3.1 in
/usr/local/lib/python3.11/dist-packages (from cartopy) (3.7.0)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (4.55.3)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (1.4.8)
Requirement already satisfied: pillow>=8 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (11.1.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (3.2.1)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.11/dist-packages (from matplotlib>=3.6-
>cartopy) (2.8.2)
Requirement already satisfied: certifi in
/usr/local/lib/python3.11/dist-packages (from pyproj>=3.3.1->cartopy)
(2024.12.14)
```

```
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7-
>matplotlib>=3.6->cartopy) (1.17.0)
MLP has 109,386.0 params and uses 109,184.0 FLOPs
CNN has 101,578.0 params and uses 7,459,968.0 FLOPs
```

Question 10

Are these results what you would have expected? Do they explain the performance difference between running on the CPU and GPU? Why or why not? Edit the cell below to answer.

Expected Results:

MLP Parameters and FLOPs: The MLP typically has fewer parameters and FLOPs compared to a CNN when processing an MNIST sample. This is because MLPs consist of fully connected layers that do not capture spatial hierarchies like convolutional layers do. CNN Parameters and FLOPs: CNNs generally have more parameters and FLOPs due to the convolutional and pooling layers, which perform computations over localized regions of the input. Performance Difference Between CPU and GPU:

MLP Performance: Since MLPs have fewer parameters and FLOPs, the performance difference between CPU and GPU is minimal. The computational workload is light, and the overhead of transferring data to the GPU might even outweigh the benefits of parallelism. CNN Performance: CNNs benefit significantly from GPU acceleration. Their computational complexity, driven by convolutional operations and large matrix multiplications, is highly parallelizable, which suits the GPU's architecture. Explaining the Results:

The profiling results (parameters and FLOPs) align with the observed performance differences. CNNs are more demanding in terms of computation, which is why GPUs provide a substantial speedup. For MLPs, the workload may not fully utilize the GPU's parallel processing capabilities, especially for small datasets like MNIST. Conclusion:

The results are consistent with expectations. CNNs leverage GPU architecture effectively due to their parallelizable operations, while MLPs show less dramatic improvements since their computational workload is simpler and often fits within the CPU's capabilities.