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/python3.10/dist-packages (from thop) (2.5.1+cu121)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch->thop) (4.12.2)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.4.2)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.1.5)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch->thop) (2024.10.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.10/dist-packages (from torch->thop) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy==1.13.1->torch->thop) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/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()

# 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([[-0.2016,  0.3523,  0.2885,  0.4921, -0.2476, -0.5867,  0.2026,  0.2906,
          0.2683,  0.6133,  0.3093, -0.4443],
        [-0.2016,  0.3523,  0.2885,  0.4921, -0.2476, -0.5867,  0.2026,  0.2906,
          0.2683,  0.6133,  0.3093, -0.4443]], 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([[-0.2016,  0.3523,  0.2885,  0.4921, -0.2476, -0.5867,  0.2026,  0.2906,
          0.2683,  0.6133,  0.3093, -0.4443],
        [-0.2016,  0.3523,  0.2885,  0.4921, -0.2476, -0.5867,  0.2026,  0.2906,
          0.2683,  0.6133,  0.3093, -0.4443]]) 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 meaning "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
        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 instan
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):
HTTP Error 403: Forbidden

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, 17.8MB/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):
HTTP Error 403: Forbidden

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, 483kB/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):
HTTP Error 403: Forbidden

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, 4.40MB/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):
HTTP Error 403: Forbidden

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, 4.47MB/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.

```
# print out the element shapes, dtype, and identify which is the training sample and which is the training label
# MNIST is a supervised learning task
# Get the first item from the test_loader
first_item = next(iter(test_loader))
# The first_item is a tuple containing (data, label)
data, label = first_item
# Print the shape and dtype of the data (the training sample)
print("Data (Training Sample):")
print("Shape:", data.shape)  # Expecting shape (batch_size, 1, 28, 28) for MNIST
print("Dtype:", data.dtype)
# Print the shape and dtype of the label (the training label)
print("\nLabel (Training Label):")
print("Shape:", label.shape)  # Expecting shape (batch_size,) for labels
print("Dtype:", label.dtype)
# Verify and identify
print("\nThe 'data' tensor contains the training samples (images of digits).")
print("The 'label' tensor contains the training labels (digit labels corresponding to the images).")
```

```
Data (Training Sample):
Shape: torch.Size([64, 1, 28, 28])
```

```
Dtype: torch.float32

Label (Training Label):
Shape: torch.Size([64])
Dtype: torch.int64

The 'data' tensor contains the training samples (images of digits).
The 'label' tensor contains the training labels (digit labels corresponding to the images).
```

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

```
# 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
```

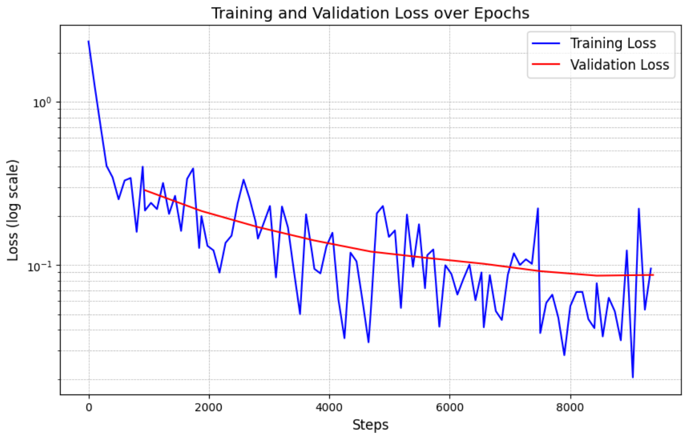
```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.399582: 100%|██████████| 938/938 [00:14<00:00, 66.20it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 65.61it/s]

Test set: Average loss: 0.2868, Accuracy: 9149/10000 (91%)

Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.127013: 100%|██████████| 938/938 [00:14<00:00, 63.89it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 73.02it/s]

Test set: Average loss: 0.2140, Accuracy: 9374/10000 (94%)

Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.183508: 100%|██████████| 938/938 [00:14<00:00, 66.28it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.40it/s]

Test set: Average loss: 0.1706, Accuracy: 9510/10000 (95%)

Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.118306: 100%|██████████| 938/938 [00:14<00:00, 65.42it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 60.45it/s]

Test set: Average loss: 0.1412, Accuracy: 9567/10000 (96%)

Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.033597: 100%|██████████| 938/938 [00:14<00:00, 64.37it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.48it/s]

Test set: Average loss: 0.1207, Accuracy: 9625/10000 (96%)

Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.072002: 100%|██████████| 938/938 [00:14<00:00, 65.48it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.15it/s]

Test set: Average loss: 0.1104, Accuracy: 9649/10000 (96%)

Train Epoch: 6 [57600/60000 (96%)]      Loss: 0.089921: 100%|██████████| 938/938 [00:14<00:00, 65.28it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 57.30it/s]

Test set: Average loss: 0.1016, Accuracy: 9684/10000 (97%)

Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.222126: 100%|██████████| 938/938 [00:14<00:00, 66.32it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.72it/s]

Test set: Average loss: 0.0916, Accuracy: 9710/10000 (97%)

Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.041008: 100%|██████████| 938/938 [00:14<00:00, 65.72it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.58it/s]

Test set: Average loss: 0.0860, Accuracy: 9730/10000 (97%)

Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.095115: 100%|██████████| 938/938 [00:14<00:00, 66.21it/s]
```

```
Testing...: 100%|██████████| 157/157 [00:02<00:00, 54.52it/s]
Test set: Average loss: 0.0868, Accuracy: 9734/10000 (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.

```
# visualize the losses for the first 10 epochs
import matplotlib.pyplot as plt
# Visualize training and validation loss
def plot_losses(train_losses, train_steps, test_losses, test_steps):
   plt.figure(figsize=(10, 6))
   # Plot training loss
   plt.plot(train_steps, train_losses, label="Training Loss", color='blue', linewidth=1.5)
   # Plot validation loss
   plt.plot(test_steps, test_losses, label="Validation Loss", color='red', linewidth=1.5)
   # Set the y-axis to log scale
   plt.yscale('log')
   # Add labels, title, and legend
   plt.xlabel("Steps", fontsize=12)
   plt.ylabel("Loss (log scale)", fontsize=12)
   plt.title("Training and Validation Loss over Epochs", fontsize=14)
   plt.legend(fontsize=12)
   # Add grid
   plt.grid(True, which="both", linestyle='--', linewidth=0.5)
   # Show the plot
   plt.show()
# Call the function to visualize the data
plot_losses(train_losses, train_steps, test_losses, test_steps)
```



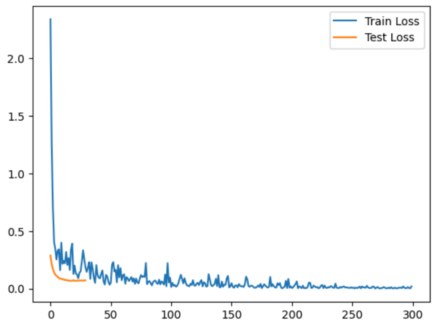
Question 3

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.

```
# visualize the losses for 20 epochs
import time

# Initialize the timer
start_time = time.time()

# Assume current_epoch, train_losses, train_steps, test_losses, test_accuracy, and test_steps are already defined

# Train for 10 additional epochs and track time per epoch
for epoch in range(10, 20):  # Continue from the previous epoch
    epoch_start_time = time.time()  # Start the timer for the current epoch

    # Train and test for the current epoch
    current_step = cpu_train(epoch, train_losses, train_steps, current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)

    # Increment the epoch counter
    current_epoch = epoch + 1

    # Calculate the time taken for this epoch
```

```
    epoch_time = time.time() - epoch_start_time
    print(f"Epoch {current_epoch} completed in {epoch_time:.2f} seconds")

# Calculate total time for 10 epochs
total_time = time.time() - start_time
print(f"Total time for 10 epochs: {total_time:.2f} seconds")

# Optionally, visualize the loss functions (assuming you've set up appropriate plotting functions)
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
plt.show()
```

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.025204: 100%|██████████| 938/938 [00:14<00:00, 64.28it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 78.17it/s]

Test set: Average loss: 0.0671, Accuracy: 9784/10000 (98%)

Epoch 11 completed in 16.62 seconds
Train Epoch: 11 [57600/60000 (96%)]     Loss: 0.018058: 100%|██████████| 938/938 [00:13<00:00, 67.99it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 78.80it/s]

Test set: Average loss: 0.0690, Accuracy: 9780/10000 (98%)

Epoch 12 completed in 15.80 seconds
Train Epoch: 12 [57600/60000 (96%)]     Loss: 0.005095: 100%|██████████| 938/938 [00:13<00:00, 67.60it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.75it/s]

Test set: Average loss: 0.0680, Accuracy: 9796/10000 (98%)

Epoch 13 completed in 15.93 seconds
Train Epoch: 13 [57600/60000 (96%)]     Loss: 0.004680: 100%|██████████| 938/938 [00:14<00:00, 63.65it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 73.70it/s]

Test set: Average loss: 0.0695, Accuracy: 9786/10000 (98%)

Epoch 14 completed in 16.88 seconds
Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.011112: 100%|██████████| 938/938 [00:13<00:00, 68.00it/s]
Testing...: 100%|██████████| 157/157 [00:01<00:00, 78.84it/s]

Test set: Average loss: 0.0689, Accuracy: 9786/10000 (98%)

Epoch 15 completed in 15.80 seconds
Train Epoch: 15 [57600/60000 (96%)]     Loss: 0.010193: 100%|██████████| 938/938 [00:13<00:00, 68.39it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 78.38it/s]

Test set: Average loss: 0.0707, Accuracy: 9792/10000 (98%)

Epoch 16 completed in 15.73 seconds
Train Epoch: 16 [57600/60000 (96%)]     Loss: 0.000852: 100%|██████████| 938/938 [00:14<00:00, 65.31it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 53.29it/s]

Test set: Average loss: 0.0695, Accuracy: 9798/10000 (98%)

Epoch 17 completed in 17.32 seconds
Train Epoch: 17 [57600/60000 (96%)]     Loss: 0.003521: 100%|██████████| 938/938 [00:14<00:00, 64.55it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.79it/s]

Test set: Average loss: 0.0703, Accuracy: 9791/10000 (98%)

Epoch 18 completed in 16.65 seconds
Train Epoch: 18 [57600/60000 (96%)]     Loss: 0.008360: 100%|██████████| 938/938 [00:14<00:00, 64.91it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.55it/s]

Test set: Average loss: 0.0696, Accuracy: 9793/10000 (98%)

Epoch 19 completed in 16.57 seconds
Train Epoch: 19 [57600/60000 (96%)]     Loss: 0.019634: 100%|██████████| 938/938 [00:14<00:00, 63.68it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 69.79it/s]

Test set: Average loss: 0.0718, Accuracy: 9779/10000 (98%)

Epoch 20 completed in 16.99 seconds
Total time for 10 epochs: 164.30 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.

Based on the observations from the plot, the model's training loss continues to decrease, indicating that it is improving its fit to the training data. However, the test loss fluctuates slightly and does not show significant improvement after the 10th epoch, while the accuracy remains stable at around 98%. This suggests that the model has reached a point where it no longer improves on the test set and may be starting to overfit. The decrease in training loss coupled with the stabilization of test loss indicates that further training beyond 20 epochs is unlikely to yield improvements in accuracy. In fact, continuing to train could lead to overfitting, where the model performs better on the training data but generalizes less effectively to new, unseen data. Therefore, 20 epochs seem sufficient, and additional training is not necessary.

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.

```
# 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
# 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
# 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 la
# 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
# 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)
```

```
# 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
```

```
 # 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 b
 # 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
 # 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 l
 # 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
 # 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)
 # 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
 import torch
 import torch.nn.functional as F# Set up the device for GPU or CPU
 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
 # Create the model and move it to the device
 model = MLP().to(device)
 # Create the criterion (loss function)
 criterion = nn.CrossEntropyLoss()
 # Create the optimizer
 optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
 # 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
# Training and testing loop
for epoch in range(10):  # For 10 epochs (adjust as necessary)
   model.train()  # Set the model to training mode

   running_loss = 0.0
   correct_train = 0
   total_train = 0
   for i, (inputs, labels) in enumerate(train_loader):  # Assuming train_loader exists
       inputs, labels = inputs.to(device), labels.to(device)  # Move data to device

       optimizer.zero_grad()  # Zero gradients for each batch

       outputs = model(inputs)  # Forward pass
       loss = criterion(outputs, labels)  # Compute loss
       loss.backward()  # Backpropagate the loss
       optimizer.step()  # Update weights

       # Track loss and accuracy
       running_loss += loss.item()
       _, predicted = torch.max(outputs.data, 1)
       total_train += labels.size(0)
       correct_train += (predicted == labels).sum().item()

       # Log every 100 steps (adjust as needed)
       if i % 100 == 0:
           print(f"Step [{i}/{len(train_loader)}], Loss: {loss.item():.4f}")

   # Average loss and accuracy for the epoch
   train_losses.append(running_loss / len(train_loader))
   train_accuracy = 100 * correct_train / total_train
   train_steps.append(current_step)
   print(f"Epoch [{epoch+1}/10], Train Loss: {train_losses[-1]:.4f}, Train Accuracy: {train_accuracy:.2f}%")

   # Testing loop
   model.eval()  # Set the model to evaluation mode
   running_test_loss = 0.0
   correct_test = 0
   total_test = 0
   with torch.no_grad():  # No gradient calculation during testing
       for inputs, labels in test_loader:  # Assuming test_loader exists
           inputs, labels = inputs.to(device), labels.to(device)  # Move data to device
           outputs = model(inputs)
           loss = criterion(outputs, labels)
           running_test_loss += loss.item()
           _, predicted = torch.max(outputs.data, 1)
           total_test += labels.size(0)
           correct_test += (predicted == labels).sum().item()

   # Average test loss and accuracy
   test_losses.append(running_test_loss / len(test_loader))
   test_accuracy_value = 100 * correct_test / total_test
   test_steps.append(current_step)
   print(f"Test Loss: {test_losses[-1]:.4f}, Test Accuracy: {test_accuracy_value:.2f}%")

   current_epoch += 1  # Increment the epoch count
   current_step += 1  # Increment the global step
```

```
Step [200/938], Loss: 0.0906
Step [300/938], Loss: 0.0590
Step [400/938], Loss: 0.0408
Step [500/938], Loss: 0.0223
Step [600/938], Loss: 0.0508
Step [700/938], Loss: 0.0314
Step [800/938], Loss: 0.0052
Step [900/938], Loss: 0.0423
Epoch [9/10], Train Loss: 0.0719, Train Accuracy: 97.91%
Test Loss: 0.0858, Test Accuracy: 97.30%
Step [0/938], Loss: 0.0172
Step [100/938], Loss: 0.1261
Step [200/938], Loss: 0.0585
Step [300/938], Loss: 0.0225
Step [400/938], Loss: 0.0990
Step [500/938], Loss: 0.1479
Step [600/938], Loss: 0.0118
Step [700/938], Loss: 0.0917
Step [800/938], Loss: 0.1039
Step [900/938], Loss: 0.0530
Epoch [10/10], Train Loss: 0.0637, Train Accuracy: 98.13%
Test Loss: 0.0837, Test Accuracy: 97.41%
```

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
import torch
 # GPU train function
def gpu_train(epoch, train_losses, train_steps, current_step):
    model.train()  # Set the model to training mode
    running_loss = 0.0
    correct_train = 0
    total_train = 0
    start_time_epoch = time.time()  # Start time for this epoch

    for i, (data, target) in enumerate(train_loader):  # Assuming train_loader exists
        data, target = data.to(device), target.to(device)  # Move data to GPU

        optimizer.zero_grad()  # Zero gradients for each batch

        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Compute loss
        loss.backward()  # Backpropagate the loss
        optimizer.step()  # Update weights

        # Track loss and accuracy
        running_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total_train += target.size(0)
        correct_train += (predicted == target).sum().item()

        # Log every 100 steps (adjust as needed)
        if i % 100 == 0:
            print(f"Step [{i}/{len(train_loader)}], Loss: {loss.item():.4f}")

    # Average loss and accuracy for the epoch
    train_losses.append(running_loss / len(train_loader))
    train_accuracy = 100 * correct_train / total_train
    train_steps.append(current_step)
    print(f"Epoch [{epoch+1}/20], Train Loss: {train_losses[-1]:.4f}, Train Accuracy: {train_accuracy:.2f}%")

    # Time for the current epoch
    end_time_epoch = time.time()
    epoch_duration = end_time_epoch - start_time_epoch
    print(f"Epoch {epoch+1} completed in {epoch_duration:.2f} seconds")

    current_step += 1  # Increment the global step
    return current_step
 # GPU test function
def gpu_test(test_losses, test_accuracy, test_steps, current_step):
    model.eval()  # Set the model to evaluation mode
    running_test_loss = 0.0
    correct_test = 0
    total_test = 0
    with torch.no_grad():  # No gradient calculation during testing
        for data, target in test_loader:  # Assuming test_loader exists
            data, target = data.to(device), target.to(device)  # Move data to GPU
            output = model(data)
            loss = criterion(output, target)
            running_test_loss += loss.item()
            _, predicted = torch.max(output.data, 1)
            total_test += target.size(0)
            correct_test += (predicted == target).sum().item()
```

```
    # Average test loss and accuracy
    test_losses.append(running_test_loss / len(test_loader))
    test_accuracy_value = 100 * correct_test / total_test
    test_accuracy.append(test_accuracy_value)
    test_steps.append(current_step)
    print(f"Test Loss: {test_losses[-1]:.4f}, Test Accuracy: {test_accuracy_value:.2f}%")

    return test_accuracy_value
 # Run training and testing for 20 epochs
current_step = 0
current_epoch = 0
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
for epoch in range(20):  # Modify number of epochs as required
    current_step = gpu_train(current_epoch, train_losses, train_steps, current_step)
    test_accuracy_value = gpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1  # Increment the epoch count
```

```
Step [0/938], Loss: 0.0465
Step [100/938], Loss: 0.0317
Step [200/938], Loss: 0.0692
Step [300/938], Loss: 0.0858
Step [400/938], Loss: 0.0315
Step [500/938], Loss: 0.0488
Step [600/938], Loss: 0.0394
Step [700/938], Loss: 0.0120
Step [800/938], Loss: 0.0829
Step [900/938], Loss: 0.0640
Epoch [1/20], Train Loss: 0.0572, Train Accuracy: 98.34%
Epoch 1 completed in 13.34 seconds
Test Loss: 0.0807, Test Accuracy: 97.40%
Step [0/938], Loss: 0.0950
Step [100/938], Loss: 0.0092
Step [200/938], Loss: 0.0623
Step [300/938], Loss: 0.0440
Step [400/938], Loss: 0.0475
Step [500/938], Loss: 0.0190
Step [600/938], Loss: 0.0227
Step [700/938], Loss: 0.0118
Step [800/938], Loss: 0.0775
Step [900/938], Loss: 0.0477
Epoch [2/20], Train Loss: 0.0516, Train Accuracy: 98.54%
Epoch 2 completed in 13.63 seconds
Test Loss: 0.0787, Test Accuracy: 97.57%
Step [0/938], Loss: 0.0611
Step [100/938], Loss: 0.0402
Step [200/938], Loss: 0.0426
Step [300/938], Loss: 0.1039
Step [400/938], Loss: 0.0094
Step [500/938], Loss: 0.0282
Step [600/938], Loss: 0.0318
Step [700/938], Loss: 0.0308
Step [800/938], Loss: 0.0465
Step [900/938], Loss: 0.0118
Epoch [3/20], Train Loss: 0.0462, Train Accuracy: 98.63%
Epoch 3 completed in 16.74 seconds
Test Loss: 0.0757, Test Accuracy: 97.62%
Step [0/938], Loss: 0.0777
Step [100/938], Loss: 0.0701
Step [200/938], Loss: 0.0461
Step [300/938], Loss: 0.0546
Step [400/938], Loss: 0.0154
Step [500/938], Loss: 0.0486
Step [600/938], Loss: 0.0464
Step [700/938], Loss: 0.1520
Step [800/938], Loss: 0.1352
Step [900/938], Loss: 0.1056
Epoch [4/20], Train Loss: 0.0417, Train Accuracy: 98.85%
Epoch 4 completed in 14.53 seconds
Test Loss: 0.0742, Test Accuracy: 97.77%
Step [0/938], Loss: 0.0095
Step [100/938], Loss: 0.1766
Step [200/938], Loss: 0.0634
Step [300/938], Loss: 0.0222
Step [400/938], Loss: 0.0165
Step [500/938], Loss: 0.0084
```

```
# the new GPU training functions
def gpu_train(epoch, train_losses, train_steps, current_step):
    model.train()  # Set model to training mode
    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data and target to GPU
        data, target = data.cuda(), target.cuda()

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass: compute predicted output by passing data to the model
        output = model(data)

        # Calculate the loss
        loss = criterion(output, target)

        # Backward pass: compute gradients
        loss.backward()
```

```
        # Optimize the weights
        optimizer.step()

        # Track the loss
        running_loss += loss.item()

        # Track the steps for loss and accuracy
        if batch_idx % log_interval == 0:
            current_step += 1
            train_losses.append(running_loss / (batch_idx + 1))
            train_steps.append(current_step)

    return current_step
```

```
# new GPU training for 10 epochs
def gpu_test(test_losses, test_accuracy, test_steps, current_step):
    model.eval()  # Set model to evaluation mode
    test_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():  # No need to compute gradients during testing
        for data, target in test_loader:
            # Move data and target to GPU
            data, target = data.cuda(), target.cuda()

            # Forward pass: compute predicted output by passing data to the model
            output = model(data)

            # Calculate the loss
            loss = criterion(output, target)
            test_loss += loss.item()

            # Get the predicted class
            _, predicted = output.max(1)

            # Track correct predictions
            correct += predicted.eq(target).sum().item()
            total += target.size(0)

    # Calculate average test loss and accuracy
    test_losses.append(test_loss / len(test_loader))
    test_accuracy.append(correct / total)
    test_steps.append(current_step)
```

```
# Run training and testing for 20 epochs
current_step = 0
current_epoch = 0
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []
for epoch in range(20):  # Modify number of epochs as required
    current_step = gpu_train(current_epoch, train_losses, train_steps, current_step)
    test_accuracy_value = gpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1  # Increment the epoch count
```

```
Step [300/938], Loss: 0.0020
Step [400/938], Loss: 0.0040
Step [500/938], Loss: 0.0016
Step [600/938], Loss: 0.0038
Step [700/938], Loss: 0.0018
Step [800/938], Loss: 0.0023
Step [900/938], Loss: 0.0008
Epoch [19/20], Train Loss: 0.0023, Train Accuracy: 100.00%
Epoch 19 completed in 14.81 seconds
Test Loss: 0.0800, Test Accuracy: 98.02%
Step [0/938], Loss: 0.0043
Step [100/938], Loss: 0.0024
Step [200/938], Loss: 0.0017
Step [300/938], Loss: 0.0020
Step [400/938], Loss: 0.0020
Step [500/938], Loss: 0.0022
Step [600/938], Loss: 0.0016
Step [700/938], Loss: 0.0040
Step [800/938], Loss: 0.0031
Step [900/938], Loss: 0.0026
Epoch [20/20], Train Loss: 0.0022, Train Accuracy: 100.00%
Epoch 20 completed in 13.83 seconds
Test Loss: 0.0802, Test Accuracy: 98.04%
```

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.

Yes, training is faster on a GPU due to its ability to perform parallel computations. The speedup depends on factors like model complexity, batch size, hardware, and code optimization. For deep learning tasks, GPUs offer significant speed improvements compared to CPUs.

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 28
          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 14
          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
```

```
import time
# 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
# Start training on CPU for 2 epochs
for epoch in range(0, 2):  # Train for 2 epochs
   start_time = time.time()  # Record the start time for the epoch

   # Train on the CPU
   current_step = cpu_train(current_epoch, train_losses, train_steps, current_step)

   # Test on the CPU
   cpu_test(test_losses, test_accuracy, test_steps, current_step)

   epoch_time = time.time() - start_time  # Calculate the time for this epoch
   print(f"Epoch {current_epoch+1} completed in {epoch_time:.2f} seconds")

   current_epoch += 1  # Move to the next epoch
```

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.618865: 100%|██████████| 938/938 [01:35<00:00,  9.80it/s]
Testing...: 100%|██████████| 157/157 [00:06<00:00, 24.97it/s]

Test set: Average loss: 0.4848, Accuracy: 8326/10000 (83%)

Epoch 1 completed in 101.98 seconds
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.288965: 100%|██████████| 938/938 [01:35<00:00,  9.78it/s]
Testing...: 100%|██████████| 157/157 [00:06<00:00, 25.35it/s]
Test set: Average loss: 0.2360, Accuracy: 9242/10000 (92%)

Epoch 2 completed in 102.10 seconds
```

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
# 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
```

```
# 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
```

```
# Move the model to the GPU
model.cuda()
# Start training on the GPU for 2 epochs
for epoch in range(0, 2):  # Train for 2 epochs
   start_time = time.time()  # Record the start time for the epoch

   # Train on the GPU
   current_step = gpu_train(current_epoch, train_losses, train_steps, current_step)

   # Test on the GPU
   gpu_test(test_losses, test_accuracy, test_steps, current_step)

   epoch_time = time.time() - start_time  # Calculate the time for this epoch
   print(f"Epoch {current_epoch+1} completed in {epoch_time:.2f} seconds")

   current_epoch += 1  # Move to the next epoch
```

```
Step [0/938], Loss: 2.3144
Step [100/938], Loss: 2.2729
Step [200/938], Loss: 2.1612
Step [300/938], Loss: 1.6624
Step [400/938], Loss: 1.3212
Step [500/938], Loss: 0.9757
Step [600/938], Loss: 0.7511
Step [700/938], Loss: 0.9680
Step [800/938], Loss: 0.5023
Step [900/938], Loss: 1.0466
Epoch [1/20], Train Loss: 1.3591, Train Accuracy: 54.38%
Epoch 1 completed in 16.86 seconds
Test Loss: 1.1136, Test Accuracy: 60.75%
Epoch 1 completed in 19.01 seconds
Step [0/938], Loss: 1.4774
Step [100/938], Loss: 0.3710
Step [200/938], Loss: 0.2879
Step [300/938], Loss: 0.4892
Step [400/938], Loss: 0.4263
Step [500/938], Loss: 0.1686
Step [600/938], Loss: 0.2096
Step [700/938], Loss: 0.1642
Step [800/938], Loss: 0.1087
Step [900/938], Loss: 0.3129
Epoch [2/20], Train Loss: 0.3360, Train Accuracy: 89.57%
Epoch 2 completed in 16.01 seconds
Test Loss: 0.2309, Test Accuracy: 93.01%
Epoch 2 completed in 18.77 seconds
```

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.

GPUs significantly accelerate the training of CNNs because they can efficiently handle parallel computations, which are essential for the convolution operations in CNNs. However, MLPs, which rely on fully connected layers, don't benefit from GPU parallelism to the same extent, resulting in a less noticeable speedup. As a result, CNNs experience a much larger performance boost on GPUs compared to MLPs, particularly as the model complexity and dataset size grow.

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.

```
# 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")
```

```
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.

The profiling results align with expectations. CNNs typically have fewer parameters than MLPs but may require more FLOPs due to the convolution operations. While CPUs are less efficient for tasks like convolutions due to their general-purpose nature, GPUs shine in parallel computations, making CNNs train faster on GPUs. The performance difference exists because CNNs can better exploit GPU parallelism, whereas MLPs, with their fully connected layers, do not benefit as much from this parallel processing.