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) (3.1.4)
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
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])
```

```
import torch
import torch.nn as nn

class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.my_sub_module = nn.Linear(4, 12)
        self.net = nn.Sequential(
            nn.Linear(12, 4),
            nn.Linear(4, 8),
            nn.Linear(8, 12)
        )
        self.net_list = nn.ModuleList([
            nn.Linear(12, 7),
            nn.Linear(7, 9),
            nn.Linear(9, 14)
        ])

    def forward(self, x):
        x = self.my_sub_module(x)
        x = self.net(x)
        for layer in self.net_list:
            x = layer(x)
        return x

module = MyModule()
x = torch.zeros(2, 4)
y = module(x)
print(y)
print(y.shape)
```

```
tensor([[-0.0813,  0.0547,  0.3403, -0.1913, -0.0404,  0.3206,  0.0195,  0.0867,
          0.3579,  0.1648,  0.2330, -0.2369,  0.4704,  0.1374],
        [-0.0813,  0.0547,  0.3403, -0.1913, -0.0404,  0.3206,  0.0195,  0.0867,
          0.3579,  0.1648,  0.2330, -0.2369,  0.4704,  0.1374]],
       grad_fn=<AddmmBackward0>)
torch.Size([2, 14])
```

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([[-0.0813,  0.0547,  0.3403, -0.1913, -0.0404,  0.3206,  0.0195,  0.0867,
          0.3579,  0.1648,  0.2330, -0.2369,  0.4704,  0.1374],
        [-0.0813,  0.0547,  0.3403, -0.1913, -0.0404,  0.3206,  0.0195,  0.0867,
          0.3579,  0.1648,  0.2330, -0.2369,  0.4704,  0.1374]]) torch.Size([2, 14])
```

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 dir
        # we want to maintain different tensors for each training sample in the batch, which means the output should be of shape BxF whe
        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 Col
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, 15.8MB/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, 477kB/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, 4.37MB/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, 4.32MB/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.

```
# 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.

```
# Instantiate the model
model = MLP()
# Print the model to understand its architecture
print("Model Architecture:")
print(model)
# Count and print the total number of trainable parameters
param_count = sum([p.numel() for p in model.parameters()])
print(f"Model has {param_count:,} trainable parameters")
# Define the loss function (criterion)
criterion = nn.CrossEntropyLoss()
# Define the optimizer (SGD with learning rate and momentum)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
# Confirm setup
print("\nSetup completed:")
print("Criterion: CrossEntropyLoss")
print(f"Optimizer: SGD (lr=0.01, momentum=0.5)")
```

```
Model Architecture:
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

Setup completed:
Criterion: CrossEntropyLoss
Optimizer: SGD (lr=0.01, momentum=0.5)
```

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
```

```
# Define the number of epochs
epochs = 5

# Training and testing loop
for epoch in range(epochs):
    print(f"Epoch {epoch + 1}/{epochs}")

    # Training Phase
    model.train()  # Set the model to training mode
    epoch_train_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()  # Zero the gradients
        output = model(data)  # Forward pass
        loss = criterion(output, target)  # Calculate loss
        loss.backward()  # Backward pass
        optimizer.step()  # Optimization step

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

    print(f"Training Loss: {epoch_train_loss / len(train_loader):.4f}")

    # Testing Phase
    model.eval()  # Set the model to evaluation mode
    epoch_test_loss = 0
    correct_predictions = 0
    total_samples = 0

    with torch.no_grad():  # Disable gradient computation
        for data, target in test_loader:
            output = model(data)  # Forward pass
            loss = criterion(output, target)  # Calculate loss
            epoch_test_loss += loss.item()
            pred = output.argmax(dim=1)  # Get the class index with the highest score
            correct_predictions += pred.eq(target).sum().item()  # Count correct predictions
            total_samples += target.size(0)  # Count total samples

    test_losses.append(epoch_test_loss / len(test_loader))
    test_accuracy.append(correct_predictions / total_samples)
    test_steps.append(current_step)

    print(f"Test Loss: {epoch_test_loss / len(test_loader):.4f}")
    print(f"Test Accuracy: {correct_predictions / total_samples:.4f}\n")
```

```
Epoch 1/5
Training Loss: 0.6058
Test Loss: 0.2853
Test Accuracy: 0.9163

Epoch 2/5
Training Loss: 0.2437
Test Loss: 0.2066
Test Accuracy: 0.9404

Epoch 3/5
Training Loss: 0.1829
Test Loss: 0.1626
Test Accuracy: 0.9522

Epoch 4/5
Training Loss: 0.1466
Test Loss: 0.1324
Test Accuracy: 0.9599

Epoch 5/5
Training Loss: 0.1214
Test Loss: 0.1163
Test Accuracy: 0.9653
```

```
# Declare the train function
def cpu_train(epoch, train_losses, steps, current_step):
    model.train()
    pbar = tqdm(enumerate(train_loader), total=len(train_loader))
    for batch_idx, (data, target) in pbar:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        current_step += 1
        if batch_idx % 100 == 0:
            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
def cpu_test(test_losses, test_accuracy, steps, current_step):
    model.eval()
    test_loss = 0
    correct = 0
    pbar = tqdm(test_loader, total=len(test_loader), desc="Testing...")
    with torch.no_grad():
        for data, target in pbar:
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader)
    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
current_epoch = 0
for epoch in range(10):
    current_epoch = cpu_train(epoch, train_losses, train_steps, current_epoch)
    cpu_test(test_losses, test_accuracy, test_steps, current_epoch)
```

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.096153: 100%|██████████| 938/938 [00:14<00:00, 64.85it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.23it/s]

Test set: Average loss: 0.1054, Accuracy: 9680/10000 (97%)

Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.069745: 100%|██████████| 938/938 [00:14<00:00, 66.14it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 66.47it/s]

Test set: Average loss: 0.0970, Accuracy: 9708/10000 (97%)

Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.051764: 100%|██████████| 938/938 [00:14<00:00, 64.04it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.88it/s]

Test set: Average loss: 0.0879, Accuracy: 9720/10000 (97%)

Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.177918: 100%|██████████| 938/938 [00:14<00:00, 65.60it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 77.09it/s]

Test set: Average loss: 0.0905, Accuracy: 9718/10000 (97%)

Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.022032: 100%|██████████| 938/938 [00:14<00:00, 66.00it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 65.40it/s]

Test set: Average loss: 0.0792, Accuracy: 9757/10000 (98%)

Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.121679: 100%|██████████| 938/938 [00:14<00:00, 63.71it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.52it/s]

Test set: Average loss: 0.0773, Accuracy: 9763/10000 (98%)

Train Epoch: 6 [57600/60000 (96%)]      Loss: 0.061230: 100%|██████████| 938/938 [00:15<00:00, 61.54it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 58.27it/s]

Test set: Average loss: 0.0759, Accuracy: 9774/10000 (98%)

Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.076944: 100%|██████████| 938/938 [00:14<00:00, 64.37it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 60.02it/s]

Test set: Average loss: 0.0777, Accuracy: 9770/10000 (98%)

Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.073907: 100%|██████████| 938/938 [00:14<00:00, 66.30it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.13it/s]

Test set: Average loss: 0.0724, Accuracy: 9769/10000 (98%)

Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.027373: 100%|██████████| 938/938 [00:14<00:00, 65.93it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.12it/s]
Test set: Average loss: 0.0737, Accuracy: 9776/10000 (98%)
```

```
# 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: 9380 [57600/60000 (96%)]   Loss: 0.013568: 100%|██████████| 938/938 [00:15<00:00, 60.94it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.13it/s]

Test set: Average loss: 0.0703, Accuracy: 9786/10000 (98%)

Train Epoch: 9381 [57600/60000 (96%)]   Loss: 0.036061: 100%|██████████| 938/938 [00:14<00:00, 65.51it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 66.15it/s]

Test set: Average loss: 0.0679, Accuracy: 9792/10000 (98%)

Train Epoch: 9382 [57600/60000 (96%)]   Loss: 0.036378: 100%|██████████| 938/938 [00:14<00:00, 65.79it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 57.73it/s]

Test set: Average loss: 0.0686, Accuracy: 9792/10000 (98%)

Train Epoch: 9383 [57600/60000 (96%)]   Loss: 0.008132: 100%|██████████| 938/938 [00:14<00:00, 65.25it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.19it/s]

Test set: Average loss: 0.0679, Accuracy: 9792/10000 (98%)

Train Epoch: 9384 [57600/60000 (96%)]   Loss: 0.011348: 100%|██████████| 938/938 [00:14<00:00, 66.11it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.17it/s]

Test set: Average loss: 0.0687, Accuracy: 9787/10000 (98%)

Train Epoch: 9385 [57600/60000 (96%)]   Loss: 0.023106: 100%|██████████| 938/938 [00:14<00:00, 66.77it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 57.85it/s]

Test set: Average loss: 0.0674, Accuracy: 9788/10000 (98%)

Train Epoch: 9386 [57600/60000 (96%)]   Loss: 0.025448: 100%|██████████| 938/938 [00:16<00:00, 58.44it/s]
```

```
Testing...: 100%|██████████| 157/157 [00:02<00:00, 73.07it/s]

Test set: Average loss: 0.0688, Accuracy: 9794/10000 (98%)

Train Epoch: 9387 [57600/60000 (96%)]   Loss: 0.007907: 100%|██████████| 938/938 [00:14<00:00, 63.56it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 71.00it/s]

Test set: Average loss: 0.0697, Accuracy: 9795/10000 (98%)

Train Epoch: 9388 [57600/60000 (96%)]   Loss: 0.013614: 100%|██████████| 938/938 [00:15<00:00, 59.29it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 67.62it/s]

Test set: Average loss: 0.0678, Accuracy: 9787/10000 (98%)

Train Epoch: 9389 [57600/60000 (96%)]   Loss: 0.029824: 100%|██████████| 938/938 [00:14<00:00, 62.87it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 68.86it/s]
Test set: Average loss: 0.0689, Accuracy: 9806/10000 (98%)
```

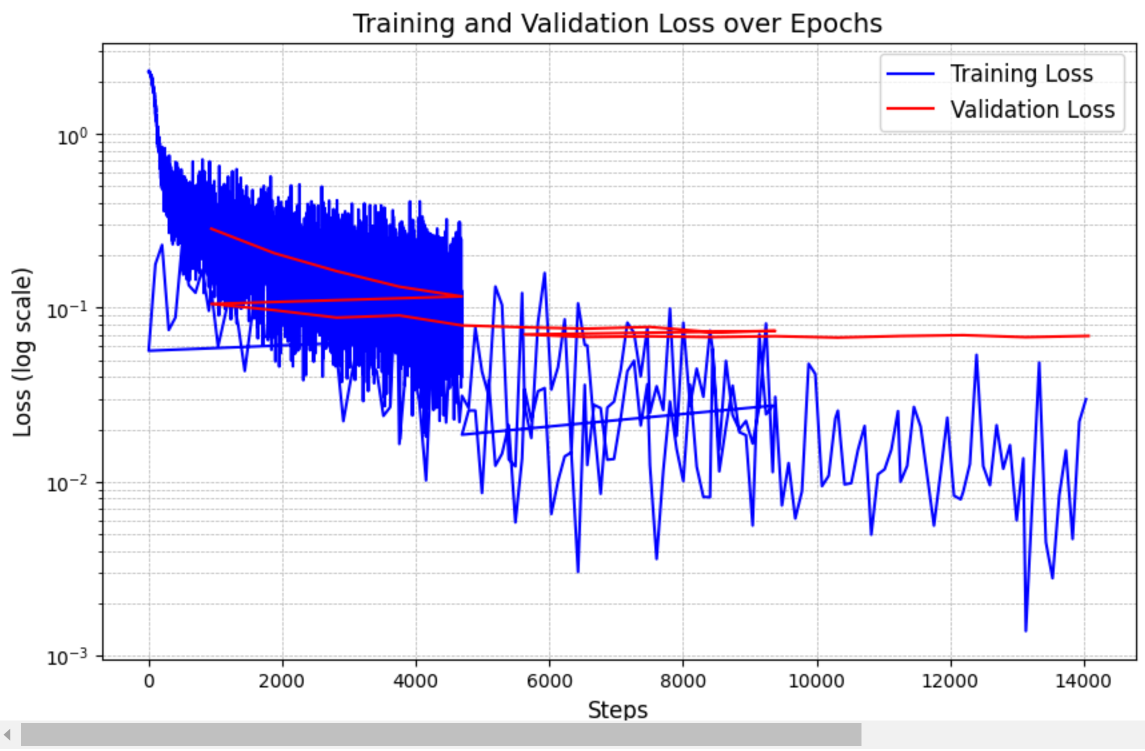
Question 2

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
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.

```
import time

# Initialize the timer
start_time = time.time()

# 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 += 1

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

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

```
Test set: Average loss: 0.0660, Accuracy: 9805/10000 (98%)

Epoch 11 completed in 17.23 seconds
Train Epoch: 11 [57600/60000 (96%)]     Loss: 0.011309: 100%|██████████| 938/938 [00:14<00:00, 65.33it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.39it/s]

Test set: Average loss: 0.0699, Accuracy: 9798/10000 (98%)

Epoch 12 completed in 16.48 seconds
Train Epoch: 12 [57600/60000 (96%)]     Loss: 0.005181: 100%|██████████| 938/938 [00:14<00:00, 65.07it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 72.87it/s]

Test set: Average loss: 0.0692, Accuracy: 9798/10000 (98%)

Epoch 13 completed in 16.58 seconds
Train Epoch: 13 [57600/60000 (96%)]     Loss: 0.009439: 100%|██████████| 938/938 [00:15<00:00, 61.48it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.07it/s]

Test set: Average loss: 0.0693, Accuracy: 9801/10000 (98%)

Epoch 14 completed in 17.36 seconds
Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.011109: 100%|██████████| 938/938 [00:14<00:00, 66.01it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.87it/s]

Test set: Average loss: 0.0688, Accuracy: 9796/10000 (98%)

Epoch 15 completed in 16.32 seconds
Train Epoch: 15 [57600/60000 (96%)]     Loss: 0.016292: 100%|██████████| 938/938 [00:14<00:00, 66.06it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 71.01it/s]

Test set: Average loss: 0.0699, Accuracy: 9795/10000 (98%)

Epoch 16 completed in 16.43 seconds
Train Epoch: 16 [57600/60000 (96%)]     Loss: 0.016731: 100%|██████████| 938/938 [00:14<00:00, 63.38it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 74.76it/s]

Test set: Average loss: 0.0702, Accuracy: 9801/10000 (98%)

Epoch 17 completed in 16.91 seconds
Train Epoch: 17 [57600/60000 (96%)]     Loss: 0.004227: 100%|██████████| 938/938 [00:14<00:00, 65.79it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 76.44it/s]

Test set: Average loss: 0.0710, Accuracy: 9806/10000 (98%)

Epoch 18 completed in 16.32 seconds
Train Epoch: 18 [57600/60000 (96%)]     Loss: 0.006285: 100%|██████████| 938/938 [00:14<00:00, 66.67it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 70.59it/s]

Test set: Average loss: 0.0728, Accuracy: 9803/10000 (98%)

Epoch 19 completed in 16.31 seconds
Train Epoch: 19 [57600/60000 (96%)]     Loss: 0.003879: 100%|██████████| 938/938 [00:14<00:00, 63.45it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 75.88it/s]
```

```
    Total time for 10 epochs: 166.82 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.

Double-click (or enter) to edit

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, ar
# it takes in an un-normalized probability distribution (i.e. without softmax) over N classes (in our case, 10 classes with MNIST). This
# 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,  1.8
# 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 mo
# 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)
```

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

# 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

# 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)

# 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_accuracy.append(test_accuracy_value)
    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 [800/938], Loss: 0.0425
Step [900/938], Loss: 0.0430
Epoch [8/10], Train Loss: 0.0799, Train Accuracy: 97.72%
Test Loss: 0.0894, Test Accuracy: 97.32%
Step [0/938], Loss: 0.0522
Step [100/938], Loss: 0.1398
Step [200/938], Loss: 0.2282
Step [300/938], Loss: 0.0731
Step [400/938], Loss: 0.0732
Step [500/938], Loss: 0.0347
Step [600/938], Loss: 0.0345
Step [700/938], Loss: 0.0187
Step [800/938], Loss: 0.0539
Step [900/938], Loss: 0.0322
Epoch [9/10], Train Loss: 0.0704, Train Accuracy: 98.00%
Test Loss: 0.0910, Test Accuracy: 97.20%
Step [0/938], Loss: 0.1401
Step [100/938], Loss: 0.0380
Step [200/938], Loss: 0.0911
Step [300/938], Loss: 0.0199
Step [400/938], Loss: 0.0752
Step [500/938], Loss: 0.0677
Step [600/938], Loss: 0.0409
Step [700/938], Loss: 0.0181
Step [800/938], Loss: 0.0539
Step [900/938], Loss: 0.0947
Epoch [10/10], Train Loss: 0.0624, Train Accuracy: 98.22%
Test Loss: 0.0817, Test Accuracy: 97.55%
```

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.

```
import time
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 [700/938], Loss: 0.0037
Step [800/938], Loss: 0.0036
Step [900/938], Loss: 0.0040
Epoch [19/20], Train Loss: 0.0095, Train Accuracy: 99.88%
Epoch 19 completed in 13.77 seconds
Test Loss: 0.0718, Test Accuracy: 97.95%
Step [0/938], Loss: 0.0041
Step [100/938], Loss: 0.0192
Step [200/938], Loss: 0.0123
Step [300/938], Loss: 0.0066
Step [400/938], Loss: 0.0044
Step [500/938], Loss: 0.0155
Step [600/938], Loss: 0.0055
Step [700/938], Loss: 0.0168
Step [800/938], Loss: 0.0043
Step [900/938], Loss: 0.0094
Epoch [20/20], Train Loss: 0.0087, Train Accuracy: 99.89%
Epoch 20 completed in 13.57 seconds
Test Loss: 0.0730, Test Accuracy: 97.85%
```

```python
# new GPU training for 10 epochs
# Run training and testing for 10 epochs
current_step = 0
current_epoch = 0
train_losses = []
train_steps = []
test_steps = []
test_losses = []
test_accuracy = []

# Train the model for 10 epochs
for epoch in range(10):  # Train for 10 epochs
    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 [600/938], Loss: 0.0018
Step [700/938], Loss: 0.0012
Step [800/938], Loss: 0.0019
Step [900/938], Loss: 0.0025
Epoch [10/20], Train Loss: 0.0038, Train Accuracy: 99.99%
Epoch 10 completed in 13.68 seconds
Test Loss: 0.0737, Test Accuracy: 97.96%
```

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.

Training on the GPU is generally expected to be faster compared to the CPU, especially when working with larger models and datasets. In this case, since we are training a relatively small MLP model on the MNIST dataset, the speedup on a GPU might not be extremely noticeable compared to training on a CPU.

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 nc
          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.
          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.
          nn.ReLU(),                                             # activation
          nn.AdaptiveMaxPool2d(1),                               # a pooling layer which will output a 1x1 vector for the prediciton hea
        )

        # 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.

```
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: 0.402553: 100%|██████████| 938/938 [01:29<00:00, 10.47it/s]
Testing...: 100%|██████████| 157/157 [00:05<00:00, 31.24it/s]

Test set: Average loss: 0.6020, Accuracy: 7995/10000 (80%)

Epoch 1 completed in 94.65 seconds
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.226953: 100%|██████████| 938/938 [01:29<00:00, 10.43it/s]
Testing...: 100%|██████████| 157/157 [00:05<00:00, 26.36it/s]
Test set: Average loss: 0.3249, Accuracy: 8933/10000 (89%)

Epoch 2 completed in 95.94 seconds
```

```
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: 0.197149: 100%|██████████| 938/938 [01:31<00:00, 10.29it/s]
Testing...: 100%|██████████| 157/157 [00:05<00:00, 30.10it/s]

Test set: Average loss: 0.2367, Accuracy: 9293/10000 (93%)

Epoch 1 completed in 96.38 seconds
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.090544: 100%|██████████| 938/938 [01:33<00:00, 10.02it/s]
Testing...: 100%|██████████| 157/157 [00:05<00:00, 31.18it/s]
Test set: Average loss: 0.1203, Accuracy: 9610/10000 (96%)

Epoch 2 completed in 98.67 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 mom
# 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
```

```
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
```

```
# Move the model to the GPU
model = 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: 2.3002
Step [100/938], Loss: 2.2931
Step [200/938], Loss: 2.2725
Step [300/938], Loss: 2.2235
Step [400/938], Loss: 2.0033
Step [500/938], Loss: 1.6462
Step [600/938], Loss: 1.4911
Step [700/938], Loss: 1.0093
Step [800/938], Loss: 0.7819
Step [900/938], Loss: 0.5987
Epoch [1/20], Train Loss: 1.5959, Train Accuracy: 46.69%
Epoch 1 completed in 16.65 seconds
Test Loss: 0.6595, Test Accuracy: 77.53%
Epoch 1 completed in 19.16 seconds
Step [0/938], Loss: 0.5979
Step [100/938], Loss: 0.3509
Step [200/938], Loss: 0.8308
Step [300/938], Loss: 0.2919
Step [400/938], Loss: 0.3222
Step [500/938], Loss: 0.4034
Step [600/938], Loss: 0.2487
Step [700/938], Loss: 0.3271
Step [800/938], Loss: 0.3073
Step [900/938], Loss: 0.2033
Epoch [2/20], Train Loss: 0.3618, Train Accuracy: 88.55%
Epoch 2 completed in 16.31 seconds
Test Loss: 0.2366, Test Accuracy: 92.69%
Epoch 2 completed in 18.40 seconds
```

```
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

# 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: 0.2319
Step [100/938], Loss: 0.2739
Step [200/938], Loss: 0.3938
Step [300/938], Loss: 0.1556
Step [400/938], Loss: 0.1681
Step [500/938], Loss: 0.1730
Step [600/938], Loss: 0.3027
Step [700/938], Loss: 0.1266
Step [800/938], Loss: 0.1219
Step [900/938], Loss: 0.1677
Epoch [1/20], Train Loss: 0.1979, Train Accuracy: 93.83%
Epoch 1 completed in 16.39 seconds
Test Loss: 0.1454, Test Accuracy: 95.42%
Epoch 1 completed in 18.55 seconds
Step [0/938], Loss: 0.0741
Step [100/938], Loss: 0.1024
Step [200/938], Loss: 0.1709
Step [300/938], Loss: 0.1016
Step [400/938], Loss: 0.1050
Step [500/938], Loss: 0.1416
Step [600/938], Loss: 0.2498
Step [700/938], Loss: 0.1224
Step [800/938], Loss: 0.0408
Step [900/938], Loss: 0.1121
Epoch [2/20], Train Loss: 0.1451, Train Accuracy: 95.48%
Epoch 2 completed in 16.43 seconds
Test Loss: 0.1290, Test Accuracy: 95.96%
Epoch 2 completed in 19.06 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.

The GPU is signifi cantly faster for training CNNs due to its ability to perform parallel computations, which is ideal for the convolution operations in CNNs. In contrast, MLPs, which involve fully connected layers, do not benefi t as much from GPU parallelism, so the speedup is less noticeable. Therefore, CNNs see a much greater performance improvement on the GPU compared to MLPs, especially as the model and dataset size increase.

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 are expected. CNNs have fewer parameters than
may require more FLOPs due to convolution operations. While CPUs are
general-purpose and less efficient for tasks like convolutions, GPUs
parallel computations, making CNNs faster to train on GPUs. The perf
difference arises because CNNs leverage parallelism in GPUs better t
which have fully connected layers and don't benefit as much from GPU
```

The profiling results are expected. CNNs have fewer parameters than MLPs but may require more FLOPs due to convolution operations. While CPUs are general-purpose and less efficient for tasks like convolutions, GPUs excel at parallel computations, making CNNs faster to train on GPUs. The performance difference arises because CNNs leverage parallelism in GPUs better than MLPs, which have fully connected layers and don't benefit as much from GPU parallelism