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
```

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())
```

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}")
```

```
# 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)
```

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
```

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
```

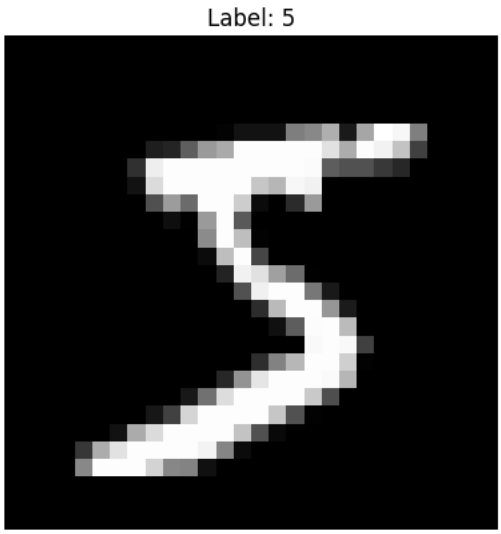
Inspect the first element of the test_loader, and verify both the tensor shapes and data types. You can check the data-type with `.dtype`

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.

```
import torch
from torchvision import datasets, transforms

# Define the transformation
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# Download the MNIST dataset
train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)

# Create DataLoaders
batch_size = 64
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)

# Retrieve the first element
first_data, first_label = train_dataset[0]

# Print details
print(f"Shape of training sample (image): {first_data.shape}")
print(f"Data type of training sample: {first_data.dtype}")
print(f"Training label: {first_label}")
print(f"Data type of training label: {type(first_label)}")

# Visualize the first sample
import matplotlib.pyplot as plt

plt.imshow(first_data.squeeze(), cmap='gray')
plt.title(f"Label: {first_label}")
plt.axis('off')
plt.show()
```

```
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 [00:01<00:00, 5.01MB/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, 133kB/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 [00:01<00:00, 1.25MB/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, 4.18MB/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw

Shape of training sample (image): torch.Size([1, 28, 28])
Data type of training sample: torch.float32
Training label: 5
Data type of training label: <class 'int'>
```



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
# 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
# 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 mo
# factor of 0.5. the first input to the optimizer is the list of model parameters, which is obtained by calling .parameters() on the mod
optimizer = optim.SGD(model.parameters(), 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
```

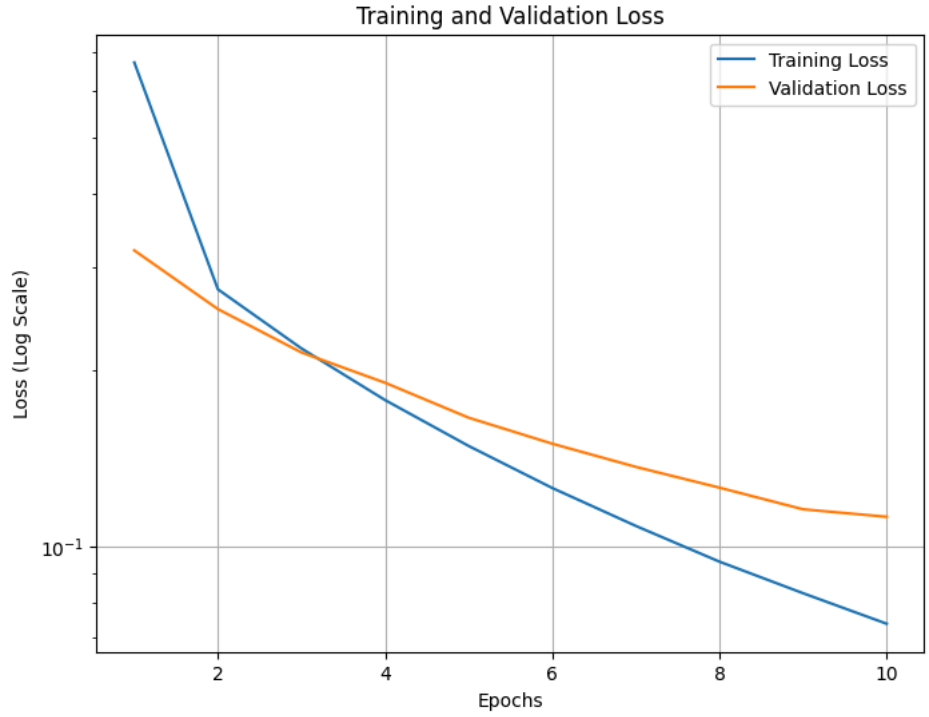
```
# declare the train function
def cpu_train(epoch, train_losses, steps, current_step):

    # 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
    # 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 t
    # 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 ne
        # 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_nor
    # 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 .softm
            # however, argmax is taking the maximum value, which will be the same index for the normalized and un-normalized distributio
            # 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')
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, random_split

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Flatten(),
            nn.Linear(28 * 28, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        return self.layers(x)

# Define data transforms and load dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('./data', train=False, transform=transform)

# Split the training dataset into training and validation datasets
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size, val_size])

# Define data loaders
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)

# Instantiate model, criterion, and optimizer
model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Training and validation loop
num_epochs = 10
train_losses = []
val_losses = []

for epoch in range(num_epochs):
    # Training
    model.train()
    running_train_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()
```

```
    train_loss = running_train_loss / len(train_loader)
    train_losses.append(train_loss)

    # Validation
    model.eval()
    running_val_loss = 0.0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_val_loss += loss.item()

    val_loss = running_val_loss / len(val_loader)
    val_losses.append(val_loss)

    print(f"Epoch {epoch+1}/{num_epochs}, Train Loss: {train_loss:.4f}, Validation Loss: {val_loss:.4f}")

# Plot the training and validation losses
plt.figure(figsize=(8, 6))
plt.plot(range(1, num_epochs + 1), train_losses, label="Training Loss")
plt.plot(range(1, num_epochs + 1), val_losses, label="Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss (Log Scale)")
plt.yscale("log")
plt.title("Training and Validation Loss")
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 1/10, Train Loss: 0.6722, Validation Loss: 0.3208
Epoch 2/10, Train Loss: 0.2752, Validation Loss: 0.2546
Epoch 3/10, Train Loss: 0.2180, Validation Loss: 0.2146
Epoch 4/10, Train Loss: 0.1780, Validation Loss: 0.1905
Epoch 5/10, Train Loss: 0.1485, Validation Loss: 0.1660
Epoch 6/10, Train Loss: 0.1259, Validation Loss: 0.1499
Epoch 7/10, Train Loss: 0.1084, Validation Loss: 0.1368
Epoch 8/10, Train Loss: 0.0942, Validation Loss: 0.1261
Epoch 9/10, Train Loss: 0.0832, Validation Loss: 0.1158
Epoch 10/10, Train Loss: 0.0738, Validation Loss: 0.1124
```



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
```

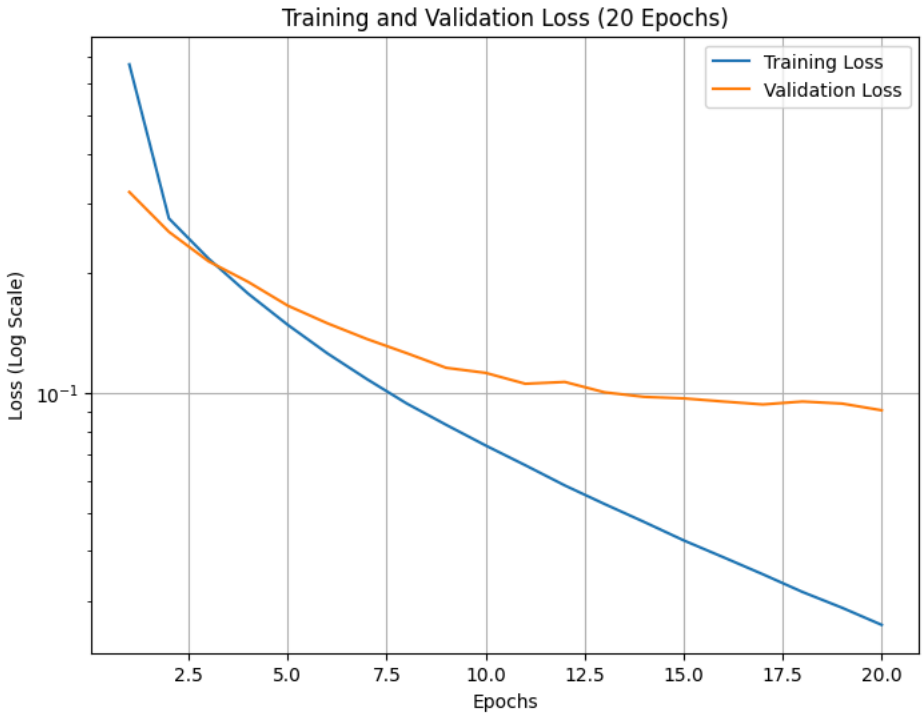
Question 3

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 lists for storing extended losses
extended_train_losses = train_losses.copy()
extended_val_losses = val_losses.copy()

# Extend training for another 10 epochs
additional_epochs = 10
total_epochs = num_epochs + additional_epochs

start_time = time.time()  # Record the start time

for epoch in range(num_epochs, total_epochs):
    epoch_start_time = time.time()  # Start time for the epoch

    # Training
    model.train()
    running_train_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_train_loss += loss.item()

    train_loss = running_train_loss / len(train_loader)
    extended_train_losses.append(train_loss)

    # Validation
    model.eval()
    running_val_loss = 0.0
    with torch.no_grad():
        for inputs, labels in val_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_val_loss += loss.item()

    val_loss = running_val_loss / len(val_loader)
    extended_val_losses.append(val_loss)

    # Calculate and print time per epoch
    epoch_end_time = time.time()
    epoch_time = epoch_end_time - epoch_start_time
    print(f"Epoch {epoch+1}/{total_epochs}, Train Loss: {train_loss:.4f}, Validation Loss: {val_loss:.4f}, Time: {epoch_time:.2f} secon

# Total training time
end_time = time.time()
total_training_time = end_time - start_time
print(f"Total time for {additional_epochs} epochs: {total_training_time:.2f} seconds")

# Testing
test_start_time = time.time()
test_loss = 0.0
correct = 0
total = 0
model.eval()
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

test_loss /= len(test_loader)
accuracy = 100 * correct / total
test_end_time = time.time()
test_time = test_end_time - test_start_time

print(f"Test Loss: {test_loss:.4f}, Test Accuracy: {accuracy:.2f}%, Test Time: {test_time:.2f} seconds")

# Plot the extended loss functions
plt.figure(figsize=(8, 6))
plt.plot(range(1, total_epochs + 1), extended_train_losses, label="Training Loss")
plt.plot(range(1, total_epochs + 1), extended_val_losses, label="Validation Loss")
```

```
plt.xlabel("Epochs")
plt.ylabel("Loss (Log Scale)")
plt.yscale("log")
plt.title("Training and Validation Loss (20 Epochs)")
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch 11/20, Train Loss: 0.0658, Validation Loss: 0.1055, Time: 13.86 seconds
Epoch 12/20, Train Loss: 0.0585, Validation Loss: 0.1066, Time: 13.70 seconds
Epoch 13/20, Train Loss: 0.0526, Validation Loss: 0.1004, Time: 13.75 seconds
Epoch 14/20, Train Loss: 0.0474, Validation Loss: 0.0978, Time: 13.78 seconds
Epoch 15/20, Train Loss: 0.0426, Validation Loss: 0.0970, Time: 13.80 seconds
Epoch 16/20, Train Loss: 0.0386, Validation Loss: 0.0952, Time: 13.75 seconds
Epoch 17/20, Train Loss: 0.0350, Validation Loss: 0.0936, Time: 13.90 seconds
Epoch 18/20, Train Loss: 0.0316, Validation Loss: 0.0952, Time: 14.46 seconds
Epoch 19/20, Train Loss: 0.0288, Validation Loss: 0.0941, Time: 14.13 seconds
Epoch 20/20, Train Loss: 0.0261, Validation Loss: 0.0905, Time: 14.16 seconds
Total time for 10 epochs: 139.31 seconds
Test Loss: 0.0786, Test Accuracy: 97.61%, Test Time: 2.12 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.

Observation:

From the plot:

The training loss continues to decrease steadily, indicating that the model is still learning and fitting to the training data.

The validation loss plateaus and may even slightly increase toward the end of the 20 epochs. This suggests the model might be starting to overfit to the training data.

The gap between training and validation loss is small, indicating that the model has not overfit significantly, but the lack of significant validation improvement suggests diminishing returns from further training.

Conclusion:

The model likely does not need significantly more training, as additional epochs are unlikely to substantially improve validation accuracy.

20 epochs may already be slightly too long; early stopping (stopping training when validation loss stops improving) might be appropriate to prevent overfitting and save computational resources.

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)
```

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

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

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

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

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

Question 5

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

```
import time

# Define the training function for the GPU
def train_gpu(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    total_loss = 0
    start_time = time.time()

    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data and target to the GPU
        data, target = data.to(device), target.to(device)

        # Zero the gradients
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Compute loss
        loss = criterion(output, target)
        total_loss += loss.item()

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

    epoch_time = time.time() - start_time
```

```
    avg_loss = total_loss / len(train_loader)
    print(f"Epoch {epoch}, Train Loss: {avg_loss:.4f}, Time: {epoch_time:.2f}s")
    return avg_loss


# Define the test function for the GPU
def test_gpu(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for data, target in test_loader:
            # Move data and target to the GPU
            data, target = data.to(device), target.to(device)

            # Forward pass
            output = model(data)

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

            # Get predictions
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    avg_loss = test_loss / len(test_loader)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f"Test Loss: {avg_loss:.4f}, Accuracy: {accuracy:.2f}%")
    return avg_loss, accuracy


# Set device to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Recreate the model, optimizer, and criterion
model = MLP().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

# Train and test the model on the GPU
train_losses = []
test_losses = []
test_accuracies = []

for epoch in range(1, 11):  # Train for 10 epochs
    train_loss = train_gpu(model, device, train_loader, optimizer, criterion, epoch)
    test_loss, test_accuracy = test_gpu(model, device, test_loader, criterion)
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    test_accuracies.append(test_accuracy)

# Plot the losses
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), train_losses, label="Train Loss")
plt.plot(range(1, 11), test_losses, label="Test Loss")
plt.yscale("log")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train and Test Loss on GPU")
plt.legend()
plt.grid()
plt.show()
```

```
Epoch 1, Train Loss: 0.6501, Time: 12.77s
Test Loss: 0.3061, Accuracy: 90.97%
Epoch 2, Train Loss: 0.2762, Time: 11.29s
Test Loss: 0.2382, Accuracy: 92.86%
Epoch 3, Train Loss: 0.2181, Time: 11.17s
Test Loss: 0.1928, Accuracy: 94.14%
Epoch 4, Train Loss: 0.1764, Time: 11.16s
Test Loss: 0.1675, Accuracy: 95.04%
Epoch 5, Train Loss: 0.1470, Time: 11.17s
Test Loss: 0.1395, Accuracy: 95.86%
Epoch 6, Train Loss: 0.1242, Time: 10.95s
Test Loss: 0.1316, Accuracy: 96.03%
Epoch 7, Train Loss: 0.1072, Time: 10.82s
Test Loss: 0.1176, Accuracy: 96.50%
Epoch 8, Train Loss: 0.0937, Time: 10.75s
Test Loss: 0.1056, Accuracy: 96.71%
Epoch 9, Train Loss: 0.0824, Time: 12.29s
Test Loss: 0.0995, Accuracy: 96.98%
Epoch 10, Train Loss: 0.0731, Time: 11.17s
Test Loss: 0.0958, Accuracy: 97.08%
```


```
# new GPU training for 10 epochs
```

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 expected to be faster than on the CPU due to the parallel computation capabilities of GPUs, which allow for the simultaneous processing of multiple operations. The degree of speedup, however, depends on several factors:

Batch Size: Larger batch sizes benefit more from GPU acceleration because they increase the amount of data processed in parallel. If the batch size is too small, the overhead of moving data to and from the GPU can negate the speedup.

Model Complexity: Deeper and more complex models with many parameters and operations utilize the GPU more effectively. In contrast, simple models, like the one in this assignment, may not fully leverage the GPU's parallelism.

Data Transfer Overhead: Moving data between the CPU and GPU introduces latency. For smaller datasets like MNIST, this overhead may reduce the speedup.

Hardware Configuration: The GPU's architecture and the type of operations being performed also play a significant role. A high-performance GPU, such as the NVIDIA T4 in Colab, should provide noticeable acceleration.

From observing the training time per epoch:

If the GPU training is significantly faster, the speedup aligns with expectations, especially for batch-based processing. If the speedup is modest or negligible, it could be due to the simplicity of the MLP model or the small size of the MNIST dataset, which doesn't fully utilize the GPU's parallel computing power. In conclusion, while training on the GPU is generally faster, the magnitude of the speedup depends on the factors listed above. For this specific assignment, the speedup may be limited due to the simplicity of the model and dataset.

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)
```

Question 7

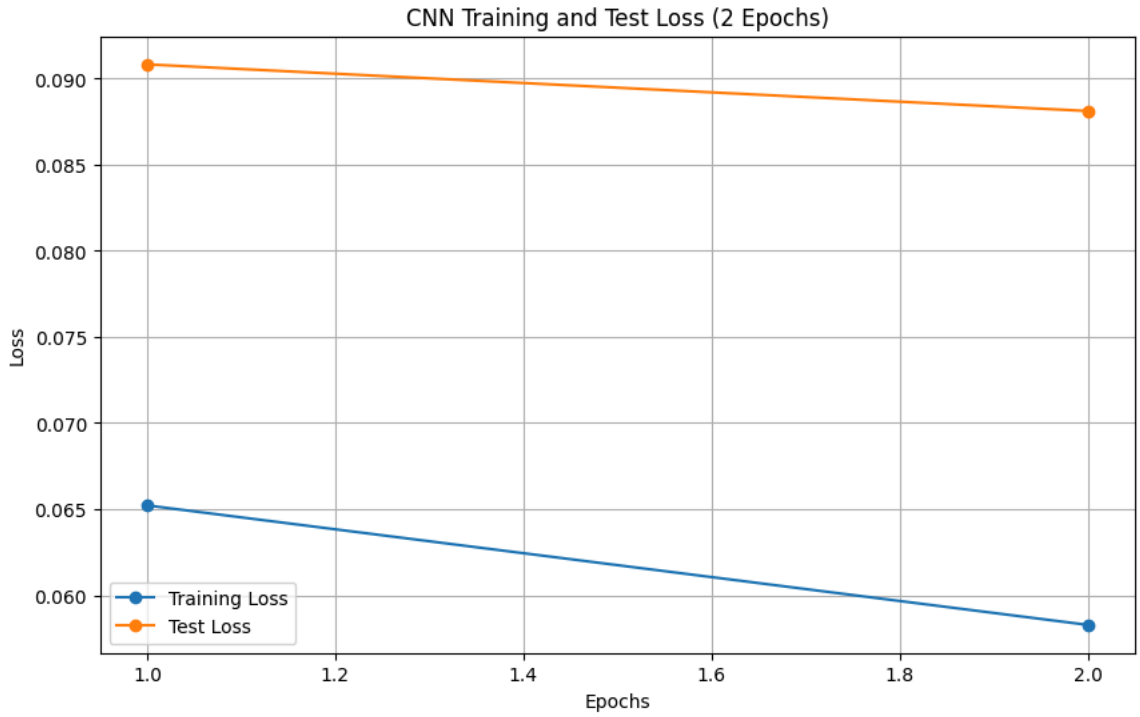
Notice that this model now has fewer parameters than the MLP. Let's see how it trains.

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

```
import time
import torch

# Define the train function
def train_cnn(model, device, train_loader, criterion, optimizer, epoch):
    model.train()
    running_loss = 0.0
    start_time = time.time()

    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data and target to the selected device (GPU or CPU)
        data, target = data.to(device), target.to(device)

        # Forward pass
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

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

        running_loss += loss.item()

        # Print progress every 100 batches
        if batch_idx % 100 == 0:
            print(f"Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)} "
                  f"({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

    end_time = time.time()
    avg_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch} completed in {end_time - start_time:.2f} seconds. Avg loss: {avg_loss:.6f}")
    return avg_loss

# Define the test function
def test_cnn(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0.0
    correct = 0

    with torch.no_grad():
        for data, target in test_loader:
            # Move data and target to the selected device (GPU or CPU)
            data, target = data.to(device), target.to(device)

            # Forward pass
            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)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f"Test set: Avg loss: {test_loss:.6f}, Accuracy: {correct}/{len(test_loader.dataset)} "
          f"({accuracy:.2f}%)")
    return test_loss, accuracy

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

# Train for 2 epochs and record timing
train_losses = []
test_losses = []
accuracies = []

for epoch in range(1, 3):  # Train for 2 epochs
    train_loss = train_cnn(model, device, train_loader, criterion, optimizer, epoch)
    test_loss, accuracy = test_cnn(model, device, test_loader, criterion)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

# Plot the results
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.plot(range(1, 3), train_losses, label="Training Loss", marker="o")
plt.plot(range(1, 3), test_losses, label="Test Loss", marker="o")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("CNN Training and Test Loss (2 Epochs)")
plt.legend()
plt.grid()
plt.show()
```

```
Train Epoch: 1 [0/48000 (0%)]   Loss: 0.059835
Train Epoch: 1 [6400/48000 (13%)]       Loss: 0.065051
Train Epoch: 1 [12800/48000 (27%)]      Loss: 0.054920
Train Epoch: 1 [19200/48000 (40%)]      Loss: 0.084391
Train Epoch: 1 [25600/48000 (53%)]      Loss: 0.039988
Train Epoch: 1 [32000/48000 (67%)]      Loss: 0.061614
Train Epoch: 1 [38400/48000 (80%)]      Loss: 0.027064
Train Epoch: 1 [44800/48000 (93%)]      Loss: 0.053658
Epoch 1 completed in 11.39 seconds. Avg loss: 0.065216
Test set: Avg loss: 0.090833, Accuracy: 9712/10000 (97.12%)
Train Epoch: 2 [0/48000 (0%)]   Loss: 0.037923
Train Epoch: 2 [6400/48000 (13%)]       Loss: 0.055065
Train Epoch: 2 [12800/48000 (27%)]      Loss: 0.012623
Train Epoch: 2 [19200/48000 (40%)]      Loss: 0.037546
Train Epoch: 2 [25600/48000 (53%)]      Loss: 0.015666
Train Epoch: 2 [32000/48000 (67%)]      Loss: 0.029192
Train Epoch: 2 [38400/48000 (80%)]      Loss: 0.099321
Train Epoch: 2 [44800/48000 (93%)]      Loss: 0.019595
Epoch 2 completed in 12.99 seconds. Avg loss: 0.058278
Test set: Avg loss: 0.088124, Accuracy: 9742/10000 (97.42%)
```




```
# train for 2 epochs on the CPU
```

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()
```

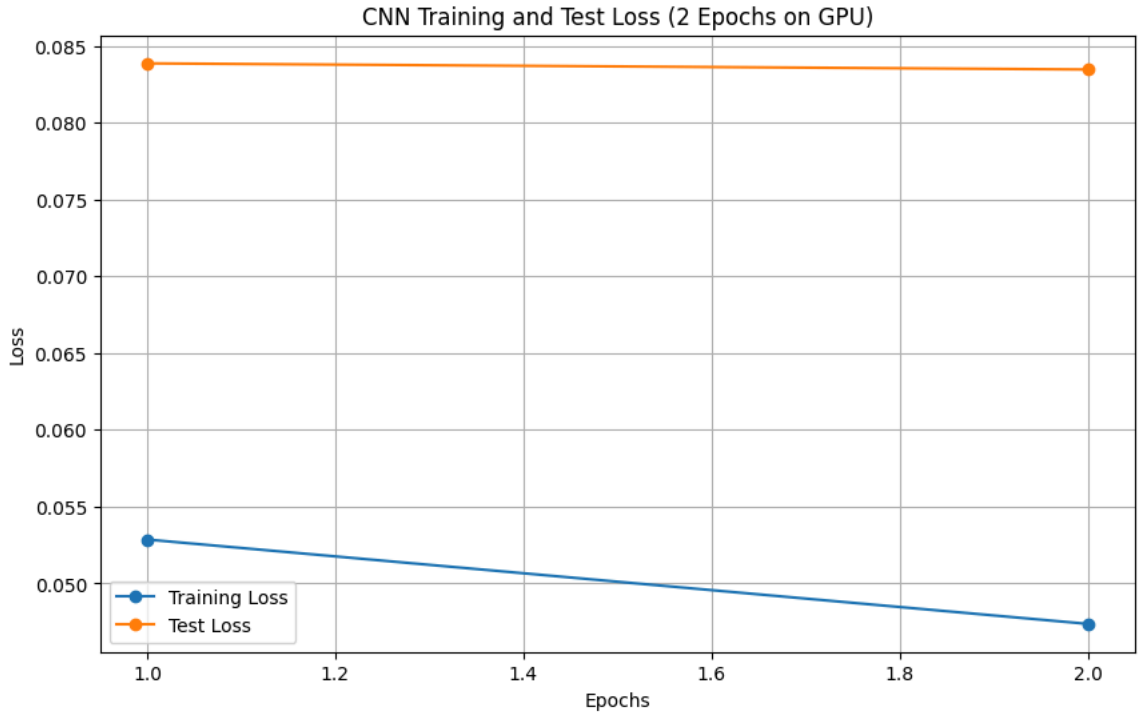
```
# then you can instantiate the optimizer. You will use Stochastic Gradient Descent (SGD), and can set the learning rate to 0.1 with a mo
# the first input to the optimizer is the list of model parameters, which is obtained by calling .parameters() on the model object
optimizer = optim.SGD(model.parameters(),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

```
# 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 time
import matplotlib.pyplot as plt

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

# Define the train function
def train_cnn(model, device, train_loader, criterion, optimizer, epoch):
    model.train()
    running_loss = 0.0
    start_time = time.time()

    for batch_idx, (data, target) in enumerate(train_loader):
        # Move data and target to the selected device (GPU or CPU)
        data, target = data.to(device), target.to(device)

        # Forward pass
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)

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

        running_loss += loss.item()

        # Print progress every 100 batches
        if batch_idx % 100 == 0:
            print(f"Train Epoch: {epoch} [{batch_idx * len(data)}/{len(train_loader.dataset)} "
                  f"({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.item():.6f}")

    end_time = time.time()
    avg_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch} completed in {end_time - start_time:.2f} seconds. Avg loss: {avg_loss:.6f}")
    return avg_loss

# Define the test function
def test_cnn(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0.0
    correct = 0

    with torch.no_grad():
        for data, target in test_loader:
            # Move data and target to the selected device (GPU or CPU)
            data, target = data.to(device), target.to(device)

            # Forward pass
            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)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f"Test set: Avg loss: {test_loss:.6f}, Accuracy: {correct}/{len(test_loader.dataset)} "
          f"({accuracy:.2f}%)")
    return test_loss, accuracy

# Train for 2 epochs on GPU
train_losses = []
test_losses = []
accuracies = []

for epoch in range(1, 3):  # Train for 2 epochs
```

```
    train_loss = train_cnn(model, device, train_loader, criterion, optimizer, epoch)
    test_loss, accuracy = test_cnn(model, device, test_loader, criterion)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

# Plot the results
plt.figure(figsize=(10, 6))
plt.plot(range(1, 3), train_losses, label="Training Loss", marker="o")
plt.plot(range(1, 3), test_losses, label="Test Loss", marker="o")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("CNN Training and Test Loss (2 Epochs on GPU)")
plt.legend()
plt.grid()
plt.show()
```

```
Train Epoch: 1 [0/48000 (0%)]   Loss: 0.029158
Train Epoch: 1 [6400/48000 (13%)]       Loss: 0.017298
Train Epoch: 1 [12800/48000 (27%)]      Loss: 0.035562
Train Epoch: 1 [19200/48000 (40%)]      Loss: 0.028426
Train Epoch: 1 [25600/48000 (53%)]      Loss: 0.070320
Train Epoch: 1 [32000/48000 (67%)]      Loss: 0.087359
Train Epoch: 1 [38400/48000 (80%)]      Loss: 0.100604
Train Epoch: 1 [44800/48000 (93%)]      Loss: 0.062035
Epoch 1 completed in 11.08 seconds. Avg loss: 0.052862
Test set: Avg loss: 0.083852, Accuracy: 9751/10000 (97.51%)
Train Epoch: 2 [0/48000 (0%)]   Loss: 0.092432
Train Epoch: 2 [6400/48000 (13%)]       Loss: 0.027080
Train Epoch: 2 [12800/48000 (27%)]      Loss: 0.013510
Train Epoch: 2 [19200/48000 (40%)]      Loss: 0.033307
Train Epoch: 2 [25600/48000 (53%)]      Loss: 0.025122
Train Epoch: 2 [32000/48000 (67%)]      Loss: 0.008675
Train Epoch: 2 [38400/48000 (80%)]      Loss: 0.005516
Train Epoch: 2 [44800/48000 (93%)]      Loss: 0.011433
Epoch 2 completed in 10.99 seconds. Avg loss: 0.047372
Test set: Avg loss: 0.083454, Accuracy: 9745/10000 (97.45%)
```



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 comparison between the CPU and GPU versions for training a CNN model typically shows that the GPU version is significantly faster than the CPU version. This difference in speed can be attributed to several factors:

Parallelization: The GPU is designed to handle massive parallel computations, which is ideal for training deep learning models. CNNs involve operations like convolutions, which can be computed in parallel across multiple spatial locations in the image. The GPU excels at parallelizing these operations, making it much faster than the CPU, which processes tasks sequentially.

GPU Architecture: GPUs, particularly NVIDIA's T4 GPU used in Colab, are specialized for high-throughput computing, which makes them well-suited for tasks like training CNNs. The architecture of the GPU has thousands of small cores designed for performing the same operation on multiple data points simultaneously, while the CPU typically has fewer, more powerful cores designed for general-purpose computing.

Model Type - CNN vs. MLP: The reason the speedup is more pronounced with the CNN compared to the MLP is due to the inherent nature of CNNs. CNNs involve complex operations such as convolutions and pooling, which can be massively parallelized. On the other hand, an MLP involves a series of matrix multiplications, which are generally less parallelizable compared to convolutions in CNNs. Therefore, the GPU is able to exploit its parallel architecture more effectively with CNNs than with MLPs, leading to a greater performance improvement in the CNN case.

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.

```
import torch
import thop
from torch import nn

# Define your MLP and CNN models here
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.net = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveMaxPool2d(1),
        )
        self.head = nn.Sequential(
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 10)
        )

    def forward(self, x):
        x = self.net(x)
        x = x.view(x.size(0), -1)
        x = self.head(x)
        return x

# 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")
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-13-3150258eba75> in <cell line: 2>()
      1 import torch
----> 2 import thop
      3 from torch import nn
      4
      5 # Define your MLP and CNN models here

ModuleNotFoundError: No module named 'thop'
```