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
```

```
Requirement already satisfied: thop in /usr/local/lib/python3.10/dist-packages (0.1.1.post2209072238)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from thop) (2.4.1+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: sympy in /usr/local/lib/python3.10/dist-packages (from torch->thop) (1.13.3)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->thop) (3.3)
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) (2024.6.1)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->thop) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch->thop) (1.3.0)
```

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: 2.4.1+cu121 with CUDA= True
```

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

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

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

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

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

A Brief Introduction to PyTorch

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

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

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

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

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

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

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

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

    # we can also add lists of modules, for example, the sequential layer
    self.net = nn.Sequential(  # this layer type takes in a collection of modules rather than a list
        nn.Linear(4,4),
        nn.Linear(4,8),
        nn.Linear(8,12)
    )

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

=== Listing parameters from the state_dict ===
my_param: torch.Size([4, 8])
foo: torch.Size([1])
my_sub_module.weight: torch.Size([12, 8])
my_sub_module.bias: torch.Size([12])
net.0.weight: torch.Size([4, 4])
net.0.bias: torch.Size([4])
net.1.weight: torch.Size([8, 4])
net.1.bias: torch.Size([8])
net.2.weight: torch.Size([12, 8])
net.2.bias: torch.Size([12])
net_list.0.weight: torch.Size([7, 7])
net_list.0.bias: torch.Size([7])
net_list.1.weight: torch.Size([9, 7])
net_list.1.bias: torch.Size([9])
net_list.2.weight: torch.Size([14, 9])
net_list.2.bias: torch.Size([14])
```

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

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

```
tensor([[ 0.1432, -0.1804, -0.1987, -0.0337,  0.3491, -0.5157,  0.0237,  0.3379,
          0.5432, -0.7059, -0.5272, -0.3631],
        [ 0.1432, -0.1804, -0.1987, -0.0337,  0.3491, -0.5157,  0.0237,  0.3379,
          0.5432, -0.7059, -0.5272, -0.3631]], grad_fn=<AddBackward0>) torch.Size([2, 12])
```

Please check the cell below to notice the following:

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

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

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

# then you can print the result and shape
```

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

```
tensor([[ 0.1432, -0.1804, -0.1987, -0.0337,  0.3491, -0.5157,  0.0237,  0.3379,
          0.5432, -0.7059, -0.5272, -0.3631],
        [ 0.1432, -0.1804, -0.1987, -0.0337,  0.3491, -0.5157,  0.0237,  0.3379,
          0.5432, -0.7059, -0.5272, -0.3631]]) torch.Size([2, 12])
```

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

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

Multi-Layer-Perceptron (MLP) Prediction of MNIST

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

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

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

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

        # project-in and apply a non-linearity (ReLU activation function)
        x = torch.relu(self.fc1(x))
        # x is of shape Bx128

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

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

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

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

# let's download the train and test datasets, applying the above transform - this will get saved locally into ./data, which is in the Colab
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.

```
# Get the first item
first_item = next(iter(test_loader))

# 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
print(first_item[0].shape, first_item[0].dtype)
print(first_item[1].shape, first_item[1].dtype)
```

```
torch.Size([64, 1, 28, 28]) torch.float32
torch.Size([64]) torch.int64
```

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

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

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

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

# for a critereon (loss) function, you will use Cross-Entropy Loss. This is the most common criterion used for multi-class prediction,
# and is also used by tokenized transformer models it takes in an un-normalized probability distribution (i.e. without softmax) over
# N classes (in our case, 10 classes with MNIST). This distribution is then compared to an integer label which is < N.
# For MNIST, the prediction might be [-0.0056, -0.2044,  1.1726,  0.0859,  1.8443, -0.9627,  0.9785, -1.0752, 1.1376,  1.8220], with the lab
# 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 moment
# 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 c
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
```

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

Finally, you can define a training, and test loop

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


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

    # set the model in training mode - this doesn't do anything for us right now, but it is good practiced and needed with other layers such
    # batch norm and dropout
    model.train()

    # Create tqdm progress bar to help keep track of the training progress
```

```python
    pbar = tqdm(enumerate(train_loader), total=len(train_loader))

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

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

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

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

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

        # increment the step count
        current_step += 1

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

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

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

    return current_step

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

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

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

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

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

    test_loss /= len(test_loader)

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

    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}'
          f' ({100. * correct / len(test_loader.dataset):.0f}%)\n')
```

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

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.222179: 100%|██████████| 938/938 [00:26<00:00, 34.74it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 47.08it/s]

Test set: Average loss: 0.2752, Accuracy: 9202/10000 (92%)

Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.171576: 100%|██████████| 938/938 [00:17<00:00, 54.35it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 63.64it/s]

Test set: Average loss: 0.2062, Accuracy: 9391/10000 (94%)

Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.269579: 100%|██████████| 938/938 [00:18<00:00, 52.03it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 61.59it/s]

Test set: Average loss: 0.1647, Accuracy: 9514/10000 (95%)

Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.119205: 100%|██████████| 938/938 [00:17<00:00, 54.82it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 62.69it/s]

Test set: Average loss: 0.1436, Accuracy: 9573/10000 (96%)

Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.088436: 100%|██████████| 938/938 [00:17<00:00, 52.35it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 62.51it/s]

Test set: Average loss: 0.1211, Accuracy: 9653/10000 (97%)

Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.184387: 100%|██████████| 938/938 [00:18<00:00, 51.96it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 48.14it/s]

Test set: Average loss: 0.1081, Accuracy: 9670/10000 (97%)

Train Epoch: 6 [57600/60000 (96%)]      Loss: 0.185497: 100%|██████████| 938/938 [00:17<00:00, 55.17it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 62.55it/s]

Test set: Average loss: 0.1015, Accuracy: 9689/10000 (97%)

Train Epoch: 7 [57600/60000 (96%)]      Loss: 0.057727: 100%|██████████| 938/938 [00:17<00:00, 54.50it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 49.12it/s]

Test set: Average loss: 0.0951, Accuracy: 9713/10000 (97%)

Train Epoch: 8 [57600/60000 (96%)]      Loss: 0.178419: 100%|██████████| 938/938 [00:16<00:00, 55.27it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 63.62it/s]

Test set: Average loss: 0.0924, Accuracy: 9716/10000 (97%)

Train Epoch: 9 [57600/60000 (96%)]      Loss: 0.030724: 100%|██████████| 938/938 [00:17<00:00, 53.37it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 60.84it/s]
Test set: Average loss: 0.0842, Accuracy: 9740/10000 (97%)
```

Question 2

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

```
# visualize the losses for the first 10 epochs
epochs = range(1, 11)
plt.plot(epochs, train_losses[0:10], label='Training Loss', color='blue')
plt.plot(epochs, test_losses[0:10], label='Validation Loss', color='red')

# Add labels and title
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')

# Use a logarithmic scale on the y-axis
plt.yscale('log')

# Add a legend to differentiate between training and validation
plt.legend()

# Display the plot
plt.show()
```

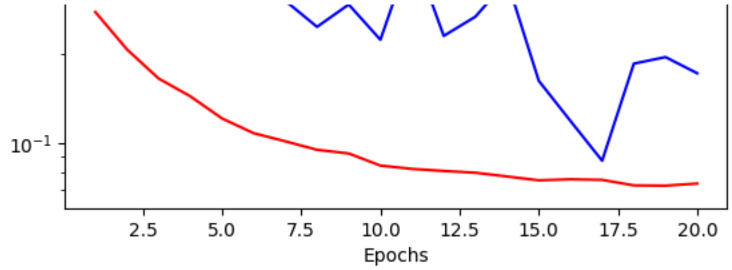
Question 3

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

```
# train for 10 epochs with time per epoch
for epoch in range(10, 20):
    start_time = time.time()
    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
    end_time = time.time()
    print(f"Time per epoch: {end_time - start_time} seconds")
    print(" ")

#visualize epchos
epochs = range(1, 21)
plt.plot(epochs, train_losses[0:20], label='Training Loss', color='blue')
plt.plot(epochs, test_losses[0:20], label='Validation Loss', color='red')

# Add labels and title
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')

# Use a logarithmic scale on the y-axis
plt.yscale('log')

# Add a legend to differentiate between training and validation
plt.legend()
```

```
# Display the plot
plt.show()
```

```
Train Epoch: 10 [57600/60000 (96%)]     Loss: 0.049199: 100%|██████████| 938/938 [00:24<00:00, 38.87it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 64.46it/s]

Test set: Average loss: 0.0820, Accuracy: 9741/10000 (97%)

Time per epoch: 26.582042932510376 seconds

Train Epoch: 11 [57600/60000 (96%)]     Loss: 0.039082: 100%|██████████| 938/938 [00:16<00:00, 55.44it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 59.58it/s]

Test set: Average loss: 0.0808, Accuracy: 9735/10000 (97%)

Time per epoch: 19.575411319732666 seconds

Train Epoch: 12 [57600/60000 (96%)]     Loss: 0.041487: 100%|██████████| 938/938 [00:17<00:00, 53.34it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 64.15it/s]

Test set: Average loss: 0.0796, Accuracy: 9749/10000 (97%)

Time per epoch: 20.04898452758789 seconds

Train Epoch: 13 [57600/60000 (96%)]     Loss: 0.024652: 100%|██████████| 938/938 [00:16<00:00, 55.46it/s]
Testing...: 100%|██████████| 157/157 [00:03<00:00, 51.48it/s]

Test set: Average loss: 0.0774, Accuracy: 9759/10000 (98%)

Time per epoch: 19.984562873840332 seconds

Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.070544: 100%|██████████| 938/938 [00:17<00:00, 52.64it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 63.34it/s]

Test set: Average loss: 0.0751, Accuracy: 9762/10000 (98%)

Time per epoch: 20.310160160064697 seconds

Train Epoch: 15 [57600/60000 (96%)]     Loss: 0.067022: 100%|██████████| 938/938 [00:16<00:00, 55.24it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 53.59it/s]

Test set: Average loss: 0.0757, Accuracy: 9761/10000 (98%)

Time per epoch: 19.928783655166626 seconds

Train Epoch: 16 [57600/60000 (96%)]     Loss: 0.059051: 100%|██████████| 938/938 [00:16<00:00, 56.78it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 65.07it/s]

Test set: Average loss: 0.0754, Accuracy: 9762/10000 (98%)

Time per epoch: 18.94910717010498 seconds

Train Epoch: 17 [57600/60000 (96%)]     Loss: 0.028077: 100%|██████████| 938/938 [00:17<00:00, 53.49it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 60.47it/s]

Test set: Average loss: 0.0722, Accuracy: 9772/10000 (98%)

Time per epoch: 20.150938034057617 seconds

Train Epoch: 18 [57600/60000 (96%)]     Loss: 0.020279: 100%|██████████| 938/938 [00:16<00:00, 55.64it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 64.07it/s]

Test set: Average loss: 0.0721, Accuracy: 9773/10000 (98%)

Time per epoch: 19.327306509017944 seconds

Train Epoch: 19 [57600/60000 (96%)]     Loss: 0.006061: 100%|██████████| 938/938 [00:17<00:00, 52.73it/s]
Testing...: 100%|██████████| 157/157 [00:02<00:00, 63.81it/s]

Test set: Average loss: 0.0733, Accuracy: 9784/10000 (98%)

Time per epoch: 20.259864330291748 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.

While the model have less loss with longer training, the efficiecy of the train per epochs decrease significalntly after 10 epoch.

Moving to the GPU

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

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

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

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

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

# for a critereon (loss) funciton, we will use Cross-Entropy Loss. This is the most common critereon used for multi-class prediction, and is
# it takes in an un-normalized probability distribution (i.e. without softmax) over N classes (in our case, 10 classes with MNIST). This dis
# 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]
# 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 moment
# 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.

```
# new GPU training for 10 epochs
def gpu_train(epoch, train_losses, steps, current_step):
    # set the model in training mode - this doesn't do anything for us right now, but it is good practiced and needed with other layers such a
    # 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))
    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()

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

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

        # Update step and log loss
        current_step += 1
        train_losses.append(loss.item())
        steps.append(current_step)

    print(f"Epoch [{epoch+1}], Loss: {sum(train_losses[-len(pbar):])/len(pbar):.4f}")

    return current_step  # Return updated step count

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

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

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

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

    test_loss /= len(test_loader)

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

    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}'
          f' ({100. * correct / len(test_loader.dataset):.0f}%)\n')
```

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 GPU is a lot faster on larger models due to paralleziation. However the gpu train does not have significant advantage on smaller models. The speed up is depends on the how model suits for the paralleization.

Another Model Type: CNN

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

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

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

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

Let's define a simple CNN below.

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

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

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

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


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

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

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

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

        # the output here is Bx10
        return x

# create the model
model = CNN()

# print the model and the parameter count
print(model)
param_count = sum([p.numel() for p in model.parameters()])
```

```
print(f"Model has {param_count:,} trainable parameters")

# the loss function
criterion = nn.CrossEntropyLoss()

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

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

Question 7

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

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

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


# train for 2 epochs on the CPU
for epoch in range(1, 2):
    current_step = cpu_train(current_epoch, train_losses, train_steps, current_step)
    cpu_test(test_losses, test_accuracy, test_steps, current_step)
    current_epoch += 1
```

```
Train Epoch: 0 [57600/60000 (96%)]      Loss: 0.684577: 100%|██████████| 938/938 [01:14<00:00, 12.61it/s]
Testing...: 100%|██████████| 157/157 [00:05<00:00, 28.07it/s]
Test set: Average loss: 0.6609, Accuracy: 7856/10000 (79%)
```

Question 8

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