How to submit the homework

This homework has been provided to you as an .ipynb. We expect you to complete the notebook and submit the following on canvas:

- the completed notebook itself with your answers filled in (including any source code used to answer the question)
- a .pdf file of the notebook that shows all your answers. This will make grading easier and I will appreciate that.

If you submit only one of these, you will lose points. No late homework will be graded. No submissions via email or other media will be graded.

Question 1 (1 point)

Is the following module capable of implementing the function $y = x_1 + x_2$ for all $y, x_1, x_2 \in \mathbb{R}$? If so, explain why. If not, explain why not and provide a modified *AddModule* that can implement this function, given the right weights. Also, give a set of weights (including the bias) that would make the (possibly modified) *AddModule* actually implement $y = x_1 + x_2$.

```
class AddModule(torch.nn.Module):

  # Define the model architecture
  def __init__(self, use_good_weights=False):
    super(AddModule, self).__init__()

    self.layer_1 = torch.nn.Linear(2,1)

  def forward(self, x):
      return torch.nn.Hardtanh(self.layer_1(x))
```

ANSWER

No because range for Hardtanh function is [-1,1], so when $x_1 + x_2 < -1$ or $x_1 + x_2 > 1$, the model cannot produce correct results. A simple way to correct it is removing activation function; code provided below. Right weight is $w = [1, 1]$ and bias $b = 0$, the model outputs is $x_1 * 1 + x_2 * 1 + 0$, which is $x_1 + x_2$.

```
class AddModule(torch.nn.Module):

  # Define the model architecture
  def __init__(self, use_good_weights=False):
    super(AddModule, self).__init__()

    self.layer_1 = torch.nn.Linear(2,1)

  def forward(self, x):
      return self.layer_1(x)
```

Question 2 (3 points)

In this question, you will do your best to make a network to embody the function $y = x_1 * x_2$. Here, assume $x_1, x_2 \in \mathbb{R}$ and $x_1, x_2 \in (-1000, 1000)$.

Assume you are starting from random weight initialization. Feel free to use any of the Non-linear Activations (weighted sum, nonlinearity) in torch.nn. Use as many layers as you like. Make the layers as big as you like.

- Note, you are not allowed to simply put some variant of $output = x_1 * x_2$ in your forward function.

You will make a training dataset and a test dataset using the provided dataset generator. Train the network on the training set and, once trained, test it on the test data. Make sure your test set is at minimum 1000 examples. Use mean-squared-error as your objective function. Then answer the following questions.

1. What was the best mean-squared-error you got on the training data?
2. What was the best mean-squared-error you got on the test data?
3. What challenges or difficulties did you encounter in implementing and training this network?

ANSWER

1.

The best mean MSE loss I got on training data (average MSE loss over one epoch, training set size is 900 examples, batch size is 10) is $7.75e + 22$

2.

The best mean MSE loss I got on test data (average MSE loss over one epoch, test set size is 100) is $8.24e + 22$

3.

I modified multivariate_normal to uniform according to campuswire. Model setup is input layer size 2, hidden layer size 100, and output layer size 1, using Sigmoid activation, SGD optimizers, batch size 100, epoch 20, training set size 9000 and test set size 1000.

Both losses on training set and test set are unacceptably large, and there is no such trend where training or testing loss stably goes down. The reason is that weights learned for one training step doesn't fit data in another training step, so the loss function won't decrease, gradient descent will not move loss to minimum; instead the gradient will go back and forth because each batch data does not fit current weights; this leads to non-decreasing loss and a model with nothing learned.

Model definition

```
class MultiplyModule(torch.nn.Module):
    # Define the model architecture
    def __init__(self, use_good_weights=False):
        super(MultiplyModule, self).__init__()
        self.model = nn.Sequential(
            torch.nn.Linear(2,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,1),
        )

    def forward(self, x):
        return self.model(x)
```

In [1]:

```
from IPython.display import Image
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.nn.functional import normalize
from torch.utils.data import Dataset, DataLoader
import numpy as np
from numpy import pi
import seaborn as sns
import matplotlib.pyplot as plt

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
np.random.seed(1)
print(device)
print(torch.cuda.get_device_name())
```

```
cuda:0
NVIDIA GeForce RTX 2070 SUPER
```

In [2]:

```
# HERE'S A DATASET GENERATOR TO HELP YOU TEST/TRAIN.
class MultiplyDataset(torch.utils.data.Dataset):
    """MULTIPLY Dataset."""

    def __init__(self, num_examples, max_abs=1000):
        """create a dataset of the form x_1 + x_2 = y. The input x_1, x_2 is a
        pair of values drawn from the default range [-1000, 1000]. The output y is a scalar.

        PARAMETERS
        ----------
        num_examples    An integer determining how much data we'll generate
        max_abs         The largest absolute value a datapoint can have
        """
        self.length = num_examples

        # make a circular unit Gaussian and draw samples from it
#         data = np.random.multivariate_normal(mean=[0,0],cov=[[1,0],[0,1]],size=self.length)
        data = np.random.uniform(low=-1000., high=1000., size=(self.length, 2))

        data *= max_abs
        # figure out the label (i.e. the result of the multiplication)
        label = np.multiply(data.T[0],data.T[1])

        # turn it into a tensor
        self.data =  torch.tensor(data).to(device, dtype=torch.float32)
        self.label = torch.tensor(label).to(device, dtype=torch.float32)

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        return self.data[idx], self.label[idx]
```

In [3]:

```
# YOUR CODE GOES HERE
class MultiplyModule(torch.nn.Module):

    # Define the model architecture
    def __init__(self, use_good_weights=False):
        super(MultiplyModule, self).__init__()
        self.model = nn.Sequential(
            torch.nn.Linear(2,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,1),
        )

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


def train_model(model, data, target, opti):
    model.train()
    opti.zero_grad()
    output = model(data)
    loss = F.mse_loss(output.squeeze(), target)
#     print(loss)
    loss.backward()
    opti.step()



def test_model(model, data, target):

    model.eval()
    with torch.no_grad():
        output = model(data)
        output = output.squeeze()
        loss = F.mse_loss(output, target)

        loss = round(loss.item(),8)

    return loss
```

In [4]:

```
bs=100
epochs=20

multModel = MultiplyModule().to(device)
optimizer = torch.optim.SGD(multModel.parameters(), lr=0.01)


train_ds = MultiplyDataset(9000)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True, drop_last=False)

test_ds = MultiplyDataset(1000)
test_dl = DataLoader(test_ds, batch_size=bs, shuffle=True, drop_last=False)

train_lst, test_lst = [], []

for i in range(epochs):
    for (xxx, yyy) in train_dl:
#         print(xxx, yyy)
        train_model(multModel, xxx, yyy, optimizer)

    train_loss = test_model(multModel, train_ds[:][0], train_ds[:][1])
    train_lst.append(train_loss)

    test_loss = test_model(multModel, test_ds[:][0], test_ds[:][1])
    test_lst.append(test_loss)

    print(f"epoch {i} train loss is {train_loss}, test loss is {test_loss}")
```

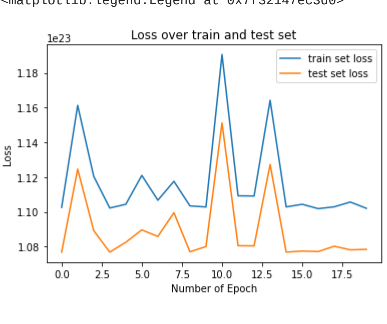
```
epoch 0 train loss is 1.1026161166251334e+23, test loss is 1.0768394038699082e+23
epoch 1 train loss is 1.1612380318227344e+23, test loss is 1.1246188129926222e+23
epoch 2 train loss is 1.1204646025243431e+23, test loss is 1.0892381739320294e+23
epoch 3 train loss is 1.1022379043284269e+23, test loss is 1.0768525543808201e+23
epoch 4 train loss is 1.1043551365852463e+23, test loss is 1.0825286211351803e+23
epoch 5 train loss is 1.120974500074154e+23, test loss is 1.0896547568975612e+23
epoch 6 train loss is 1.1067070063826517e+23, test loss is 1.085864347307181e+23
epoch 7 train loss is 1.1176026550331417e+23, test loss is 1.0995628562217613e+23
epoch 8 train loss is 1.103450273348115e+23, test loss is 1.0770615214035301e+23
epoch 9 train loss is 1.1028256240797987e+23, test loss is 1.080061279043329e+23
epoch 10 train loss is 1.190510168392747e+23, test loss is 1.1511720362955986e+23
epoch 11 train loss is 1.1092555033397881e+23, test loss is 1.0805599175940715e+23
epoch 12 train loss is 1.10913075363011e+23, test loss is 1.0804718271853602e+23
epoch 13 train loss is 1.1641932038262224e+23, test loss is 1.1272727841530316e+23
epoch 14 train loss is 1.1028588606450487e+23, test loss is 1.0768800263385471e+23
epoch 15 train loss is 1.1043752226395844e+23, test loss is 1.0774746816333451e+23
epoch 16 train loss is 1.1018550983601004e+23, test loss is 1.07720365500777e+23
epoch 17 train loss is 1.1029288465832 58e+23, test loss is 1.0802442152601929e+23
epoch 18 train loss is 1.1056610003331986e+23, test loss is 1.0781855297985293e+23
epoch 19 train loss is 1.1020509148718984e+23, test loss is 1.0784545748402684e+23
```

In [5]:

```
plt.clf()
plt.plot(train_lst,  label='train set loss')
plt.plot(test_lst,   label='test set loss')
plt.ylabel("Loss")
plt.xlabel("Number of Epoch")
plt.title("Loss over train and test set")
plt.legend()
```

Out[5]:

```
<matplotlib.legend.Legend at 0x7f32147ec3d0>
```



Question 3 (2 points)

We're now going to think about how easy it is to solve the general multiplication problem for real numbers: $y = x_1 * x_2$, when $x_1, x_2 \in \mathbb{R}$ are not bounded to the limited range (-1000,1000).

Define a "simple" feed-forward neural network as one where each layer l takes input from only the previous layer $l - 1$. Let's assume our simple feed-forward network only uses "standard" nodes, which take a weighted sum $z = \mathbf{w}^T\mathbf{x}$ of inputs $\mathbf{x}$, given weights $\mathbf{w}$, and then apply a differentiable activation function $f()$ to z. Example "standard" nodes include ReLu, Leaky ReLU, Sigmoid, TanH, and the linear/identity function.

Define "correctly" performing multiplication as estimating $\hat{y} = x_1 * x_2$ to 2 decimal places of precision (i.e. $|\hat{y} - y| < 0.01$). Here, $\hat{y}$ is the network's result and y is the true answer.

Is it possible to make a "simple" neural network that can correctly perform multiplication for any arbitrary pair $x_1, x_2 \in \mathbb{R}$? **Support your answer.**

Hint: Think about what a "standard" node calculates. Think about how you would implement multiplication using addition.

Answer

For x_1, x_2 positive, we can use log to transform multiplication to addition $log(ab) = log(a) + log(b)$, and in Question 1 I have already demonstrated addition is possible via neural net; I also support this point by providing a model below, the MSE loss is significantly low (implying $|\hat{y} - y| < 0.01$)), proving addition over transformed data is doable. However, log function does not take negative inputs, so I will say multiplication over arbitrary real number is not feasible.

Model structure, all hyperparameters setting same as Question 2:

```
class MultiplyTransformModule(torch.nn.Module):

    # Define the model architecture
    def __init__(self, use_good_weights=False):
        super(MultiplyTransformModule, self).__init__()
        self.model = nn.Sequential(
            torch.nn.Linear(2,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,1)
        )

    def forward(self, x):
        res = self.model(x)
        return res
```

In [6]:

```
class MultiplyTransformDataset(torch.utils.data.Dataset):
    """MULTIPLY Dataset."""

    def __init__(self, num_examples, max_abs=1000, transform=None):
        """create a dataset of the form x_1 + x_2 = y. The input x_1, x_2 is a
        pair of values drawn from the default range [-1000, 1000]. The output y is a scalar.

        PARAMETERS
        ----------
        num_examples    An integer determining how much data we'll generate
        max_abs         The largest absolute value a datapoint can have
        """
        self.length = num_examples

        # make a circular unit Gaussian and draw samples from it
#         data = np.random.multivariate_normal(mean=[0,0],cov=[[1,0],[0,1]],size=self.length)
        data = np.random.uniform(low=1., high=1000., size=(self.length, 2))

        data *= max_abs
        data = np.log(data)
        # figure out the label (i.e. the result of the multiplication)
        label = np.log(np.multiply(data.T[0],data.T[1]))

        # turn it into a tensor
        self.data =  torch.tensor(data).to(device, dtype=torch.float32)
        self.label = torch.tensor(label).to(device, dtype=torch.float32)

    def __len__(self):
        return self.length

    def __getitem__(self, idx):
        return self.data[idx], self.label[idx]

class MultiplyTransformModule(torch.nn.Module):

    # Define the model architecture
    def __init__(self, use_good_weights=False):
        super(MultiplyTransformModule, self).__init__()
        self.model = nn.Sequential(
            torch.nn.Linear(2,100),
            nn.Sigmoid(),
            torch.nn.Linear(100,1)
        )

    def forward(self, x):

        res = self.model(x)
        return res

def test_model_transform(model, data, target):

    model.eval()
    with torch.no_grad():
        output = torch.exp(model(data))
        output = output.squeeze()
        loss = F.mse_loss(output, target)

        loss = round(loss.item(),8)

    return loss
```

In [7]:

```
bs=100
epochs=20

multModel = MultiplyTransformModule().to(device)
optimizer = torch.optim.SGD(multModel.parameters(), lr=0.01)


train_ds = MultiplyTransformDataset(9000)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True, drop_last=False)

test_ds = MultiplyTransformDataset(1000)
test_dl = DataLoader(test_ds, batch_size=bs, shuffle=True, drop_last=False)
train_lst, test_lst = [], []
for i in range(epochs):
    for (xxx, yyy) in train_dl:
#         xxx = torch.log(xxx)
#         yyy = torch.log(yyy)
#         print(xxx, yyy)
        train_model(multModel, xxx, yyy, optimizer)

    train_loss = test_model(multModel, train_ds[:][0], train_ds[:][1])
    train_lst.append(train_loss)
    test_loss = test_model(multModel, test_ds[:][0], test_ds[:][1])
    test_lst.append(test_loss)
    print(f"epoch {i} train loss is {train_loss}, test loss is {test_loss}")
```

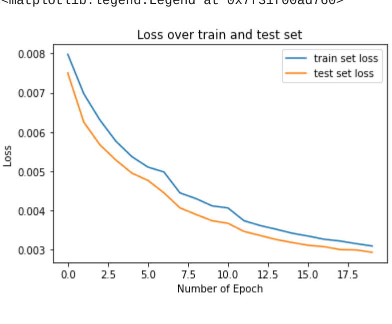
```
epoch 0 train loss is 0.00797877, test loss is 0.0074996
epoch 1 train loss is 0.00697133, test loss is 0.00624631
epoch 2 train loss is 0.00630535, test loss is 0.00567148
epoch 3 train loss is 0.00576186, test loss is 0.00528141
epoch 4 train loss is 0.00536786, test loss is 0.00494725
epoch 5 train loss is 0.00510185, test loss is 0.00476132
epoch 6 train loss is 0.00497706, test loss is 0.00444613
epoch 7 train loss is 0.00444298, test loss is 0.00406065
epoch 8 train loss is 0.00429792, test loss is 0.00389301
epoch 9 train loss is 0.00411287, test loss is 0.00373262
epoch 10 train loss is 0.00405794, test loss is 0.00366715
epoch 11 train loss is 0.00373261, test loss is 0.00345941
epoch 12 train loss is 0.00361095, test loss is 0.00335708
epoch 13 train loss is 0.00351588, test loss is 0.00325329
epoch 14 train loss is 0.00341481, test loss is 0.00317708
epoch 15 train loss is 0.0033419, test loss is 0.00310688
epoch 16 train loss is 0.00326017, test loss is 0.00307057
epoch 17 train loss is 0.00321486, test loss is 0.00299629
epoch 18 train loss is 0.00314723, test loss is 0.00298804
epoch 19 train loss is 0.0030881, test loss is 0.00292721
```

In [8]:

```
plt.clf()
plt.plot(train_lst,  label='train set loss')
plt.plot(test_lst,   label='test set loss')
plt.ylabel("Loss")
plt.xlabel("Number of Epoch")
plt.title("Loss over train and test set")
plt.legend()
```

Out[8]:

```
<matplotlib.legend.Legend at 0x7f31f00ad760>
```



Question 4 (2 points)

Suppose you want to build a fully convolutional network, YouNet, which converts an image with cropped ImageNet dimensions (256, 256), to MNIST dimensions (28, 28), and back to (256, 256). This network contains a convolutional layer that maps an image from (256, 256) -> (28, 28), and a transposed convolutional layer that maps an image from (28, 28) -> (256, 256).

In [9]:

```
import torch
import torch.nn as nn
from typing import Tuple

class YouNet(nn.Module):
    def __init__(self,
                 kernel_1: Tuple[int, int],
                 kernel_2: Tuple[int, int],
                 stride_1: Tuple[int, int] = (1, 1),
                 stride_2: Tuple[int, int] = (1, 1)):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 1, kernel_1, stride=stride_1)
        self.conv2 = nn.ConvTranspose2d(1, 1, kernel_2, stride=stride_2)

    def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        mnist = self.conv1(x)
        imagenet = self.conv2(mnist)
        return mnist, imagenet
```

1. Find valid kernel sizes for the convolutional layers when `stride=(1, 1)`. By 'valid', we mean that using the kernel results in a `mnist.shape` and an `imagenet.shape` that pass the assert statement below.

In [10]:

```
kernel_1 = (229,229)# YOUR ANSWER GOES HERE
kernel_2 = (229,229)# YOUR ANSWER GOES HERE

network = YouNet(kernel_1, kernel_2)
mnist, imagenet = network(torch.zeros(1, 1, 256, 256))

assert mnist.shape == (1, 1, 28, 28)
assert imagenet.shape == (1, 1, 256, 256)
```

1. Find valid kernel sizes for when `stride=(8, 8)`

In [11]:

```
kernel_1 = (40, 40)# YOUR ANSWER GOES HERE
kernel_2 = (40, 40) # YOUR ANSWER GOES HERE

network = YouNet(kernel_1, kernel_2, stride_1=(8, 8), stride_2=(8, 8))
mnist, imagenet = network(torch.zeros(1, 1, 256, 256))

assert mnist.shape == (1, 1, 28, 28)
assert imagenet.shape == (1, 1, 256, 256)
```

1. Suppose instead of processing an image of size (256, 256) with the YouNet you implemented in part 2, you want to process an input image of size (257, 257). What would the sizes of the two processed output images be? Why doesn't the imagenet output have dimensionality (257, 257)? (Hint: Does the strided convolution process all the rows and columns of the original image?)

In [12]:

```
kernel_1 = (40, 40)# YOUR ANSWER GOES HERE
kernel_2 = (40, 40) # YOUR ANSWER GOES HERE

network = YouNet(kernel_1, kernel_2, stride_1=(8, 8), stride_2=(8, 8))
mnist, imagenet = network(torch.zeros(1, 1, 257, 257))

assert mnist.shape == (1, 1, 28, 28)
assert imagenet.shape == (1, 1, 256, 256)
```

Answer

The dimensions for two processed output images are the same as part 2; this can be checked from above code snippets. Since stride is 8, and 257%8 == 1, so last row and column is visited by the kernel.

1. Suppose you are processing an image of size (264, 264) with the YouNet implemented in part 2. What would be the sizes of the two processed images output by the network? For an image of this size, does the imagenet output have the same size as the input?

In [13]:

```
kernel_1 = (40, 40)# YOUR ANSWER GOES HERE
kernel_2 = (40, 40) # YOUR ANSWER GOES HERE

network = YouNet(kernel_1, kernel_2, stride_1=(8, 8), stride_2=(8, 8))
mnist, imagenet = network(torch.zeros(1, 1, 264, 264))

assert mnist.shape == (1, 1, 29, 29)
assert imagenet.shape == (1, 1, 264, 264)
```

Answer

Two output image shapes are (29, 29) and (264, 264); this be can checked by above code snippets; imagenet output has the same size as its input. We can calculate this using the formula provided on pytorch conv2d/convtranspose2d documentation page.
For convolution 2d, $H_{out} = \left\lfloor \frac{264 + 2 * 0 - 1 * (40 - 1) - 1}{8} + 1 \right\rfloor = 29$, and we are using square image, width is same as height.

For convolution transpose 2d, $H_{out} = (29 - 1) * 8 - 2 * 0 + 1 * (40 - 1) + 0 + 1 = 264$, and we are using square image, width is same as height.

Int this case 264%8=0, so the kernel covers the whole image; the kernel visits every row and column of an image, so the imagenet output can back to same shape as input.

Question 5 (3 points)

The paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift is in our class readings list and describes one of the most popular normalization approaches.

Specifically, you are to reproduce the experiment in **section 4.1** of the paper, providing an output figure like **figure 1(a)**.

This means making a network module that has **the architecture described in section 4.1** of the paper. You will need to create two alternative versions of your network module: one model that has at least one batch normalization layer, another module that has no batch normalization layer. Pytorch provides a handy function or two to help with this.

- Note, you don't have to duplicate their weight initializations. Using the default weight initializations is fine.

The famous MNIST dataset is available in torchvision.datasets. You can download it just by declarng the dataset and specifying download=true. See the torchvision.dataset docs for more on that.

- Note, torchvision.datasets has both test and train datasets available for MNIST.
- Note, they never specify in the paper HOW BIG the testing (I think they mean VALIDATION, actually) set they use is. Yours could be just a couple of hundred examples.
- Note that you don't have to run the test after every training step. Every 20 training steps would be fine.

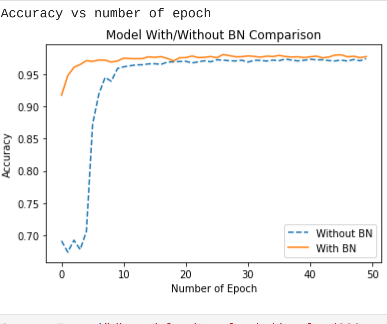
1. Put your graph similar to **figure 1(a)** from the paper below.

In [7]:

```
temp = Image("/home/alen/Downloads/download222.png")
print("Accuracy vs number of epoch")
temp
```

```
Accuracy vs number of epoch
```

Out[7]:

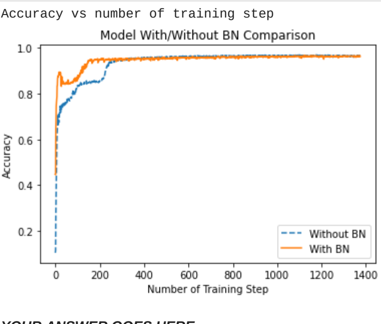


In [8]:

```
temp = Image("/home/alen/Downloads/download333.png")
print("Accuracy vs number of training step")
temp
```

```
Accuracy vs number of training step
```

Out[8]:



YOUR ANSWER GOES HERE

1. Put your analysis of the effectiveness of batch normalization for your network and dataset below. Did you duplicate their results?

Yes I did reproduce the paper's results. The dataset is MNIST, I use 55000 examples as training set and 5000 examples as validation set, above diagram is based on those 60000 examples, which are originally mnist_train_full from pytorch. From above diagram we can see the model without BN converges much slower comparing to the model with BN layers. The model with BN reaches over 90% accuracy on epoch 1, whereas the model without BN roughly achieved same accuracy on epoch 8. According to paper, internal covariate shift is the problem where input distribution of each layer changes (due to last layer's outputs change), the model becomes hard to update, especially in the saturated area (flat region) of activation functions (gradient is small here). Batchnorm lies before activation layers; it makes each layer's input stable and using alpha, beta to linear transform normalized data to avoid data falls in pure linear area in activation functions to make model remain its expressive power.

YOUR ANSWER GOES HERE

1. Put all your code (including network modules, data loaders, testing and training code) below.

In [1]:

```
import time
import torch
import torch.nn as nn
import torchvision
import torchvision.datasets as datasets
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
# from IPython.display import Image
```

In [2]:

```
data_dir = "./data/"
# download MNIST "test" dataset
mnist_test = torchvision.datasets.MNIST(data_dir, train=False, download=True)

# download MNIST "train" dataset and set aside a portion for validation
mnist_train_full = datasets.MNIST(data_dir, train=True, download=True)
mnist_train, mnist_val = torch.utils.data.random_split(mnist_train_full, [55000, 5000])

type(mnist_test), type(mnist_train), type(mnist_val)
```

Out[2]:

```
(torchvision.datasets.mnist.MNIST,
 torch.utils.data.dataset.Subset,
 torch.utils.data.dataset.Subset)
```

In [3]:

```
class MNISTNetwork1(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
                    nn.Linear(28*28, 100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 10),
                    nn.Sigmoid()
                    )

    def forward(self, x):
        batch_size, channels, width, height = x.size()
        x = x.view(batch_size, -1)
        res = self.model(x)
        return res

class MNISTNetwork2(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
                    nn.Linear(28*28, 100),
                    nn.BatchNorm1d(100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 100),
                    nn.BatchNorm1d(100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 100),
                    nn.BatchNorm1d(100),
                    nn.Sigmoid(),
                    torch.nn.Linear(100, 10),
                    nn.Sigmoid()
                    )

    def forward(self, x):
        batch_size, channels, width, height = x.size()
        x = x.view(batch_size, -1)
        res = self.model(x)
        return res
```

In [4]:
```python
# device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
def training_loop(model, save_path, epochs, batch_size, device="cuda:0"):
    """
    Train a neural network model for digit recognition on the MNIST dataset.

    Parameters
    ----------
    save_path (str):  path/filename for model checkpoint, e.g. 'my_model.pt'

    epochs (int):     number of iterations through the whole dataset for training

    batch_size (int): size of a single batch of inputs

    device (str):     device on which tensors are placed; should be 'cpu' or 'cuda'.
                      More on this in the next section!

    Returns
    -------
    model (nn.Module): final trained model

    save_path (str):   path/filename for model checkpoint, so that we can load our model
                       later to test on unseen data

    device (str):      the device on which we carried out training, so we can match it
                       when we test the final model on unseen data later
    """

    model.to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999))

    # make a new directory in which to download the MNIST dataset
    data_dir = "./data/"

    # initialize a Transform object to prepare our data
    transform = torchvision.transforms.Compose([
        torchvision.transforms.ToTensor(),
#         lambda x: x>0,
        lambda x: x.float(),
    ])

    # load MNIST "test" dataset from disk
    mnist_test = torchvision.datasets.MNIST(data_dir, train=False, download=False, transform=transform)

    # load MNIST "train" dataset from disk and set aside a portion for validation
    mnist_train_full = datasets.MNIST(data_dir, train=True, download=False, transform=transform)
    mnist_train, mnist_val = torch.utils.data.random_split(mnist_train_full, [55000, 5000])

    # initialize a DataLoader object for each dataset
    train_dataloader = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=True)
    val_dataloader = torch.utils.data.DataLoader(mnist_val, batch_size=batch_size, shuffle=False)
    test_dataloader = torch.utils.data.DataLoader(mnist_test, batch_size=1, shuffle=False)

    # a PyTorch categorical cross-entropy loss object
    loss_fn = torch.nn.CrossEntropyLoss()

    # time training process
    st = time.time()

    acc_lst=[]
    acc_lst_ts =[]
    i_train_step = 0
    # time to start training!
    for epoch_idx, epoch in enumerate(range(epochs)):

        # keep track of best validation accuracy; if improved upon, save checkpoint
        best_acc = 0.0

        # loop through the entire dataset once per epoch
        train_loss = 0.0
        train_acc = 0.0
        train_total = 0
        model.train()
        for batch_idx, batch in enumerate(train_dataloader):

            # clear gradients
            optimizer.zero_grad()

            # unpack data and labels
            x, y = batch
            x = x.to(device)  # we'll cover this in the next section!
            y = y.to(device)  # we'll cover this in the next section!

            # generate predictions and compute loss
            output = model(x)  # (batch_size, 10)
            loss = loss_fn(output, y)

            # compute accuracy
            preds = output.argmax(dim=1)
            acc = preds.eq(y).sum().item()/len(y)

            # compute gradients and update model parameters
            loss.backward()
            optimizer.step()

            # update statistics
            train_loss += (loss * len(x))
            train_acc += (acc * len(x))
            train_total += len(x)

            i_train_step += 1

# NOTES uncomments below code if you want to check on validation set every 20 training step
            if i_train_step % 20 == 0:
                model.eval()
                val_acc = 0.0
                val_total = 0
                for batch_idx, batch in enumerate(val_dataloader):
                    with torch.no_grad():
                        x, y = batch
                        x = x.to(device)
                        y = y.to(device)
                        output = model(x)
                        preds = output.argmax(dim=1)
                        acc = preds.eq(y).sum().item()/len(y)
                        val_acc += (acc * len(x))
                        val_total += len(x)
                val_acc /= val_total
                acc_lst_ts.append(val_acc)


        train_loss /= train_total
        train_acc /= train_total


        # perform validation once per epoch
        val_loss = 0.0
        val_acc = 0.0
        val_total = 0

        model.eval()
        for batch_idx, batch in enumerate(val_dataloader):
            with torch.no_grad():
                x, y = batch
                x = x.to(device)
                y = y.to(device)
                output = model(x)
                loss = loss_fn(output, y)
                preds = output.argmax(dim=1)
                acc = preds.eq(y).sum().item()/len(y)
                val_loss += (loss * len(x))
                val_acc += (acc * len(x))
                val_total += len(x)
        val_loss /= val_total
        val_acc /= val_total
        acc_lst.append(val_acc)
        print(f"Epoch {epoch_idx + 1}: val loss {val_loss :0.3f}, val acc {val_acc :0.3f}, train loss {train_lo


    print(f"Total training time (s): {time.time() - st :0.3f}")
#     print(i_train_step, ' ===================')

    return model, save_path, acc_lst, acc_lst_ts
```

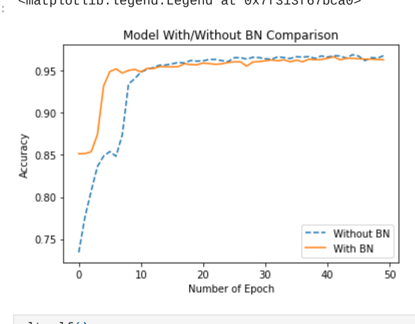
In [5]:
```python
modNoBN = MNISTNetwork1()
modBN = MNISTNetwork2()

# run our training loop
modelnobn, save_pathnobn, acc_lst_nobn, acc_ts_nobn = training_loop(modNoBN, "nobn", 50, 100)
print('==================================================================')
modelbn, save_pathbn, acc_lst_bn, acc_ts_bn = training_loop(modBN, "bn", 50, 100)
```

```
Epoch 1: val loss 1.683, val acc 0.734, train loss 1.882, train acc 0.590
Epoch 2: val loss 1.615, val acc 0.776, train loss 1.638, train acc 0.758
Epoch 3: val loss 1.592, val acc 0.807, train loss 1.600, train acc 0.790
Epoch 4: val loss 1.549, val acc 0.836, train loss 1.564, train acc 0.828
Epoch 5: val loss 1.538, val acc 0.849, train loss 1.538, train acc 0.852
Epoch 6: val loss 1.532, val acc 0.854, train loss 1.529, train acc 0.859
Epoch 7: val loss 1.528, val acc 0.848, train loss 1.523, train acc 0.864
Epoch 8: val loss 1.519, val acc 0.874, train loss 1.513, train acc 0.872
Epoch 9: val loss 1.507, val acc 0.934, train loss 1.502, train acc 0.926
Epoch 10: val loss 1.503, val acc 0.941, train loss 1.494, train acc 0.955
Epoch 11: val loss 1.501, val acc 0.948, train loss 1.489, train acc 0.964
Epoch 12: val loss 1.498, val acc 0.952, train loss 1.486, train acc 0.968
Epoch 13: val loss 1.498, val acc 0.954, train loss 1.483, train acc 0.972
Epoch 14: val loss 1.498, val acc 0.957, train loss 1.482, train acc 0.975
Epoch 15: val loss 1.497, val acc 0.957, train loss 1.480, train acc 0.977
Epoch 16: val loss 1.497, val acc 0.958, train loss 1.479, train acc 0.979
Epoch 17: val loss 1.496, val acc 0.960, train loss 1.477, train acc 0.982
Epoch 18: val loss 1.496, val acc 0.959, train loss 1.476, train acc 0.983
Epoch 19: val loss 1.493, val acc 0.962, train loss 1.476, train acc 0.984
Epoch 20: val loss 1.495, val acc 0.961, train loss 1.475, train acc 0.985
Epoch 21: val loss 1.495, val acc 0.962, train loss 1.474, train acc 0.986
Epoch 22: val loss 1.494, val acc 0.963, train loss 1.473, train acc 0.987
Epoch 23: val loss 1.493, val acc 0.963, train loss 1.473, train acc 0.988
Epoch 24: val loss 1.495, val acc 0.962, train loss 1.472, train acc 0.988
Epoch 25: val loss 1.496, val acc 0.961, train loss 1.472, train acc 0.989
Epoch 26: val loss 1.492, val acc 0.965, train loss 1.471, train acc 0.989
Epoch 27: val loss 1.494, val acc 0.966, train loss 1.471, train acc 0.990
Epoch 28: val loss 1.493, val acc 0.964, train loss 1.470, train acc 0.990
Epoch 29: val loss 1.492, val acc 0.966, train loss 1.470, train acc 0.991
Epoch 30: val loss 1.492, val acc 0.966, train loss 1.470, train acc 0.991
Epoch 31: val loss 1.494, val acc 0.964, train loss 1.470, train acc 0.991
Epoch 32: val loss 1.493, val acc 0.964, train loss 1.470, train acc 0.991
Epoch 33: val loss 1.492, val acc 0.967, train loss 1.470, train acc 0.991
Epoch 34: val loss 1.494, val acc 0.966, train loss 1.469, train acc 0.992
Epoch 35: val loss 1.493, val acc 0.965, train loss 1.469, train acc 0.992
Epoch 36: val loss 1.492, val acc 0.967, train loss 1.469, train acc 0.992
Epoch 37: val loss 1.491, val acc 0.966, train loss 1.469, train acc 0.992
Epoch 38: val loss 1.492, val acc 0.967, train loss 1.469, train acc 0.993
Epoch 39: val loss 1.493, val acc 0.965, train loss 1.469, train acc 0.993
Epoch 40: val loss 1.492, val acc 0.968, train loss 1.468, train acc 0.993
Epoch 41: val loss 1.492, val acc 0.966, train loss 1.468, train acc 0.993
Epoch 42: val loss 1.491, val acc 0.968, train loss 1.468, train acc 0.993
Epoch 43: val loss 1.493, val acc 0.968, train loss 1.468, train acc 0.994
Epoch 44: val loss 1.492, val acc 0.965, train loss 1.468, train acc 0.993
Epoch 45: val loss 1.491, val acc 0.969, train loss 1.468, train acc 0.993
Epoch 46: val loss 1.492, val acc 0.967, train loss 1.468, train acc 0.993
Epoch 47: val loss 1.495, val acc 0.962, train loss 1.468, train acc 0.993
Epoch 48: val loss 1.493, val acc 0.966, train loss 1.468, train acc 0.993
Epoch 49: val loss 1.492, val acc 0.965, train loss 1.468, train acc 0.993
Epoch 50: val loss 1.491, val acc 0.968, train loss 1.467, train acc 0.994
Total training time (s): 786.761
==================================================================
Epoch 1: val loss 1.560, val acc 0.851, train loss 1.724, train acc 0.814
Epoch 2: val loss 1.539, val acc 0.852, train loss 1.569, train acc 0.853
Epoch 3: val loss 1.531, val acc 0.854, train loss 1.553, train acc 0.855
Epoch 4: val loss 1.525, val acc 0.874, train loss 1.533, train acc 0.879
Epoch 5: val loss 1.522, val acc 0.932, train loss 1.528, train acc 0.915
Epoch 6: val loss 1.511, val acc 0.949, train loss 1.516, train acc 0.944
Epoch 7: val loss 1.508, val acc 0.952, train loss 1.508, train acc 0.952
Epoch 8: val loss 1.508, val acc 0.947, train loss 1.502, train acc 0.956
Epoch 9: val loss 1.505, val acc 0.950, train loss 1.501, train acc 0.958
Epoch 10: val loss 1.503, val acc 0.952, train loss 1.497, train acc 0.960
Epoch 11: val loss 1.504, val acc 0.949, train loss 1.497, train acc 0.960
Epoch 12: val loss 1.503, val acc 0.953, train loss 1.494, train acc 0.963
Epoch 13: val loss 1.502, val acc 0.953, train loss 1.493, train acc 0.964
Epoch 14: val loss 1.501, val acc 0.955, train loss 1.490, train acc 0.967
Epoch 15: val loss 1.501, val acc 0.955, train loss 1.490, train acc 0.967
Epoch 16: val loss 1.500, val acc 0.955, train loss 1.488, train acc 0.970
Epoch 17: val loss 1.499, val acc 0.955, train loss 1.488, train acc 0.969
Epoch 18: val loss 1.498, val acc 0.958, train loss 1.486, train acc 0.971
Epoch 19: val loss 1.498, val acc 0.957, train loss 1.486, train acc 0.972
Epoch 20: val loss 1.497, val acc 0.957, train loss 1.485, train acc 0.973
Epoch 21: val loss 1.497, val acc 0.959, train loss 1.484, train acc 0.975
Epoch 22: val loss 1.497, val acc 0.958, train loss 1.483, train acc 0.975
Epoch 23: val loss 1.497, val acc 0.958, train loss 1.482, train acc 0.976
Epoch 24: val loss 1.497, val acc 0.958, train loss 1.481, train acc 0.977
Epoch 25: val loss 1.496, val acc 0.960, train loss 1.481, train acc 0.977
Epoch 26: val loss 1.495, val acc 0.961, train loss 1.480, train acc 0.979
Epoch 27: val loss 1.496, val acc 0.961, train loss 1.480, train acc 0.979
Epoch 28: val loss 1.500, val acc 0.955, train loss 1.479, train acc 0.980
Epoch 29: val loss 1.496, val acc 0.960, train loss 1.479, train acc 0.980
Epoch 30: val loss 1.495, val acc 0.961, train loss 1.478, train acc 0.981
Epoch 31: val loss 1.495, val acc 0.962, train loss 1.477, train acc 0.982
Epoch 32: val loss 1.493, val acc 0.963, train loss 1.477, train acc 0.983
Epoch 33: val loss 1.494, val acc 0.962, train loss 1.477, train acc 0.983
Epoch 34: val loss 1.493, val acc 0.963, train loss 1.476, train acc 0.983
Epoch 35: val loss 1.495, val acc 0.961, train loss 1.476, train acc 0.983
Epoch 36: val loss 1.494, val acc 0.963, train loss 1.476, train acc 0.984
Epoch 37: val loss 1.494, val acc 0.961, train loss 1.475, train acc 0.984
Epoch 38: val loss 1.493, val acc 0.964, train loss 1.474, train acc 0.985
Epoch 39: val loss 1.493, val acc 0.963, train loss 1.475, train acc 0.985
Epoch 40: val loss 1.494, val acc 0.963, train loss 1.474, train acc 0.986
Epoch 41: val loss 1.493, val acc 0.965, train loss 1.474, train acc 0.986
Epoch 42: val loss 1.492, val acc 0.967, train loss 1.474, train acc 0.987
Epoch 43: val loss 1.493, val acc 0.963, train loss 1.473, train acc 0.987
Epoch 44: val loss 1.493, val acc 0.965, train loss 1.473, train acc 0.987
Epoch 45: val loss 1.492, val acc 0.965, train loss 1.473, train acc 0.987
Epoch 46: val loss 1.494, val acc 0.964, train loss 1.472, train acc 0.988
Epoch 47: val loss 1.494, val acc 0.964, train loss 1.473, train acc 0.987
Epoch 48: val loss 1.493, val acc 0.964, train loss 1.472, train acc 0.989
Epoch 49: val loss 1.493, val acc 0.963, train loss 1.472, train acc 0.989
Epoch 50: val loss 1.494, val acc 0.963, train loss 1.472, train acc 0.988
Total training time (s): 838.413
```

In [6]:
```python
plt.clf()
plt.plot(acc_lst_nobn,linestyle='dashed', label='Without BN')
plt.plot(acc_lst_bn,   label='With BN')
plt.ylabel("Accuracy")
plt.xlabel("Number of Epoch")
plt.title("Model With/Without BN Comparison")
plt.legend()
```

Out[6]:
```
<matplotlib.legend.Legend at 0x7f313f67bca0>
```



In [7]:
```python
plt.clf()
plt.plot(acc_ts_nobn,linestyle='dashed', label='Without BN')
plt.plot(acc_ts_bn,   label='With BN')
plt.ylabel("Accuracy")
plt.xlabel("Number of Training Step")
plt.title("Model With/Without BN Comparison")
plt.legend()
```

Out[7]:
```
<matplotlib.legend.Legend at 0x7f313f63eeb0>
```