This can be run run on Google Colab using this link

MNIST Classifiers (Convolutional Neural Networks and Fully Connected Networks)

Optional: Installing Wandb to see cool analysis of you code. You can go through the documentation here. We will do it for this assignment to get a taste of the GPU and CPU utilizations. If this is creating problems to your code, please comment out all the wandb lines from the notebook

```
# Uncomment the below line to install wandb (optinal)
# !pip install wandb
# Uncomment the below line to install torchinfo (https://github.com/TylerYep/torchinfo) [Mandatory]
!pip install torchinfo
```

```
    Requirement already satisfied: torchinfo in /usr/local/lib/python3.10/dist-packages (1.8.0)
```

```
%%bash

wget -N https://cs7150.baulab.info/2022-Fall/data/mnist-classify.pth
```

```
    --2023-10-18 21:50:43--  https://cs7150.baulab.info/2022-Fall/data/mnist-classify.pth
    Resolving cs7150.baulab.info (cs7150.baulab.info)... 35.232.255.106
    Connecting to cs7150.baulab.info (cs7150.baulab.info)|35.232.255.106|:443... connected.
    HTTP request sent, awaiting response... 304 Not Modified
    File ‘mnist-classify.pth’ not modified on server. Omitting download.
```

```
# Importing libraries
import matplotlib.pyplot as plt

import torch
import torchvision
from torchvision import transforms
from torch.utils.data import DataLoader,random_split,Subset
from torch import nn
import torch.nn.functional as F
import torch.optim as optim
from torchinfo import summary
import numpy as np
import datetime

from typing import List
from collections import OrderedDict
import math
```

```
# Create an account at https://wandb.ai/site and paste the api key here (optional)
# import wandb
# wandb.init(project="hw3.1-ConvNets")
```

```
    wandb: Currently logged in as: wellzhang1217 (7150). Use `wandb login --relogin` to force
    Tracking run with wandb version 0.15.12
    Run data is saved locally in /content/wandb/run-20231018_214449-fhpmgdn8
    Syncing run dashing-thunder-3 to Weights & Biases (docs)
    View project at https://wandb.ai/7150/hw3.1-ConvNets
    View run at https://wandb.ai/7150/hw3.1-ConvNets/runs/fhpmgdn8
    Display W&B run
```

Some helper functions to view network parameters

```
def view_network_parameters(model):
    # Visualise the number of parameters
    tensor_list = list(model.state_dict().items())
    total_parameters = 0
    print('Model Summary\n')
    for layer_tensor_name, tensor in tensor_list:
        total_parameters += int(torch.numel(tensor))
        print('{}: {} elements'.format(layer_tensor_name, torch.numel(tensor)))
    print(f'\nTotal Trainable Parameters: {total_parameters}!')
```

```
def view_network_shapes(model, input_shape):
    print(summary(conv_net, input_size=input_shape))
```

Fully Connected Network for Image Classification

Let's build a simple fully connected network!

```
def simple_fc_net():
    model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(1*28*28,8*28*28),
        nn.ReLU(),
        nn.Linear(8*28*28,16*14*14),
        nn.ReLU(),
        nn.Linear(16*14*14,32*7*7),
        nn.ReLU(),
        nn.Linear(32*7*7,288),
        nn.ReLU(),
        nn.Linear(288,64),
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax())
    return model
```

```
fc_net = simple_fc_net()
```

```
view_network_parameters(fc_net)
```

```
Model Summary

1.weight: 4917248 elements
1.bias: 6272 elements
3.weight: 19668992 elements
3.bias: 3136 elements
5.weight: 4917248 elements
5.bias: 1568 elements
7.weight: 451584 elements
7.bias: 288 elements
9.weight: 18432 elements
9.bias: 64 elements
11.weight: 640 elements
11.bias: 10 elements

Total Trainable Parameters: 29985482!
```

```
view_network_parameters(fc_net)
```

```
Model Summary

1.weight: 4917248 elements
1.bias: 6272 elements
3.weight: 19668992 elements
3.bias: 3136 elements
5.weight: 4917248 elements
5.bias: 1568 elements
7.weight: 451584 elements
7.bias: 288 elements
9.weight: 18432 elements
9.bias: 64 elements
11.weight: 640 elements
11.bias: 10 elements

Total Trainable Parameters: 29985482!
```

```
from torchinfo import summary
summary(fc_net, input_size=(1, 1, 28,28))
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/container.py:217: UserWarning: Implicit dimension choice for log_softmax has bee
  input = module(input)
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 10]                   --
├─Flatten: 1-1                           [1, 784]                  --
├─Linear: 1-2                            [1, 6272]                 4,923,520
├─ReLU: 1-3                              [1, 6272]                 --
├─Linear: 1-4                            [1, 3136]                 19,672,128
├─ReLU: 1-5                              [1, 3136]                 --
├─Linear: 1-6                            [1, 1568]                 4,918,816
├─ReLU: 1-7                              [1, 1568]                 --
├─Linear: 1-8                            [1, 288]                  451,872
├─ReLU: 1-9                              [1, 288]                  --
├─Linear: 1-10                           [1, 64]                   18,496
├─ReLU: 1-11                             [1, 64]                   --
├─Linear: 1-12                           [1, 10]                   650
├─LogSoftmax: 1-13                       [1, 10]                   --
==========================================================================================
Total params: 29,985,482
Trainable params: 29,985,482
```

```
Non-trainable params: 0
Total mult-adds (M): 29.99
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 119.94
Estimated Total Size (MB): 120.04
==========================================================================================
```

Exercise: Now try to add different layers and see how the network parameters vary. Does adding layers reduce the parameters? Does the number of hidden neurons in the layers affect the total trainable parameters?

Add a few sentences on your observations while using various architectures

```
# add an extra layer
def modified_simple_fc_net1():
    model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(1*28*28,8*28*28),
        nn.ReLU(),
        nn.Linear(8*28*28,16*14*14),
        nn.ReLU(),
        nn.Linear(16*14*14,32*7*7),
        nn.ReLU(),
        nn.Linear(32*7*7,288),
        nn.ReLU(),
        nn.Linear(288,100), # extra layer added
        nn.ReLU(),
        nn.Linear(100,64),
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax())
    return model


# increase the number of hidden neurons
def modified_simple_fc_net2():
    model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(1*28*28,8*28*28),
        nn.ReLU(),
        nn.Linear(8*28*28,16*14*14),
        nn.ReLU(),
        nn.Linear(16*14*14,32*7*7),
        nn.ReLU(),
        nn.Linear(32*7*7,512),
        nn.ReLU(),
        nn.Linear(512,64),
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax())
    return model


modified_fc_net1 = modified_simple_fc_net1()
modified_fc_net2 = modified_simple_fc_net2()


view_network_parameters(modified_fc_net1)
view_network_parameters(modified_fc_net2)
```

```
Model Summary

1.weight: 4917248 elements
1.bias: 6272 elements
3.weight: 19668992 elements
3.bias: 3136 elements
5.weight: 4917248 elements
5.bias: 1568 elements
7.weight: 451584 elements
7.bias: 288 elements
9.weight: 28800 elements
9.bias: 100 elements
11.weight: 6400 elements
11.bias: 64 elements
13.weight: 640 elements
13.bias: 10 elements

Total Trainable Parameters: 30002350!
Model Summary

1.weight: 4917248 elements
1.bias: 6272 elements
3.weight: 19668992 elements
3.bias: 3136 elements
```

```
5.weight: 4917248 elements
5.bias: 1568 elements
7.weight: 802816 elements
7.bias: 512 elements
9.weight: 32768 elements
9.bias: 64 elements
11.weight: 640 elements
11.bias: 10 elements

Total Trainable Parameters: 30351274!
```

Please type your answer here ...

Adding more layers will always increase the total number of parameters since each layer has its own weights and biases. The additional layer in modified_fc_net_ introduced more parameters. The number of hidden neurons in layers does affect total number of trainable parameters.

Convolutional Neural Network for Image Classification

Let's build a simple CNN to classify our images. **Exercise 3.1.1:** In the function below please add the conv/Relu/Maxpool layers to match the shape of FC-Net. Suppose at the some layer the FC-Net has `28*28*16` dimension, we want your conv_net to have `16 X 28 X 28` shape at the same numbered layer.
Extra-credit: Try not to use MaxPool2d !

```
def simple_conv_net():
    model = nn.Sequential(
        nn.Conv2d(1,8,kernel_size=3,padding=1),
        nn.ReLU(),
        nn.MaxPool2d(2,2),
        # TO-DO: Add layers below
        nn.Conv2d(8, 16, kernel_size=3, padding=1),
        nn.ReLU(),
        nn.MaxPool2d(2, 2),
        nn.Conv2d(16, 32, kernel_size=3, padding=1),
        nn.ReLU(),
        # TO-DO, what will your shape be after you flatten? Fill it in place of None
        nn.Flatten(),
        nn.Linear(32 * 7 * 7, 288),
        nn.ReLU(),
        nn.Linear(288, 64),
        # Do not change the code below
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax(dim=1))
    return model
```

```
conv_net = simple_conv_net()
```

```
view_network_parameters(conv_net)
```

```
Model Summary

0.weight: 72 elements
0.bias: 8 elements
3.weight: 1152 elements
3.bias: 16 elements
6.weight: 4608 elements
6.bias: 32 elements
9.weight: 451584 elements
9.bias: 288 elements
11.weight: 18432 elements
11.bias: 64 elements
13.weight: 640 elements
13.bias: 10 elements

Total Trainable Parameters: 476906!
```

```
view_network_shapes(conv_net, input_shape=(1,1,28,28))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Sequential                               [1, 10]                   --
├─Conv2d: 1-1                            [1, 8, 28, 28]            80
├─ReLU: 1-2                              [1, 8, 28, 28]            --
├─MaxPool2d: 1-3                         [1, 8, 14, 14]            --
├─Conv2d: 1-4                            [1, 16, 14, 14]           1,168
├─ReLU: 1-5                              [1, 16, 14, 14]           --
├─MaxPool2d: 1-6                         [1, 16, 7, 7]             --
├─Conv2d: 1-7                            [1, 32, 7, 7]             4,640
```

```
    ├─ReLU: 1-8                              [1, 32, 7, 7]             --
    ├─Flatten: 1-9                           [1, 1568]                 --
    ├─Linear: 1-10                           [1, 288]                  451,872
    ├─ReLU: 1-11                             [1, 288]                  --
    ├─Linear: 1-12                           [1, 64]                   18,496
    ├─ReLU: 1-13                             [1, 64]                   --
    ├─Linear: 1-14                           [1, 10]                   650
    ├─LogSoftmax: 1-15                       [1, 10]                   --
    ==========================================================================================
    Total params: 476,906
    Trainable params: 476,906
    Non-trainable params: 0
    Total mult-adds (M): 0.99
    ==========================================================================================
    Input size (MB): 0.00
    Forward/backward pass size (MB): 0.09
    Params size (MB): 1.91
    Estimated Total Size (MB): 2.00
    ==========================================================================================
```

```
def simple_conv_net_no_maxpool():
    model = nn.Sequential(
        nn.Conv2d(1, 8, kernel_size=3, padding=1, stride=2),  # Stride of 2 to halve dimensions: 28x28 -> 14x14
        nn.ReLU(),

        nn.Conv2d(8, 16, kernel_size=3, padding=1, stride=2),  # Stride of 2 to halve dimensions: 14x14 -> 7x7
        nn.ReLU(),

        nn.Conv2d(16, 32, kernel_size=3, padding=1),  # No stride here, to keep 7x7 dimensions
        nn.ReLU(),

        nn.Flatten(),
        nn.Linear(32 * 7 * 7, 288),
        nn.ReLU(),

        nn.Linear(288, 64),
        nn.ReLU(),

        nn.Linear(64,10),
        nn.LogSoftmax(dim=1))
    return model
```

```
conv_net_no_maxpool = simple_conv_net_no_maxpool()
```

```
view_network_parameters(conv_net_no_maxpool)
```

```
    Model Summary

    0.weight: 72 elements
    0.bias: 8 elements
    2.weight: 1152 elements
    2.bias: 16 elements
    4.weight: 4608 elements
    4.bias: 32 elements
    7.weight: 451584 elements
    7.bias: 288 elements
    9.weight: 18432 elements
    9.bias: 64 elements
    11.weight: 640 elements
    11.bias: 10 elements

    Total Trainable Parameters: 476906!
```

```
view_network_shapes(conv_net_no_maxpool, input_shape=(1,1,28,28))
```

```
    ==========================================================================================
    Layer (type:depth-idx)                   Output Shape              Param #
    ==========================================================================================
    Sequential                               [1, 10]                   --
    ├─Conv2d: 1-1                            [1, 8, 28, 28]            80
    ├─ReLU: 1-2                              [1, 8, 28, 28]            --
    ├─MaxPool2d: 1-3                         [1, 8, 14, 14]            --
    ├─Conv2d: 1-4                            [1, 16, 14, 14]           1,168
    ├─ReLU: 1-5                              [1, 16, 14, 14]           --
    ├─MaxPool2d: 1-6                         [1, 16, 7, 7]             --
    ├─Conv2d: 1-7                            [1, 32, 7, 7]             4,640
    ├─ReLU: 1-8                              [1, 32, 7, 7]             --
    ├─Flatten: 1-9                           [1, 1568]                 --
    ├─Linear: 1-10                           [1, 288]                  451,872
    ├─ReLU: 1-11                             [1, 288]                  --
    ├─Linear: 1-12                           [1, 64]                   18,496
    ├─ReLU: 1-13                             [1, 64]                   --
    ├─Linear: 1-14                           [1, 10]                   650
    ├─LogSoftmax: 1-15                       [1, 10]                   --
    ==========================================================================================
    Total params: 476,906
    Trainable params: 476,906
    Non-trainable params: 0
    Total mult-adds (M): 0.99
    ==========================================================================================
    Input size (MB): 0.00
    Forward/backward pass size (MB): 0.09
    Params size (MB): 1.91
    Estimated Total Size (MB): 2.00
    ==========================================================================================
```

Exercise 3.1.2: Why is the final layer a log softmax? What is a softmax function? Can we use ReLU instead of softmax? If yes, what would you do different? If not, tell us why. If you think there is a different answer, feel free to use this space to chart it down

Please type your answer here ...

Why log softmax: Traditional softmax computationally unstable. As the numbers are too big the exponents will probably blow up (computer cannot handle such big numbers) giving Nan as output. Also, dividing large numbers from , can be numerically unstable. The use of log probabilities means representing probabilities on a logarithmic scale, instead of the standard unit interval. Since the probabilities of independent events multiply, and logarithms convert multiplication to addition, log probabilities of independent events add. Log probabilities are thus practical for computations. (Reference: https://medium.com/@AbhiramiVS/softmax-vs-logsoftmax-eb94254445a2)

Softmax Function:Softmax function is used to convert a vector of real numbers into a probability distribution. For each element in the input vector, the softmax function computes the exponential of that element divided by the sum of the exponentials of all the elements in the vector. The resulting output is a probability distribution over n classes, and the sum of the output values will be 1.

ReLU?: I don't think ReLU is recommended since it doesn't generate a probability distribution. As it also allows unbounded positive values, making it less suitable for producing a final classification score. If we are able to set some threshold for classification tasks, ReLU might also work. But the interpretability might be an issue.

Exercise 3.1.3: What is the ratio of number of parameters of Conv-net to number of parameters of FC-Net
$\frac{p_{conv-net}}{p_{fc-net}}$ = 0.159
Do you see the difference ?!

```
476906 / 29985482
```

```
0.015904563415055327
```

The ratio shows the number of parameters used by a conv model to achieve a same model structure is inly about 1 percent of the number of parameters used by a fc-net.

Exercise 3.1.4: Now try to add different layers and see how the network parameters vary. Does adding layers reduce the parameters? Does the number of hidden neurons in the layers affect the total trainable parameters? Use the `build_custom_fc_net` function given below. You do not have to understand the working of it.

Add a few sentences on your observations while using various architectures

```
def build_custom_fc_net(inp_dim: int, out_dim: int, hidden_fc_dim: List[int]):
    '''
    Inputs :

    inp_dim: Shape of the input dimensions (in MNIST case 28*28)
    out_dim: Desired classification classes (in MNIST case 10)
    hidden_fc_dim: List of the intermediate dimension shapes (list of integers). Try different values and see the shapes'

    Return: nn.Sequential (final custom model)
    '''
    assert type(hidden_fc_dim) == list, "Please define hidden_fc_dim as list of integers"
    layers = []
    layers.append((f'flatten', nn.Flatten()))
    # If no hidden layer is required
    if len(hidden_fc_dim) == 0:
        layers.append((f'linear',nn.Linear(math.prod(inp_dim),out_dim)))
        layers.append((f'activation',nn.LogSoftmax()))
    else:
        # Loop over hidden dimensions and add layers
        for idx, dim in enumerate(hidden_fc_dim):
            if idx == 0:
                layers.append((f'linear_{idx+1}',nn.Linear(math.prod(inp_dim),dim)))
                layers.append((f'activation_{idx+1}',nn.ReLU()))
            else:
                layers.append((f'linear_{idx+1}',nn.Linear(hidden_fc_dim[idx-1],dim)))
                layers.append((f'activation_{idx+1}',nn.ReLU()))
        layers.append((f'linear_{idx+2}',nn.Linear(dim,out_dim)))
        layers.append((f'activation_{idx+2}',nn.LogSoftmax()))

    model =  nn.Sequential(OrderedDict(layers))
    return model

# TO-DO build different networks (atleast 3) and see the parameters
#(You don't have to understand the function above. It is a generic way to build a FC-Net)


fc_net_custom1 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc_dim=[128,64,32])
view_network_parameters(fc_net_custom1)

fc_net_custom2 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc_dim=[512, 256])
view_network_parameters(fc_net_custom2)

fc_net_custom3 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc_dim=[256])
view_network_parameters(fc_net_custom3)
```

```
    Model Summary

    linear_1.weight: 100352 elements
    linear_1.bias: 128 elements
    linear_2.weight: 8192 elements
    linear_2.bias: 64 elements
    linear_3.weight: 2048 elements
    linear_3.bias: 32 elements
    linear_4.weight: 320 elements
    linear_4.bias: 10 elements

    Total Trainable Parameters: 111146!
    Model Summary

    linear_1.weight: 401408 elements
    linear_1.bias: 512 elements
    linear_2.weight: 131072 elements
    linear_2.bias: 256 elements
    linear_3.weight: 2560 elements
    linear_3.bias: 10 elements

    Total Trainable Parameters: 535818!
    Model Summary

    linear_1.weight: 200704 elements
    linear_1.bias: 256 elements
    linear_2.weight: 2560 elements
    linear_2.bias: 10 elements

    Total Trainable Parameters: 203530!
```

Let's train the models to see their performace

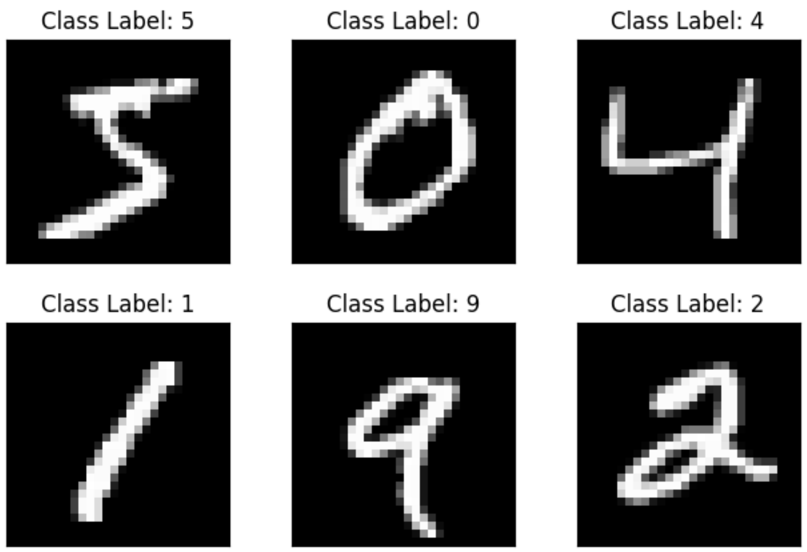
```
# downloading mnist into folder
data_dir = 'data' # make sure that this folder is created in your working dir
# transform the PIL images to tensor using torchvision.transform.toTensor method
train_data = torchvision.datasets.MNIST(data_dir, train=True, download=True, transform=torchvision.transforms.Compose([torchvision.transforms.T
test_data  = torchvision.datasets.MNIST(data_dir, train=False, download=True, transform=torchvision.transforms.Compose([torchvision.transforms.
print(f'Datatype of the dataset object: {type(train_data)}')
# check the length of dataset
n_train_samples = len(train_data)
print(f'Number of samples in training data: {len(train_data)}')
print(f'Number of samples in test data: {len(test_data)}')
# Check the format of dataset
#print(f'Foramt of the dataset: \n {train_data}')

val_split = .2
batch_size=256

train_data_, val_data = random_split(train_data, [int(n_train_samples*(1-val_split)), int(n_train_samples*val_split)])

train_loader = torch.utils.data.DataLoader(train_data_, batch_size=batch_size,shuffle=True)
val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,shuffle=True)
```

```
    Datatype of the dataset object: <class 'torchvision.datasets.mnist.MNIST'>
    Number of samples in training data: 60000
    Number of samples in test data: 10000
```

Displaying the loaded dataset

```
import matplotlib.pyplot as plt
```

```
fig = plt.figure()
for i in range(6):
  plt.subplot(2, 3, i+1)
  plt.tight_layout()
  plt.imshow(train_data[i][0][0], cmap='gray', interpolation='none')
  plt.title("Class Label: {}".format(train_data[i][1]))
  plt.xticks([])
  plt.yticks([])
```



Function to train the model

```
def train_model(model, train_loader, device, loss_fn, optimizer, input_dim=(-1,1,28,28)):
    model.train()
    # Initiate a loss monitor
    train_loss = []
    # Iterate the dataloader (we do not need the label values, this is unsupervised learning and not supervised classification)
    for images, labels in train_loader: # the variable `labels` will be used for customised training
        # reshape input
        images = torch.reshape(images,input_dim)
        images = images.to(device)
        labels = labels.to(device)
        # predict the class
        predicted = model(images)
        loss = loss_fn(predicted, labels)
        # Backward pass (back propagation)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # wandb.log({"Training Loss": loss})
        # wandb.watch(model)
        train_loss.append(loss.detach().cpu().numpy())
    return np.mean(train_loss)
```

Function to test the model

```
# Testing Function
def test_model(model, test_loader, device, loss_fn, input_dim=(-1,1,28,28)):
    # Set evaluation mode for encoder and decoder
    model.eval()
    with torch.no_grad(): # No need to track the gradients
        # Define the lists to store the outputs for each batch
        predicted = []
        actual = []
        for images, labels in test_loader:
            # reshape input
            images = torch.reshape(images,input_dim)
            images = images.to(device)
            labels = labels.to(device)
            ## predict the label
            pred = model(images)
            # Append the network output and the original image to the lists
            predicted.append(pred.cpu())
            actual.append(labels.cpu())
        # Create a single tensor with all the values in the lists
        predicted = torch.cat(predicted)
        actual = torch.cat(actual)
        # Evaluate global loss
        val_loss = loss_fn(predicted, actual)
    return val_loss.data
```

Before we start training let's delete the huge FC-Net we built and build a reasonable FC-Net (You learnt why such larger networks are not reasonable in the previous notebook)

```
del fc_net, fc_net_custom1, fc_net_custom2, fc_net_custom3
torch.cuda.empty_cache()
# Building a reasonable fully connected network
fc_net = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc_dim=[128,64,32])
```

Exercise 3.1.5: Code the `weight_init_xavier` function by referring to https://pytorch.org/docs/stable/nn.init.html. Replace the weight initializations to your own function.

```
### Set the random seed for reproducible results
torch.manual_seed(0)
# Choosing a device based on the env and torch setup
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print(f'Selected device: {device}')

def weight_init_zero(m):
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.constant_(m.weight, 0.0)
        m.bias.data.fill_(0.01)

def weight_init_xavier(m):
    '''
    TO-DO: please add code below to add xavier uniform initialization and remove the 'pass'
    '''
    if isinstance(m, nn.Linear) or isinstance(m, nn.Conv2d):
        torch.nn.init.xavier_uniform_(m.weight)
        m.bias.data.fill_(0.01)


fc_net.to(device)
conv_net.to(device)

# Apply the weight initialization
fc_net.apply(weight_init_zero)
conv_net.apply(weight_init_zero)

# Apply the xavier weight initialization
#TO-DO: Add your function here
fc_net.apply(weight_init_xavier)
conv_net.apply(weight_init_xavier)


# Take the parameters for optimiser
params_to_optimize_fc = [
    {'params': fc_net.parameters()}
]

params_to_optimize_conv = [
    {'params': conv_net.parameters()}
]
### Define the loss function
loss_fn = torch.nn.NLLLoss()
### Define an optimizer (both for the encoder and the decoder!)
lr= 0.001

optim_fc = torch.optim.Adam(params_to_optimize_fc, lr=lr, weight_decay=1e-05)
optim_conv = torch.optim.Adam(params_to_optimize_conv, lr=lr, weight_decay=1e-05)
num_epochs = 30
# wandb.config = {
#   "learning_rate": lr,
#   "epochs": num_epochs,
#   "batch_size": batch_size
# }
```

```
    Selected device: cuda
```

Training the Convolutional Neural Networks

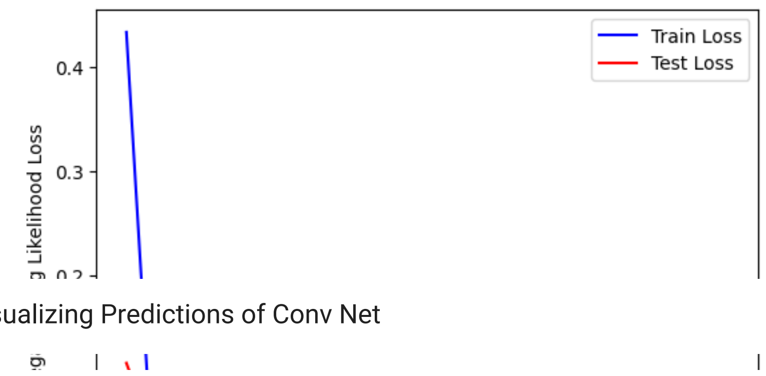
```
print('Conv Net training started')
history_conv = {'train_loss':[],'val_loss':[]}
start_time = datetime.datetime.now()

for epoch in range(num_epochs):
    ### Training

    train_loss = train_model(
        model=conv_net,
        train_loader=train_loader,
        device=device,
        loss_fn=loss_fn,
        optimizer=optim_conv,
        input_dim=(-1,1,28,28))
    ### Validation  (use the testing function)
    val_loss = test_model(
        model=conv_net,
        test_loader=test_loader,
        device=device,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
    # Print Losses
    print(f'Epoch {epoch+1}/{num_epochs} : train loss {train_loss:.3f} \t val loss {val_loss:.3f}')
    history_conv['train_loss'].append(train_loss)
    history_conv['val_loss'].append(val_loss)


print(f'Conv Net training done in {(datetime.datetime.now()-start_time).total_seconds():.3f} seconds!')
```

```
Conv Net training started
Epoch 1/30 : train loss 0.433    val loss 0.116
Epoch 2/30 : train loss 0.105    val loss 0.062
Epoch 3/30 : train loss 0.070    val loss 0.050
Epoch 4/30 : train loss 0.054    val loss 0.042
Epoch 5/30 : train loss 0.043    val loss 0.042
Epoch 6/30 : train loss 0.034    val loss 0.047
Epoch 7/30 : train loss 0.029    val loss 0.028
Epoch 8/30 : train loss 0.025    val loss 0.029
Epoch 9/30 : train loss 0.020    val loss 0.036
Epoch 10/30 : train loss 0.018   val loss 0.034
Epoch 11/30 : train loss 0.018   val loss 0.027
Epoch 12/30 : train loss 0.013   val loss 0.033
Epoch 13/30 : train loss 0.012   val loss 0.038
Epoch 14/30 : train loss 0.011   val loss 0.028
Epoch 15/30 : train loss 0.008   val loss 0.039
Epoch 16/30 : train loss 0.010   val loss 0.035
Epoch 17/30 : train loss 0.007   val loss 0.028
Epoch 18/30 : train loss 0.009   val loss 0.032
Epoch 19/30 : train loss 0.007   val loss 0.030
Epoch 20/30 : train loss 0.007   val loss 0.028
Epoch 21/30 : train loss 0.004   val loss 0.031
Epoch 22/30 : train loss 0.005   val loss 0.029
Epoch 23/30 : train loss 0.005   val loss 0.039
Epoch 24/30 : train loss 0.006   val loss 0.036
Epoch 25/30 : train loss 0.003   val loss 0.035
Epoch 26/30 : train loss 0.007   val loss 0.029
Epoch 27/30 : train loss 0.007   val loss 0.030
Epoch 28/30 : train loss 0.007   val loss 0.028
Epoch 29/30 : train loss 0.003   val loss 0.032
Epoch 30/30 : train loss 0.005   val loss 0.032
Conv Net training done in 216.709 seconds!
```

Visualizing Training Progress of Conv Net (Also check out your wandb.ai homepage)

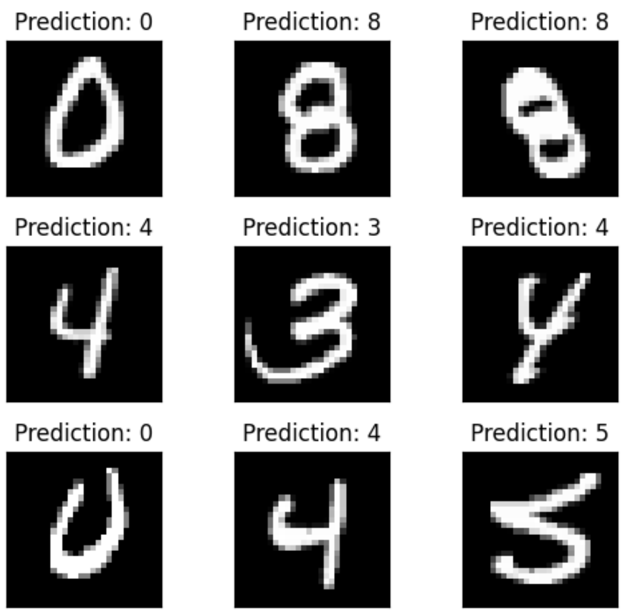
```
fig = plt.figure()
plt.plot(history_conv['train_loss'], color='blue')
plt.plot(history_conv['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

```
Text(0, 0.5, 'Negative Log Likelihood Loss')
```



Visualizing Predictions of Conv Net

```
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
with torch.no_grad():
    example_data = example_data.to(device)
    output = conv_net(example_data)
example_data = example_data.cpu().detach().numpy()
fig = plt.figure(figsize=(5,5))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray',interpolation='none')
    plt.title("Prediction: {}".format(
    output.data.max(1, keepdim=True)[1][i].item()))
    plt.xticks([])
    plt.yticks([])
```



Training the Fully-Connected Neural Networks

Exercise 3.1.6: Train the fully connected neural network and analyse it

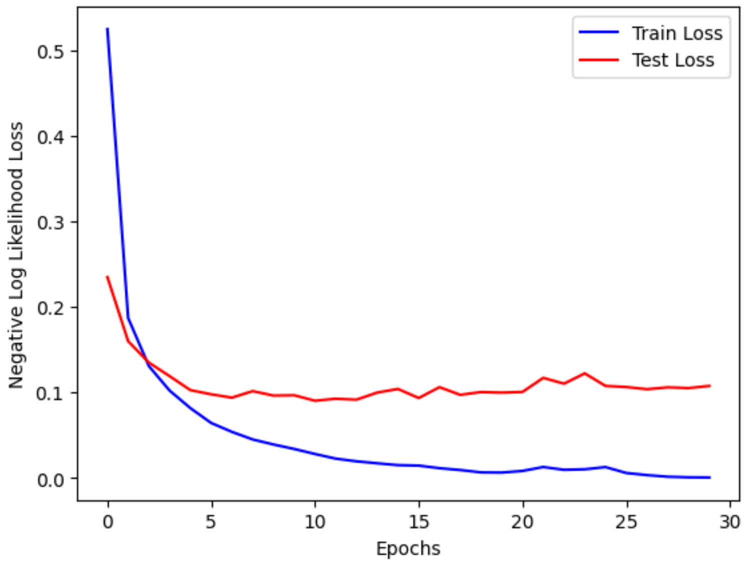
```
#TO-DO:Train the fc_net here
print('FC Net training started')
history_fc = {'train_loss':[],'val_loss':[]}
start_time = datetime.datetime.now()

for epoch in range(num_epochs):
    ### Training

    train_loss = train_model(
        model=fc_net,
        train_loader=train_loader,
        device=device,
        loss_fn=loss_fn,
        optimizer=optim_fc,
        input_dim=(-1,1,28,28))
    ### Validation  (use the testing function)
    val_loss = test_model(
        model=fc_net,
        test_loader=test_loader,
        device=device,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
    # Print Losses
    print(f'Epoch {epoch+1}/{num_epochs} : train loss {train_loss:.3f} \t val loss {val_loss:.3f}')
    history_fc['train_loss'].append(train_loss)
    history_fc['val_loss'].append(val_loss)


print(f'FC Net training done in {(datetime.datetime.now()-start_time).total_seconds():.3f} seconds!')
```

```
    FC Net training started
    /usr/local/lib/python3.10/dist-packages/torch/nn/modules/container.py:217: UserWarning: Implicit dimension choice for log_softmax has beer
      input = module(input)
    Epoch 1/30 : train loss 0.524    val loss 0.234
    Epoch 2/30 : train loss 0.187    val loss 0.160
    Epoch 3/30 : train loss 0.130    val loss 0.135
    Epoch 4/30 : train loss 0.102    val loss 0.119
    Epoch 5/30 : train loss 0.081    val loss 0.102
    Epoch 6/30 : train loss 0.064    val loss 0.098
    Epoch 7/30 : train loss 0.054    val loss 0.094
    Epoch 8/30 : train loss 0.045    val loss 0.101
    Epoch 9/30 : train loss 0.039    val loss 0.096
    Epoch 10/30 : train loss 0.034   val loss 0.096
    Epoch 11/30 : train loss 0.028   val loss 0.090
    Epoch 12/30 : train loss 0.022   val loss 0.092
    Epoch 13/30 : train loss 0.019   val loss 0.091
    Epoch 14/30 : train loss 0.017   val loss 0.100
    Epoch 15/30 : train loss 0.015   val loss 0.104
    Epoch 16/30 : train loss 0.014   val loss 0.093
    Epoch 17/30 : train loss 0.011   val loss 0.106
    Epoch 18/30 : train loss 0.009   val loss 0.097
    Epoch 19/30 : train loss 0.006   val loss 0.100
    Epoch 20/30 : train loss 0.006   val loss 0.100
    Epoch 21/30 : train loss 0.008   val loss 0.100
    Epoch 22/30 : train loss 0.013   val loss 0.117
    Epoch 23/30 : train loss 0.009   val loss 0.110
    Epoch 24/30 : train loss 0.010   val loss 0.122
    Epoch 25/30 : train loss 0.013   val loss 0.107
    Epoch 26/30 : train loss 0.006   val loss 0.106
    Epoch 27/30 : train loss 0.003   val loss 0.104
    Epoch 28/30 : train loss 0.001   val loss 0.106
    Epoch 29/30 : train loss 0.001   val loss 0.105
    Epoch 30/30 : train loss 0.000   val loss 0.107
    FC Net training done in 211.915 seconds!
```

Visualizing Training Progress of FC Net (Check out your wandb.ai project webpage)

```
# TODO - Visualize the training progress of fc_net
fig = plt.figure()
plt.plot(history_fc['train_loss'], color='blue')
plt.plot(history_fc['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

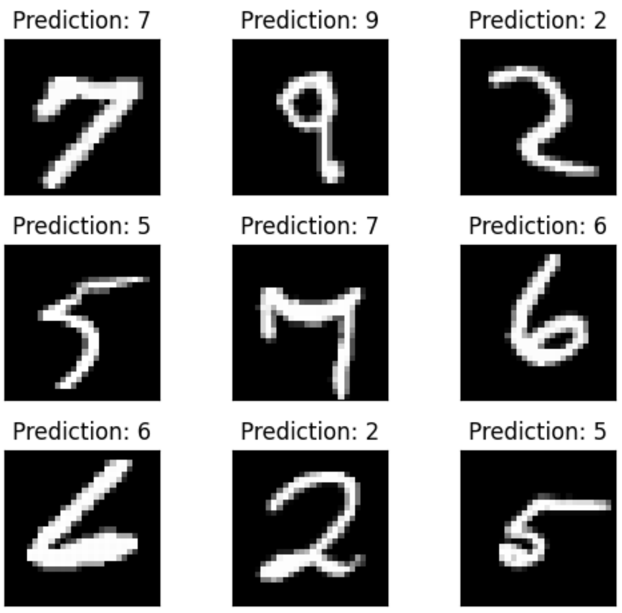
```
    Text(0, 0.5, 'Negative Log Likelihood Loss')
```



Visualizing Predictions of FC Net

```
# TODO - Visualise the predictions of fc_net
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
with torch.no_grad():
    example_data = example_data.to(device)
    output = fc_net(example_data)
example_data = example_data.cpu().detach().numpy()
fig = plt.figure(figsize=(5,5))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray',interpolation='none')
    plt.title("Prediction: {}".format(
    output.data.max(1, keepdim=True)[1][i].item()))
    plt.xticks([])
    plt.yticks([])
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/container.py:217: UserWarning: Implicit dimension choice for log_softmax has been
  input = module(input)
```



Exercise 3.1.7: What are the training times for each of the model? Did both the models take similar times? If yes, why? Shouldn't CNN train faster given it's number of weights to train?

Conv net took about 217 seconds to finsih training, while it only took 212 seconds to train the fully connected network. Both models took similar times. I think this could be caused by the fact that training time is influenced by a couple of factors and not only dependent on the parameter size. Other hyperparameters such as learning rate and hardware capability could be major factors that influence training time. In this case, we happen to set them the same for both models.

```
#Please type your answer here ...
```

Let's see how the models perform under translation

In principle, one of the advantages of convolutions is that they are equivariant under translation which means that a function composed out of convolutions should invariant under translation.

Exercise 3.1.8: In practice, however, we might not see perfect invariance under translation. What aspect of our network leads to imperfect invariance?

This is due to the sliding window operation of the convolution which processes local patches of the input image using the same kernel. Therefore, if a pattern is moved to another location in the image, the same kernel might not produce an activation for the pattern in its new position.

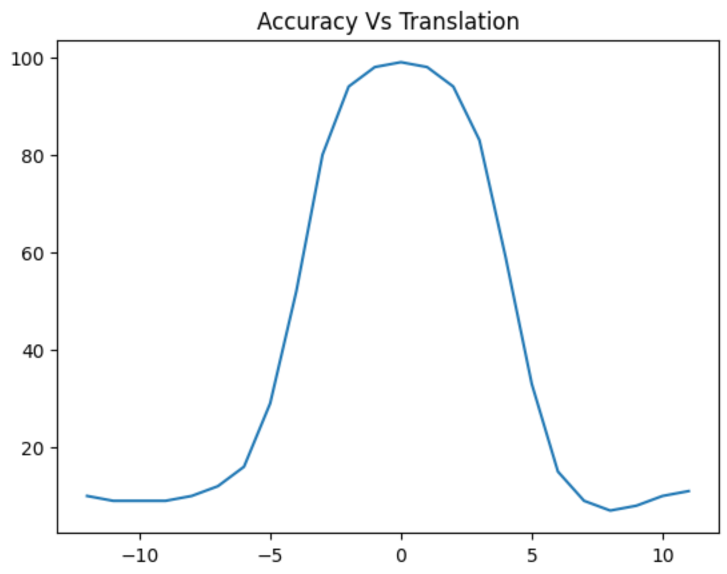
We will next measure the sensitivity of the convolutional network to translation in practice, and we will compare it to the fully-connected version.

```
## function to check accuracies for unit translation
def shiftVsAccuracy(model, test_loader, device, loss_fn, shifts = 12, input_dim=(-1,1,28,28)):
    # Set evaluation mode for encoder and decoder
    accuracies = []
    shifted = []
    for i in range(-shifts,shifts):
        model.eval()
        correct = 0
        total = 0
        with torch.no_grad(): # No need to track the gradients
            # Define the lists to store the outputs for each batch
            predicted = []
            actual = []
            for images, labels in test_loader:
                # reshape input
                images = torch.roll(images,shifts=i, dims=2)
                if i == 0:
                    pass
                elif i > 0:
                    images[:,:,:i,:] = 0
                else:
                    images[:,:,i:,:] = 0
                images = torch.reshape(images,input_dim)
                images = images.to(device)
                labels = labels.to(device)
                ## predict the label
                pred = model(images)
                # Append the network output and the original image to the lists
                _ , pred = torch.max(pred.data, 1)
                total += labels.size(0)
                correct += (pred == labels).sum().item()
                predicted.append(pred.cpu())
                actual.append(labels.cpu())
            shifted.append(images[0][0].cpu())
            acc = 100 * correct // total
            accuracies.append(acc)
    return accuracies,shifted
```

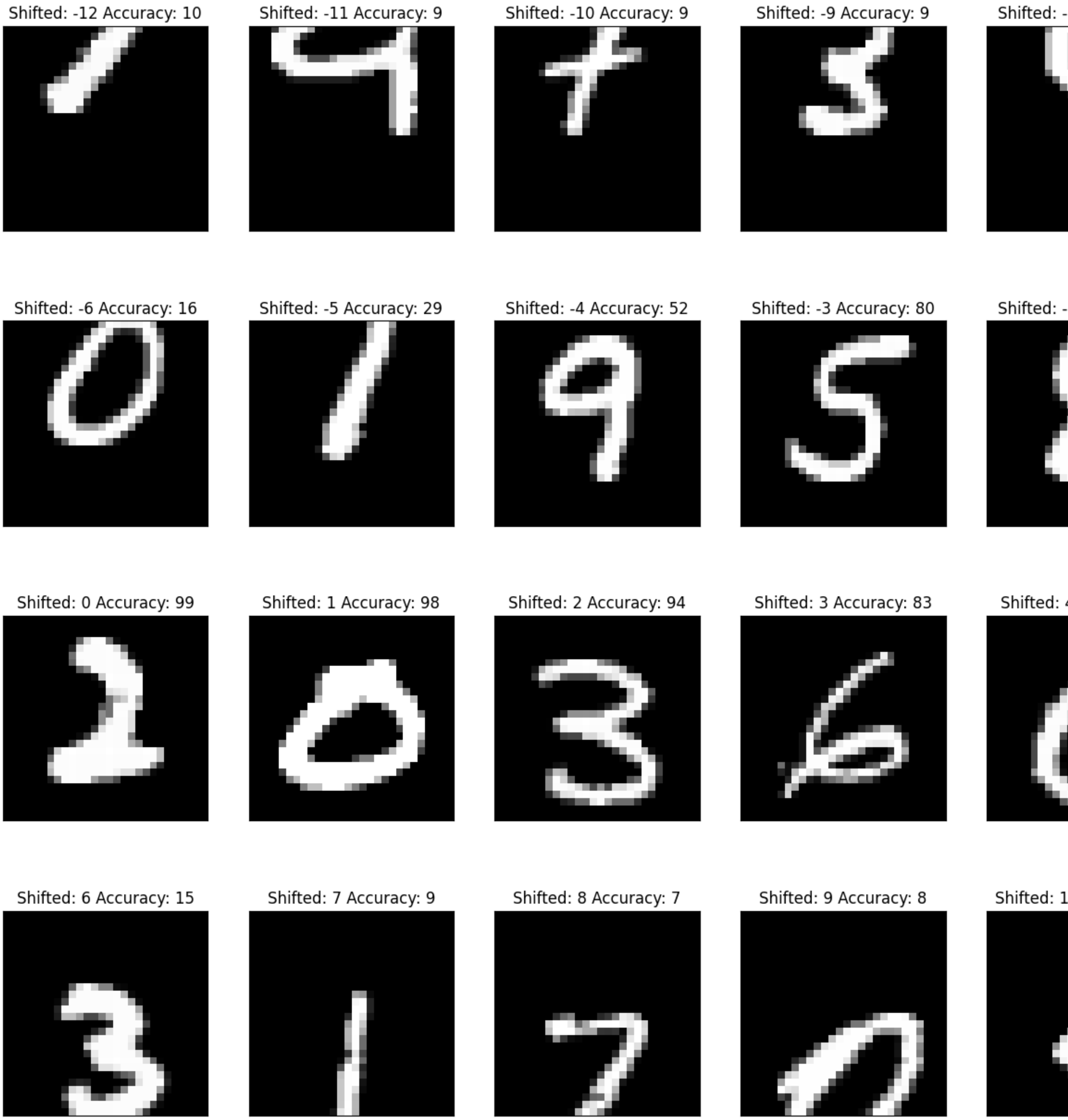
```
accuracies,shifted = shiftVsAccuracy(
        model=conv_net,
        test_loader=test_loader,
        device=device,
        shifts=12,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
```

```
shifts = np.arange(-12,12)
plt.plot(shifts,accuracies)
plt.title('Accuracy Vs Translation')
```

```
Text(0.5, 1.0, 'Accuracy Vs Translation')
```



```
fig = plt.figure(figsize=(20,20))
plt_num = 0
for i in range(-12,12):
    plt.subplot(5,6,plt_num+1)
    plt.imshow(shifted[plt_num], cmap='gray',interpolation='none')
    plt.title(f"Shifted: {i} Accuracy: {accuracies[plt_num]}")
    plt.xticks([])
    plt.yticks([])
    plt_num+=1
```



Exercise 3.1.8: Do the same for FC-Net and plot the accuracies. Is the rate of accuracy degradation same as Conv-Net? Can you justify why this happened?
Clue: You might want to look at the way convolution layers process information

The fc net's rate of accuracy degradation with translation is not the same as the Conv-Net. As we can see from the two graphies below, the accuracy for this model is highest when there's no translation, indicating that the model performs best when the test images are not shifted. The model lost the ability to track particular patterns of each number once shifting happens. Likewise, FC net follow similar trend, while it has relatively higher accuracy ar aounrd -10 mark. It does not have a smooth decline in accuracy with translation. There are noticeable dips in its accuracy.

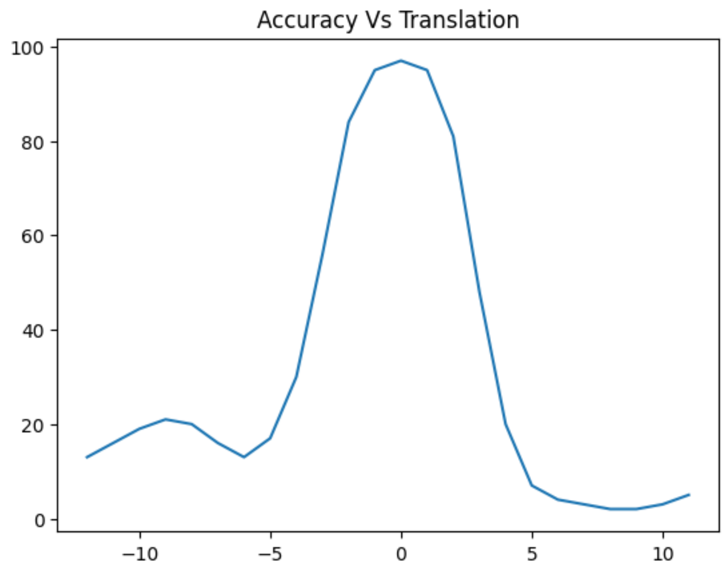
I think one possible reason causing such erratic behavior of a fc net after translation is that its inability to handle and process local spatial patterns, while conv net can. Fully connected layers treat each pixel of the image independently. Thus, even minor translations can lead to a significant drop in accuracy since the spatial relationships between pixels, crucial for recognition, get disrupted. It's also why we have a smoother line for conv net, as they can better process local pattern infomation.

```
# To-DO Write your code below
fc_accuracies,fc_shifted = shiftVsAccuracy(
        model=fc_net,
        test_loader=test_loader,
        device=device,
        shifts=12,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
```

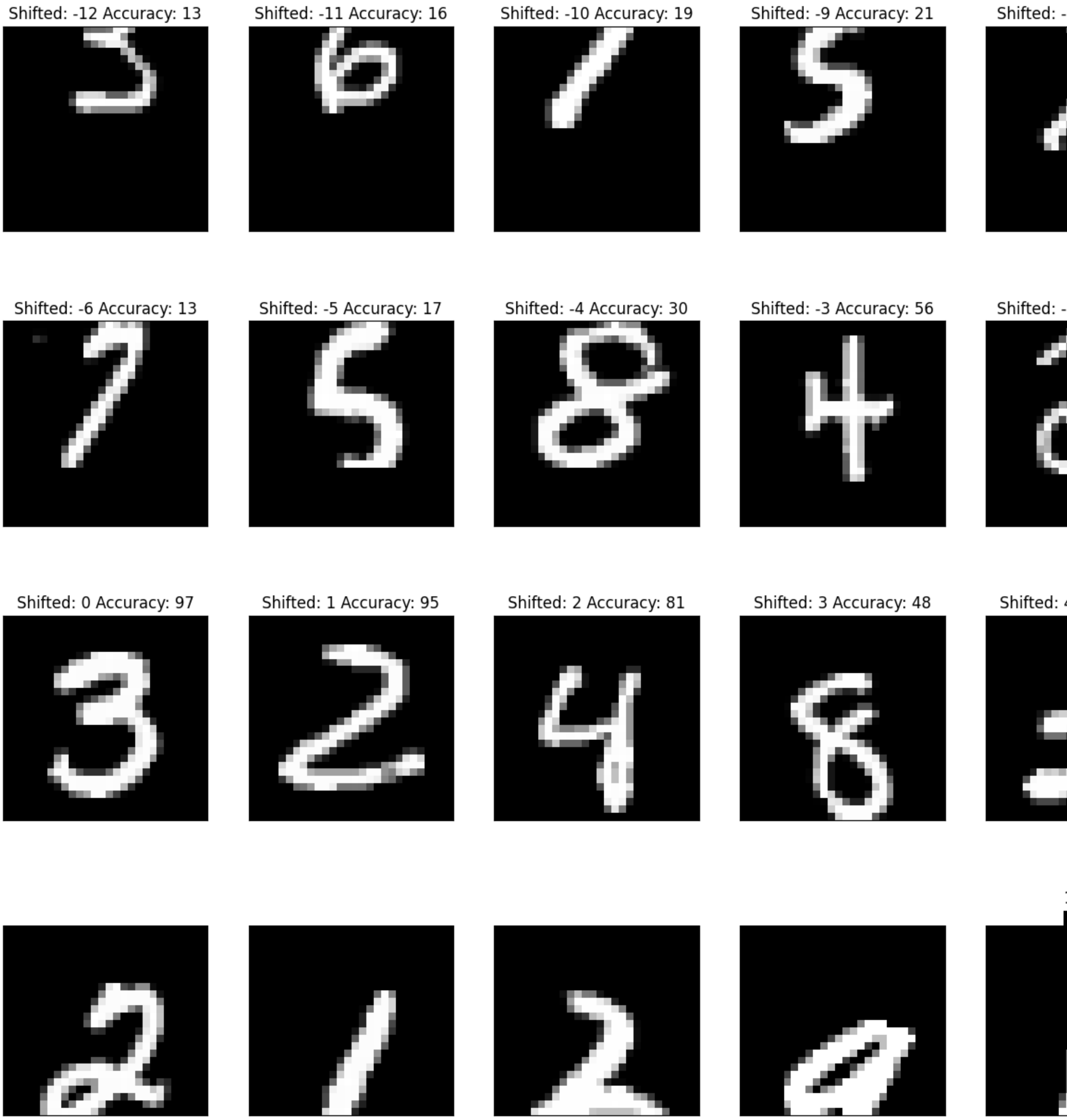
```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/container.py:217: UserWarning: Implicit dimension choice for log_softmax has been
  input = module(input)
```

```
shifts = np.arange(-12,12)
plt.plot(shifts,fc_accuracies)
plt.title('Accuracy Vs Translation')
```

```
Text(0.5, 1.0, 'Accuracy Vs Translation')
```



```
fig = plt.figure(figsize=(20,20))
plt_num = 0
for i in range(-12,12):
    plt.subplot(5,6,plt_num+1)
    plt.imshow(fc_shifted[plt_num], cmap='gray',interpolation='none')
    plt.title(f"Shifted: {i} Accuracy: {fc_accuracies[plt_num]}")
    plt.xticks([])
    plt.yticks([])
    plt_num+=1
```


Shifted: -12 Accuracy: 13
Shifted: -11 Accuracy: 16
Shifted: -10 Accuracy: 19
Shifted: -9 Accuracy: 21
Shifted: -
Shifted: -6 Accuracy: 13
Shifted: -5 Accuracy: 17
Shifted: -4 Accuracy: 30
Shifted: -3 Accuracy: 56
Shifted: -
Shifted: 0 Accuracy: 97
Shifted: 1 Accuracy: 95
Shifted: 2 Accuracy: 81
Shifted: 3 Accuracy: 48
Shifted: 4