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
```

```
Collecting wandb
  Downloading wandb-0.15.12-py3-none-any.whl (2.1 MB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.1/2.1 MB 11.4 MB/s eta
0:00:00
ent already satisfied: Click!=8.0.0,>=7.1 in
/usr/local/lib/python3.10/dist-packages (from wandb) (8.1.7)
Collecting GitPython!=3.1.29,>=1.0.0 (from wandb)
  Downloading GitPython-3.1.40-py3-none-any.whl (190 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 190.6/190.6 kB 14.1 MB/s eta
0:00:00
ent already satisfied: requests<3,>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (2.31.0)
Requirement already satisfied: psutil>=5.0.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (5.9.5)
Collecting sentry-sdk>=1.0.0 (from wandb)
  Downloading sentry_sdk-1.32.0-py2.py3-none-any.whl (240 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 241.0/241.0 kB 16.5 MB/s eta
0:00:00
 wandb)
  Downloading docker_pycreds-0.4.0-py2.py3-none-any.whl (9.0 kB)
Requirement already satisfied: PyYAML in
/usr/local/lib/python3.10/dist-packages (from wandb) (6.0.1)
Collecting pathtools (from wandb)
  Downloading pathtools-0.1.2.tar.gz (11 kB)
  Preparing metadata (setup.py) ...  wandb)
  Downloading setproctitle-1.3.3-cp310-cp310-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_17_x86_64.manylinux
2014_x86_64.whl (30 kB)
Requirement already satisfied: setuptools in
/usr/local/lib/python3.10/dist-packages (from wandb) (67.7.2)
Requirement already satisfied: appdirs>=1.4.3 in
/usr/local/lib/python3.10/dist-packages (from wandb) (1.4.4)
```

```
Requirement already satisfied: protobuf!=4.21.0,<5,>=3.19.0 in
/usr/local/lib/python3.10/dist-packages (from wandb) (3.20.3)
Requirement already satisfied: six>=1.4.0 in
/usr/local/lib/python3.10/dist-packages (from docker-pycreds>=0.4.0-
>wandb) (1.16.0)
Collecting gitdb<5,>=4.0.1 (from GitPython!=3.1.29,>=1.0.0->wandb)
  Downloading gitdb-4.0.11-py3-none-any.whl (62 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.7/62.7 kB 8.9 MB/s eta
0:00:00
ent already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0-
>wandb) (3.3.0)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0-
>wandb) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0-
>wandb) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests<3,>=2.0.0-
>wandb) (2023.7.22)
Collecting smmap<6,>=3.0.1 (from gitdb<5,>=4.0.1->GitPython!
=3.1.29,>=1.0.0->wandb)
  Downloading smmap-5.0.1-py3-none-any.whl (24 kB)
Building wheels for collected packages: pathtools
  Building wheel for pathtools (setup.py) ... e=pathtools-0.1.2-py3-
none-any.whl size=8791
sha256=ef05ef85ea53b44e249af37d10399ab549a68ef87184a33ab8c904e37c079f3
0
  Stored in directory:
/root/.cache/pip/wheels/e7/f3/22/152153d6eb222ee7a56ff8617d80ee5207207
a8c00a7aab794
Successfully built pathtools
Installing collected packages: pathtools, smmap, setproctitle, sentry-
sdk, docker-pycreds, gitdb, GitPython, wandb
Successfully installed GitPython-3.1.40 docker-pycreds-0.4.0 gitdb-
4.0.11 pathtools-0.1.2 sentry-sdk-1.32.0 setproctitle-1.3.3 smmap-
5.0.1 wandb-0.15.12
Collecting torchinfo
  Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.8.0
```

```
%%bash

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

```
--2023-10-21 01:31:30--
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... 200 OK
Length: 1078198 (1.0M)
Saving to: ‘mnist-classify.pth’

     0K .......... .......... .......... .......... ..........  4%
616K 2s
    50K .......... .......... .......... .......... ..........  9%
1.26M 1s
   100K .......... .......... .......... .......... .......... 14%
3.87M 1s
   150K .......... .......... .......... .......... .......... 18%
2.00M 1s
   200K .......... .......... .......... .......... .......... 23%
8.98M 1s
   250K .......... .......... .......... .......... .......... 28%
11.7M 0s
   300K .......... .......... .......... .......... .......... 33%
7.22M 0s
   350K .......... .......... .......... .......... .......... 37%
2.24M 0s
   400K .......... .......... .......... .......... .......... 42%
19.2M 0s
   450K .......... .......... .......... .......... .......... 47%
13.6M 0s
   500K .......... .......... .......... .......... .......... 52%
24.7M 0s
   550K .......... .......... .......... .......... .......... 56%
20.5M 0s
   600K .......... .......... .......... .......... .......... 61%
29.7M 0s
   650K .......... .......... .......... .......... .......... 66%
14.5M 0s
   700K .......... .......... .......... .......... .......... 71%
16.1M 0s
   750K .......... .......... .......... .......... .......... 75%
2.38M 0s
   800K .......... .......... .......... .......... .......... 80%
24.4M 0s
   850K .......... .......... .......... .......... .......... 85%
114M 0s
   900K .......... .......... .......... .......... .......... 90%
156M 0s
   950K .......... .......... .......... .......... .......... 94%
16.8M 0s
  1000K .......... .......... .......... .......... .......... 99%
127M 0s
  1050K ..                                                    100%
```

```
5.45T=0.2s

2023-10-21 01:31:30 (4.28 MB/s) - ‘mnist-classify.pth’ saved
[1078198/1078198]


# 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")

<IPython.core.display.Javascript object>

wandb: Logging into wandb.ai. (Learn how to deploy a W&B server
locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here:
https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press
ctrl+c to quit:

 ..........

wandb: Appending key for api.wandb.ai to your netrc file: /root/.netrc

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

```
<wandb.sdk.wandb_run.Run at 0x7d4ec5f4b8b0>
```

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!
```

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

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/
module.py:1518: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)

======================================================================
====================
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?

Observation:

Increasing depth i.e. adding layers is directly proportional to increase in the number of total parameters. No, adding layers does not reduce parameters. I have also tried adding dropout layers and normalization layers like batch normalization, layer normalization and group normalization but I observed that it doesn't reduces the parameters but increases the parameters slightly. Increasing width i.e. number of hidden neurons in the layers is proportional to the total trainable parameters.

```python
import torch.nn as nn

def extended_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, 144),
        nn.ReLU(),
        nn.Linear(144, 64),
        nn.ReLU(),
        nn.Linear(64, 10),
        nn.ReLU(),
        nn.LogSoftmax()
    )
    return model

fc_net_1 = extended_fc_net()

view_network_parameters(fc_net_1)
```

```
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: 41472 elements
9.bias: 144 elements
11.weight: 9216 elements
11.bias: 64 elements
13.weight: 640 elements
13.bias: 10 elements

Total Trainable Parameters: 30017882!
```

```python
from torchinfo import summary
summary(fc_net_1, input_size=(1, 1, 28, 28))
```

```
======================================================================
====================
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, 144]
41,616
├─ReLU: 1-11                             [1, 144]                  --
├─Linear: 1-12                           [1, 64]
9,280
├─ReLU: 1-13                             [1, 64]                   --
├─Linear: 1-14                           [1, 10]                   650
├─ReLU: 1-15                             [1, 10]                   --
├─LogSoftmax: 1-16                       [1, 10]                   --
======================================================================
```

```
====================
Total params: 30,017,882
Trainable params: 30,017,882
Non-trainable params: 0
Total mult-adds (M): 30.02
======================================================================
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 120.07
Estimated Total Size (MB): 120.17
======================================================================
====================
```

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.Conv2d(8,16, kernel_size=3, padding = 1, stride = 2),
        nn.ReLU(),
        nn.Conv2d(16, 32, kernel_size=3, padding = 1, stride = 2),
        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, 64),
        # Do not change the code below
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax())
    return model

conv_net = simple_conv_net()

view_network_parameters(conv_net)
```

```
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: 100352 elements
7.bias: 64 elements
9.weight: 640 elements
9.bias: 10 elements

Total Trainable Parameters: 106954!
```

```
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]            --
├─Conv2d: 1-3                            [1, 16, 14, 14]
1,168
├─ReLU: 1-4                              [1, 16, 14, 14]           --
├─Conv2d: 1-5                            [1, 32, 7, 7]
4,640
├─ReLU: 1-6                              [1, 32, 7, 7]             --
├─Flatten: 1-7                           [1, 1568]                 --
├─Linear: 1-8                            [1, 64]
100,416
├─ReLU: 1-9                              [1, 64]                   --
├─Linear: 1-10                           [1, 10]                   650
├─LogSoftmax: 1-11                       [1, 10]                   --
========================================================================
====================
Total params: 106,954
Trainable params: 106,954
Non-trainable params: 0
Total mult-adds (M): 0.62
========================================================================
====================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 0.43
Estimated Total Size (MB): 0.52
========================================================================
====================
```

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

Explanation:

The final layer of a neural network is a log softmax function because it converts the raw outputs of the network into probabilities that can be interpreted as the likelihood of each class. This makes it easier to interpret the output of the network and to make decisions based on that output.

The softmax function is a mathematical function that takes a vector of real numbers as input and returns a vector of real numbers as output. The output values are all positive and sum to 1. The softmax function is calculated as follows:

softmax(x) = $ \frac{e^{x}}{\sum{e^{x}}} $

We cannot use a ReLU function instead of a softmax function in the final layer because the ReLU function does not output probabilities. The ReLU function simply outputs the input value if it is positive and 0 otherwise. This is not suitable for classification tasks, where we need to be able to output the probability of each class.

If we were to use a ReLU function in the final layer of a neural network, we would need to use a different loss function, such as the cross-entropy loss function. The cross-entropy loss function is a loss function that is specifically designed for classification tasks.

However, it is important to note that the cross-entropy loss function is not as numerically stable as the log softmax function. This means that it can be more difficult to train neural networks that use the cross-entropy loss function.

In general, it is best to use a log softmax function in the final layer of a neural network for classification tasks. This is because the log softmax function outputs probabilities and is numerically stable.

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}} = \frac{106,954}{29,985,482} = 0.00356685945 $ Do you see the difference ?!

Yes, I see the difference as CNNs are designed to leverage weight sharing and local connectivity to handle image data efficiently with a relatively small number of parameters compared to fully connected networks. This reduced parameter count is one of the reasons why CNNs are effective for image classification tasks, as they can capture local patterns in images without the need for massive parameterization.

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.

Observation:

We can observe that when we change the hidden_fc_dim that is the dimension shape it affects the change in total parameters. We see that adding more layers and increasing the hidden neuron count in the layers will lead to a higher number of trainable parameters, potentially increasing the model's capacity to capture complex patterns in the data.

```
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=[256, 128, 64])
view_network_parameters(fc_net_custom2)

fc_net_custom3 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, 
```

```
hidden_fc_dim=[64, 32, 16])
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: 200704 elements
linear_1.bias: 256 elements
linear_2.weight: 32768 elements
linear_2.bias: 128 elements
linear_3.weight: 8192 elements
linear_3.bias: 64 elements
linear_4.weight: 640 elements
linear_4.bias: 10 elements

Total Trainable Parameters: 242762!
Model Summary

linear_1.weight: 50176 elements
linear_1.bias: 64 elements
linear_2.weight: 2048 elements
linear_2.bias: 32 elements
linear_3.weight: 512 elements
linear_3.bias: 16 elements
linear_4.weight: 160 elements
linear_4.bias: 10 elements

Total Trainable Parameters: 53018!
```

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.ToTen
```

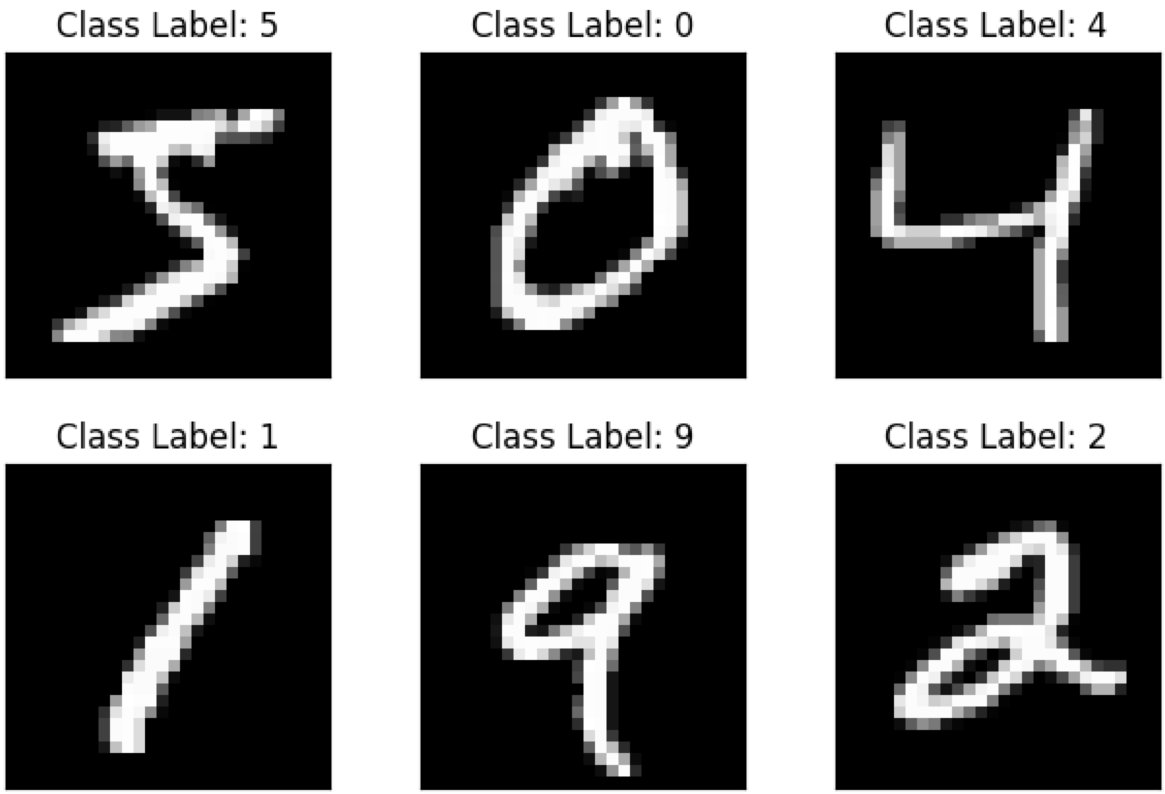
```
sor()]))
test_data  = torchvision.datasets.MNIST(data_dir, train=False,
download=True,
transform=torchvision.transforms.Compose([torchvision.transforms.ToTen
sor()]))
print(f'Datatype of the dataset object: {type(train_data)}')
# check the length of dataset
n_train_samples = len(train_data)
print(f'Number 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)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz to data/MNIST/raw/train-images-idx3-ubyte.gz

100%|██████████| 9912422/9912422 [00:00<00:00, 173501056.82it/s]

Extracting data/MNIST/raw/train-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz to data/MNIST/raw/train-labels-idx1-ubyte.gz

100%|██████████| 28881/28881 [00:00<00:00, 20576812.27it/s]

Extracting data/MNIST/raw/train-labels-idx1-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
to data/MNIST/raw/t10k-images-idx3-ubyte.gz


100%|██████████| 1648877/1648877 [00:00<00:00, 45563734.21it/s]
```

```
Extracting data/MNIST/raw/t10k-images-idx3-ubyte.gz to data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
to data/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|██████████| 4542/4542 [00:00<00:00, 3902997.08it/s]

Extracting data/MNIST/raw/t10k-labels-idx1-ubyte.gz to data/MNIST/raw

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

```python
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
```

```python
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):
    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

```python
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
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/
module.py:1518: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  return self._call_impl(*args, **kwargs)
```

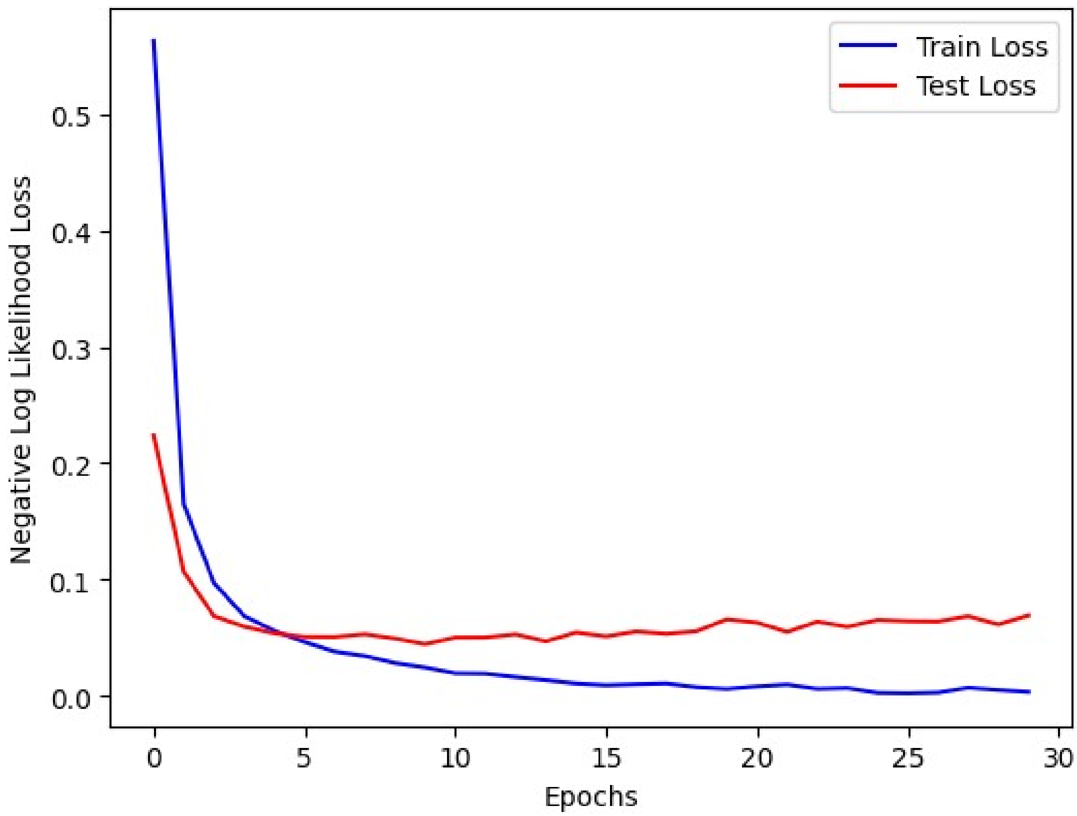
```
Epoch 1/30 : train loss 0.563     val loss 0.224
Epoch 2/30 : train loss 0.165     val loss 0.107
Epoch 3/30 : train loss 0.097     val loss 0.069
Epoch 4/30 : train loss 0.069     val loss 0.060
```

```
Epoch 5/30 : train loss 0.056     val loss 0.054
Epoch 6/30 : train loss 0.046     val loss 0.051
Epoch 7/30 : train loss 0.038     val loss 0.051
Epoch 8/30 : train loss 0.034     val loss 0.053
Epoch 9/30 : train loss 0.028     val loss 0.049
Epoch 10/30 : train loss 0.025    val loss 0.045
Epoch 11/30 : train loss 0.019    val loss 0.050
Epoch 12/30 : train loss 0.019    val loss 0.050
Epoch 13/30 : train loss 0.016    val loss 0.053
Epoch 14/30 : train loss 0.014    val loss 0.047
Epoch 15/30 : train loss 0.011    val loss 0.055
Epoch 16/30 : train loss 0.009    val loss 0.051
Epoch 17/30 : train loss 0.010    val loss 0.055
Epoch 18/30 : train loss 0.011    val loss 0.053
Epoch 19/30 : train loss 0.008    val loss 0.056
Epoch 20/30 : train loss 0.006    val loss 0.066
Epoch 21/30 : train loss 0.008    val loss 0.063
Epoch 22/30 : train loss 0.010    val loss 0.055
Epoch 23/30 : train loss 0.006    val loss 0.064
Epoch 24/30 : train loss 0.007    val loss 0.059
Epoch 25/30 : train loss 0.003    val loss 0.065
Epoch 26/30 : train loss 0.002    val loss 0.064
Epoch 27/30 : train loss 0.003    val loss 0.064
Epoch 28/30 : train loss 0.007    val loss 0.069
Epoch 29/30 : train loss 0.005    val loss 0.062
Epoch 30/30 : train loss 0.003    val loss 0.069
Conv Net training done in 437.097 seconds!
```

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

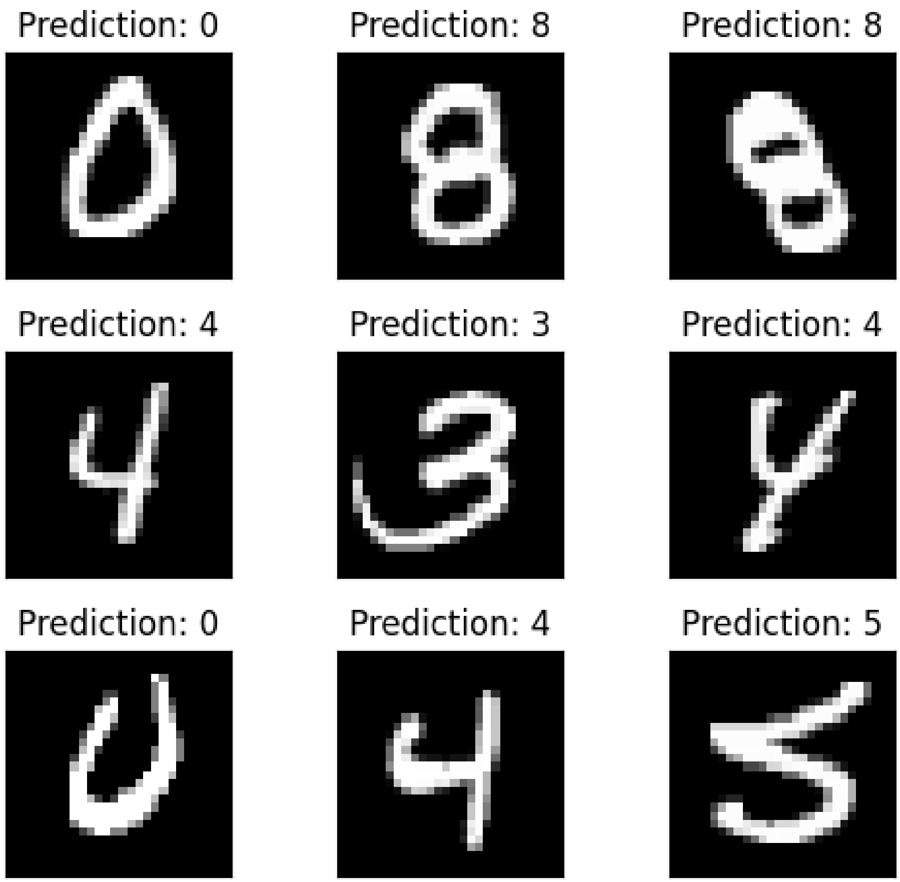
```
fig = plt.figure()
plt.plot(history_conv['train_loss'], color='blue')
plt.plot(history_conv['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

```
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(
```

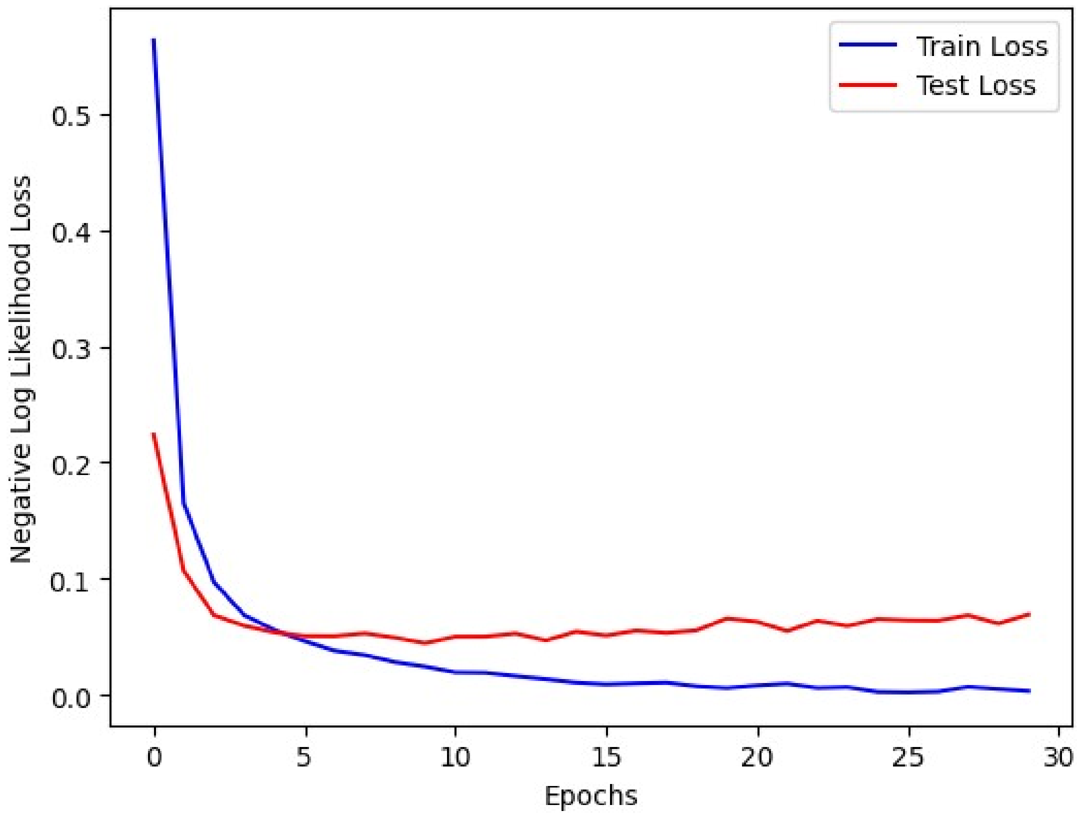
```python
        model=fc_net,
        test_loader=test_loader,
        device=device,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
    # Print Losses
    print(f'Epoch {epoch+1}/{num_epochs} : train loss {train_loss:.3f}
\t val loss {val_loss:.3f}')
    history_fc['train_loss'].append(train_loss)
    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
Epoch 1/30 : train loss 0.495    val loss 0.224
Epoch 2/30 : train loss 0.183    val loss 0.152
Epoch 3/30 : train loss 0.130    val loss 0.123
Epoch 4/30 : train loss 0.098    val loss 0.116
Epoch 5/30 : train loss 0.079    val loss 0.105
Epoch 6/30 : train loss 0.065    val loss 0.099
Epoch 7/30 : train loss 0.056    val loss 0.095
Epoch 8/30 : train loss 0.046    val loss 0.086
Epoch 9/30 : train loss 0.038    val loss 0.086
Epoch 10/30 : train loss 0.032   val loss 0.088
Epoch 11/30 : train loss 0.030   val loss 0.098
Epoch 12/30 : train loss 0.025   val loss 0.087
Epoch 13/30 : train loss 0.021   val loss 0.090
Epoch 14/30 : train loss 0.017   val loss 0.083
Epoch 15/30 : train loss 0.015   val loss 0.094
Epoch 16/30 : train loss 0.012   val loss 0.088
Epoch 17/30 : train loss 0.011   val loss 0.103
Epoch 18/30 : train loss 0.010   val loss 0.093
Epoch 19/30 : train loss 0.010   val loss 0.106
Epoch 20/30 : train loss 0.009   val loss 0.097
Epoch 21/30 : train loss 0.008   val loss 0.107
Epoch 22/30 : train loss 0.014   val loss 0.103
Epoch 23/30 : train loss 0.006   val loss 0.099
Epoch 24/30 : train loss 0.004   val loss 0.098
Epoch 25/30 : train loss 0.002   val loss 0.101
Epoch 26/30 : train loss 0.001   val loss 0.097
Epoch 27/30 : train loss 0.001   val loss 0.100
Epoch 28/30 : train loss 0.001   val loss 0.099
Epoch 29/30 : train loss 0.000   val loss 0.099
Epoch 30/30 : train loss 0.000   val loss 0.100
FC Net training done in 392.673 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_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')
```

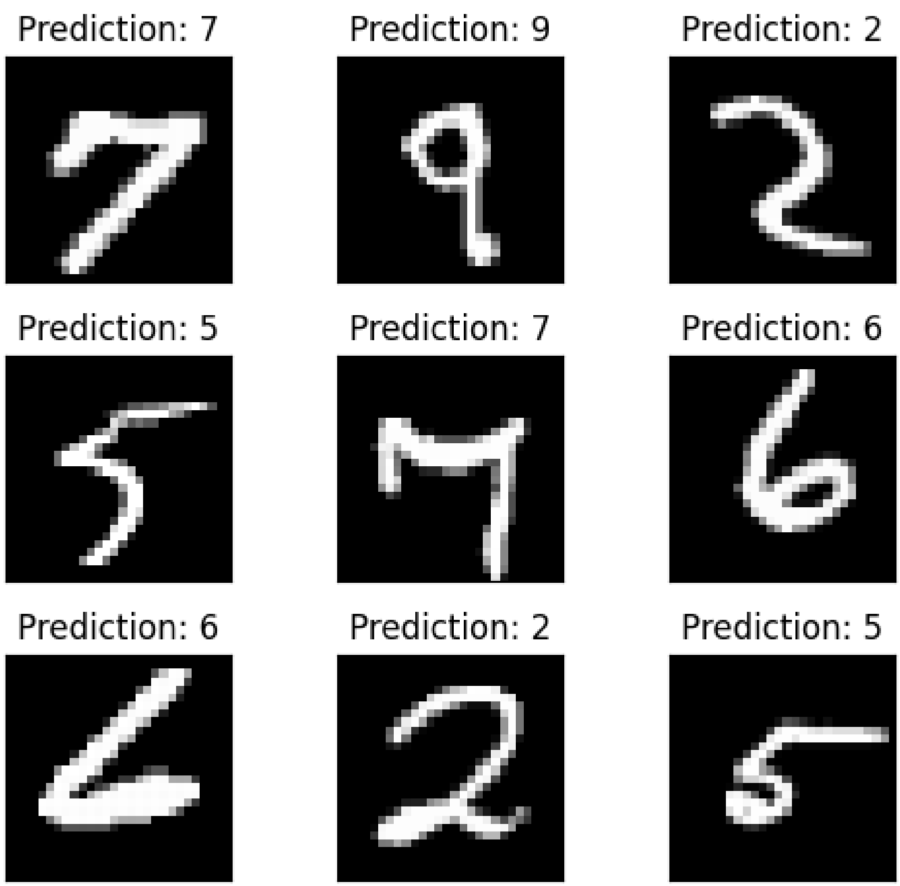
```
Text(0, 0.5, 'Negative Log Likelihood Loss')
```



Visualizing Predictions of 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([])
```



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 437.097 seconds to run.

Fc_net took 392.673 seconds to run.

Yes, the two models took similar time(1 minute difference) to train, even though the CNN has more weights to train. This is because the CNN is able to take advantage of the spatial structure of the input images, which can lead to more efficient computation. CNN model are also more complex as compared to FCN.

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?

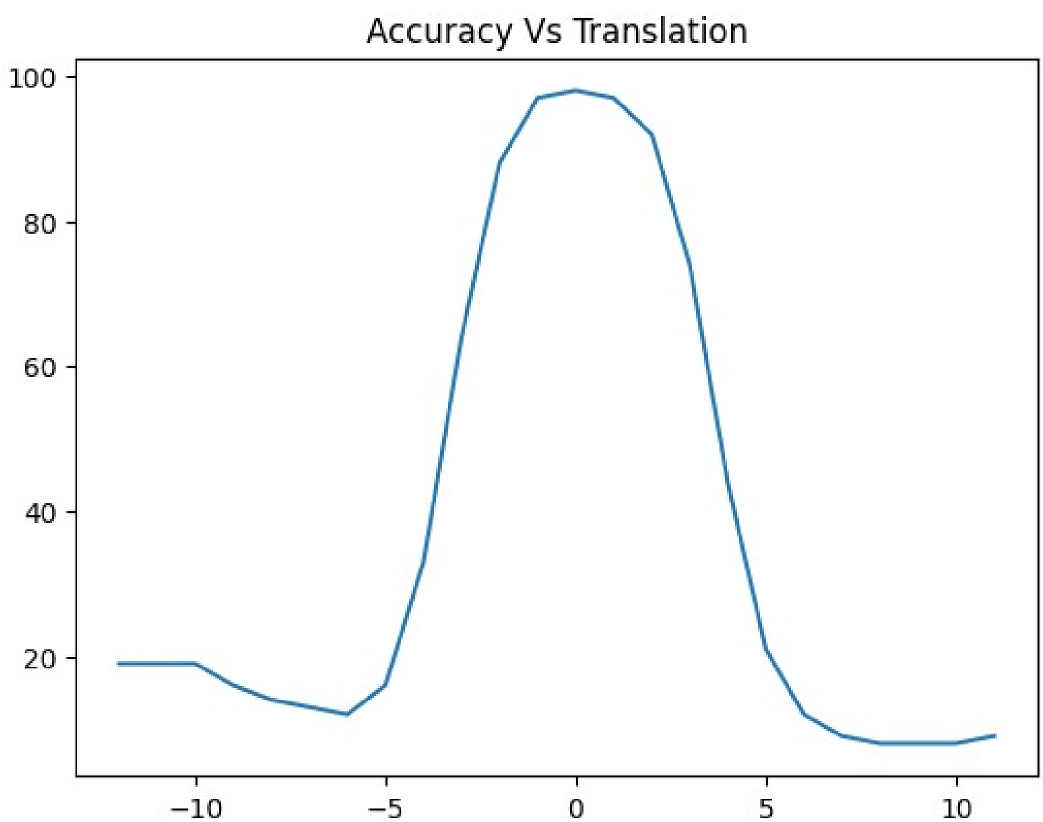
Answer:

Convolutional neural networks may not be perfectly invariant under translation due to pooling layers, non-linear activation functions, and the training data. To improve translation invariance, we can use strided convolutions instead of pooling layers, use batch normalization, and use data augmentation techniques.

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

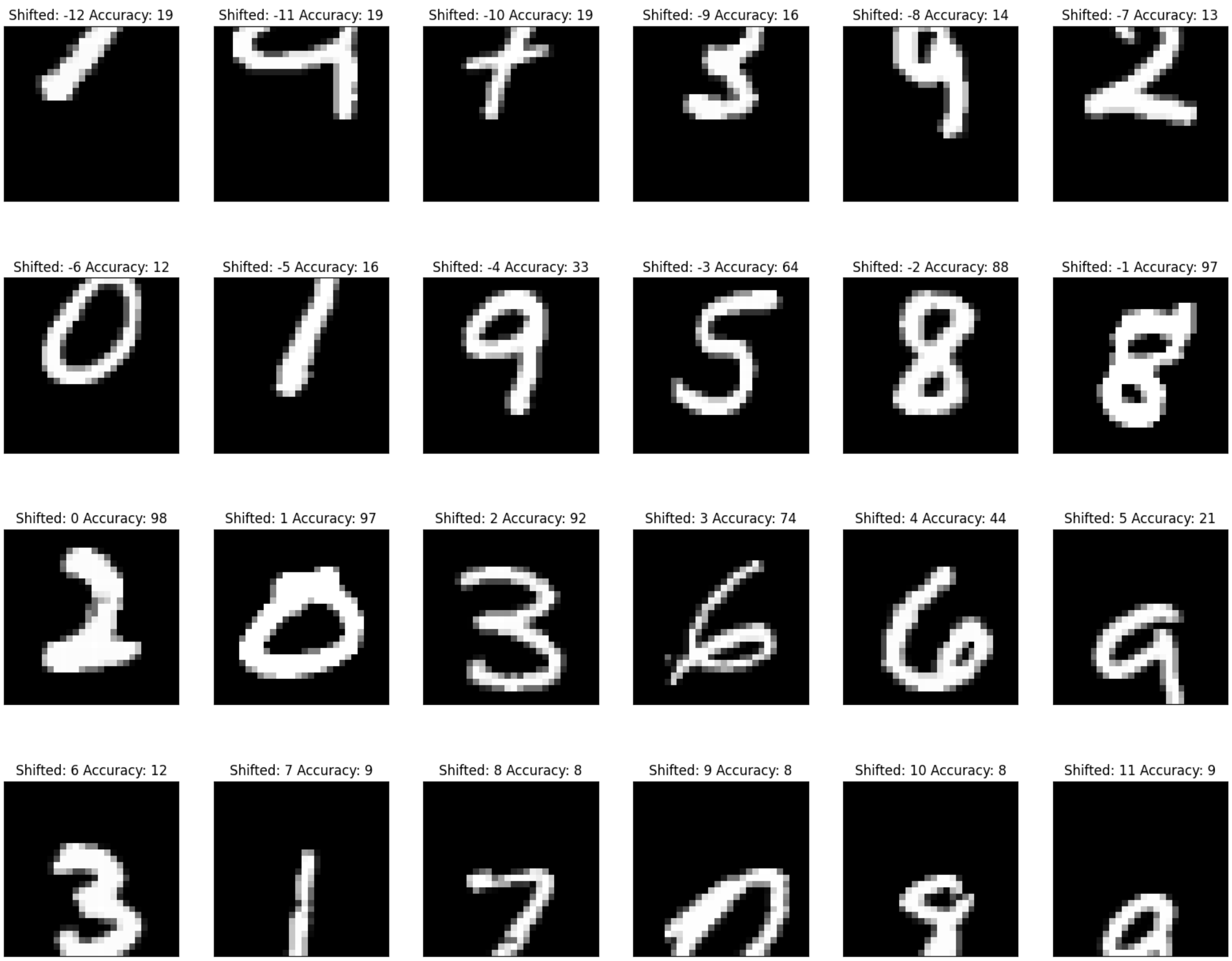
```
## function to check accuracies for unit translation
def shiftVsAccuracy(model, test_loader, device, loss_fn, shifts = 12,
input_dim=(-1,1,28,28)):
    # Set evaluation mode for encoder and decoder
    accuracies = []
    shifted = []
    for i in range(-shifts,shifts):
        model.eval()
        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
```

```python
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
```



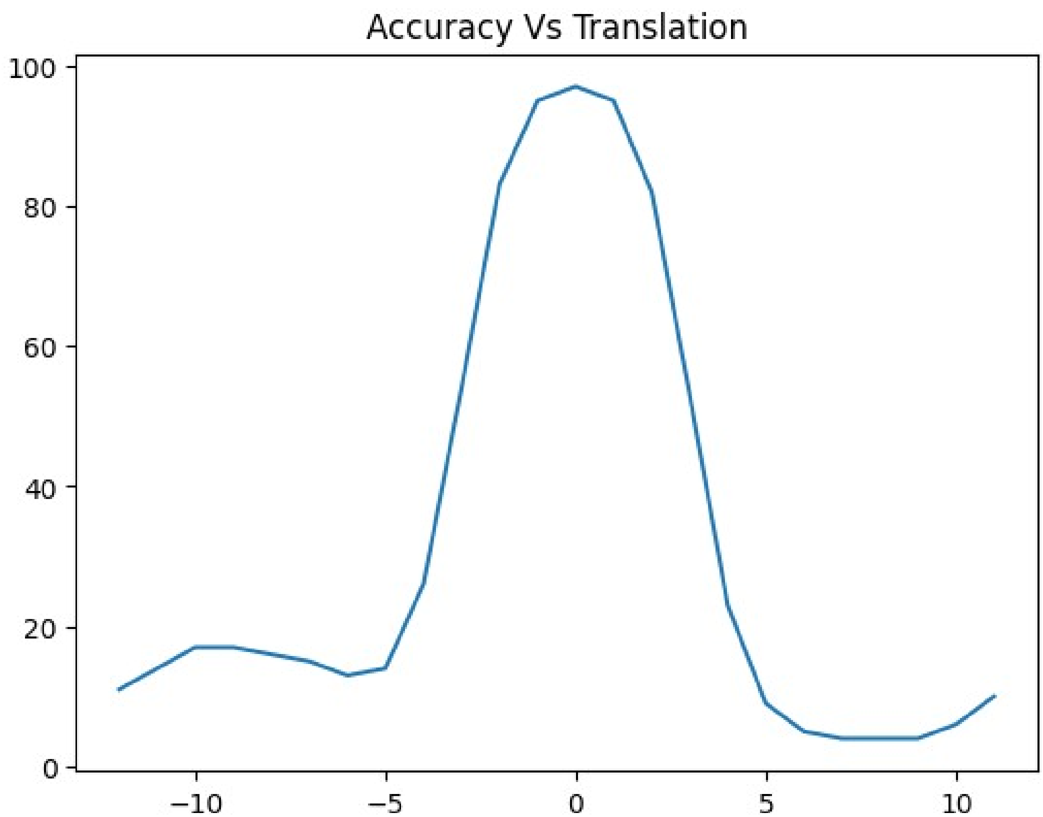
Exercise 3.1.8: Do the same for FC-Net and plot the accuracies. Is the rate of accuracy degradation same as Conv-Net? Can you justify why this happened? Clue: You might want to look at the way convolution layers process information

Answer:

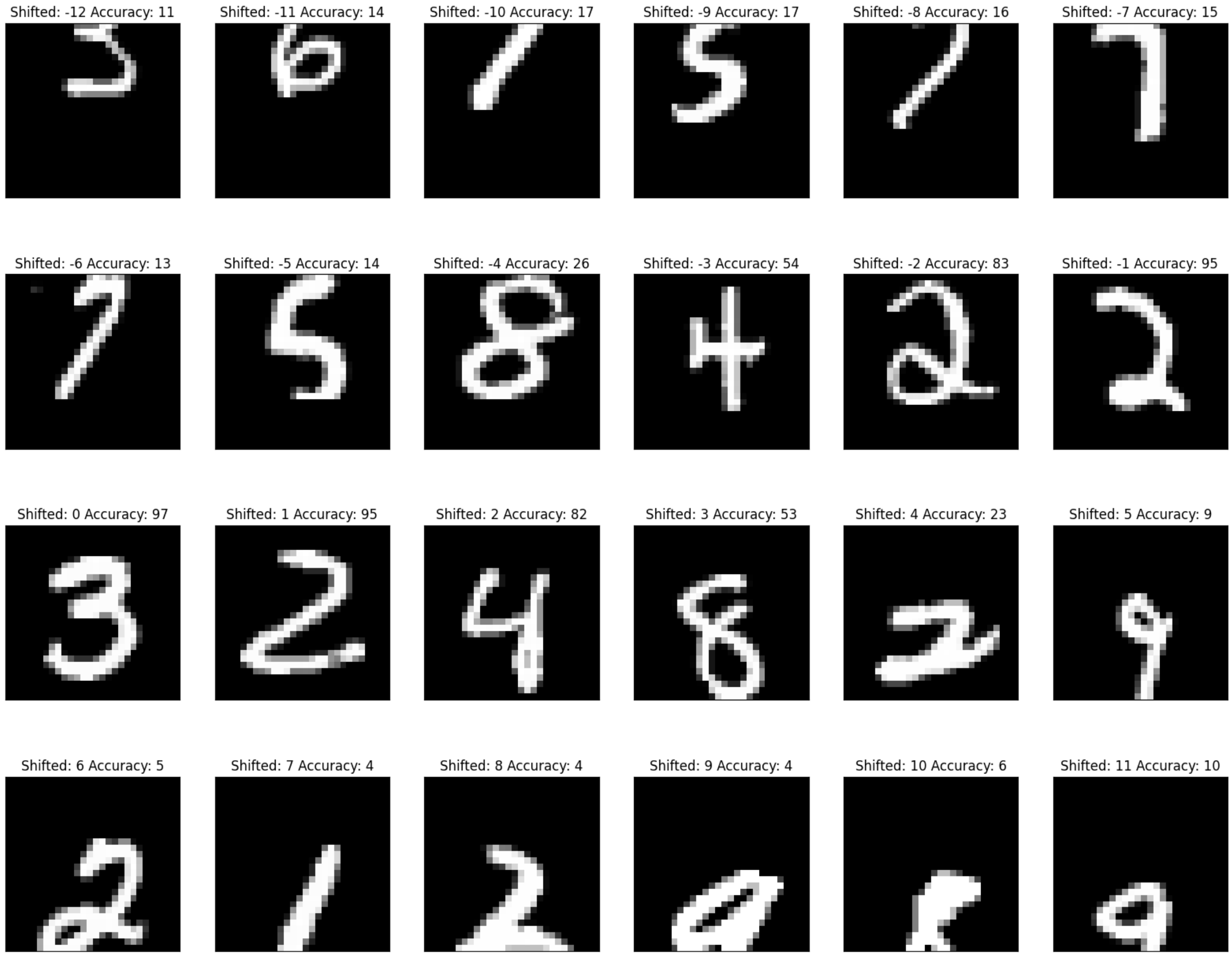
Yes, the plots look similar. The similar rate of accuracy degradation between the FC-Net and Conv-Net can be attributed to the way convolution layers process information. Convolutional layers in Conv-Nets capture local spatial relationships and exhibit a degree of translation invariance through shared weights. In contrast, FC-Nets, with fully connected layers, lack this inherent translation invariance and are more sensitive to spatial variations, resulting in a comparable loss of accuracy when dealing with translations.

```
accuracies,shifted = shiftVsAccuracy(
        model=fc_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
```



Shifted: -12 Accuracy: 11
Shifted: -11 Accuracy: 14
Shifted: -10 Accuracy: 17
Shifted: -9 Accuracy: 17
Shifted: -8 Accuracy: 16
Shifted: -7 Accuracy: 15
Shifted: -6 Accuracy: 13
Shifted: -5 Accuracy: 14
Shifted: -4 Accuracy: 26
Shifted: -3 Accuracy: 54
Shifted: -2 Accuracy: 83
Shifted: -1 Accuracy: 95
Shifted: 0 Accuracy: 97
Shifted: 1 Accuracy: 95
Shifted: 2 Accuracy: 82
Shifted: 3 Accuracy: 53
Shifted: 4 Accuracy: 23
Shifted: 5 Accuracy: 9
Shifted: 6 Accuracy: 5
Shifted: 7 Accuracy: 4
Shifted: 8 Accuracy: 4
Shifted: 9 Accuracy: 4
Shifted: 10 Accuracy: 6
Shifted: 11 Accuracy: 10