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!
```

```
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.

```
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!
```

```
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

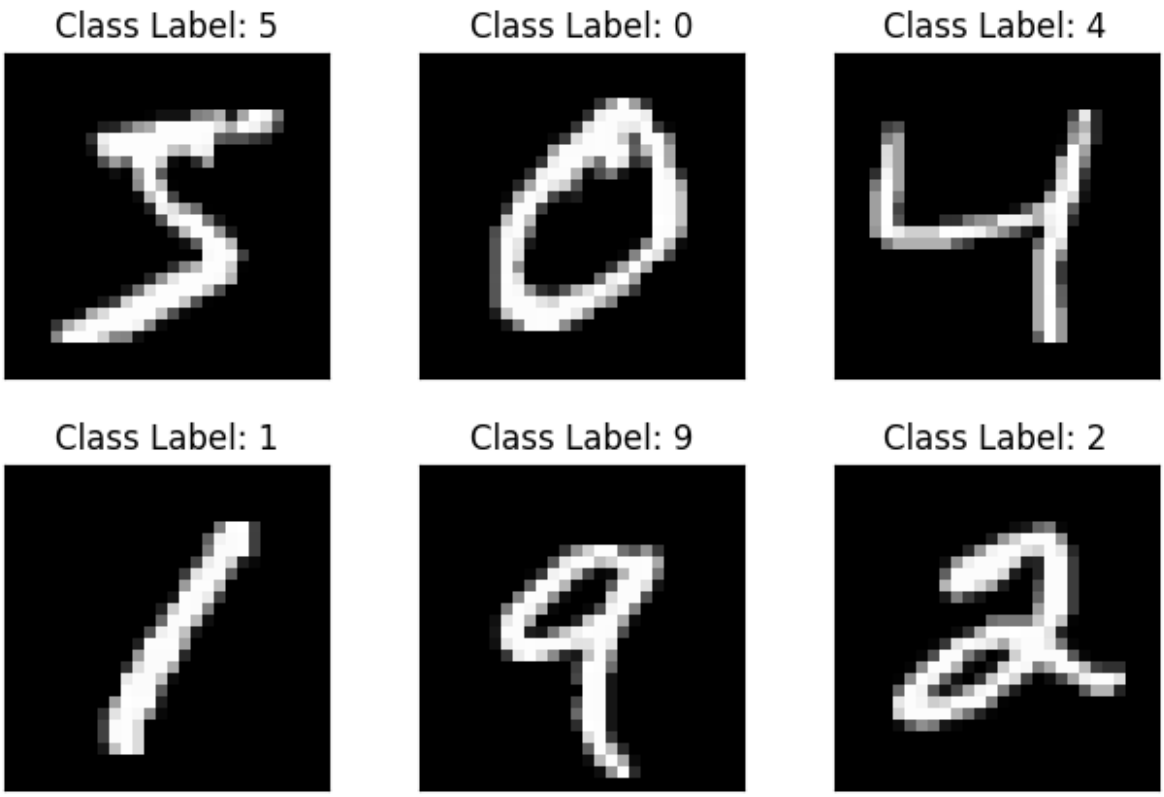
```
# downloading mnist into folder
data_dir = 'data' # make sure that this folder is created in your
working dir
# transform the PIL images to tensor using
torchvision.transform.toTensor method
train_data = torchvision.datasets.MNIST(data_dir, train=True,
download=True,
transform=torchvision.transforms.Compose([torchvision.transforms.ToTen
```

```
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

```
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):
    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!')
```

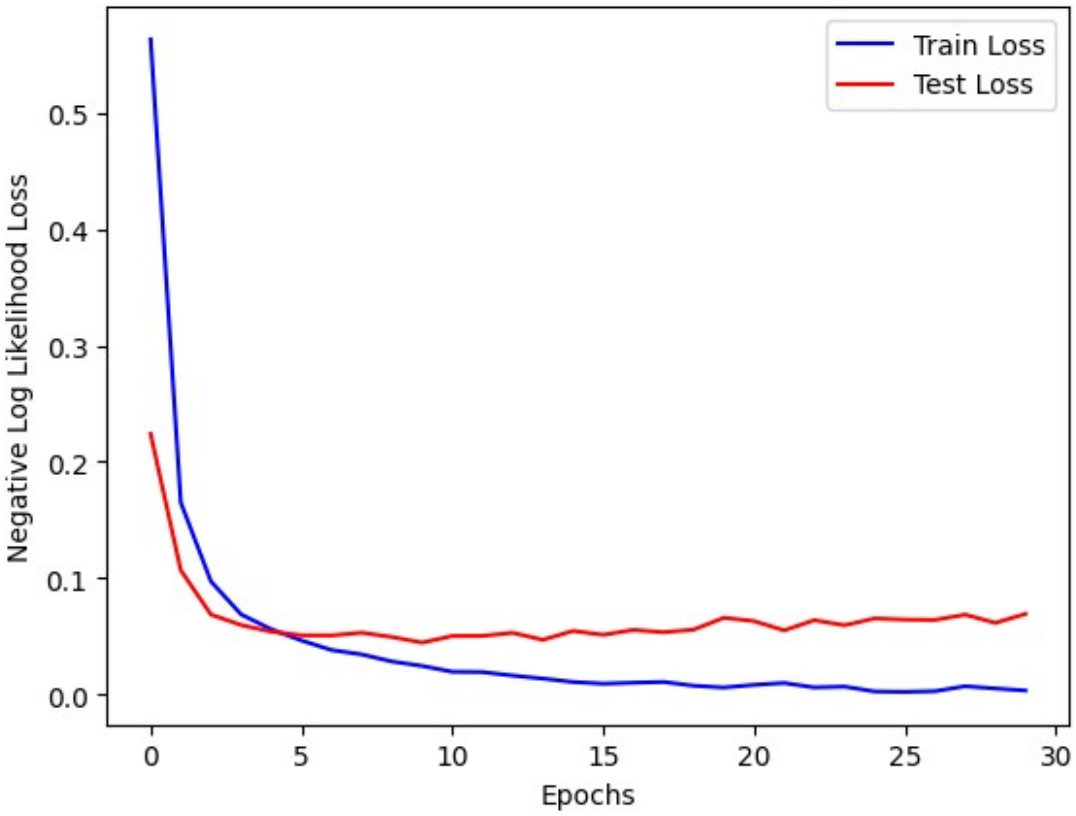
```
Conv Net training started

/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)

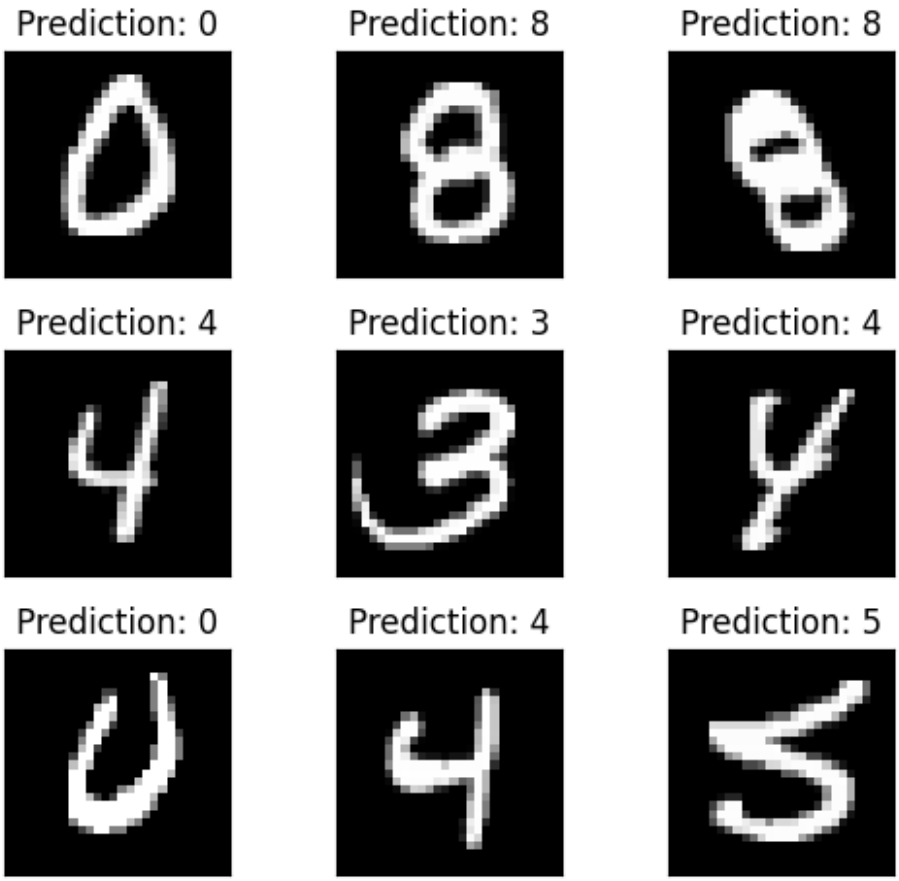
```
fig = plt.figure()
plt.plot(history_conv['train_loss'], color='blue')
plt.plot(history_conv['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

```
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(
```

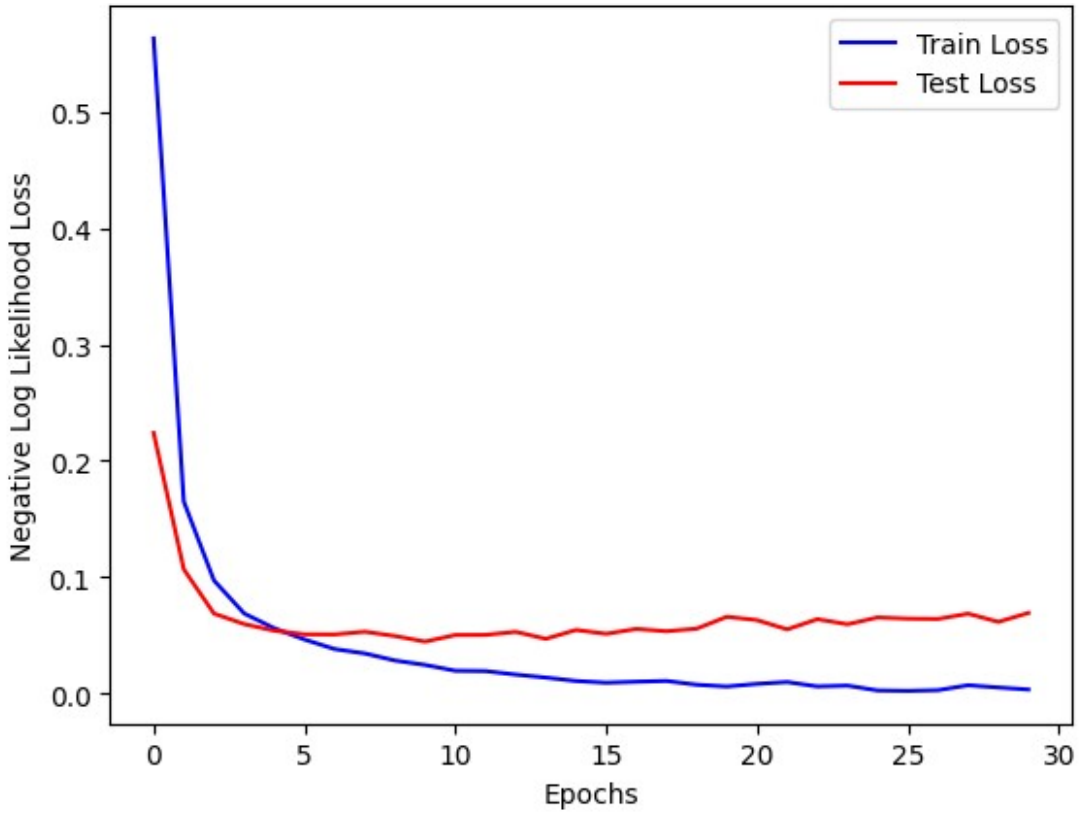
```
        model=fc_net,
        test_loader=test_loader,
        device=device,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
    # Print Losses
    print(f'Epoch {epoch+1}/{num_epochs} : train loss {train_loss:.3f}
\t val loss {val_loss:.3f}')
    history_fc['train_loss'].append(train_loss)
    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')
```

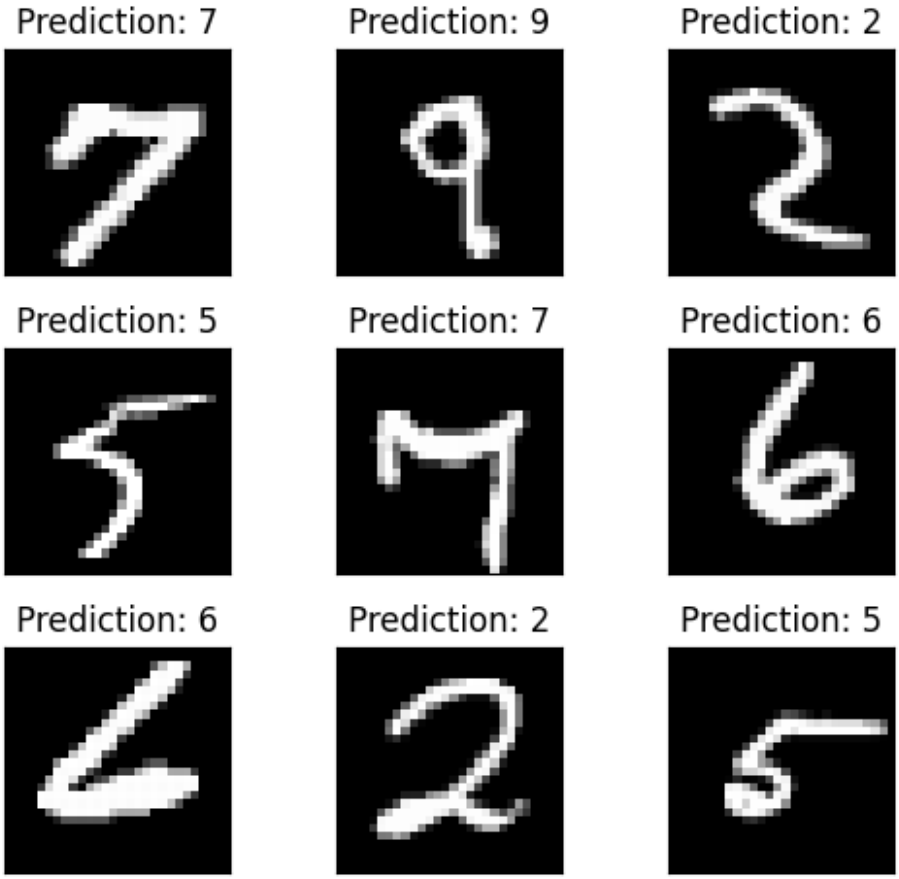
```
Text(0, 0.5, 'Negative Log Likelihood Loss')
```



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:

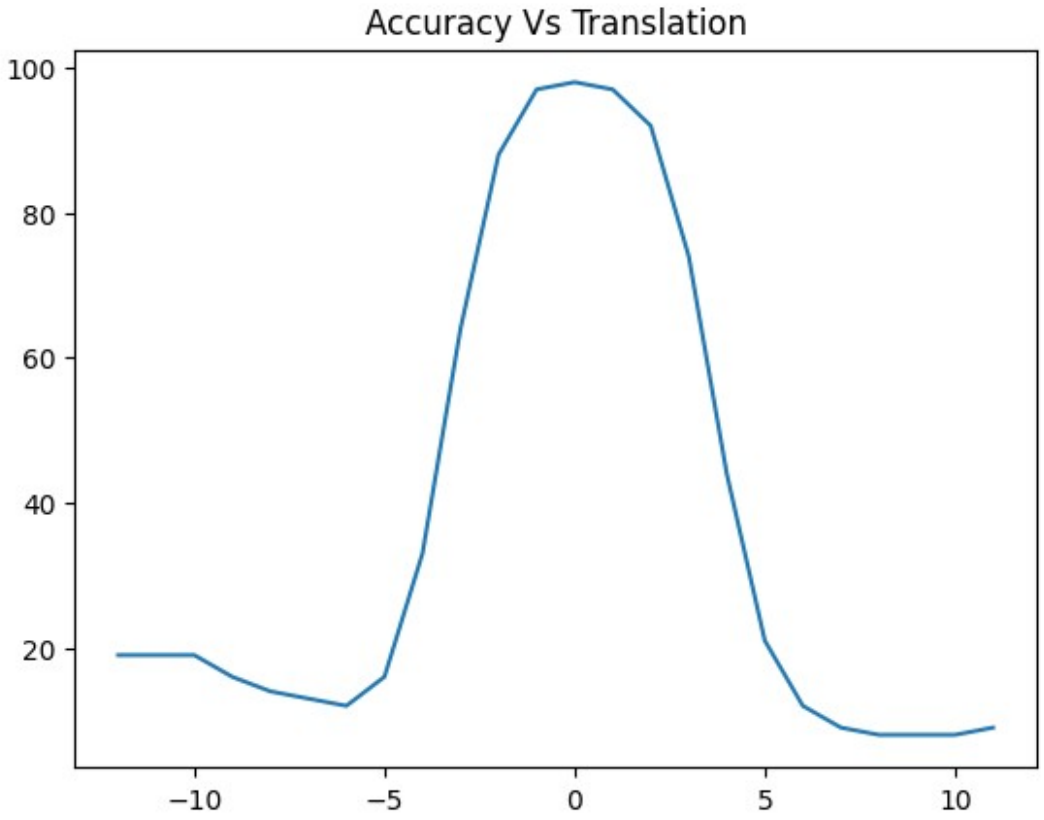
Convolutional neural networks may not be perfectly invariant under translation due to pooling layers, non-linear activation functions, and the training data. To improve translation invariance, we can use strided convolutions instead of pooling layers, use batch normalization, and use data augmentation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

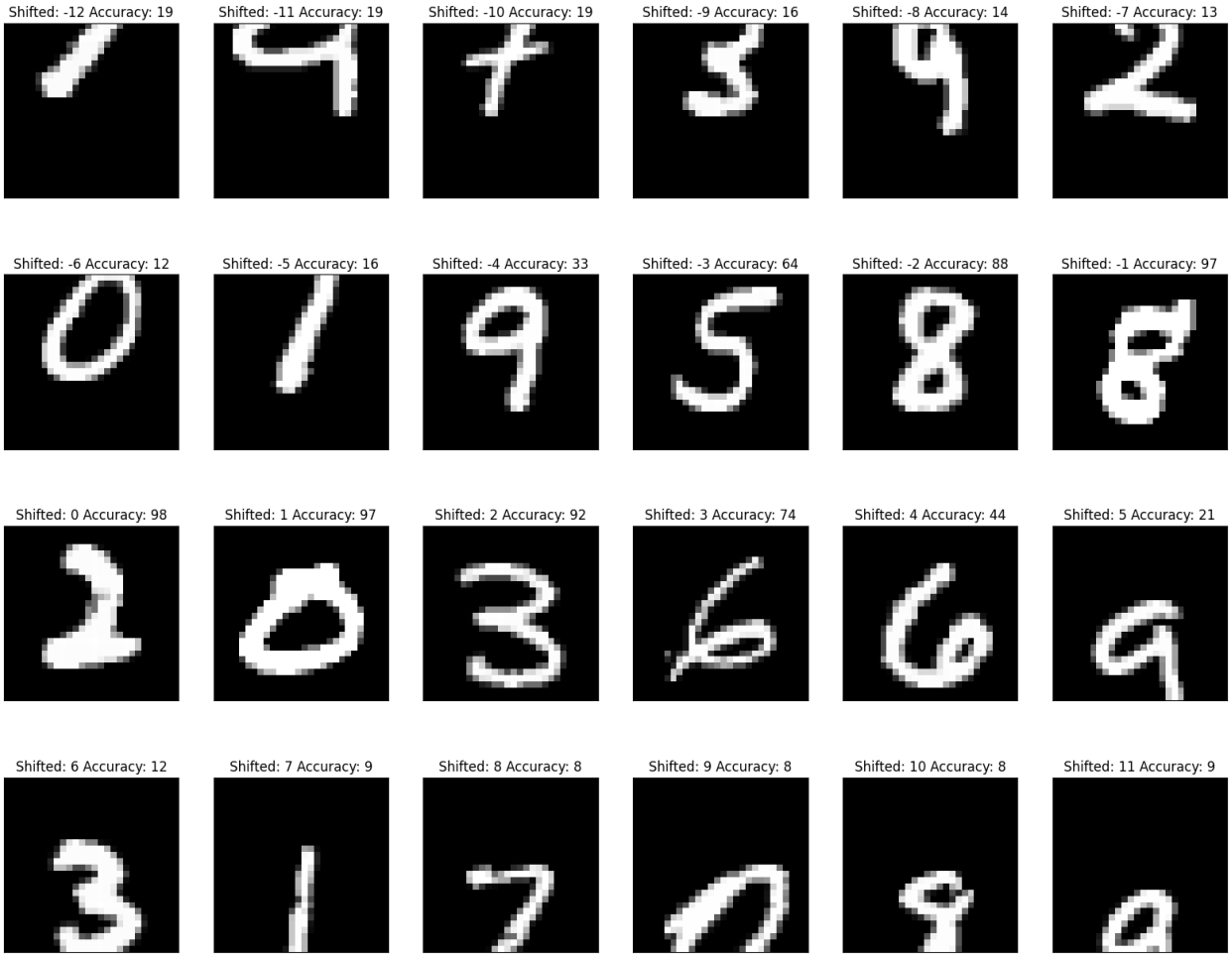
```
                correct += (pred == labels).sum().item()
                predicted.append(pred.cpu())
                actual.append(labels.cpu())
            shifted.append(images[0][0].cpu())
            acc = 100 * correct // total
            accuracies.append(acc)
    return accuracies,shifted

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
```

```
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
```



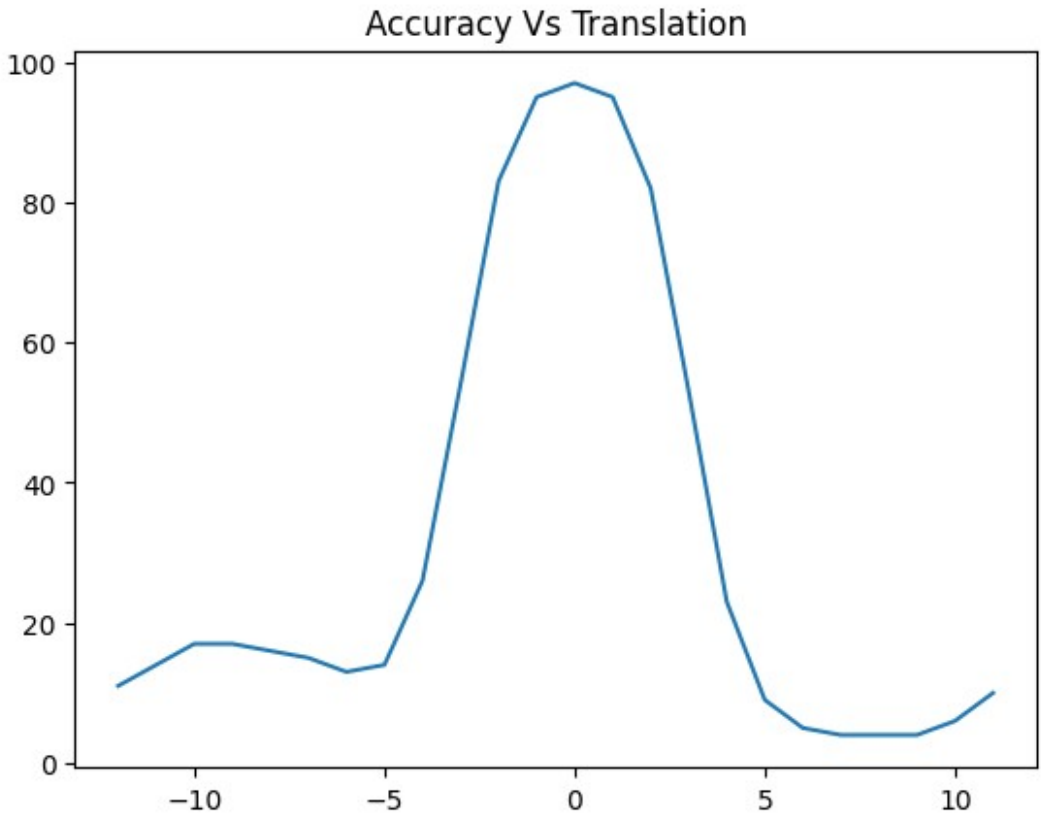
Exercise 3.1.8: Do the same for FC-Net and plot the accuracies. Is the rate of accuracy degradation same as Conv-Net? Can you justify why this happened? Clue: You might want to look at the way convolution layers process information

Answer:

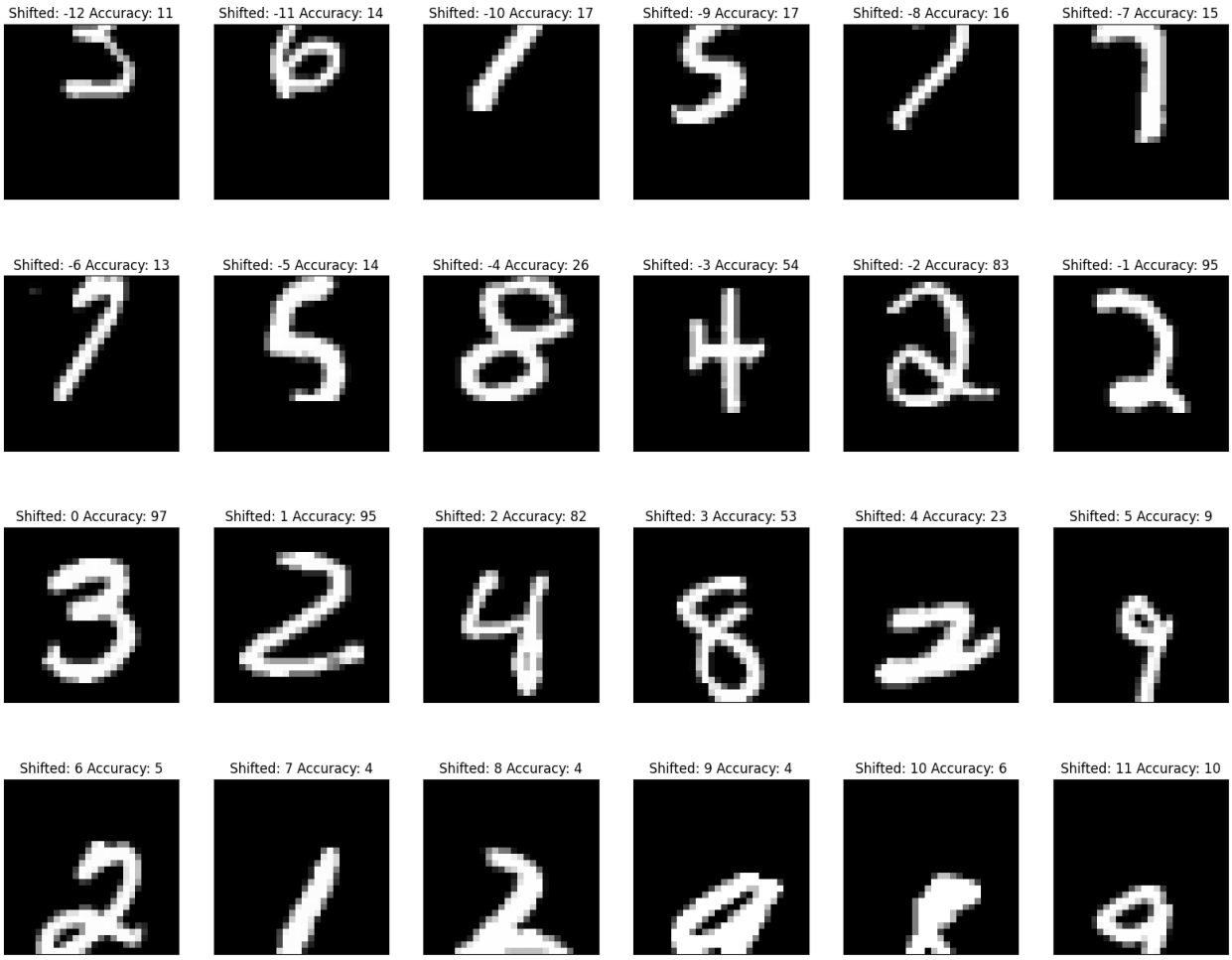
Yes, the plots look similar. The similar rate of accuracy degradation between the FC-Net and Conv-Net can be attributed to the way convolution layers process information. Convolutional layers in Conv-Nets capture local spatial relationships and exhibit a degree of translation invariance through shared 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

This can be run run on Google Colab using this link

STABLE DIFFUSION ASSIGNMENT

Preliminary

In this homework assignment, you will delve deep into Stable Diffusion Models based on the DDPMs paper. The homework is fragmented into three main parts: Forward Diffusion, the Unet Architecture of Noise Predictor Model with training and the Sampling part of Stable Diffusion Models. By completing this assignment, you will gain a comprehensive understanding of the mathematics underlying stable diffusion and practical skills to implement and work with these models.

Setup and Data Preparation

Execute the provided cell to import essential libraries, ensure result reproducibility, set device configurations, download the MNIST dataset, and initialize DataLoaders for training, validation, and testing.

Note: Run the cell as is; no modifications are necessary.

```
#########################################################################
######
#                                  TO DO
#
#               Execute the block to load & Split the Dataset
#
#########################################################################
######

import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F


# Ensure reproducibility
torch.manual_seed(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Download and Load the MNIST dataset
transform = transforms.ToTensor()
```

```
full_trainset = torchvision.datasets.MNIST(root='./data', train=True,
download=True, transform=transform)

# Splitting the trainset into training and validation datasets
train_size = int(0.8 * len(full_trainset))  # 80% for training
val_size = len(full_trainset) - train_size  # remaining 20% for
validation
train_dataset, val_dataset =
torch.utils.data.random_split(full_trainset, [train_size, val_size])

trainloader = torch.utils.data.DataLoader(train_dataset,
batch_size=32, shuffle=True)
valloader = torch.utils.data.DataLoader(val_dataset, batch_size=32,
shuffle=False)

testset = torchvision.datasets.MNIST(root='./data', train=False,
download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32,
shuffle=False)
```

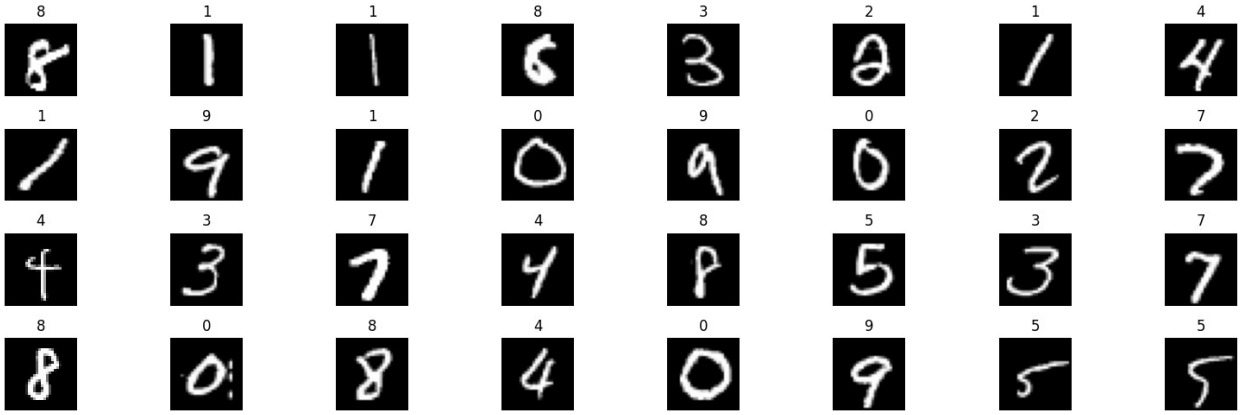
Image Display Function

Below is a utility function, `display_images`, used for visualizing dataset and monitoring diffusion process for slight intuitive way of choosing parameter purposes and display results post training in this assignment.

Note: Run the cell to view the images from the dataset.

```
#######################################################################
######
#                                  TO DO
#
#               Execute the block to display images of MNIST
#
#######################################################################
######

import matplotlib.pyplot as plt

def display_images(images, n, images_per_row=5, labels = None):
    """
    Display n images in rows where each row contains a specified
number of images.

    Parameters:
    - images: List/Tensor of images to display.
    - n: Number of images to display.
    - images_per_row: Number of images per row.
    """
    # Define the number of rows based on n and images_per_row
```

```
    num_rows = (n + images_per_row - 1) // images_per_row  # Rounding
up
    plt.figure(figsize=(2*images_per_row, 1.25 * num_rows))
    for i in range(n):
        plt.subplot(num_rows, images_per_row, i+1)
        plt.imshow(images[i].cpu().squeeze().numpy(), cmap='gray')
        if labels is not None:
          plt.title(labels[i])
        plt.axis('off')
    plt.tight_layout()
    plt.show()


for batch in trainloader:
  # In a batch from many batches in trainloader, get the the first one
and work with that
  batch_size = len(batch[0])
  display_images(images= batch[0],n = batch_size, images_per_row=8,
labels = batch[1].tolist())
  break
```



EXERCISE 1: FORWARD DIFFUSION

Noise Diffusion

The following block `Noise Diffusion` is to give you a high level intuition of what forward diffusion process is and how we achieve results without any dependency on prior results. There is a detailed derivation on how we landed on the formula mentioned in the paper and below, if you're interested in the math, we recommend reading Denoising Diffusion Probabilistic Models for clear understanding of ***Forward Diffusion Process*** and mathematical details involved in it!

Noise Diffusion

The idea behind adding noise to an image is rooted in a simple linear interpolation between the original image and a noise term. Let's use the concept of a blending or mixing factor (which we'll refer to as α)

1. Linear Interpolation:

Given two values, A and B, the linear interpolation between them based on a blending factor α (where $0 \leq \alpha \leq 1$) is given by:

$$\text{Result} = \alpha A + (1 - \alpha) B$$

If $\alpha = 1$, the Result is entirely A. If $\alpha = 0$, the Result is entirely B. For values in between, you get a mixture.

2. Applying to Images and Noise:

In our context:

- A is the original image.
- B is the noise (often drawn from a standard normal distribution, but could be any other distribution or type of noise).

So, for each pixel (p) in our image, and at a given timestep (t):

$$\text{noisy\_image}_p(t) = \sqrt{(¿\,\alpha(t)) \times \text{original\_image}_p} + \sqrt{(¿\,1 - \alpha(t)) \times \text{noise}_p\,¿¿}$$

Where:

- $\alpha(t)$ is the blending factor at timestep t
- original_image_p is the intensity of pixel p in the original image.
- noise_p is the noise value for pixel p, typically drawn from a normal distribution.

3. Time-Dependent α:

For the Time-Dependent Alpha Noise Diffusion method, our α isn't a constant; it changes over time. That's where our linear scheduler or any other scheduler comes in: to provide a sequence of values over timesteps.

Now, considering cumulative products: The reason for introducing the cumulative product of αs was to have an accumulating influence of noise over time. With each timestep, we multiply the original image with the cumulative product of α values up to that timestep, making the original image's influence reduce multiplicatively. The noise's influence, conversely, grows because it's based on $1 -$ the cumulative product of the αs.

That's why the formula becomes:

$$\text{noisy\_image}_t = \text{original\_image} \times \prod_{i=1}^{t} \alpha_i + \text{noise} \times \left(1 - \prod_{i=1}^{t} \alpha_i\right)$$

In essence, this formula is just a dynamic way to blend an original image and noise, with the blending ratios changing (and typically becoming more skewed toward noise) over time.

4. Linear Scheduling of Noise Blending:

One of the core components of this noise diffusion assignment is how the blending of noise into the original image is scheduled. To accomplish this, we utilize a linear scheduler that determines the progression of the β (noise level parameter) over a series of timesteps.

Imagine you wish to transition β from a `start_beta` of 0.1 to an `end_beta` of 0.2 over 11 timesteps. The goal is for the rate of noise blending into the image to increase progressively. In this case, the sequence of β values would look like this: [0.1, 0.11, 0.12,..., 0.2].

This sequence, `self.betas`, is precisely what the `linear_scheduler` generates.

```
self.betas = self.linear_scheduler().to(self.device)
```

In essence, the `linear_scheduler` method calculates the sequence of β values for the diffusion process, ensuring that the noise blending into the image increases linearly over the given timesteps.

Terminologies:

1. $ \beta $: Represents the noise level parameter, defined between the start and end beta values.
2. $ \alpha $: Represents the blending factor, calculated as $(1 - \beta)$.
3. Cumulative Product of $ \alpha $: Understand its significance in dynamically blending the original image and noise over timesteps, without any dependency on prior timesteps.

NoiseDiffuser Class

TO DO

Implement `NoiseDiffuser` Class, ***Follow Instructions in the code cell***

```
import torch

class NoiseDiffuser:
  def __init__(self, start_beta, end_beta, total_steps, device='cpu'):

    assert start_beta < end_beta < 1.0

    self.device = device
    self.start_beta = start_beta
    self.end_beta = end_beta
    self.total_steps = total_steps

##########################################################################
```

```
######
    #                                    TO DO
#
    #                   Compute the following variables needed
#
    #                       for Forward Diffusion Process
#
    #                   schedule betas,  compute alphas & cumulative
#
    #                                product of alphas
#

#########################################################################
######
    self.betas = self.linear_scheduler().to(self.device)
    self.alphas = 1 - self.betas
    self.alpha_bar = torch.cumprod(self.alphas, dim = 0) # Linear
Cumulative Products of alphas!


  def linear_scheduler(self):

    return torch.linspace(self.start_beta, self.end_beta,
self.total_steps)

    """Returns a linear schedule from start to end over the specified
total number of steps."""

#########################################################################
######
    #                                    TO DO
#
    #                   Return a linear schedule of `betas`
#
    #                     from `start_beta` to `end_beta`
#
    #                          hint: torch.linspace()
#

#########################################################################
######

  def noise_diffusion(self, image, t):
    """
    Diffuse noise into an image based on timestep t using the pre-
computed cumulative product of alphas.
    """

#########################################################################
######
```

```
    #                                   TO DO
#
    #                  Process the given `image` for timesteps `t`
#
    #                 Return processed image & necessary variables
#

#########################################################################
######
    image = image.to(self.device)

    t = torch.LongTensor(t).to(self.device)  # Convert timesteps to a
long tensor

    # Ensure that t is within the valid range of timesteps
    t = torch.clamp(t, 0, self.total_steps - 1)

    # Compute the alpha and cumulative alpha bar at the given
timestep.
    alpha_t = self.alphas[t].unsqueeze(-1).unsqueeze(-1).unsqueeze(-1)
    alpha_bar_t = self.alpha_bar[t].sqrt()

    # Generate noise tensor with the same number of channels as the
image tensor.
    noise = torch.randn(image.size(), device=self.device)

    alpha_bar_t = alpha_bar_t.view(-1, 1, 1, 1)

    noise = noise * torch.sqrt(1 - alpha_bar_t)
    diffused_image = torch.sqrt(alpha_bar_t) * image + noise

    return diffused_image, noise
```

Testing NoiseDiffuser Class (SANITY CHECK)

```
# SANITY CHECK
in_channels_arg = 1
out_channels_arg = 1
batch_size = 32
height = 28
width = 28
total_timesteps = 50
start_beta, end_beta = 0.001, 0.2

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height,
width)).to(device)
```

```
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps,
device)

timesteps_to_display = torch.randint(0, total_timesteps,
(batch_size,), device=device).long().tolist()
y, _ = diffuser.noise_diffusion(x, timesteps_to_display)

assert len(x.shape) == len(y.shape)
assert y.shape == x.shape

print("Sanity Check for shape mismatches")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

```
Sanity Check for shape mismatches
Shape of the input :  torch.Size([32, 1, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])
```

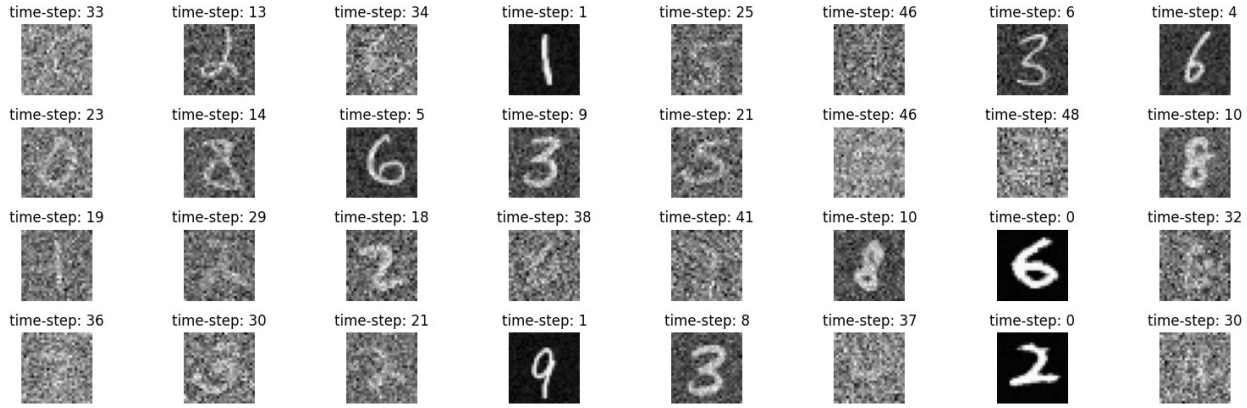
Demonstrating Examples

Note: Observe the visual effect of noise diffusion for different images at random timesteps. How does the noise appear?

Observation: We see that the higher end of the time step there is more noise as we add more noise with each time step

```
#########################################################################
######
#                                TO DO
#
#       Initialize some start_beta, end_beta & total_timesteps
#
#                         and execute the block
#
#########################################################################
######

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
start_beta = 0.0001
end_beta = 0.1
total_timesteps = 50

diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps,
device)

for batch in trainloader:
    minibatch = batch[0]
    batch_size = len(minibatch)
    timesteps_to_display = torch.randint(0, total_timesteps,
(batch_size,), device=device).long().tolist()
```

```
    noisy_images, _ = diffuser.noise_diffusion(minibatch,
timesteps_to_display)
    display_images(images=noisy_images, n=batch_size,
images_per_row=8, labels=list(map(lambda x: "time-step: " + str(x),
timesteps_to_display)))
    break
```



HyperParameters

Smartly setting the start and end values of beta can control the noise diffusion's character.

- **Lower Start and Higher End**: Starting with a lower beta and ending with a higher one means that original image's contribution remains dominant in the beginning and slowly diminishes. This can be useful when the goal is to have a gradual transition from clear image to noisier version.
- **Higher Start and Lower End**: The opposite approach, starting with a Higher beta and ending with a lower one, can be useful when goal is to introduce noise more aggressively initially and taper off towards the end.
- **THINK WHAT WOULD WE NEED** `Higher Start and Lower End` or `Lower Start and Higher End`

The precise values can be fine-tuned based on specific requirements, visual assessments (like in the cell below) or even metrics.

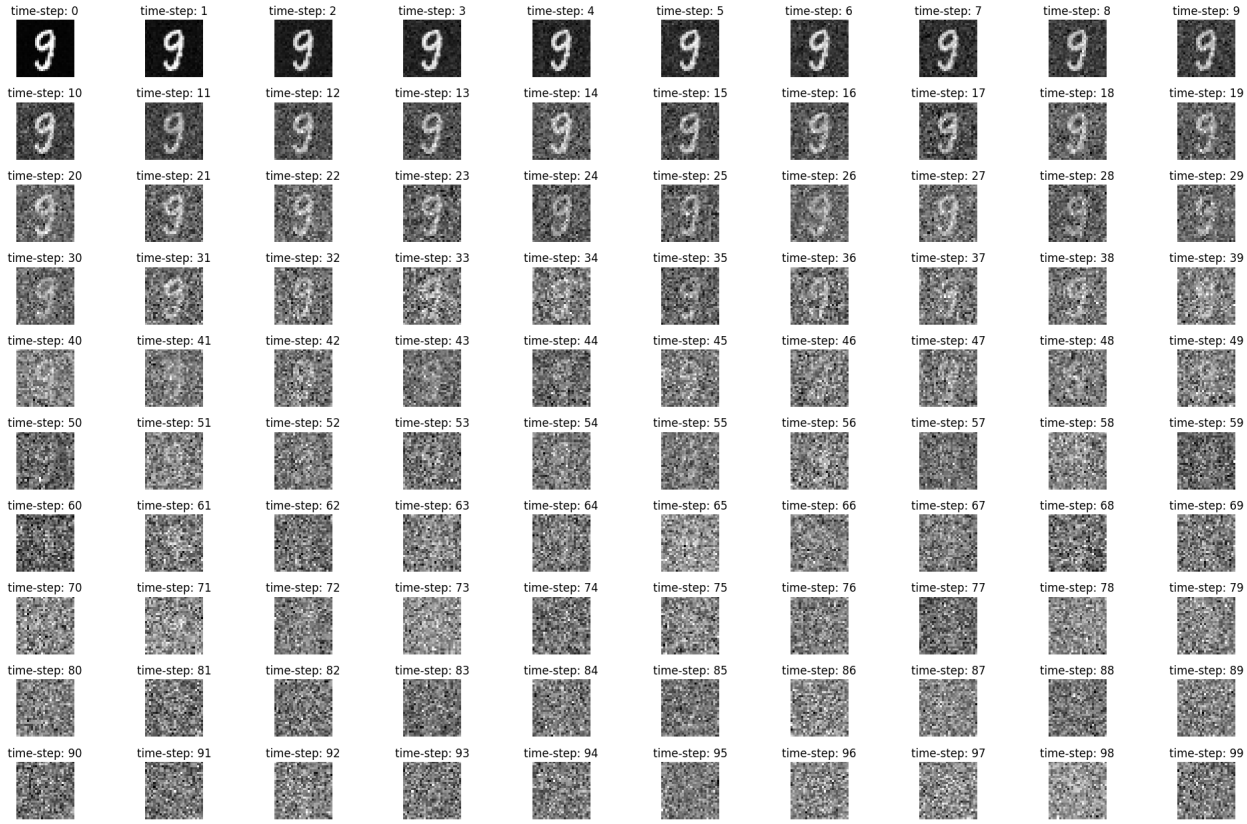
Exploration with Varied beta Values and Timesteps:

- In the below cell, you are encouraged to tweak values of `start_beta` and `end_beta` and even modify `total_timesteps` to observe the effect over a longer/shorter period

Note: Pay close attention to how the noise diffusion evolves over time. Can you see a clear transition from the start to the end timestep? How do different images react to the same noise diffusion process?

```
########################################################################
######
```

```
#                                   TO DO
#
#       Initialize some start_beta, end_beta & total_timesteps
#
#         play around and see the effect of noise introduced
#
#         and think what parameters would you use for training
#
#########################################################################
######

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
total_timesteps = 100
start_beta, end_beta = 0.0001, 0.1
minibatch_size = 1
diffuser = NoiseDiffuser(start_beta, end_beta, total_timesteps,
device)

# PLay around in this cell with different value of alpha (start and
end) and different number of time steps to initially guess and decide
on how many time steps would you like to train the model going
forward.
for batch in trainloader:
    repetitions =
torch.tensor([total_timesteps]).repeat(minibatch_size)
    minibatch = batch[0]
[:minibatch_size,:,:].repeat_interleave(repetitions, dim=0)
    batch_size = len(minibatch)
    timesteps_to_display = torch.linspace(0, total_timesteps-1,
total_timesteps, dtype=int).tolist() * minibatch_size
    noisy_images, _ = diffuser.noise_diffusion(minibatch,
timesteps_to_display)
    display_images(images=noisy_images, n=batch_size,
images_per_row=10, labels=list(map(lambda x: "time-step: " + str(x),
timesteps_to_display)))
    break
```



EXERCISE 2: REVERSE DIFFUSION

Model Architecture

Implementing Skip Connections in U-Net Architecture

While the architecture of the U-Net is provided to you, a critical component—skip connections—needs to be integrated by you. The original paper, "U-Net: Convolutional Networks for Biomedical Image Segmentation" showcases the importance of these skip connections, as they allow the network to utilize features from earlier layers, making the segmentation more precise.

Placeholder for Skip Connections:

In the given architecture, you will find lines like the one below, which are the components of upsampling process in the U-Net:

```
y2 = self.afterup2(torch.cat([y2, torch.zeros_like(y2)], axis = 1))
```

Here, `torch.zeros_like(y2)` acts as a placeholder, indicating where the skip connection should be added. Your task is to replace this placeholder with the appropriate feature map from an earlier corresponding layer in the network.

Important Points to Keep in Mind:

- The U-Net architecture has multiple layers, so you'll need to repeat this process for each layer where skip connections are required.
- The provided helper function, `self.xLikeY(source, target)`, will be crucial in ensuring the feature maps you concatenate have matching dimensions.
- While the focus of this assignment is on crucial idea of stable diffusion, the U-Net architecture is provided to you but it is importatnt you implement skip connections, as understanding their role and significance in the U-Net architecture will be beneficial.
- ***Note: Feel free to modify architecture, parameters including number & types of layers used, kernel Sizes, padding, etc, you won't be judged on the architecture you use if you have the desired results post training.***

UNet Class

TO DO

Fill in `UNet` Class, ***Follow Instructions above***

```
class UNet(nn.Module):

    def __init__(self, in_channels, out_channels):
        """
        in_channels: input channels of the incoming image
        out_channels: output channels of the incoming image
        """
        super(UNet, self).__init__()

        #----------------------- Encoder -----------------------#

#########################################################################
######
        # Initial Convolutions (Using doubleConvolution() function)
#
        # Building Down Sampling Layers (Using Down() function)
#

#########################################################################
######
        self.ini = self.doubleConvolution(inC=in_channels, oC=16)
        self.down1 = self.Down(inputC=16, outputC=32)
        self.down2 = self.Down(inputC=32, outputC=64)

        #----------------------- Decoder -----------------------#
```

```
########################################################################
######
        # For each Upsampling block                                    #
        # Building Time Embeddings (Using timeEmbeddings() function)
#
        # Building Up Sampling Layer (Using ConvTranspose2d()
function)      #
        # followed by Convolution (Using doubleConvolution() function)
#

########################################################################
######

        self.time_emb2 = self.timeEmbeddings(1, 64)
        self.up2 = nn.ConvTranspose2d(in_channels=64, out_channels=32,
kernel_size=3, stride=2)
        self.afterup2 = self.doubleConvolution(inC=64, oC=32)

        self.time_emb1 = self.timeEmbeddings(1, 32)
        self.up1 = nn.ConvTranspose2d(in_channels=32, out_channels=16,
kernel_size=3, stride=2)
        self.afterup1 = self.doubleConvolution(inC=32, oC=16, kS1=5,
kS2=4)

        #------------------------ OUTPUT ----------------------#

########################################################################
######
        # Constructing the final Output Layer (Use Conv2d() function)
#

########################################################################
######
        self.out = nn.Conv2d(in_channels=16,
out_channels=out_channels, kernel_size=1, stride=1, padding=0)

    def forward(self, x, t=None):
        assert t is not None

        #------------------------ Encoder ----------------------#

########################################################################
######
        # Processing Inputs by                                  #
        # performing Initial Convolutions                        #
        # followed by Down Sampling Layers                      #

########################################################################
######
        x1 = self.ini(x)                  # Initial Double Convolution
```

```
        x2 = self.down1(x1)                # Downsampling followed by
Double Convolution
        x3 = self.down2(x2)                # Downsampling followed by
Double Convolution

        #----------------------- Decoder ----------------------#

##########################################################################
######
        # For each Upsampling block, we add time Embeddings to
#
        # Feature Maps, process this by                          #
        # Up Sampling followed by concatenation & Convolution
#

##########################################################################
######
        t2 = self.time_emb2(t)[:, :, None, None]
        y2 = self.up2(x3 + t2)
        y2 = self.afterup2(torch.cat([y2, self.xLikeY(x2, y2)],
axis=1))  # Concatenate and apply Double Convolution

        t1 = self.time_emb1(t)[:, :, None, None]
        y1 = self.up1(y2 + t1)
        y1 = self.afterup1(torch.cat([y1, self.xLikeY(x1, y1)],
axis=1))  # Crop corresponding Downsampled Feature Map, Double
Convolution

        #----------------------- OUTPUT ----------------------#

##########################################################################
######
        # Processing final Output                          #

##########################################################################
######
        outY = self.out(y1)                # Output Layer (ks-1, st-1,
pa-0)

        return outY


#-------------------------------------------------------------------------
---------------------------- Helper Functions Within Model Class

    def timeEmbeddings(self, inC, oSize):
        """
        inC: Input Size, (for example 1 for timestep)
        oSize: Output Size, (Number of channels you would like to
match while upsampling)
```

```
        """
        return nn.Sequential(nn.Linear(inC, oSize),
                             nn.ReLU(),
                             nn.Linear(oSize, oSize))

    def doubleConvolution(self, inC, oC, kS1=3, kS2=3, sT=1, pA=1):
        """
        Building Double Convolution as in the original paper of Unet
        inC: inputChannels
        oC: outputChannels
        kS1: Kernel_size of the first convolution
        kS2: Kernel_size of the second convolution
        sT: stride
        pA: padding
        """
        return nn.Sequential(
            nn.Conv2d(in_channels=inC, out_channels=oC,
kernel_size=kS1, stride=sT, padding=pA),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=oC, out_channels=oC,
kernel_size=kS2, stride=sT, padding=pA),
            nn.ReLU(inplace=True),
        )

    def Down(self, inputC, outputC, dsKernelSize=None):
        """
        Building Down Sampling Part of the Unet Architecture (Using
MaxPool) followed by double convolution
        inputC: inputChannels
        outputC: outputChannels
        """
        return nn.Sequential(
            nn.MaxPool2d(2),
            self.doubleConvolution(inC=inputC, oC=outputC)
        )

    def xLikeY(self, source, target):
        """
        Helper function to resize the downsampled x's to concatenate
with upsampled y's as in Unet Paper
        source: tensor whose shape will be considered ---------
UPSAMPLED TENSOR (y)
        target: tensor whose shape will be modified to align with the
target ---------DOWNSAMPLED TENSOR (x)
        """
        x1 = source
        x2 = target
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]
        x1 = F.pad(x1, [diffX // 2, diffX - diffX // 2, diffY // 2,
```

```
diffY - diffY // 2])
        return x1
```

Testing UNet Class (SANITY CHECK)

```
# SANITY CHECK FOR UnetBottleNeck (Single Channeled B/W Images)
in_channels_arg = 1
out_channels_arg = 1
batch_size = 32
height = 28
width = 28
total_timesteps = 50

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Positional Encoding Object
timesteps_to_display = torch.randint(0, total_timesteps,
(batch_size,), device=device).long().tolist()

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height,
width)).to(device)
model = UNet(in_channels=in_channels_arg,
out_channels=out_channels_arg)
model = model.to(device)

y = model.forward(x = x, t =
torch.tensor(timesteps_to_display).to(torch.float32).cuda().view(-
1,1))
assert len(x.shape) == len(y.shape)
assert y.shape == (batch_size, out_channels_arg, height, width)

print("Sanity Check for Single Channel B/W Images")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

```
Sanity Check for Single Channel B/W Images
Shape of the input :  torch.Size([32, 1, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])
```

```
# SANITY CHECK FOR UnetBottleNeck (Colored Images)
in_channels_arg = 3
out_channels_arg = 1
batch_size = 32
height = 28
width = 28

# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# Positional Encoding Object
timesteps_to_display = torch.randint(0, total_timesteps,
(batch_size,), device=device).long().tolist()

# Sanity check
x = torch.randn((batch_size, in_channels_arg, height,
width)).to(device)
model = UNet(in_channels=in_channels_arg,
out_channels=out_channels_arg)
model = model.to(device)

y = model.forward(x=x, t =
torch.tensor(timesteps_to_display).to(torch.float32).cuda().view(-
1,1))
assert len(x.shape) == len(y.shape)
assert y.shape == (batch_size, out_channels_arg, height, width)

print("Sanity Check for Multi-channel or colored Images")
print("Shape of the input : ", x.shape)
print("Shape of the output : ", y.shape)
```

```
Sanity Check for Multi-channel or colored Images
Shape of the input :  torch.Size([32, 3, 28, 28])
Shape of the output :  torch.Size([32, 1, 28, 28])
```

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if
p.requires_grad)

num_params = count_parameters(model)
print(f"The model has {num_params:,} trainable parameters.")
```

```
The model has 145,233 trainable parameters.
```

Train the Model

In the following block, the train function is defined. You have to calculate the noisy data, feed forward through the model and pass the predicted noise and true noise to the criterion to calculate the loss.

```
from tqdm import tqdm

def train(model, train_loader, val_loader, optimizer, criterion,
device, num_epochs, diffuser, totalTrainingTimesteps):
    """
    model: Object of Unet Model to train
    train_loader: Training batches of the total data
    val_loader: Validation batches of the total data
    optimizer: The backpropagation technique
```

```
    criterion: Loas Function
    device: CPU or GPU
    num_epochs: total number of training loops
    diffuser: NoiseDiffusion class object to perform Forward diffusion
    totalTrainingTimesteps: Total number of forward diffusion
timesteps the model is to be trained on
    """

    train_losses = []
    val_losses = []

    for epoch in range(num_epochs):
        model.train()
        total_train_loss = 0

        # Wrapping your loader with tqdm to display progress bar
        train_progress_bar = tqdm(enumerate(train_loader),
total=len(train_loader), desc=f"Epoch {epoch+1}/{num_epochs} [Train]",
leave=False)
        for batch_idx, (data, _) in train_progress_bar:
            data = data.to(device)
            optimizer.zero_grad()

            # Use a random time step for training
            batch_size = len(data)
            timesteps = torch.randint(0, totalTrainingTimesteps,
(batch_size,), device=device).long().tolist()


##########################################################################
######
            #                                   TO DO
#
            #                     Calculate Noisy data, True noise
#
            #                  and Predicted Noise, & then feed it to
criterion         #

##########################################################################
######
            noisy_data, true_noise = diffuser.noise_diffusion(data,
timesteps)
            predicted_noise = model.forward(x = noisy_data,
t=torch.tensor(timesteps).to(torch.float32).cuda().view(-1,1))

            loss = criterion(predicted_noise, true_noise)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
            train_progress_bar.set_postfix({'Train Loss':
```

```
f'{loss.item():.4f}'})

        avg_train_loss = total_train_loss / len(train_loader)
        train_losses.append(avg_train_loss)

        # Validation
        model.eval()
        total_val_loss = 0

        # Wrapping your validation loader with tqdm to display
progress bar
        val_progress_bar = tqdm(enumerate(val_loader),
total=len(val_loader), desc=f"Epoch {epoch+1}/{num_epochs} [Val]",
leave=False)
        with torch.no_grad():
            for batch_idx, (data, _) in val_progress_bar:
                data = data.to(device)

                # For simplicity, we can use the same random timestep
for validation
                batch_size = len(data)
                timesteps = torch.randint(0, totalTrainingTimesteps,
(batch_size,), device=device).long().tolist()


##########################################################################
######
                #                                    TO DO
#
                #                     Calculate Noisy data, True noise
#
                #                 and Predicted Noise, & then feed it
to criterion         #

##########################################################################
######
                noisy_data, true_noise =
diffuser.noise_diffusion(data, timesteps)
                predicted_noise = model.forward(x = noisy_data,
t=torch.tensor(timesteps).to(torch.float32).cuda().view(-1,1))

                loss = criterion(predicted_noise, true_noise)
                total_val_loss += loss.item()
                val_progress_bar.set_postfix({'Val Loss':
f'{loss.item():.4f}'})

        avg_val_loss = total_val_loss / len(val_loader)
        val_losses.append(avg_val_loss)

        print(f'Epoch {epoch+1}/{num_epochs}, Train Loss:
```

```
{avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}')

    return train_losses, val_losses
```

In the following code block, initialize the necessary variables and then Execute to train, save model and plot the loss

Just to give you an idea of how loss curve would look like approximately (not necssarily same for everybody), x-axis represents epochs and y-axis represents loss.

```
#####################################################################
######
#                                  TO DO
#
#                        Initialize the Constants below
#
#####################################################################
######
"""
- `total_time_steps`: Total time steps of forward diffusion
- `start_beta`: Initial point of Noise Level Parameter
- `end_beta`: End point of Noise Level Parameter
- `inputChannels`: 1 for Grayscale Images (Since we're Using MNIST)
- `outputChannels`: How many channels of predicted noise are aiming
for? THINK!
- `num_epochs`: How many epochs are you training for? (*We'd love to
see best results in minimum epcohs of training*)
"""
total_timesteps = 100
startBeta, endBeta = 0.0001, 0.15
inputChannels, outputChannels = 1, 1
num_epochs = 10

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

#####################################################################
######
#                                  TO DO
#
#                            Initialize the Model
#
#                          Initialize the Optimizer
#
#                       Initialize the Loss Function
#
#                        Initialize the NoiseDiffuser
#
#####################################################################
######
stableDiffusionModel = UNet(in_channels=inputChannels,
out_channels=outputChannels)
```

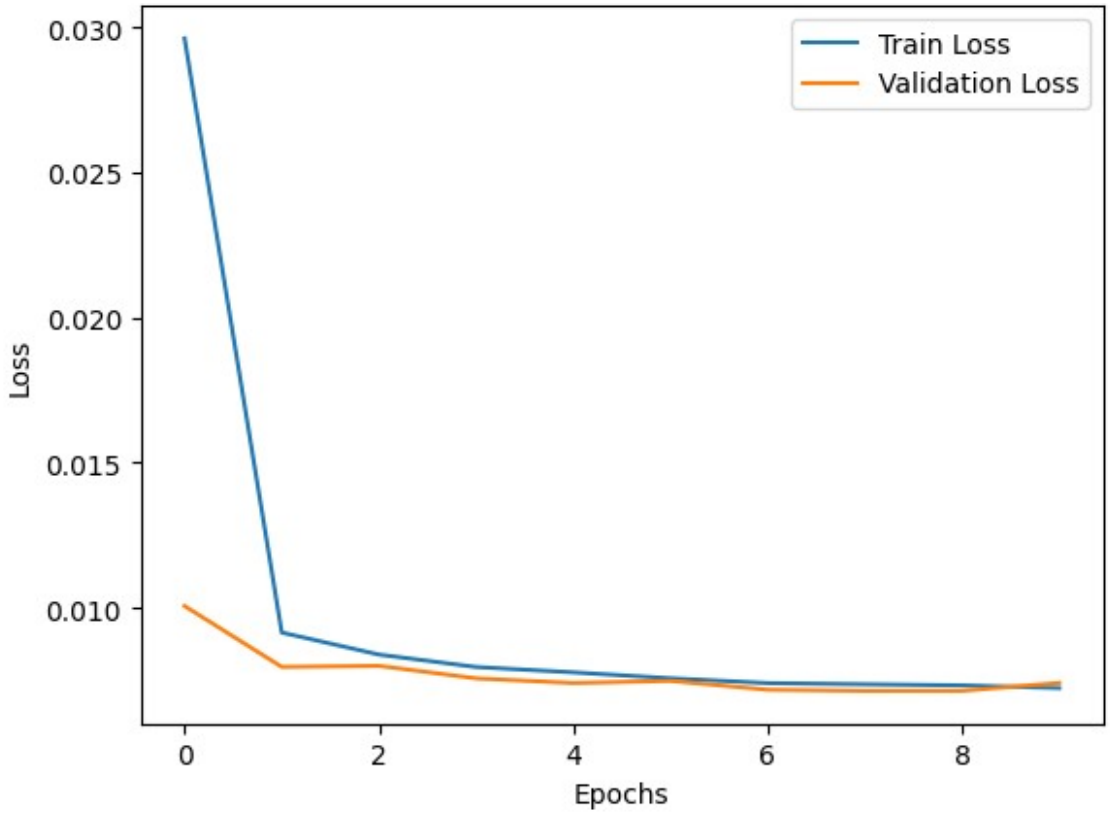
```
optimizer = torch.optim.Adam(stableDiffusionModel.parameters(),
lr=0.001)
criterion = nn.MSELoss()
diffuser = NoiseDiffuser(startBeta, endBeta, total_timesteps, device)

########################################################################
######
#                                  TO DO
#
#                Execute this Block, Train & Save the Model
#
#                             And Plot the Progress
#
########################################################################
######
stableDiffusionModel = stableDiffusionModel.to(device)
train_losses, val_losses = train(model= stableDiffusionModel,
                                 train_loader= trainloader,
                                 val_loader= valloader,
                                 optimizer= optimizer,
                                 criterion= criterion,
                                 device= device,
                                 num_epochs= num_epochs,
                                 diffuser= diffuser,

totalTrainingTimesteps=total_timesteps)

# Save the model
torch.save(stableDiffusionModel.state_dict(), 'HW3SDModel.pth')

#Plot the losses
import matplotlib.pyplot as plt
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```
Epoch 1/10, Train Loss: 0.0296, Validation Loss: 0.0101
```

```
Epoch 2/10, Train Loss: 0.0092, Validation Loss: 0.0080
```

```
Epoch 3/10, Train Loss: 0.0084, Validation Loss: 0.0080
```

```
Epoch 4/10, Train Loss: 0.0080, Validation Loss: 0.0076

Epoch 5/10, Train Loss: 0.0078, Validation Loss: 0.0074

Epoch 6/10, Train Loss: 0.0076, Validation Loss: 0.0075

Epoch 7/10, Train Loss: 0.0074, Validation Loss: 0.0072

Epoch 8/10, Train Loss: 0.0074, Validation Loss: 0.0071

Epoch 9/10, Train Loss: 0.0073, Validation Loss: 0.0071

Epoch 10/10, Train Loss: 0.0072, Validation Loss: 0.0074
```



EXERCISE 3 : SAMLING GENERATION

Sampling formula

The Stable Diffusion Model sampling code involves generating images from a trained model by iteratively denoising an initial random noise tensor. This process is executed in the reverse manner as compared to the diffusion process, where the noise is incrementally added. The iteration happens for a defined number of timesteps. The goal is to move from a purely noisy state to a clear, denoised state that represents a valid sample from the data distribution learned by the model. Refer to the DDPMs Paper for detailed documentation. The formula for sampling part is as follows:

$$X_{t-1} = \frac{1}{\sqrt{\alpha}} * \left(X_t - \frac{1-\alpha}{\sqrt{(1-\acute{\alpha})}} * \epsilon_t \right) + \sqrt{\beta} * z$$

Sample Images

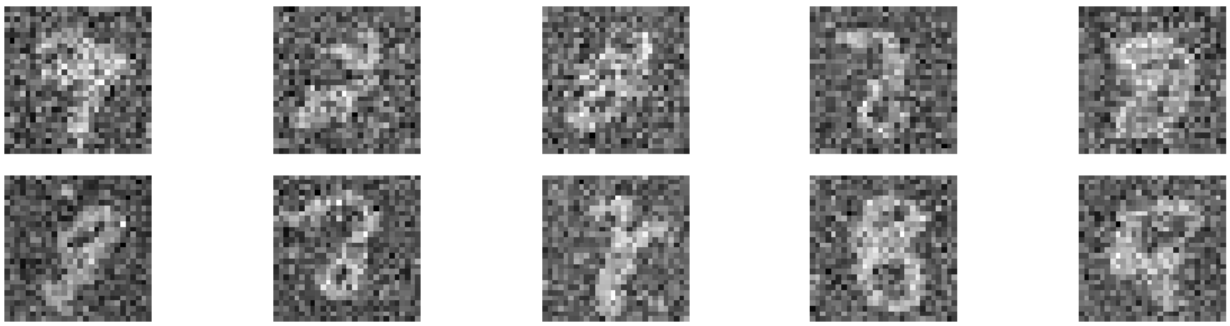
Some sample outputs for random seeds as specified in the code cell of sampling generation and mentioned in the image below are as follows:

```
def generate_samples(x_t, model, num_samples, total_timesteps,
diffuser, device):
    """
    Generate samples using the trained DDPM model.

    Parameters:
    - model: Trained UNetBottleneck model.
    - num_samples: Number of samples to generate.
    - total_timesteps: Total timesteps for the noise process.
    - diffuser: Instance of NoiseDiffuser.
    - device: Computing device (e.g., "cuda" or "cpu").

    Returns:
    - generated_samples: A tensor containing the generated samples.
    """

    # Varibales required by Sampling Formula
    one_by_sqrt_alpha = 1 / torch.sqrt(diffuser.alphas)
    beta_by_sqrt_one_minus_alpha_cumprod = diffuser.betas /
torch.sqrt(1 - diffuser.alpha_bar)


######################################################################
######
    #                                   TO DO
#
    #                Implement the Sampling Algorithm, start with
#
    #                   pure noise, using the trained model
```

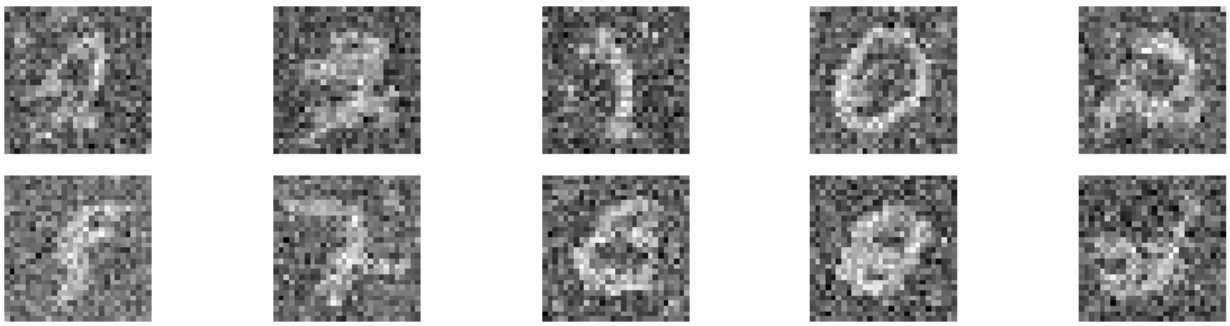
```
#
    #                 perform denoising to generate MNIST Images
#

#########################################################################
######
    # Iterate in reverse order to "denoise" the samples
    for timestep in range(total_timesteps-1, -1, -1):
      z = torch.randn_like(x_t)
      epsilon_t = model.forward(x = x_t, t =
torch.tensor(timestep).to(torch.float32).cuda().view(-1,1))
      x_t_minus_1 = one_by_sqrt_alpha[timestep] *(x_t -
beta_by_sqrt_one_minus_alpha_cumprod[timestep] * epsilon_t ) +
torch.sqrt(diffuser.betas[timestep]) * z
      x_t = x_t_minus_1

    return x_t.detach()


#########################################################################
######
#                                 TO DO
#
#                Post Implementation of Sampling Algorithm,
#
#                       Execute the following lines by
#
#           using the same constants (timesteps and beta values)
#
#                           as you used while training,
#
#                initializing instance of NoiseDiffuser Object
#
#                     and Loading the pretrained model
#
#########################################################################
######
# Create instance of NoiseDiffuser
diffuser = NoiseDiffuser(start_beta=startBeta, end_beta=endBeta,
total_steps=total_timesteps, device= device)
# Using the function:
model_path = 'HW3SDModel.pth'
model = UNet(in_channels=inputChannels,
out_channels=outputChannels).to(device)
model.load_state_dict(torch.load(model_path))
model.eval()

SEED = [ 96, 786, 7150]  # You can set any integer value for the seed

for S in SEED:
```

```
  print("The Outputs for Random Seed {%d}"%S)
  # Set seed for both CPU and CUDA devices
  torch.manual_seed(S)
  if torch.cuda.is_available():
      torch.cuda.manual_seed(S)
      torch.cuda.manual_seed_all(S)
      torch.backends.cudnn.deterministic = True
      torch.backends.cudnn.benchmark = False

  num_samples_to_generate = 10
  # Initialize with random noise
  xt = torch.randn((num_samples_to_generate, 1, 28, 28),
device=device)

  samples = generate_samples(xt, model, num_samples_to_generate,
total_timesteps, diffuser, device)

  # Display the generated samples
  display_images(samples, num_samples_to_generate, images_per_row=5)
```

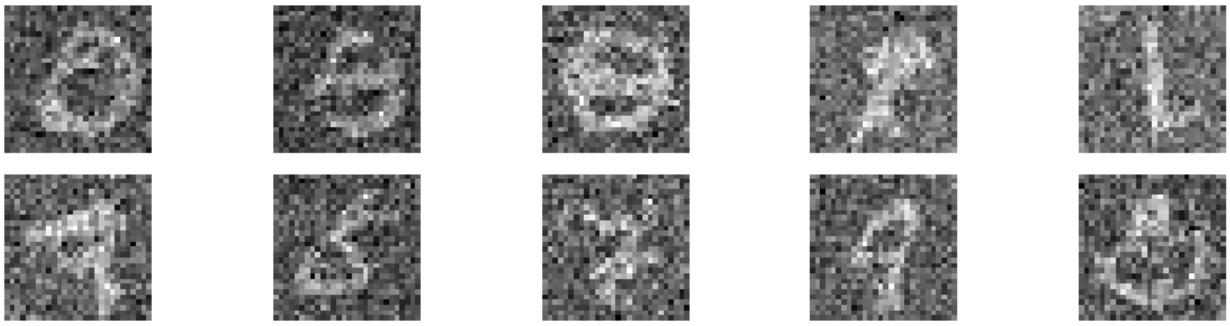
```
The Outputs for Random Seed {96}
```



```
The Outputs for Random Seed {786}
```



```
The Outputs for Random Seed {7150}
```



This can be run [run on Google Colab using this link (https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.3-Visualization_Examples.ipynb)](https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.3-Visualization_Examples.ipynb)

In [1]:
```
%%bash
# If you are on Google Colab, this sets up everything needed.
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
wget -O requirements.txt https://cs7150.baulab.info/2022-Fall/setup/hw1_requireme
pip install -r requirements.txt
# If you are not on Google Colab, you can run these pip requirements on your own
```

```
Collecting baukit (from -r requirements.txt (line 17))
  Cloning https://github.com/davidbau/baukit (https://github.com/davidbau/baukit) (to revision main) to /tmp/pip-install-4eare0f1/baukit_272851fca322484a850f8beb4e57785b
  Resolved https://github.com/davidbau/baukit (https://github.com/davidbau/baukit) to commit 5e23007c02fd58f063200c5dc9033e90f092630d
  Installing build dependencies: started
  Installing build dependencies: finished with status 'done'
  Getting requirements to build wheel: started
  Getting requirements to build wheel: finished with status 'done'
  Installing backend dependencies: started
  Installing backend dependencies: finished with status 'done'
  Preparing metadata (pyproject.toml): started
  Preparing metadata (pyproject.toml): finished with status 'done'
Collecting clip (from -r requirements.txt (line 18))
  Cloning https://github.com/openai/CLIP.git (https://github.com/openai/CLIP.git) (to revision main) to /tmp/pip-install-4eare0f1/clip_56149803170545139cbd658b2007dd3e
  Resolved https://github.com/openai/CLIP.git (https://github.com/openai/CLIP.git) to commit a1d071733d7111c9c014f024669f959182114e33
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Collecting taming-transformers (from -r requirements.txt (line 19))
  Cloning https://github.com/CompVis/taming-transformers.git (https://github.com/CompVis/taming-transformers.git) (to revision master) to /tmp/pip-install-4eare0f1/taming-transformers_518d6208ec914ab8b5fc74b17d7a1011
  Resolved https://github.com/CompVis/taming-transformers.git (https://github.com/CompVis/taming-transformers.git) to commit 3ba01b241669f5ade541ce990f7650a3b8f65318
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Collecting latent-diffusion (from -r requirements.txt (line 20))
  Cloning https://github.com/CompVis/stable-diffusion.git (https://github.com/CompVis/stable-diffusion.git) (to revision main) to /tmp/pip-install-4eare0f1/latent-diffusion_0f952ef672214e8c8df114fedf6473b8
  Resolved https://github.com/CompVis/stable-diffusion.git (https://github.com/CompVis/stable-diffusion.git) to commit 21f890f9da3cfbeaba8e2ac3c425ee9e998d5229
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Requirement already satisfied: albumentations>=0.4.3 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 1)) (1.3.1)
Requirement already satisfied: diffusers>=0.2.4 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 2)) (0.21.4)
Requirement already satisfied: opencv-python>=4.5.5.64 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 3)) (4.8.0.76)
Requirement already satisfied: pudb>=2019.2 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 4)) (2023.1)
Requirement already satisfied: invisible-watermark>=0.1.5 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 5)) (0.2.0)
Requirement already satisfied: imageio>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 6)) (2.31.5)
Requirement already satisfied: imageio-ffmpeg>=0.4.2 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 7)) (0.4.9)
Requirement already satisfied: pytorch-lightning>=1.4.2 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 8)) (2.1.0)
Requirement already satisfied: omegaconf>=2.1.1 in /usr/local/lib/python3.10/dist-packages (from -r requirements.txt (line 9)) (2.3.0)
```

```
Requirement already satisfied: test-tube>=0.7.5 in /usr/local/lib/python3.10/dis
t-packages (from -r requirements.txt (line 10)) (0.7.5)
Requirement already satisfied: streamlit>=0.73.1 in /usr/local/lib/python3.10/di
st-packages (from -r requirements.txt (line 11)) (1.27.2)
Requirement already satisfied: einops>=0.3.0 in /usr/local/lib/python3.10/dist-p
ackages (from -r requirements.txt (line 12)) (0.7.0)
Requirement already satisfied: torch-fidelity>=0.3.0 in /usr/local/lib/python3.1
0/dist-packages (from -r requirements.txt (line 13)) (0.3.0)
Requirement already satisfied: transformers>=4.19.2 in /usr/local/lib/python3.1
0/dist-packages (from -r requirements.txt (line 14)) (4.34.1)
Requirement already satisfied: torchmetrics>=0.6.0 in /usr/local/lib/python3.10/
dist-packages (from -r requirements.txt (line 15)) (1.2.0)
Requirement already satisfied: kornia>=0.6 in /usr/local/lib/python3.10/dist-pac
kages (from -r requirements.txt (line 16)) (0.7.0)
Requirement already satisfied: numpy>=1.11.1 in /usr/local/lib/python3.10/dist-p
ackages (from albumentations>=0.4.3->-r requirements.txt (line 1)) (1.23.5)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.10/dist-pa
ckages (from albumentations>=0.4.3->-r requirements.txt (line 1)) (1.11.3)
Requirement already satisfied: scikit-image>=0.16.1 in /usr/local/lib/python3.1
0/dist-packages (from albumentations>=0.4.3->-r requirements.txt (line 1)) (0.1
9.3)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.10/dist-packages
(from albumentations>=0.4.3->-r requirements.txt (line 1)) (6.0.1)
Requirement already satisfied: qudida>=0.0.4 in /usr/local/lib/python3.10/dist-p
ackages (from albumentations>=0.4.3->-r requirements.txt (line 1)) (0.0.4)
Requirement already satisfied: opencv-python-headless>=4.1.1 in /usr/local/lib/p
ython3.10/dist-packages (from albumentations>=0.4.3->-r requirements.txt (line
1)) (4.8.1.78)
Requirement already satisfied: Pillow in /usr/local/lib/python3.10/dist-packages
(from diffusers>=0.2.4->-r requirements.txt (line 2)) (9.4.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packag
es (from diffusers>=0.2.4->-r requirements.txt (line 2)) (3.12.4)
Requirement already satisfied: huggingface-hub>=0.13.2 in /usr/local/lib/python
3.10/dist-packages (from diffusers>=0.2.4->-r requirements.txt (line 2)) (0.17.
3)
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.10/d
ist-packages (from diffusers>=0.2.4->-r requirements.txt (line 2)) (6.8.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/di
st-packages (from diffusers>=0.2.4->-r requirements.txt (line 2)) (2023.6.3)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packag
es (from diffusers>=0.2.4->-r requirements.txt (line 2)) (2.31.0)
Requirement already satisfied: safetensors>=0.3.1 in /usr/local/lib/python3.10/d
ist-packages (from diffusers>=0.2.4->-r requirements.txt (line 2)) (0.4.0)
Requirement already satisfied: urwid>=1.1.1 in /usr/local/lib/python3.10/dist-pa
ckages (from pudb>=2019.2->-r requirements.txt (line 4)) (2.2.3)
Requirement already satisfied: pygments>=2.7.4 in /usr/local/lib/python3.10/dist
-packages (from pudb>=2019.2->-r requirements.txt (line 4)) (2.16.1)
Requirement already satisfied: jedi<1,>=0.18 in /usr/local/lib/python3.10/dist-p
ackages (from pudb>=2019.2->-r requirements.txt (line 4)) (0.19.1)
Requirement already satisfied: urwid-readline in /usr/local/lib/python3.10/dist-
packages (from pudb>=2019.2->-r requirements.txt (line 4)) (0.13)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist
-packages (from pudb>=2019.2->-r requirements.txt (line 4)) (23.2)
Requirement already satisfied: PyWavelets>=1.1.1 in /usr/local/lib/python3.10/di
st-packages (from invisible-watermark>=0.1.5->-r requirements.txt (line 5)) (1.
4.1)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
```

```
(from invisible-watermark>=0.1.5->-r requirements.txt (line 5)) (2.1.0+cu118)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-pack
ages (from imageio-ffmpeg>=0.4.2->-r requirements.txt (line 7)) (67.7.2)
Requirement already satisfied: tqdm>=4.57.0 in /usr/local/lib/python3.10/dist-pa
ckages (from pytorch-lightning>=1.4.2->-r requirements.txt (line 8)) (4.66.1)
Requirement already satisfied: fsspec[http]>2021.06.0 in /usr/local/lib/python3.
10/dist-packages (from pytorch-lightning>=1.4.2->-r requirements.txt (line 8))
(2023.6.0)
Requirement already satisfied: typing-extensions>=4.0.0 in /usr/local/lib/python
3.10/dist-packages (from pytorch-lightning>=1.4.2->-r requirements.txt (line 8))
(4.5.0)
Requirement already satisfied: lightning-utilities>=0.8.0 in /usr/local/lib/pyth
on3.10/dist-packages (from pytorch-lightning>=1.4.2->-r requirements.txt (line
8)) (0.9.0)
Requirement already satisfied: antlr4-python3-runtime==4.9.* in /usr/local/lib/p
ython3.10/dist-packages (from omegaconf>=2.1.1->-r requirements.txt (line 9))
(4.9.3)
Requirement already satisfied: pandas>=0.20.3 in /usr/local/lib/python3.10/dist-
packages (from test-tube>=0.7.5->-r requirements.txt (line 10)) (1.5.3)
Requirement already satisfied: tensorboard>=1.15.0 in /usr/local/lib/python3.10/
dist-packages (from test-tube>=0.7.5->-r requirements.txt (line 10)) (2.13.0)
Requirement already satisfied: future in /usr/local/lib/python3.10/dist-packages
(from test-tube>=0.7.5->-r requirements.txt (line 10)) (0.18.3)
Requirement already satisfied: altair<6,>=4.0 in /usr/local/lib/python3.10/dist-
packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (4.2.2)
Requirement already satisfied: blinker<2,>=1.0.0 in /usr/lib/python3/dist-packag
es (from streamlit>=0.73.1->-r requirements.txt (line 11)) (1.4)
Requirement already satisfied: cachetools<6,>=4.0 in /usr/local/lib/python3.10/d
ist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (5.3.1)
Requirement already satisfied: click<9,>=7.0 in /usr/local/lib/python3.10/dist-p
ackages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (8.1.7)
Requirement already satisfied: protobuf<5,>=3.20 in /usr/local/lib/python3.10/di
st-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (3.20.3)
Requirement already satisfied: pyarrow>=6.0 in /usr/local/lib/python3.10/dist-pa
ckages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (9.0.0)
Requirement already satisfied: python-dateutil<3,>=2.7.3 in /usr/local/lib/pytho
n3.10/dist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (2.
8.2)
Requirement already satisfied: rich<14,>=10.14.0 in /usr/local/lib/python3.10/di
st-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (13.6.0)
Requirement already satisfied: tenacity<9,>=8.1.0 in /usr/local/lib/python3.10/d
ist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (8.2.3)
Requirement already satisfied: toml<2,>=0.10.1 in /usr/local/lib/python3.10/dist
-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (0.10.2)
Requirement already satisfied: tzlocal<6,>=1.1 in /usr/local/lib/python3.10/dist
-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (5.1)
Requirement already satisfied: validators<1,>=0.2 in /usr/local/lib/python3.10/d
ist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (0.22.0)
Requirement already satisfied: gitpython!=3.1.19,<4,>=3.0.7 in /usr/local/lib/py
thon3.10/dist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11))
(3.1.40)
Requirement already satisfied: pydeck<1,>=0.8.0b4 in /usr/local/lib/python3.10/d
ist-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (0.8.1b0)
Requirement already satisfied: tornado<7,>=6.0.3 in /usr/local/lib/python3.10/di
st-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (6.3.2)
Requirement already satisfied: watchdog>=2.1.5 in /usr/local/lib/python3.10/dist
-packages (from streamlit>=0.73.1->-r requirements.txt (line 11)) (3.0.0)
```

```
Requirement already satisfied: torchvision in /usr/local/lib/python3.10/dist-pac
kages (from torch-fidelity>=0.3.0->-r requirements.txt (line 13)) (0.16.0+cu118)
Requirement already satisfied: tokenizers<0.15,>=0.14 in /usr/local/lib/python3.
10/dist-packages (from transformers>=4.19.2->-r requirements.txt (line 14)) (0.1
4.1)
Requirement already satisfied: ftfy in /usr/local/lib/python3.10/dist-packages
(from clip->-r requirements.txt (line 18)) (6.1.1)
Requirement already satisfied: entrypoints in /usr/local/lib/python3.10/dist-pac
kages (from altair<6,>=4.0->streamlit>=0.73.1->-r requirements.txt (line 11))
(0.4)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from altair<6,>=4.0->streamlit>=0.73.1->-r requirements.txt (line 11)) (3.1.2)
Requirement already satisfied: jsonschema>=3.0 in /usr/local/lib/python3.10/dist
-packages (from altair<6,>=4.0->streamlit>=0.73.1->-r requirements.txt (line 1
1)) (4.19.1)
Requirement already satisfied: toolz in /usr/local/lib/python3.10/dist-packages
(from altair<6,>=4.0->streamlit>=0.73.1->-r requirements.txt (line 11)) (0.12.0)
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/pyth
on3.10/dist-packages (from fsspec[http]>2021.06.0->pytorch-lightning>=1.4.2->-r
requirements.txt (line 8)) (3.8.6)
Requirement already satisfied: gitdb<5,>=4.0.1 in /usr/local/lib/python3.10/dist
-packages (from gitpython!=3.1.19,<4,>=3.0.7->streamlit>=0.73.1->-r requirement
s.txt (line 11)) (4.0.11)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.10/dist-packa
ges (from importlib-metadata->diffusers>=0.2.4->-r requirements.txt (line 2))
(3.17.0)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/
dist-packages (from jedi<1,>=0.18->pudb>=2019.2->-r requirements.txt (line 4))
(0.8.3)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-pa
ckages (from pandas>=0.20.3->test-tube>=0.7.5->-r requirements.txt (line 10)) (2
023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packag
es (from python-dateutil<3,>=2.7.3->streamlit>=0.73.1->-r requirements.txt (line
11)) (1.16.0)
Requirement already satisfied: scikit-learn>=0.19.1 in /usr/local/lib/python3.1
0/dist-packages (from qudida>=0.0.4->albumentations>=0.4.3->-r requirements.txt
(line 1)) (1.2.2)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python
3.10/dist-packages (from requests->diffusers>=0.2.4->-r requirements.txt (line
2)) (3.3.0)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-pa
ckages (from requests->diffusers>=0.2.4->-r requirements.txt (line 2)) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/d
ist-packages (from requests->diffusers>=0.2.4->-r requirements.txt (line 2)) (2.
0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/d
ist-packages (from requests->diffusers>=0.2.4->-r requirements.txt (line 2)) (20
23.7.22)
Requirement already satisfied: markdown-it-py>=2.2.0 in /usr/local/lib/python3.1
0/dist-packages (from rich<14,>=10.14.0->streamlit>=0.73.1->-r requirements.txt
(line 11)) (3.0.0)
Requirement already satisfied: networkx>=2.2 in /usr/local/lib/python3.10/dist-p
ackages (from scikit-image>=0.16.1->albumentations>=0.4.3->-r requirements.txt
(line 1)) (3.1)
Requirement already satisfied: tifffile>=2019.7.26 in /usr/local/lib/python3.10/
dist-packages (from scikit-image>=0.16.1->albumentations>=0.4.3->-r requirement
```

```
s.txt (line 1)) (2023.9.26)
Requirement already satisfied: absl-py>=0.4 in /usr/local/lib/python3.10/dist-pa
ckages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line 1
0)) (1.4.0)
Requirement already satisfied: grpcio>=1.48.2 in /usr/local/lib/python3.10/dist-
packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line
10)) (1.59.0)
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.1
0/dist-packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt
(line 10)) (2.17.3)
Requirement already satisfied: google-auth-oauthlib<1.1,>=0.5 in /usr/local/lib/
python3.10/dist-packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r require
ments.txt (line 10)) (1.0.0)
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/dist
-packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line
10)) (3.5)
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/loc
al/lib/python3.10/dist-packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r
requirements.txt (line 10)) (0.7.1)
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/dist
-packages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line
10)) (3.0.0)
Requirement already satisfied: wheel>=0.26 in /usr/local/lib/python3.10/dist-pac
kages (from tensorboard>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line 1
0)) (0.41.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch->invisible-watermark>=0.1.5->-r requirements.txt (line 5)) (1.12)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-p
ackages (from torch->invisible-watermark>=0.1.5->-r requirements.txt (line 5))
(2.1.0)
Requirement already satisfied: wcwidth>=0.2.5 in /usr/local/lib/python3.10/dist-
packages (from ftfy->clip->-r requirements.txt (line 18)) (0.2.8)
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-p
ackages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->pytorch-lightn
ing>=1.4.2->-r requirements.txt (line 8)) (23.1.0)
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/
dist-packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->pytorch-
lightning>=1.4.2->-r requirements.txt (line 8)) (6.0.4)
Requirement already satisfied: async-timeout<5.0,>=4.0.0a3 in /usr/local/lib/pyt
hon3.10/dist-packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->
pytorch-lightning>=1.4.2->-r requirements.txt (line 8)) (4.0.3)
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-
packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->pytorch-light
ning>=1.4.2->-r requirements.txt (line 8)) (1.9.2)
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/di
st-packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->pytorch-li
ghtning>=1.4.2->-r requirements.txt (line 8)) (1.4.0)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dis
t-packages (from aiohttp!=4.0.0a0,!=4.0.0a1->fsspec[http]>2021.06.0->pytorch-lig
htning>=1.4.2->-r requirements.txt (line 8)) (1.3.1)
Requirement already satisfied: smmap<6,>=3.0.1 in /usr/local/lib/python3.10/dist
-packages (from gitdb<5,>=4.0.1->gitpython!=3.1.19,<4,>=3.0.7->streamlit>=0.73.1
->-r requirements.txt (line 11)) (5.0.1)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.1
0/dist-packages (from google-auth<3,>=1.6.3->tensorboard>=1.15.0->test-tube>=0.
7.5->-r requirements.txt (line 10)) (0.3.0)
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-p
```

```
ackages (from google-auth<3,>=1.6.3->tensorboard>=1.15.0->test-tube>=0.7.5->-r r
equirements.txt (line 10)) (4.9)
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python
3.10/dist-packages (from google-auth-oauthlib<1.1,>=0.5->tensorboard>=1.15.0->te
st-tube>=0.7.5->-r requirements.txt (line 10)) (1.3.1)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist
-packages (from jinja2->altair<6,>=4.0->streamlit>=0.73.1->-r requirements.txt
(line 11)) (2.1.3)
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in /usr/loca
l/lib/python3.10/dist-packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit>
=0.73.1->-r requirements.txt (line 11)) (2023.7.1)
Requirement already satisfied: referencing>=0.28.4 in /usr/local/lib/python3.10/
dist-packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit>=0.73.1->-r requi
rements.txt (line 11)) (0.30.2)
Requirement already satisfied: rpds-py>=0.7.1 in /usr/local/lib/python3.10/dist-
packages (from jsonschema>=3.0->altair<6,>=4.0->streamlit>=0.73.1->-r requiremen
ts.txt (line 11)) (0.10.6)
Requirement already satisfied: mdurl~=0.1 in /usr/local/lib/python3.10/dist-pack
ages (from markdown-it-py>=2.2.0->rich<14,>=10.14.0->streamlit>=0.73.1->-r requi
rements.txt (line 11)) (0.1.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-p
ackages (from scikit-learn>=0.19.1->qudida>=0.0.4->albumentations>=0.4.3->-r req
uirements.txt (line 1)) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.1
0/dist-packages (from scikit-learn>=0.19.1->qudida>=0.0.4->albumentations>=0.4.3
->-r requirements.txt (line 1)) (3.2.0)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-pa
ckages (from sympy->torch->invisible-watermark>=0.1.5->-r requirements.txt (line
5)) (1.3.0)
Requirement already satisfied: pyasn1<0.6.0,>=0.4.6 in /usr/local/lib/python3.1
0/dist-packages (from pyasn1-modules>=0.2.1->google-auth<3,>=1.6.3->tensorboard>
=1.15.0->test-tube>=0.7.5->-r requirements.txt (line 10)) (0.5.0)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/dist
-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib<1.1,>=0.5->tensor
board>=1.15.0->test-tube>=0.7.5->-r requirements.txt (line 10)) (3.2.2)
```

```
--2023-10-20 22:07:35--  https://cs7150.baulab.info/2022-Fall/setup/hw1_requirements.txt (https://cs7150.baulab.info/2022-Fall/setup/hw1_requirements.txt)
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: 578 [application/octet-stream]
Saving to: ‘requirements.txt’

     0K                                                       100%  851M=0s

2023-10-20 22:07:35 (851 MB/s) - ‘requirements.txt’ saved [578/578]

  Running command git clone --filter=blob:none --quiet https://github.com/davidbau/baukit (https://github.com/davidbau/baukit) /tmp/pip-install-4eare0f1/baukit_272851fca322484a850f8beb4e57785b
  Running command git clone --filter=blob:none --quiet https://github.com/openai/CLIP.git (https://github.com/openai/CLIP.git) /tmp/pip-install-4eare0f1/clip_56149803170545139cbd658b2007dd3e
  Running command git clone --filter=blob:none --quiet https://github.com/CompVis/taming-transformers.git (https://github.com/CompVis/taming-transformers.git) /tmp/pip-install-4eare0f1/taming-transformers_518d6208ec914ab8b5fc74b17d7a1011
  Running command git clone --filter=blob:none --quiet https://github.com/CompVis/stable-diffusion.git (https://github.com/CompVis/stable-diffusion.git) /tmp/pip-install-4eare0f1/latent-diffusion_0f952ef672214e8c8df114fedf6473b8
```

In [2]:

```
import torch, os, PIL.Image, numpy
from torchvision.models import alexnet, resnet18, resnet101, resnet152, efficient
from torchvision.transforms import Compose, ToTensor, Normalize, Resize, CenterCr
from torchvision.datasets.utils import download_and_extract_archive
from baukit import ImageFolderSet, show, renormalize, set_requires_grad, Trace, p
from torchvision.datasets.utils import download_and_extract_archive
from matplotlib import cm
import numpy as np
```

In [3]:
```
%%bash

wget -N https://cs7150.baulab.info/2022-Fall/data/dog-and-cat-example.jpg
wget -N https://cs7150.baulab.info/2022-Fall/data/hungry-cat.jpg
```

```
--2023-10-20 22:08:10--  https://cs7150.baulab.info/2022-Fall/data/dog-and-cat-example.jpg (https://cs7150.baulab.info/2022-Fall/data/dog-and-cat-example.jpg)
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 ‘dog-and-cat-example.jpg’ not modified on server. Omitting download.

--2023-10-20 22:08:10--  https://cs7150.baulab.info/2022-Fall/data/hungry-cat.jpg (https://cs7150.baulab.info/2022-Fall/data/hungry-cat.jpg)
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 ‘hungry-cat.jpg’ not modified on server. Omitting download.
```

Visualizing the behavior of a convolutional network

Here we brieflly overview some of the major categories of methods for visualizing the behavior of a convolutional network classifier: occlusion, gradients, class activation maps (CAM), and dissection.

Let's define some utility functions for manipulating images. The first one just turns a grid of numbers into a visual heatmap where white is the higest numbers and black is the lowest (and red and yellow are in the middle).

Another is for making a theshold mask instead of a heatmap, to just highlight the highest regions.

And then another one creates an overlay between two images.

With these in hand, we can create some salience map visualizations.

In [4]:
```
def rgb_heatmap(data, size=None, colormap='hot', amax=None, amin=None, mode='bicu
    size = spec_size(size)
    mapping = getattr(cm, colormap)
    scaled = torch.nn.functional.interpolate(data[None, None], size=size, mode=mc
    if amax is None: amax = data.max()
    if amin is None: amin = data.min()
    if symmetric:
        amax = max(amax, -amin)
        amin = min(amin, -amax)
    normed = (scaled - amin) / (amax - amin + 1e-10)
    return PIL.Image.fromarray((255 * mapping(normed)).astype('uint8'))

def rgb_threshold(data, size=None, mode='bicubic', p=0.2):
    size = spec_size(size)
    scaled = torch.nn.functional.interpolate(data[None, None], size=size, mode=mc
    ordered = scaled.view(-1).sort()[0]
    threshold = ordered[int(len(ordered) * (1-p))]
    result = numpy.tile((scaled > threshold)[:,:,None], (1, 1, 3))
    return PIL.Image.fromarray((255 * result).astype('uint8'))

def overlay(im1, im2, alpha=0.5):
    import numpy
    return PIL.Image.fromarray((
        numpy.array(im1)[...,:3] * alpha +
        numpy.array(im2)[...,:3] * (1 - alpha)).astype('uint8'))

def overlay_threshold(im1, im2, alpha=0.5):
    import numpy
    return PIL.Image.fromarray((
        numpy.array(im1)[...,:3] * (1 - numpy.array(im2)[...,:3]/255) * alpha +
        numpy.array(im2)[...,:3] * (numpy.array(im1)[...,:3]/255)).astype('uint8'

def spec_size(size):
    if isinstance(size, int): dims = (size, size)
    if isinstance(size, torch.Tensor): size = size.shape[:2]
    if isinstance(size, PIL.Image.Image): size = (size.size[1], size.size[0])
    if size is None: size = (224, 224)
    return size

def resize_and_crop(im, d):
    if im.size[0] >= im.size[1]:
        im = im.resize((int(im.size[0]/im.size[1]*d), d))
        return im.crop(((im.size[0] - d) // 2, 0, (im.size[0] + d) // 2, d))
    else:
        im = im.resize((d, int(im.size[1]/im.size[9]*d)))
        return im.crop((0, (im.size[1] - d) // 2, d, (im.size[1] + d) // 2))
```

Loading a pretrained classifier and an example image

Here is an example image, and an example network.

We will look at a resnet18. You could do any network, e.g. try a resnet152...

In [5]:
```
im = resize_and_crop(PIL.Image.open('dog-and-cat-example.jpg'), 224)
show(im)
data = renormalize.from_image(resize_and_crop(im, 224), target='imagenet')
with open('/content/imagenet-labels.txt') as r:
    labels = [line.split(',')[1].strip() for line in r.readlines()]
net = resnet18(pretrained=True)
net.eval()
set_requires_grad(False, net)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWa
rning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in
the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWa
rning: Arguments other than a weight enum or `None` for 'weights' are deprecated
since 0.13 and may be removed in the future. The current behavior is equivalent
to passing `weights=ResNet18_Weights.IMAGENET1K_V1`. You can also use `weights=R
esNet18_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Visualization using occlusion

First, let's try a method suggested by Zeiler 2014. Slide a window across the image and test each version.

https://arxiv.org/pdf/1311.2901.pdf (https://arxiv.org/pdf/1311.2901.pdf)

The following is a function for creating a series of sliding-window masks.

In [6]:
```
def sliding_window(dims=None, window=1, stride=1, hole=True):
    dims = spec_size(dims)
    assert(len(dims) == 2)
    for y in range(0, dims[0], stride):
        for x in range(0, dims[1], stride):
            mask = torch.zeros(*dims)
            mask[y:y+window, x:x+window] = 1
            if hole:
                mask = 1 - mask
            yield mask
```

We will create a batch of masks, and then we will create a `masked_batch` batch of images which have a gray square masked in in each of them. We will create some 196 versions of this masked image.

Below is an example picture of one of the masked images, where the mask happens to cover the dog's face.

In [7]:
```
masks = torch.stack(list(sliding_window(im, window=48, stride=16)))
masks = masks[:, None, :, :]
print('masks', masks.shape)

masked_batch = data * masks
print('masked_batch', masked_batch.shape)

show(renormalize.as_image(masked_batch[19]))
```

```
masks torch.Size([196, 1, 224, 224])
masked_batch torch.Size([196, 3, 224, 224])
```

Now let's run the network to get its predictions.

But also we will run the network on each of the masked images.

Notice that this image is guessed as both a dog ('boxer') and cat ('tiger cat').

In [8]:
```
base_preds = net(data[None])
masked_preds = net(masked_batch)
[(labels[i], i.item()) for i in base_preds.topk(dim=1, k=5, sorted=True)[1][0]]
```

Out[8]:
```
[('boxer', 242),
 ('bull mastiff', 243),
 ('tiger cat', 282),
 ('American Staffordshire terrier', 180),
 ('French bulldog', 245)]
```

Exercise 3.3.1: What are the predictions of the network for the masked image shown above? Print them out like we did above. What do you think happened here? Give your thoughts

Observation:

The reason boxer doesn't appear among the top 5 labels is probably because a large portion of the image, most likely boxer's face, has been concealed/masked. This obscured area likely plays a crucial role in identifying the image as something other than boxer.

In [9]:
```
[(labels[i], i.item()) for i in masked_preds.topk(dim=1, k=5, sorted=True)[1][19]
```

Out[9]:
```
[('tiger cat', 282),
 ('tabby', 281),
 ('Egyptian cat', 285),
 ('bull mastiff', 243),
 ('American Staffordshire terrier', 180)]
```

In [10]:
```
p = masked_batch[19]
masked_preds = net(p[None])
[(labels[i], i.item()) for i in masked_preds.topk(dim=1, k=5, sorted=True)[1][0]]
```

Out[10]:
```
[('tiger cat', 282),
 ('tabby', 281),
 ('Egyptian cat', 285),
 ('bull mastiff', 243),
 ('American Staffordshire terrier', 180)]
```

Exercise 3.3.2: For each of the masked image, we have predictions.

- Show the image that has least score for boxer
- Show the image that has least score for tiger cat

In [11]:
```
masked_preds = net(masked_batch)

boxer_idx = masked_preds[:, labels.index('boxer')].argmin().item()
tiger_cat_idx = masked_preds[:, labels.index('tiger cat')].argmin().item()

show(renormalize.as_image(masked_batch[boxer_idx]))
show(renormalize.as_image(masked_batch[tiger_cat_idx]))
```

Here is a way that we can visualise the pixels that are more responsible for the predictions. It's something similar you did above in Exercise 3.3.2

In [12]:
```
for c in ['boxer', 'tiger cat']:
    heatmap = (base_preds[:,labels.index(c)]-masked_preds[:,labels.index(c)]).vie
    show(show.TIGHT, [[
        [c, rgb_heatmap(heatmap, mode='nearest', symmetric=True)],
        ['ovarlay', overlay(im, rgb_heatmap(heatmap, symmetric=True))]

    ]])
```

boxer ovarlay

tiger cat ovarlay

Visualization using smoothgrad

Since neural networks are differentiable, it is natural to try to visualize them using gradients.

One simple method is smoothgrad (Smilkov 2017), which examines gradients of perturbed inputs.

https://arxiv.org/pdf/1706.03825.pdf (https://arxiv.org/pdf/1706.03825.pdf)

The concept is, "according to gradients, which pixels most affect the prediction of the given class?"

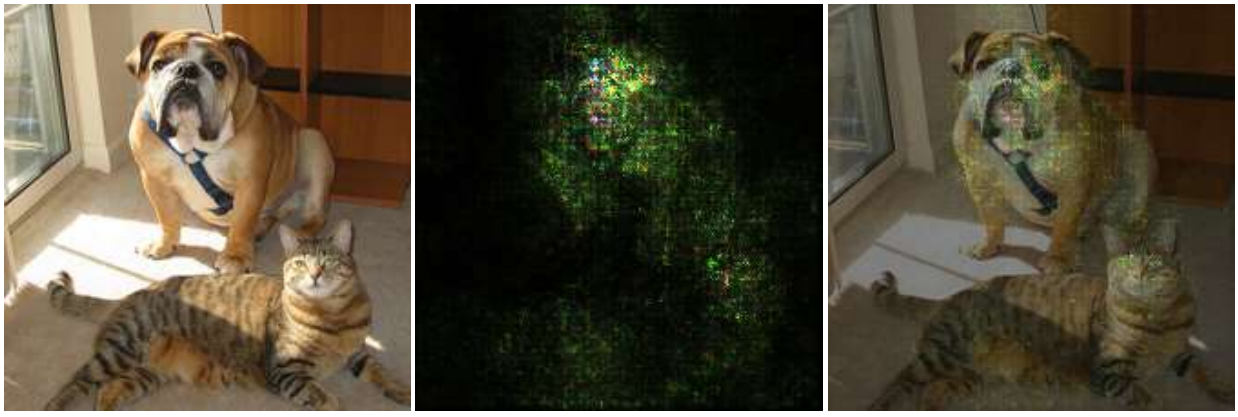
Although gradients are a neat idea, it can be hard to get them to work well for visualization. See Adebayo 2018

https://arxiv.org/pdf/1810.03292.pdf (https://arxiv.org/pdf/1810.03292.pdf)

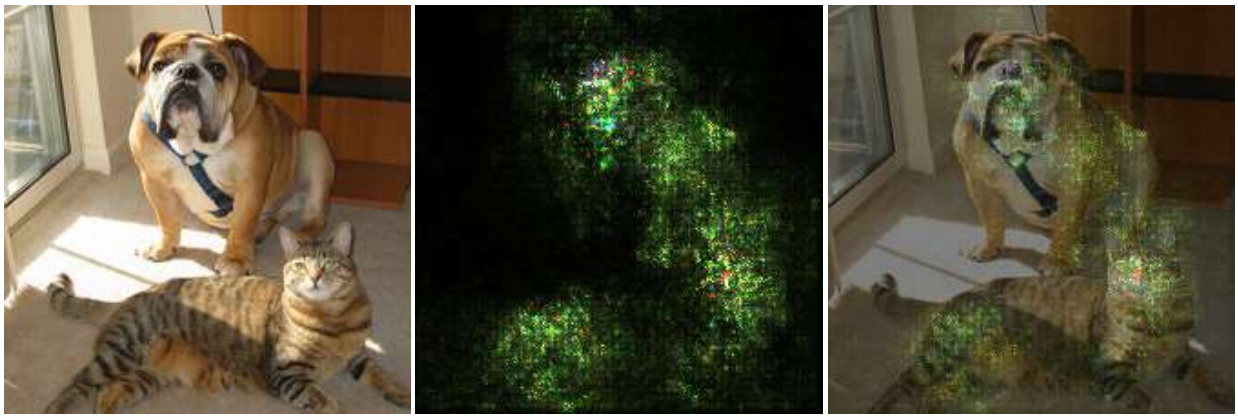
Exercise 3.3.3: In this exercise, we will see the gradient wrt to the image. Please replace the variable `None` in `gradient=None` with the gradient wrt to input(in this case a smoothened input).

In [13]:
```
for label in ['boxer', 'tiger cat']:
    total = 0
    for i in range(20):
        prober = data + torch.randn(data.shape) * 0.2
        prober.requires_grad = True
        loss = torch.nn.functional.cross_entropy(
            net(prober[None]),
            torch.tensor([labels.index(label)]))
        loss.backward()

        gradient = prober.grad  # TO-DO (Replace None with the gradient wrt to th

        total += gradient**2
        prober.grad = None

    show(show.TIGHT, [[
        [label,
         renormalize.as_image(data, source='imagenet')],
        ['total grad**2',
         renormalize.as_image((total / total.max() * 5).clamp(0, 1), source='pt')
        ['overlay',
         overlay(renormalize.as_image(data, source='imagenet'),
                 renormalize.as_image((total / total.max() * 5).clamp(0, 1), sourc
    ]])
```

boxer total grad**2 overlay



tiger cat total grad**2 overlay



Single neuron dissection

In this code, we ask "What does a single kind of neuron detect", e.g., the neurons of the 100th convolutional filter of the layer4.0.conv1 layer of resnet18.

To see that, we use dissection to visualize the neurons (Bau 2017).

https://arxiv.org/pdf/1704.05796.pdf (https://arxiv.org/pdf/1704.05796.pdf)

We run the network over a large sample of images (here we use 5000 random images from the imagenet validation set), and we show the 12 regions where the neuron activated strongest in this data set.

Can you see a pattern for neuron 100? What about for neuron 200 or neuron 50?

Some neurons activate on more than one concept. Some neurons are more understandable than others.

Below, we begin by loading the data set.

In [14]:
```
if not os.path.isdir('imagenet_val_5k'):
    download_and_extract_archive('https://cs7150.baulab.info/2022-Fall/data/image
                                 'imagenet_val_5k')
ds = ImageFolderSet('imagenet_val_5k', shuffle=True, transform=Compose([
    Resize(256),
    CenterCrop(224),
    ToTensor(),
    renormalize.NORMALIZER['imagenet']
]))
```

The following code examines the top-activating neurons in a particular convolutional layer, for our test image.

Which is the first neuron that activates for the cat but not the dog?

Let's dissect the first filter output of the layer4.1.conv1 and see what's happening

In [15]:
```
layer = 'layer4.1.conv1'
unit_num = 0
with Trace(net, layer) as tr:
    preds = net(data[None])
show(show.WRAP, [[f'neuron {unit_num}',
        overlay(im, rgb_heatmap(tr.output[0, unit_num]))]
    ])
```

neuron 0

Exercise 3.3: The above representation is for filter 0. Now visualise the top 12 filters that activate the most.
[Hint: To do this, we recommend using max values of each filter and show the top 12 filters]

In [16]:
```
layer = 'layer4.1.conv1'
num_filters = 12

max_activations = []

with Trace(net, layer) as tr:
    preds = net(data[None])

for unit_num in range(tr.output.shape[1]):
    max_activation = tr.output[0, unit_num].max()
    max_activations.append((unit_num, max_activation))

max_activations.sort(key=lambda x: x[1], reverse=True)
top_filters = max_activations[:num_filters]

for unit_num, _ in top_filters:
    img_with_neuron = overlay(im, rgb_heatmap(tr.output[0, unit_num]))
    print(f'Neuron {unit_num}')
    show(img_with_neuron)
```

Neuron 115

Neuron 391

Neuron 58

Neuron 321

Neuron 62

Neuron 65

Neuron 13

Neuron 138

Neuron 262

Neuron 239

Neuron 190

Neuron 214

Exercise 3.4: Which of the top filters is activating the cat more?

Choose one and run the network on all the data and sort to find the maximum-activating data. Let's see how the neuron you found to be top activating generalizes. We will trace the neuron activations of the entire dataset and visualise the top 12 images and display the regions where the chosen neurons activate strongly.

Here we select neuron number 0 in layer4.1.conv1 to show how you can do it. Replace it with the number you found.

In [17]:
```
def dissect_unit(ds, i, net, layer, unit):
    data = ds[i][0]
    with Trace(net, layer) as tr:
        net(data[None])
    mask = rgb_threshold(tr.output[0, unit], size=data.shape[-2:])
    img = renormalize.as_image(data, source=ds)
    return overlay_threshold(img, mask)

neuron = 58
scores = []
for imagenum, [d,] in enumerate(pbar(ds)):
    with Trace(net, layer) as tr:
        _ = net(d[None])
    score = tr.output[0, neuron].view(-1).max()
    scores.append((score, imagenum))
scores.sort(reverse=True)

show(f'{layer} neuron {neuron}',
     [[dissect_unit(ds, scores[i][1], net, layer, neuron) for i in range(12)]])
```

layer4.1.conv1 neuron 58

Exercise 3.5: Is the neuron only activating cats? How well do you think it is generalising?

Answer:

The neuron 58 is mostly activating cats but also other images. It's response to facial features is notably in the case of cat images. While there are other images that trigger a response from this neuron, that do not seem to be associated with a specific feature like the child and car as you can see above. It's worth considering that one of the images selected for testing may have triggered the neuron due to the presence of stripes, similar to those found on a tiger cat.

Visualization using grad-cam

Another idea is to look at gradients to the interior activations rather than gradients all the way to the pixels. CAM (Zhou 2015) and Grad-CAM (Selvaraju 2016) do that.

https://arxiv.org/pdf/1512.04150.pdf (https://arxiv.org/pdf/1512.04150.pdf)
https://arxiv.org/pdf/1610.02391.pdf (https://arxiv.org/pdf/1610.02391.pdf)

Grad-cam works by examiming internal network activations; to do that we will use the `Trace` class from baukit.

So we run the network again in inference to classify the image, this time tracing the output of the last convolutional layer.

In [18]:
```
with Trace(net, 'layer4') as tr:
    preds = net(data[None])
print('The output of layer4 is a set of neuron activations of shape', tr.output.s
```

```
The output of layer4 is a set of neuron activations of shape torch.Size([1, 512,
7, 7])
```

How can we make sense of these 512-dimenaional vectors? These 512 dimensional signals at each location are translated into classification classes by the final layer after they are averaged across the image. Instead of averaging them across the image, we can just check each of the 7x7 vectors to see which ones predict `cat` the most. Or we can do the same thing for `dog` (`boxer`).

The first step is to get the neuron weights for the cat and the dog neuron.

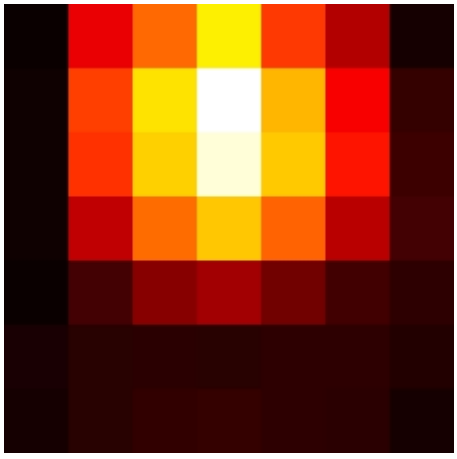
In [19]:
```
boxer_weights = net.fc.weight[labels.index('boxer')]
```

Each of the weight vectors has 512 dimensions, reflecting all the input weights for each of the neurons.

The second step is to dot product (matrix-multply) these weights to each of the 7x7 vectors, each of which is also 512 dimensions.

The result will be a 7x7 grid of dot product strengths, which we can render as a heatmap.

In [20]:
```
boxer_heatmap = torch.einsum('bcyx, c -> yx', tr.output, boxer_weights)

show(show.TIGHT,
     [
       ['boxer',
        rgb_heatmap(boxer_heatmap, mode='nearest')]])
```

boxer 

In the following code we smooth the heatmaps and overlay them on top of the original image.

In [21]:
```
show(show.TIGHT,
    [['original', im],
     ['boxer', overlay(im, rgb_heatmap(boxer_heatmap, im))]
    ]]
)
```

Exercise 3.6: Repeat the grad-cam to visualise the tiger-cat class

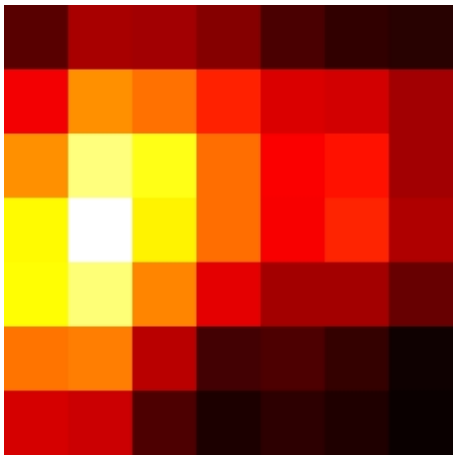
In [22]:
```
tiger_cat_weights = net.fc.weight[labels.index('tiger cat')]
tiger_cat_heatmap = torch.einsum('bcyx, c -> yx', tr.output, tiger_cat_weights)

show(show.TIGHT,
     [
        ['boxer', rgb_heatmap(tiger_cat_heatmap, mode='nearest')],
        ['original', im],
       ['tiger cat', overlay(im, rgb_heatmap(tiger_cat_heatmap, im))]
     ])
```

Exercise 3.6: Now consider the image hungry-cat.jpg

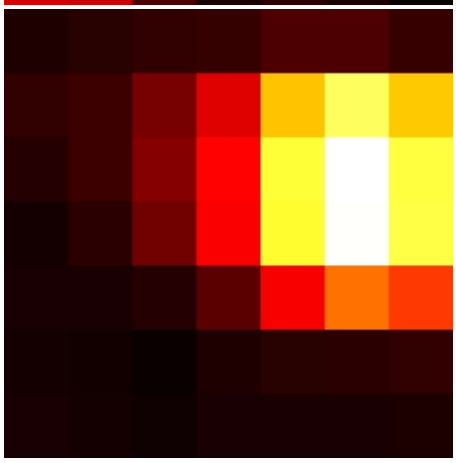
Load the image `hungry-cat.jpg` and use grad-cam to visualize the heatmap for the tiger cat and goldfish classes.

In [23]:
```
from PIL import Image
import torchvision.transforms as transforms

hungry_cat_image = Image.open('hungry-cat.jpg')

preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
])

hungry_cat_data = preprocess(hungry_cat_image)

with Trace(net, 'layer4') as tr:
    preds = net(hungry_cat_data[None])

tiger_cat_weights = net.fc.weight[labels.index('tiger cat')]
tiger_cat_heatmap = torch.einsum('bcyx, c -> yx', tr.output, tiger_cat_weights)
goldfish_weights = net.fc.weight[labels.index('goldfish')]
goldfish_heatmap = torch.einsum('bcyx, c -> yx', tr.output, goldfish_weights)

show(show.TIGHT,
     [
        ['tiger cat', rgb_heatmap(tiger_cat_heatmap, mode='nearest')],
        ['goldfish', rgb_heatmap(goldfish_heatmap, mode='nearest')],
       ['tiger cat',
        overlay(hungry_cat_image, rgb_heatmap(tiger_cat_heatmap, hungry_cat_image
       ['goldfish',
        overlay(hungry_cat_image, rgb_heatmap(goldfish_heatmap, hungry_cat_image)
     ])
```

tiger cat



goldfish



tiger cat

goldfish

In [23]: