This can be run [run on Google Colab using this link (https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.1-Classifiers.ipynb)](https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.1-Classifiers.ipynb)

# 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

In [1]:
```
# Uncomment the below line to install wandb (optinal)
# !pip install wandb
# Uncomment the below line to install torchinfo (https://github.com/TylerYep/
# !pip install torchinfo
```

In [2]:
```
%%bash

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

```
--2023-10-19 22:02:51--  https://cs7150.baulab.info/2022-Fall/data/mnist-cl
assify.pth (https://cs7150.baulab.info/2022-Fall/data/mnist-classify.pth)
Connecting to 10.99.0.130:3128... connected.
Proxy request sent, awaiting response... 304 Not Modified
File ‘mnist-classify.pth’ not modified on server. Omitting download.
```

In [3]:
```
# 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
```

In [4]:
```
import os
os.environ["WANDB_API_KEY"] = "76649aed02536ffcaa99378dd2c2cfa5fd558e78"
```

In [5]:
```
# Create an account at https://wandb.ai/site and paste the api key here (opti
import wandb
wandb.init(project="hw3.1-ConvNets")
```

```
Failed to detect the name of this notebook, you can set it manually with th
e WANDB_NOTEBOOK_NAME environment variable to enable code saving.
wandb: Currently logged in as: surya-thiru001 (dl-hw). Use `wandb login --r
elogin` to force relogin
```

Tracking run with wandb version 0.15.12

Run data is saved locally in `/home/krishnamurthy.sur/projects/DLHW/hw3/wandb/run-20231019_220303-2eedyixh`

Syncing run **rosy-snowflake-7 (https://wandb.ai/dl-hw/hw3.1-ConvNets/runs/2eedyixh)** to Weights & Biases (https://wandb.ai/dl-hw/hw3.1-ConvNets) (docs (https://wandb.me/run))

View project at https://wandb.ai/dl-hw/hw3.1-ConvNets (https://wandb.ai/dl-hw/hw3.1-ConvNets)

View run at https://wandb.ai/dl-hw/hw3.1-ConvNets/runs/2eedyixh (https://wandb.ai/dl-hw/hw3.1-ConvNets/runs/2eedyixh)

Out[5]: Display W&B run

## Some helper functions to view network parameters

In [6]:
```
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}!')
```

In [7]:
```
def view_network_shapes(model, input_shape):
    print(summary(model, input_size=input_shape))
```

## Fully Connected Network for Image Classification

Let's build a simple fully connected network!

In [8]:
```
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
```

In [9]:
```
fc_net = simple_fc_net()
```

In [10]:
```
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!
```

In [11]:
```
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!
```

In [12]:
```
from torchinfo import summary
summary(fc_net, input_size=(1, 1, 28,28))
```

```
/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torch/nn/module
s/container.py:217: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
```

Out[12]:

```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
Sequential                               [1, 10]                   --
├─Flatten: 1-1                           [1, 784]                  --
├─Linear: 1-2                            [1, 6272]                 4,923,52
0
├─ReLU: 1-3                              [1, 6272]                 --
├─Linear: 1-4                            [1, 3136]                 19,672,1
28
├─ReLU: 1-5                              [1, 3136]                 --
├─Linear: 1-6                            [1, 1568]                 4,918,81
6
├─ReLU: 1-7                              [1, 1568]                 --
├─Linear: 1-8                            [1, 288]                  451,872
├─ReLU: 1-9                              [1, 288]                  --
├─Linear: 1-10                           [1, 64]                   18,496
├─ReLU: 1-11                             [1, 64]                   --
├─Linear: 1-12                           [1, 10]                   650
├─LogSoftmax: 1-13                       [1, 10]                   --
==========================================================================
===============
Total params: 29,985,482
Trainable params: 29,985,482
Non-trainable params: 0
Total mult-adds (M): 29.99
==========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.09
Params size (MB): 119.94
Estimated Total Size (MB): 120.04
==========================================================================
===============
```

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

*Add a few sentences on your observations while using various architectures*

In [13]:
```
# add layers

model = nn.Sequential(
        nn.Flatten(),
        nn.Linear(1*28*28,8*28*28),
        nn.ReLU(),
        nn.Linear(8*28*28,30*14*14),
        nn.ReLU(),
        nn.Linear(30*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())
summary(model, input_size=(1, 1, 28,28))
```

Out[13]:
```
==========================================================================
===============
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
===============
Sequential                               [1, 10]                   --
├─Flatten: 1-1                           [1, 784]                  --
├─Linear: 1-2                            [1, 6272]                 4,923,52
0
├─ReLU: 1-3                              [1, 6272]                 --
├─Linear: 1-4                            [1, 5880]                 36,885,2
40
├─ReLU: 1-5                              [1, 5880]                 --
├─Linear: 1-6                            [1, 1568]                 9,221,40
8
├─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: 51,501,186
Trainable params: 51,501,186
Non-trainable params: 0
Total mult-adds (M): 51.50
==========================================================================
===============
Input size (MB): 0.00
Forward/backward pass size (MB): 0.11
Params size (MB): 206.00
Estimated Total Size (MB): 206.12
==========================================================================
===============
```

In [14]:
```
#Please type your answer here ...
"""
Adding more layers increases the parameters. Activation layers, which do not
do not add more parameters when added to the network.

Increasing the hidden neurons increases the total number of trainable paramet
"""
```

Out[14]:
```
'\nAdding more layers increases the parameters. Activation layers, which do
not have any additional parameters,\ndo not add more parameters when added
to the network.\n\nIncreasing the hidden neurons increases the total number
of trainable parameters.\n'
```

## 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 !

In [15]:
```
def simple_conv_net():
    model = nn.Sequential(
        # nn.Conv2d(1,16,kernel_size=3,padding=1),
        # nn.ReLU(),
        # nn.MaxPool2d(2,2),

        # TO-DO: Add layers below
        nn.Conv2d(1,8,kernel_size=3,padding=1),
        nn.ReLU(),
        nn.Conv2d(8,16,kernel_size=6,padding=2,stride=2),
        nn.ReLU(),
        nn.Conv2d(16,32,kernel_size=2,padding=0,stride=2),
        nn.ReLU(),

        # TO-DO, what will your shape be after you flatten? Fill it in place
        nn.Flatten(),
        nn.Linear(1568,64),
        # Do not change the code below
        nn.ReLU(),
        nn.Linear(64,10),
        nn.LogSoftmax())
    return model
```

In [16]:
```
conv_net = simple_conv_net()
```

In [17]:
```
view_network_parameters(conv_net)
```

```
Model Summary

0.weight: 72 elements
0.bias: 8 elements
2.weight: 4608 elements
2.bias: 16 elements
4.weight: 2048 elements
4.bias: 32 elements
7.weight: 100352 elements
7.bias: 64 elements
9.weight: 640 elements
9.bias: 10 elements

Total Trainable Parameters: 107850!
```

In [18]:
```
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]           4,624
├─ReLU: 1-4                              [1, 16, 14, 14]           --
├─Conv2d: 1-5                            [1, 32, 7, 7]             2,080
├─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: 107,850
Trainable params: 107,850
Non-trainable params: 0
Total mult-adds (M): 1.17
==========================================================================
===============
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

We use softmax to produce a probability of distributions as an output, which apart from being a very nice way of expressing discrete outputs, is also the output format required by popular loss functions.

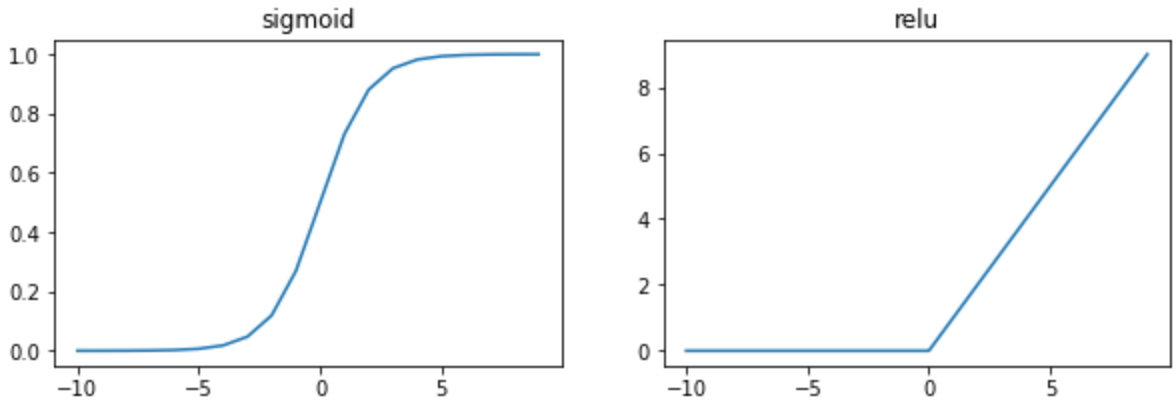
ReLU does not have the above properties and is therefore not a good output function. An example illustration is shown in the following cell.

In machine learning, when optimization involving probabilities are done, we generally use the log of the probabilities instead of the raw probabilities, as done by the log-softmax function. This has a number of practical advantages:

1. Multiplication of probabilities (common in MLE) become addition of log probabilities, which is a operation with significantly lesser time complexity
2. Log probabilities are numerically stable. With raw probabilities [0,1] there is a higher chance of precision underflow.

In [19]:
```
x = torch.arange(-10, 10)
ys = F.sigmoid(x)
yr = F.relu(x)

fig, ax = plt.subplots(1, 2, figsize=(10,3))
ax[0].plot(x, ys)
ax[0].set_title("sigmoid")
ax[1].plot(x, yr)
ax[1].set_title("relu")
```

Out[19]:
```
Text(0.5, 1.0, 'relu')
```




**Exercise 3.1.3**: What is the ratio of number of parameters of Conv-net to number of parameters of FC-Net

$\frac{p_{conv-net}}{p_{fc-net}}$ = Fill your answer

Do you see the difference ?!

0.0036

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

*Add a few sentences on your observations while using various architectures*

In [20]:

```
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 integer

    Return: nn.Sequential (final custom model)
    '''
    assert type(hidden_fc_dim) == list, "Please define hidden_fc_dim as list
    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)
                layers.append((f'activation_{idx+1}',nn.ReLU()))
            else:
                layers.append((f'linear_{idx+1}',nn.Linear(hidden_fc_dim[idx-
                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 bui


fc_net_custom1 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc
view_network_parameters(fc_net_custom1)

fc_net_custom2 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc
view_network_parameters(fc_net_custom2)

fc_net_custom3 = build_custom_fc_net(inp_dim=(1,28,28), out_dim=10, hidden_fc
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: 100352 elements
linear_1.bias: 128 elements
linear_2.weight: 16384 elements
linear_2.bias: 128 elements
linear_3.weight: 4096 elements
linear_3.bias: 32 elements
linear_4.weight: 320 elements
linear_4.bias: 10 elements

Total Trainable Parameters: 121450!
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: 4096 elements
linear_3.bias: 64 elements
linear_4.weight: 2048 elements
linear_4.bias: 32 elements
linear_5.weight: 320 elements
linear_5.bias: 10 elements

Total Trainable Parameters: 115306!
```

While both adding layers and adding more parameters result in an increased number of trainable parameters, the comparison between them might vary based on the type of network.

But generally for a fixed network, adding parameters can have more dramatic increase in parameters than adding layers. In the example above, I have illustrated this by doubling the 64 unit layer by:

1. adding another 64 units to the same layer
2. adding another layer with 64 units

We see that 1. introduces more trainable parameters.

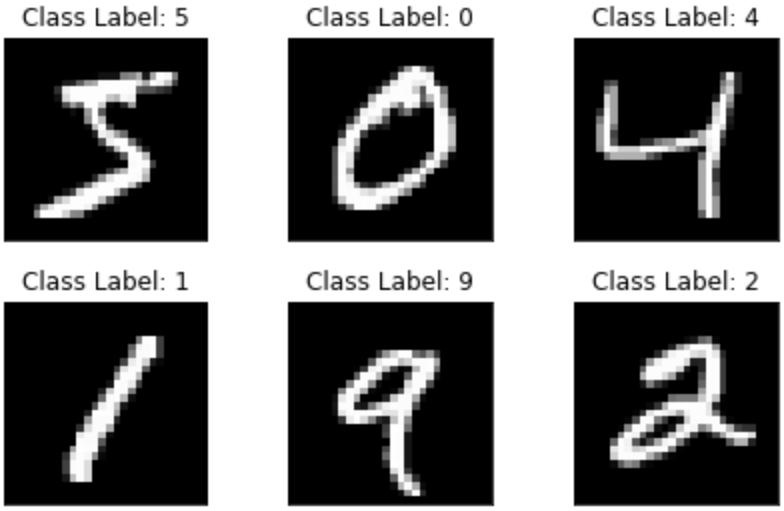
## Let's train the models to see their performace

In [21]:
```
# downloading mnist into folder
data_dir = 'data' # make sure that this folder is created in your working di
# transform the PIL images to tensor using torchvision.transform.toTensor me
train_data = torchvision.datasets.MNIST(data_dir, train=True, download=True,
test_data  = torchvision.datasets.MNIST(data_dir, train=False, download=True,
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_

train_loader = torch.utils.data.DataLoader(train_data_, batch_size=batch_size
val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,shuf
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,sh
```

```
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

In [22]:
```
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

In [23]:
```
def train_model(model, train_loader, device, loss_fn, optimizer, input_dim=(-
    model.train()
    # Initiate a loss monitor
    train_loss = []
    # Iterate the dataloader (we do not need the label values, this is unsupe
    for images, labels in train_loader: # the variable `labels` will be used
        # 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

In [24]:
```
# 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)

In [25]:
```
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=[12
```

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

In [26]:
```
### 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(
print(f'Selected device: {device}')

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

def weight_init_xavier(m):
    '''
    TO-DO: please add code below to add xavier uniform initialization and rem
    '''
    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
num_epochs = 30
wandb.config = {
  "learning_rate": lr,
  "epochs": num_epochs,
  "batch_size": batch_size
}
```

```
Selected device: cuda
```

# Training the Convolutional Neural Networks

In [27]:
```
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
    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
```

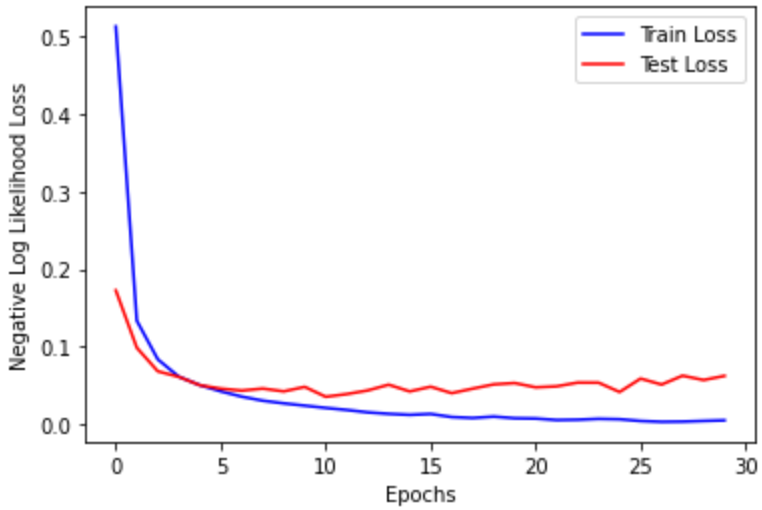
```
Conv Net training started

/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torch/nn/module
s/container.py:217: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torch/nn/module
s/container.py:217: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)

Epoch 1/30 : train loss 0.513    val loss 0.173
Epoch 2/30 : train loss 0.134    val loss 0.099
Epoch 3/30 : train loss 0.084    val loss 0.068
Epoch 4/30 : train loss 0.061    val loss 0.061
Epoch 5/30 : train loss 0.051    val loss 0.050
Epoch 6/30 : train loss 0.043    val loss 0.046
Epoch 7/30 : train loss 0.036    val loss 0.044
Epoch 8/30 : train loss 0.031    val loss 0.046
Epoch 9/30 : train loss 0.027    val loss 0.043
Epoch 10/30 : train loss 0.024   val loss 0.048
Epoch 11/30 : train loss 0.021   val loss 0.036
Epoch 12/30 : train loss 0.019   val loss 0.039

wandb: ERROR Summary data exceeds maximum size of 10.4MB. Dropping it.
```

```
Epoch 13/30 : train loss 0.015   val loss 0.044
Epoch 14/30 : train loss 0.014   val loss 0.051
Epoch 15/30 : train loss 0.012   val loss 0.042
Epoch 16/30 : train loss 0.013   val loss 0.049
Epoch 17/30 : train loss 0.009   val loss 0.040
Epoch 18/30 : train loss 0.008   val loss 0.046
Epoch 19/30 : train loss 0.010   val loss 0.052
Epoch 20/30 : train loss 0.008   val loss 0.053
Epoch 21/30 : train loss 0.008   val loss 0.048
Epoch 22/30 : train loss 0.006   val loss 0.049
Epoch 23/30 : train loss 0.006   val loss 0.054
Epoch 24/30 : train loss 0.007   val loss 0.054
Epoch 25/30 : train loss 0.007   val loss 0.042
Epoch 26/30 : train loss 0.004   val loss 0.059
Epoch 27/30 : train loss 0.003   val loss 0.051
Epoch 28/30 : train loss 0.004   val loss 0.063
Epoch 29/30 : train loss 0.004   val loss 0.057
Epoch 30/30 : train loss 0.005   val loss 0.062
Conv Net training done in 682.806 seconds!
```

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

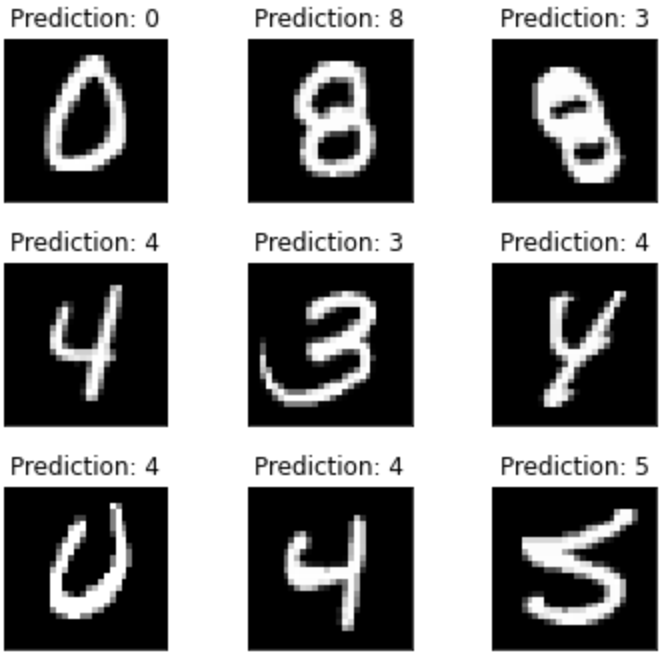
In [28]:
```
fig = plt.figure()
plt.plot(history_conv['train_loss'], color='blue')
plt.plot(history_conv['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

Out[28]:
```
Text(0, 0.5, 'Negative Log Likelihood Loss')
```



### Visualizing Predictions of Conv Net

In [29]:
```
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

In [30]:

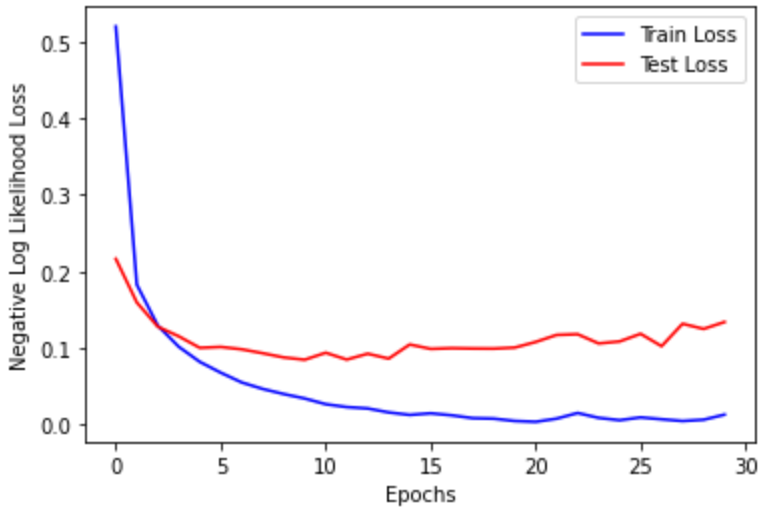
```
#TO-DO:Train the fc_net here
print('FC Net training started')
history_fc = {'train_loss':[],'val_loss':[]}
start_time = datetime.datetime.now()

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
    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_s
```

```
FC Net training started
Epoch 1/30 : train loss 0.520    val loss 0.216
Epoch 2/30 : train loss 0.183    val loss 0.160
Epoch 3/30 : train loss 0.129    val loss 0.128
Epoch 4/30 : train loss 0.102    val loss 0.115
Epoch 5/30 : train loss 0.082    val loss 0.100
Epoch 6/30 : train loss 0.068    val loss 0.101
Epoch 7/30 : train loss 0.055    val loss 0.098
Epoch 8/30 : train loss 0.047    val loss 0.093
Epoch 9/30 : train loss 0.040    val loss 0.088
Epoch 10/30 : train loss 0.034   val loss 0.085
Epoch 11/30 : train loss 0.027   val loss 0.094
Epoch 12/30 : train loss 0.023   val loss 0.085
Epoch 13/30 : train loss 0.021   val loss 0.093
Epoch 14/30 : train loss 0.016   val loss 0.086
Epoch 15/30 : train loss 0.013   val loss 0.105
Epoch 16/30 : train loss 0.015   val loss 0.099
Epoch 17/30 : train loss 0.012   val loss 0.100
Epoch 18/30 : train loss 0.008   val loss 0.099
Epoch 19/30 : train loss 0.008   val loss 0.099
Epoch 20/30 : train loss 0.005   val loss 0.101
Epoch 21/30 : train loss 0.004   val loss 0.108
Epoch 22/30 : train loss 0.008   val loss 0.117
Epoch 23/30 : train loss 0.015   val loss 0.118
Epoch 24/30 : train loss 0.009   val loss 0.106
Epoch 25/30 : train loss 0.006   val loss 0.109
Epoch 26/30 : train loss 0.009   val loss 0.119
Epoch 27/30 : train loss 0.007   val loss 0.102
Epoch 28/30 : train loss 0.005   val loss 0.132
Epoch 29/30 : train loss 0.006   val loss 0.125
Epoch 30/30 : train loss 0.013   val loss 0.134
FC Net training done in 573.883 seconds!
```

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

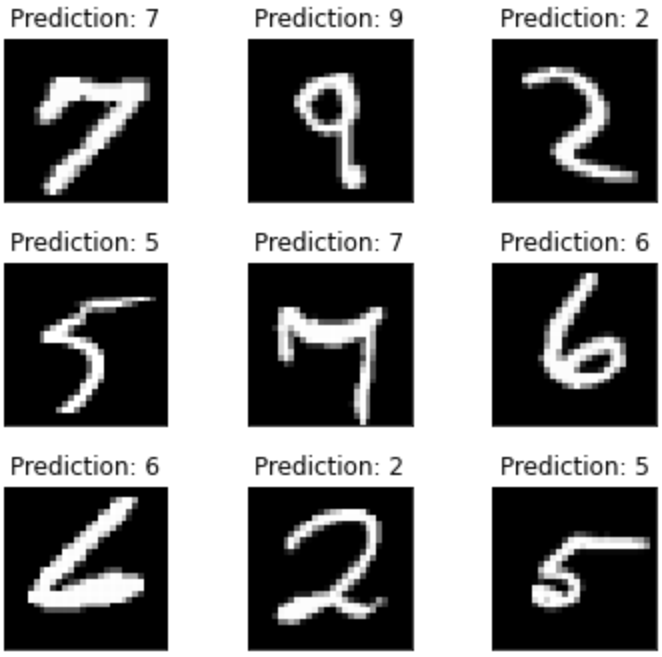
In [31]:
```
# TODO - Visualize the training progress of fc_net
fig = plt.figure()
plt.plot(history_fc['train_loss'], color='blue')
plt.plot(history_fc['val_loss'], color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('Epochs')
plt.ylabel('Negative Log Likelihood Loss')
```

Out[31]:
```
Text(0, 0.5, 'Negative Log Likelihood Loss')
```



## Visualizing Predictions of FC Net

In [32]:
```
# 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?

In [33]:
```
#Please type your answer here ...
"""
While it is true that CNNs have fewer parmeters, it is because they share the
as they perform their convolution operation across the image.

This convolution operation has a much higher time complexity compared
to the matrix multiplication used at each layer in the feed forward network.
"""
```

Out[33]:
```
'\nWhile it is true that CNNs have fewer parmaters, it is because they shar
e their parameters\nas they perform their convolution operation across the
image.\n\nThis convolution operation has a much higher time complexity comp
ared \nto the matrix multiplication used at each layer in the feed forward
network.\n'
```

## 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?

Several reasons can cause the model to perform poorly under translation.

1. When translation is manually introduced, the addition of padding elements can cause the network distribution to shift resulting in poor performance
2. Sometimes severe translation can cause the object of interest to leave the image frame, resulting in loss of information
3. In rare cases the feed forward layers at the end of the network can become a bottleneck for invariance
4. Layers like max pooling perform downsampling at the local level which can change how the features look when the image is translated. This can have a minor effect on invariance. // source: chatgpt
5. While CNNs are equivariant, if translation of an image changes its distribution drastically from the training set (probably from poor training data diversity), it can affect the performance of the model.

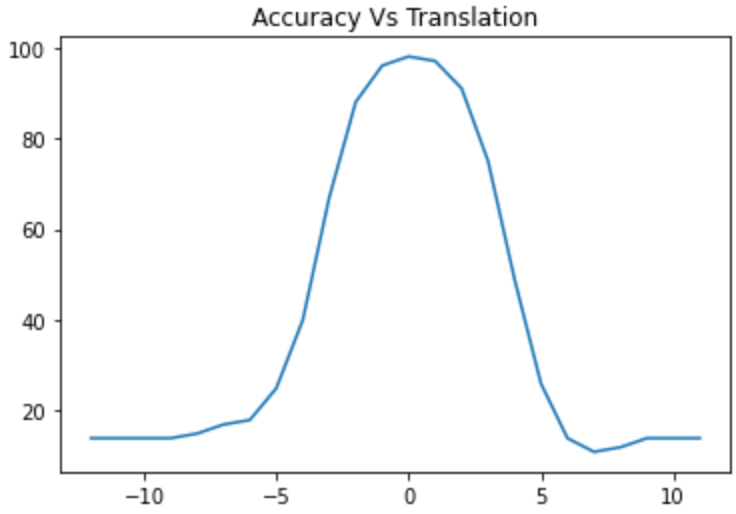
We will next measure the sensitivity of the convolutional network to translation in practice, and we will compare it to the fully-connected version.

In [34]:
```
## function to check accuracies for unit translation
def shiftVsAccuracy(model, test_loader, device, loss_fn, shifts = 12, input_d
    # 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 lis
                _ , 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
```

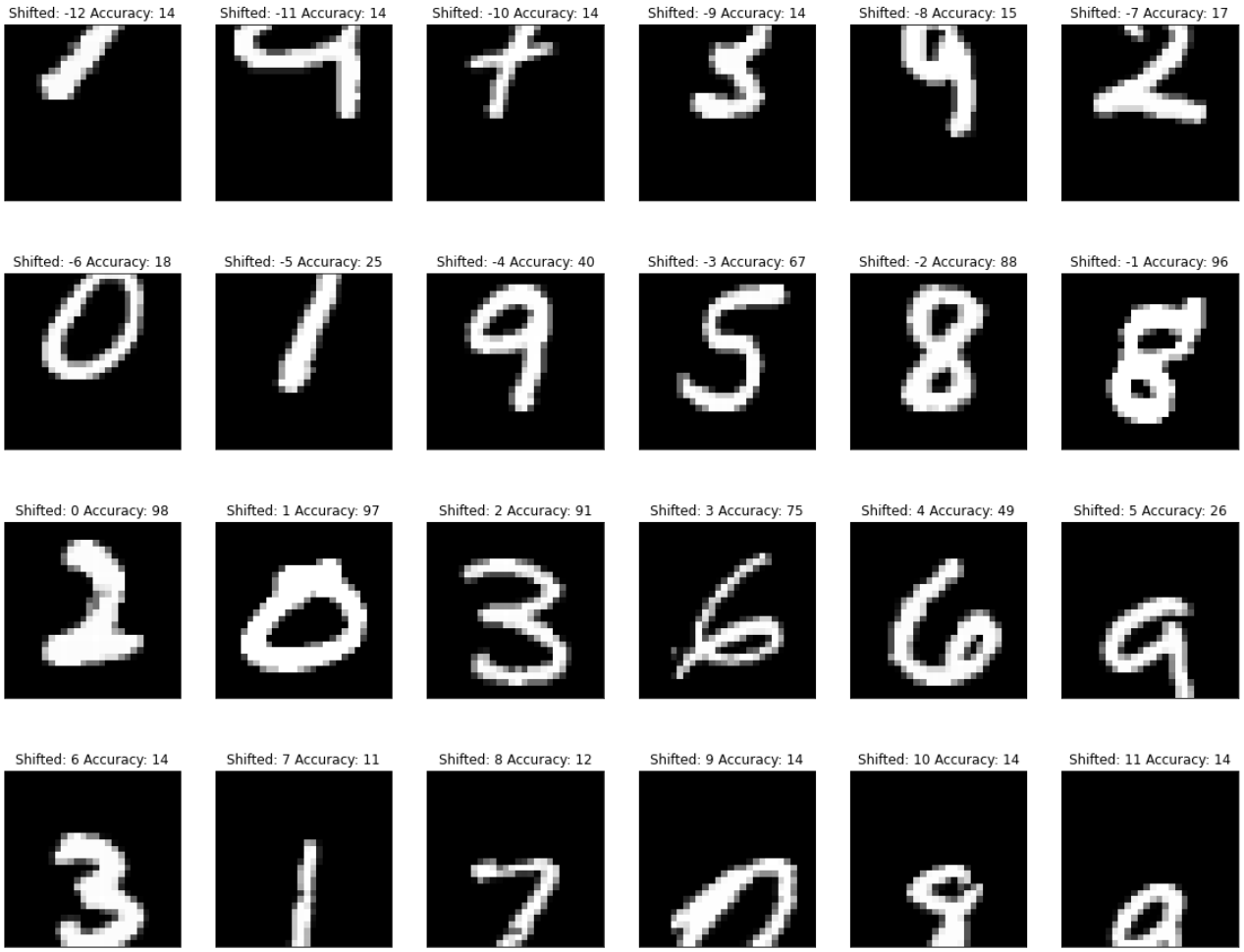
In [35]:
```
accuracies,shifted = shiftVsAccuracy(
        model=conv_net,
        test_loader=test_loader,
        device=device,
        shifts=12,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
```

In [36]:
```
shifts = np.arange(-12,12)
plt.plot(shifts,accuracies)
plt.title('Accuracy Vs Translation')
```

Out[36]:
```
Text(0.5, 1.0, 'Accuracy Vs Translation')
```



In [37]:
```
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
```



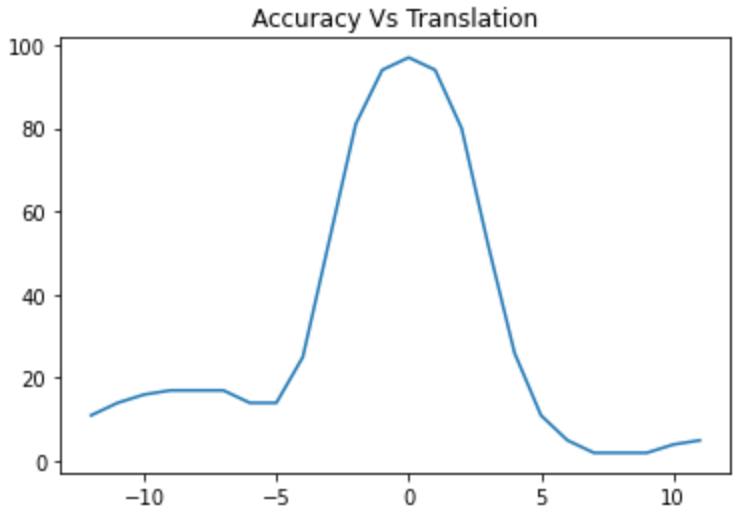

**Exercise 3.1.8:** Do the same for FC-Net and plot the accuracies. Is the rate of accuracy degradation same as Conv-Net? Can you justify why this happened?
Clue: You might want to look at the way convolution layers process information

In [38]:
```
# To-DO Write your code below
accuracies,shifted = shiftVsAccuracy(
        model=fc_net,
        test_loader=test_loader,
        device=device,
        shifts=12,
        loss_fn=loss_fn,
        input_dim=(-1,1,28,28))
```

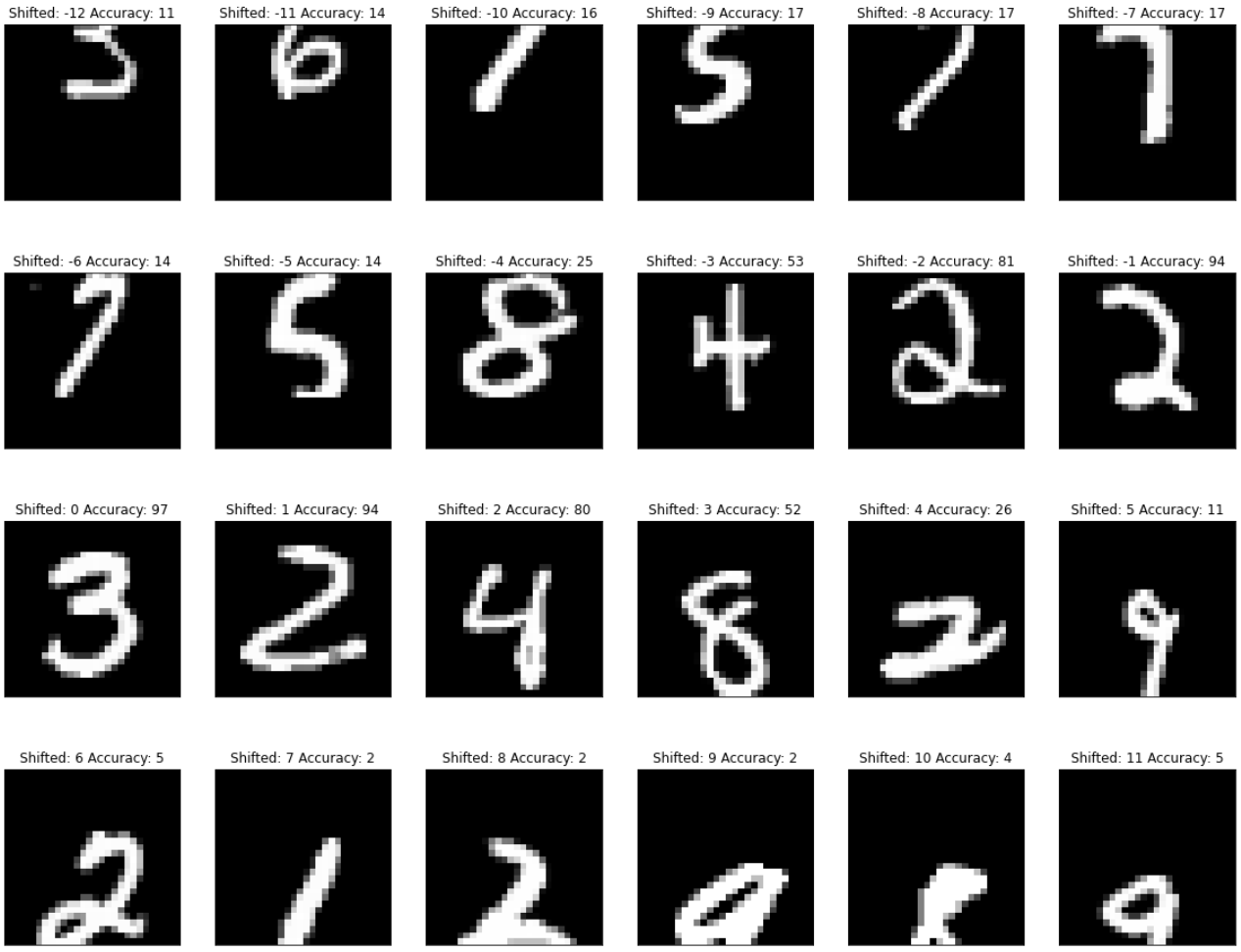
```
/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torch/nn/module
s/container.py:217: UserWarning: Implicit dimension choice for log_softmax
has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
```

In [39]:
```
shifts = np.arange(-12,12)
plt.plot(shifts,accuracies)
plt.title('Accuracy Vs Translation')
```

Out[39]:
```
Text(0.5, 1.0, 'Accuracy Vs Translation')
```



In [40]:
```
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
```




The drop in accuracy is slightly sharper in the feed-forward layers, compared to CNN which is a little gradual.

The convolution operation that obtains the cross correlation of features by convolving over the input image makes it translation invariant.

However, feed-forward networks look for certain features in specific spatial positions which is often impractical for images.

However, with the important features of the image being cut off from the image after severe translation the network's performance will begin to fall, which can be observed above.

I also hypothesize that the sharper drop for shifting down compared to shifting up in FCN is because of the fact that many of the useful features of digits are present in the lower half of the image, which the FCN model is explicitly looking for.

This can be run [run on Google Colab using this link (https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.2-Diffusion.ipynb)](https://colab.research.google.com/github/CS7150/CS7150-Homework_3/blob/main/HW3.2-Diffusion.ipynb)

# 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.***

In [1]:
```
##########################################################################
#                                  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, downloa

# 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, [tr

trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuff
valloader = torch.utils.data.DataLoader(val_dataset, batch_size=32, shuffle=F

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

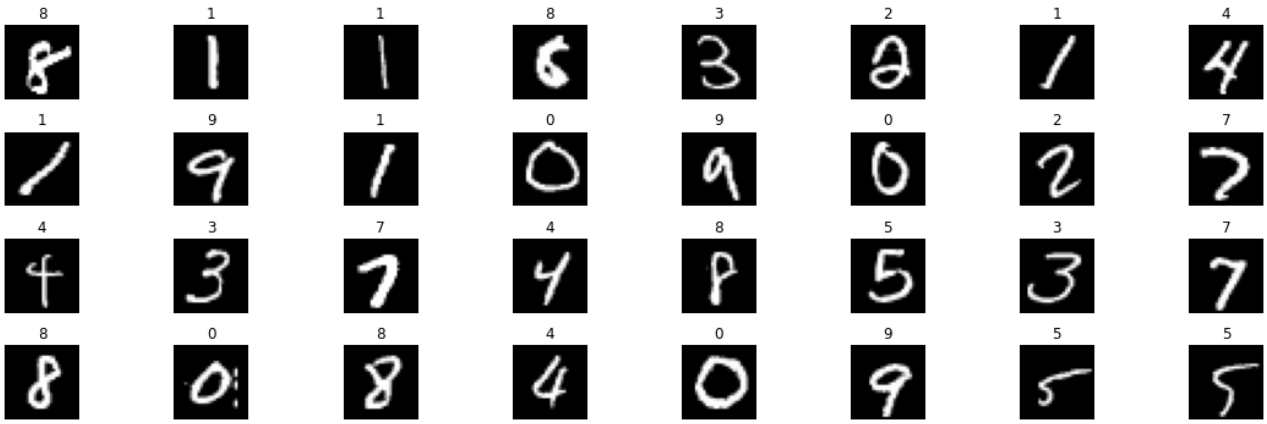
## Image Display Function

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.***

In [2]:

```
##########################################################################
#                                 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 im

    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 wo
  batch_size = len(batch[0])
  display_images(images= batch[0],n = batch_size, images_per_row=8, labels =
  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 (https://arxiv.org/abs/2006.11239) 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 $\alpha$)

***1. Linear Interpolation:***

Given two values, $A$ and $B$, the linear interpolation between them based on a blending factor $\alpha$ (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):

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

Where:

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

***3. Time-Dependent $\alpha$:***

For the Time-Dependent Alpha Noise Diffusion method, our $\alpha$ 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 $\alpha$s was to have an accumulating influence of noise over time. With each timestep, we multiply the original image with the cumulative product of $\alpha$ 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 $\alpha$s.

That's why the formula becomes:

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

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 $\beta$ (noise level parameter) over a series of timesteps.

Imagine you wish to transition $\beta$ 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 $\beta$ 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 $\beta$ 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***

---

In [3]:

```python
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(device)
    self.alphas = 1 - self.betas
    self.alpha_bar = torch.cumprod(self.alphas, dim=0) # Linear Cumulative Pr


  def linear_scheduler(self):
    """Returns a linear schedule from start to end over the specified total n
    ##########################################################################
    #                                  TO DO
    #                  Return a linear schedule of `betas`
    #                    from `start_beta` to `end_beta`
    #                         hint: torch.linspace()
    ##########################################################################
    return torch.linspace(self.start_beta, self.end_beta, self.total_steps)

  def noise_diffusion(self, image, t):
    """
    Diffuse noise into an image based on timestep t using the pre-computed cu
    """
    ##########################################################################
    #                                  TO DO
    #                 Process the given `image` for timesteps `t`
    #                Return processed image & necessary variables
    ##########################################################################
    image = image.to(self.device)
    cum_alpha_t = self.alpha_bar[t, None, None, None]
    noise = torch.randn_like(image)
    return image * cum_alpha_t + noise * (1 - cum_alpha_t), noise
```

## Testing NoiseDiffuser Class (SANITY CHECK)

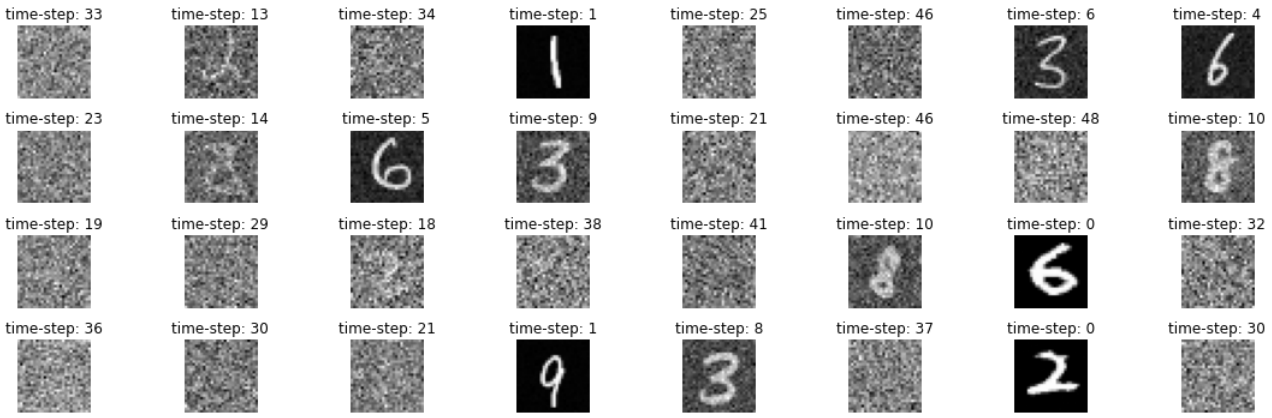
In [4]:
```
# SANITY CHECK
in_channels_arg = 1
out_channels_arg = 1
batch_size = 32
height = 28
width = 28
total_timesteps = 50
start_beta, end_beta = 0.001, 0.2


# 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,), devic
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?***

In [5]:
```
##########################################################################
#                                 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")
total_timesteps = 50
start_beta, end_beta = 0.001, 0.2
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,), d
    noisy_images,_ = diffuser.noise_diffusion(minibatch, timesteps_to_display
    display_images(images=noisy_images, n=batch_size, images_per_row=8, label
    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`

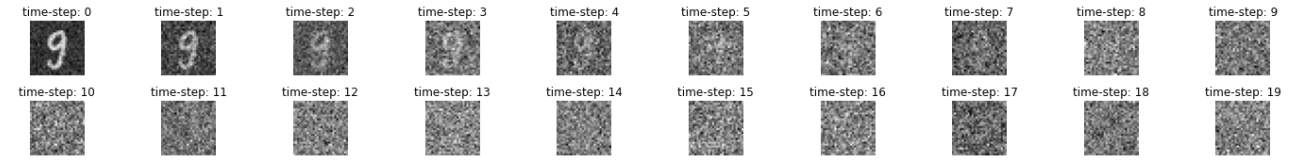
The precise values can be fine-tuned based on specific requirements, visual assessments (like in the cell below) or even metrics.

**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?***

In [6]:
```
##########################################################################
#                                 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 = 20
start_beta, end_beta = 0.1, 0.15
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
for batch in trainloader:
    repetitions = torch.tensor([total_timesteps]).repeat(minibatch_size)
    minibatch = batch[0][:minibatch_size,:,:].repeat_interleave(repetitions,
    batch_size = len(minibatch)
    timesteps_to_display = torch.linspace(0, total_timesteps-1, total_timeste
    noisy_images,_ = diffuser.noise_diffusion(minibatch, timesteps_to_display
    display_images(images=noisy_images, n=batch_size, images_per_row=10, labe
    break
```




Using a low to higher noise makes the noise addition gradual.

Using a high to lower noise makes the noise addition start aggresively. However, to my naked eyed the transition at later stages is not clearly smoother to compare to the defaults.

# 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 (https://arxiv.org/abs/1505.04597)" 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***

---

In [7]:
```
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_s
      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_s
      self.afterup1 = self.doubleConvolution(inC = 32 , oC = 16, kS1=5, kS2=4

      #----------------------- OUTPUT -----------------------#
      ###################################################################
      #            Constructing final Output Layer (Use Conv2d() function)
      ###################################################################
      self.out = nn.Conv2d(in_channels=16, out_channels=out_channels, kernel_

    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 Cor
      x3 = self.down2(x2)                 # Downsampling followed by Double Cor

      #----------------------- 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))

    t1 = self.time_emb1(t)[:,:, None, None]
    y1 = self.up1(y2 + t1)
    y1 = self.afterup1(torch.cat([y1, self.xLikeY(x1, y1)], axis = 1))

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

    return outY



  #-------------------------------------------------------------------------

  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 u
    """
    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 original paper of Unet
    inC : inputChannels
    oC : outputChannels
    kS1 : Kernel_size of first convolution
    kS2 : Kernel_size of second convolution
    sT: stride
    pA: padding
    """
    return nn.Sequential(
          nn.Conv2d(in_channels= inC, out_channels=oC, kernel_size=kS1, str
          nn.ReLU(inplace=True),
          nn.Conv2d(in_channels = oC,out_channels=oC, kernel_size=kS2, stri
          nn.ReLU(inplace=True),
    )


  def Down(self, inputC, outputC, dsKernelSize = None):
    """
    Building Down Sampling Part of the Unet Architecture (Using MaxPool) fo
    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 upsam
    source: tensor whose shape will be considered ---------UPSAMPLED TENSOR
```

```
    target: tensor whose shape will be modified to align with target ------
    """
    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 - dif
    return x1
```

## Testing UNet Class (SANITY CHECK)

In [8]:

```
# 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,), devic

# 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.floa
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])
```

In [9]:

```
# 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,), devic

# 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.float3
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])
```

In [10]:

```
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.***

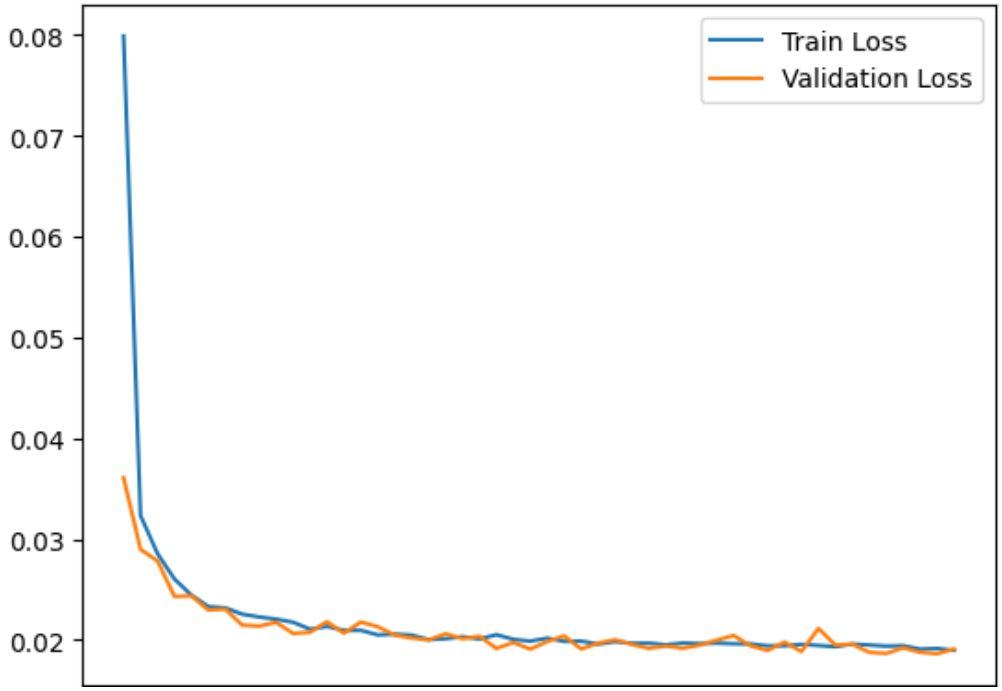
In [11]:

```python
from tqdm import tqdm

def train(model, train_loader, val_loader, optimizer, criterion, device, num_
    """
    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 m
    """

    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_lo
        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,


            ##########################################################################
            #                                  TO DO
            #                     Calculate Noisy data, True noise
            #                 and Predicted Noise, & then feed it to criteric
            ##########################################################################
            # raise NotImplementedError
            noisy_data, true_noise = diffuser.noise_diffusion(data, timesteps
            # noisy_data, true_noise = None,None
            predicted_noise = model(noisy_data, t = torch.tensor(timesteps).t


            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),
        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 val
                batch_size = len(data)
                timesteps = torch.randint(0, totalTrainingTimesteps, (batch_s

                ###########################################################
                #                                  TO DO
                #                      Calculate Noisy data, True noise
                #                  and Predicted Noise, & then feed it to crit
                ###########################################################
                # raise NotImplementedError
                # noisy_data, true_noise = None,None
                noisy_data, true_noise = diffuser.noise_diffusion(data, times
                predicted_noise = model(noisy_data, t = torch.tensor(timestep

                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

    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.



In [12]:

```
##########################################################################
#                                 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? THIN
- `num_epochs`: How many epochs are you training for? (*We'd love to see best
"""
total_timesteps = 1000
startBeta, endBeta = 1e-3, 0.02
inputChannels, outputChannels = 1, 1
num_epochs = 40

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=outputCha
optimizer = torch.optim.Adam(stableDiffusionModel.parameters(), 2e-4)
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 1/40, Train Loss: 0.2494, Validation Loss: 0.0833

Epoch 2/40, Train Loss: 0.0708, Validation Loss: 0.0588

Epoch 3/40, Train Loss: 0.0561, Validation Loss: 0.0504

Epoch 4/40, Train Loss: 0.0481, Validation Loss: 0.0479

Epoch 5/40, Train Loss: 0.0436, Validation Loss: 0.0425

Epoch 6/40, Train Loss: 0.0406, Validation Loss: 0.0407

Epoch 7/40, Train Loss: 0.0379, Validation Loss: 0.0374

Epoch 8/40, Train Loss: 0.0355, Validation Loss: 0.0354

Epoch 9/40, Train Loss: 0.0340, Validation Loss: 0.0323

Epoch 10/40, Train Loss: 0.0326, Validation Loss: 0.0325

Epoch 11/40, Train Loss: 0.0313, Validation Loss: 0.0305

Epoch 12/40, Train Loss: 0.0305, Validation Loss: 0.0307

Epoch 13/40, Train Loss: 0.0299, Validation Loss: 0.0296

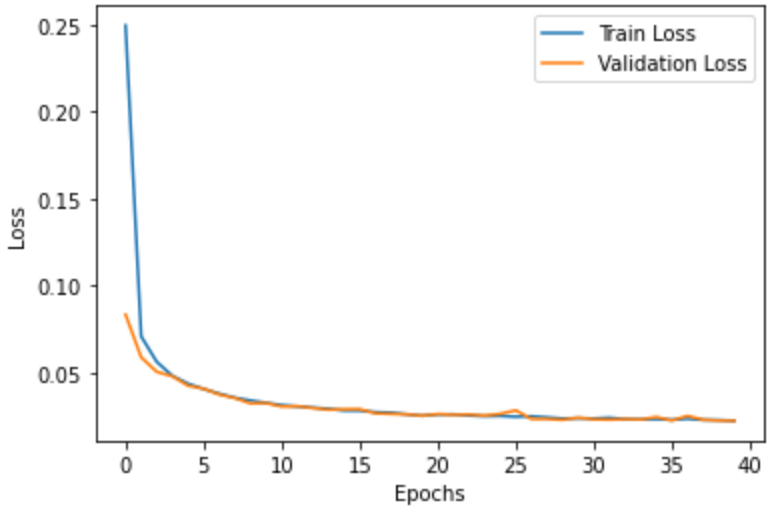
Epoch 14/40, Train Loss: 0.0292, Validation Loss: 0.0287

Epoch 15/40, Train Loss: 0.0282, Validation Loss: 0.0289

Epoch 16/40, Train Loss: 0.0281, Validation Loss: 0.0290

```
Epoch 17/40, Train Loss: 0.0273, Validation Loss: 0.0266

Epoch 18/40, Train Loss: 0.0270, Validation Loss: 0.0263

Epoch 19/40, Train Loss: 0.0260, Validation Loss: 0.0260

Epoch 20/40, Train Loss: 0.0252, Validation Loss: 0.0253

Epoch 21/40, Train Loss: 0.0259, Validation Loss: 0.0262

Epoch 22/40, Train Loss: 0.0260, Validation Loss: 0.0257

Epoch 23/40, Train Loss: 0.0253, Validation Loss: 0.0259

Epoch 24/40, Train Loss: 0.0249, Validation Loss: 0.0253

Epoch 25/40, Train Loss: 0.0251, Validation Loss: 0.0262

Epoch 26/40, Train Loss: 0.0246, Validation Loss: 0.0283

Epoch 27/40, Train Loss: 0.0249, Validation Loss: 0.0233

Epoch 28/40, Train Loss: 0.0242, Validation Loss: 0.0233

Epoch 29/40, Train Loss: 0.0236, Validation Loss: 0.0229

Epoch 30/40, Train Loss: 0.0238, Validation Loss: 0.0241

Epoch 31/40, Train Loss: 0.0236, Validation Loss: 0.0232

Epoch 32/40, Train Loss: 0.0240, Validation Loss: 0.0229

Epoch 33/40, Train Loss: 0.0232, Validation Loss: 0.0232
```

```
Epoch 34/40, Train Loss: 0.0231, Validation Loss: 0.0230

Epoch 35/40, Train Loss: 0.0232, Validation Loss: 0.0244

Epoch 36/40, Train Loss: 0.0229, Validation Loss: 0.0225

Epoch 37/40, Train Loss: 0.0234, Validation Loss: 0.0251

Epoch 38/40, Train Loss: 0.0228, Validation Loss: 0.0229

Epoch 39/40, Train Loss: 0.0227, Validation Loss: 0.0225

Epoch 40/40, Train Loss: 0.0223, Validation Loss: 0.0223
```




# EXERCISE 3 : SAMLING GENERATION

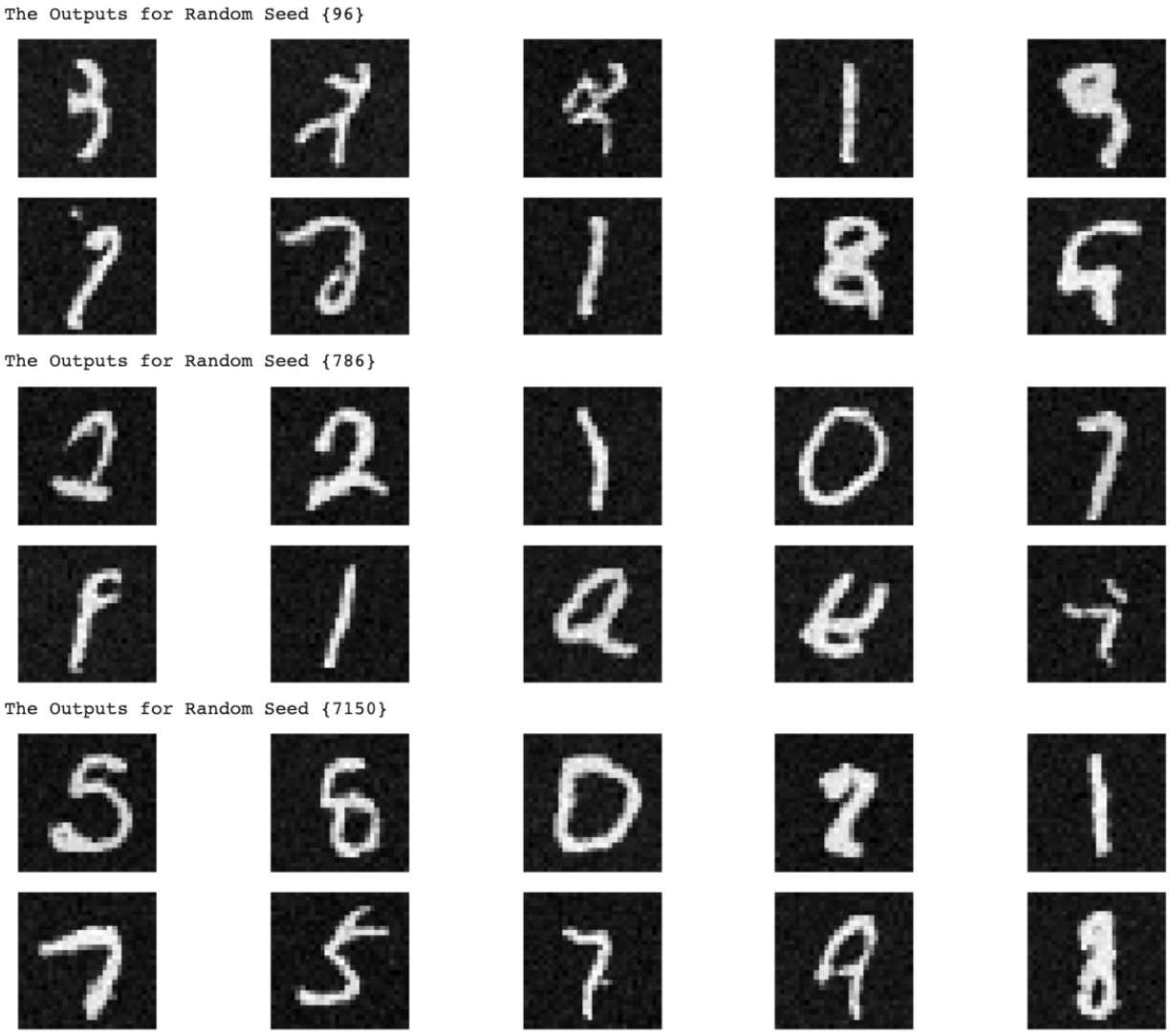
## Sampling formula

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:

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

## 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:



In [13]:
```
def generate_samples(x_t, model, num_samples, total_timesteps, diffuser, devi
    """
    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 - di

    ##########################################################################
    #                                  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(x_t,
                          t = torch.tensor(timestep)
                                  .to(torch.float32).to(device=device)
                                  .view(-1,1))

        sqrt_beta_z = torch.sqrt(diffuser.betas)[timestep] * z
        x_t_minus_1 = one_by_sqrt_alpha[timestep] * \
                (x_t - beta_by_sqrt_one_minus_alpha_cumprod[timestep] \
                     *epsilon_t) \
                + sqrt_beta_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_timest

  # 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]:
```
! pip install git+https://github.com/davidbau/baukit
```

```
Defaulting to user installation because normal site-packages is not wr
iteable
Collecting git+https://github.com/davidbau/baukit
  Cloning https://github.com/davidbau/baukit (https://github.com/david
bau/baukit) to /tmp/pip-req-build-2uxiqw6h
  Running command git clone -q https://github.com/davidbau/baukit (htt
ps://github.com/davidbau/baukit) /tmp/pip-req-build-2uxiqw6h
  Resolved https://github.com/davidbau/baukit (https://github.com/davi
dbau/baukit) to commit 5e23007c02fd58f063200c5dc9033e90f092630d
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
    Preparing wheel metadata ... done
Requirement already satisfied: numpy in /shared/centos7/anaconda3/202
2.05/lib/python3.9/site-packages (from baukit==0.0.1) (1.21.5)
Requirement already satisfied: torchvision in /home/krishnamurthy.su
r/.local/lib/python3.9/site-packages (from baukit==0.0.1) (0.15.1)
Requirement already satisfied: torch in /home/krishnamurthy.sur/.loca
l/lib/python3.9/site-packages (from baukit==0.0.1) (2.0.0)
```

In [2]:
```
import torch, os, PIL.Image, numpy
from torchvision.models import alexnet, resnet18, resnet101, resnet152, effic
from torchvision.transforms import Compose, ToTensor, Normalize, Resize, Cent
from torchvision.datasets.utils import download_and_extract_archive
from baukit import ImageFolderSet, show, renormalize, set_requires_grad, Trac
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
```

```
--2023-10-19 21:55:44--  https://cs7150.baulab.info/2022-Fall/data/dog-and-cat-example.jpg (https://cs7150.baulab.info/2022-Fall/data/dog-and-cat-example.jpg)
Connecting to 10.99.0.130:3128... connected.
Proxy request sent, awaiting response... 304 Not Modified
File ‘dog-and-cat-example.jpg’ not modified on server. Omitting download.

--2023-10-19 21:55:44--  https://cs7150.baulab.info/2022-Fall/data/hungry-cat.jpg (https://cs7150.baulab.info/2022-Fall/data/hungry-cat.jpg)
Connecting to 10.99.0.130:3128... connected.
Proxy 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='
    size = spec_size(size)
    mapping = getattr(cm, colormap)
    scaled = torch.nn.functional.interpolate(data[None, None], size=size, moc
    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, moc
    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('ui

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('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)
```

```
/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
  warnings.warn(
/home/krishnamurthy.sur/.local/lib/python3.9/site-packages/torchvision/models/_utils.py:223: UserWarning: 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=ResNet18_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

In [9]:
```
# Type your solution here
[(labels[i], i.item()) for i in masked_preds[19,None].topk(dim=1, k=5, sorted
```

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

The model was not able to detect the presence of the boxer in the image.

It seems much of the features used to identify the 'boxer' dog seems to be present in its face. With that important feature masked out, the network is not able to effectively identify the object effectively. This implies that neural networks focus on certain types of features for certain classes.

**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 [10]:
```
# Type your solution here
for c in ['boxer', 'tiger cat']:
    print(f"Image with the least score for {c}")
    classidx = labels.index(c)
    minidx = masked_preds[:,classidx].argmin()
    show(renormalize.as_image(masked_batch[minidx]))
```

```
Image with the least score for boxer
```

```
Image with the least score for tiger cat
```

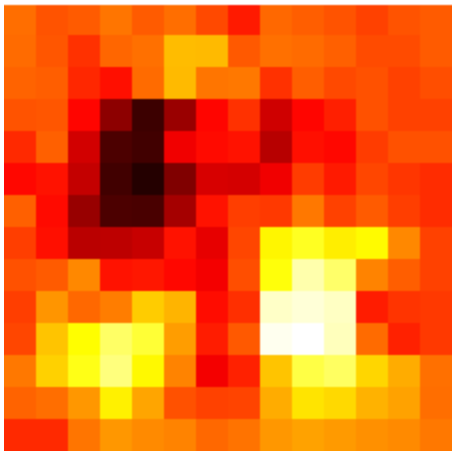
Here is a way that we can visualise the pixels that are more responsible for the predictions. It's something similar you did above in Exercise 3.3.2

In [11]:
```
for c in ['boxer', 'tiger cat']:
    heatmap = (base_preds[:,labels.index(c)]-masked_preds[:,labels.index(c)])
    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

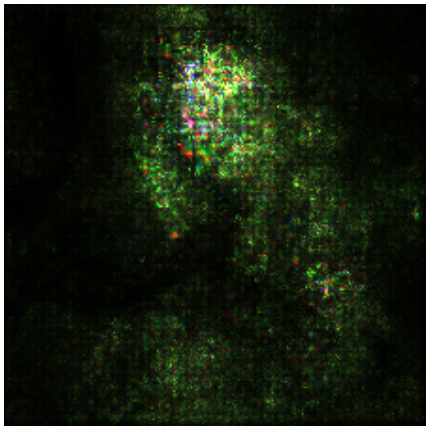
https://arxiv.org/pdf/1810.03292.pdf (https://arxiv.org/pdf/1810.03292.pdf)

**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).

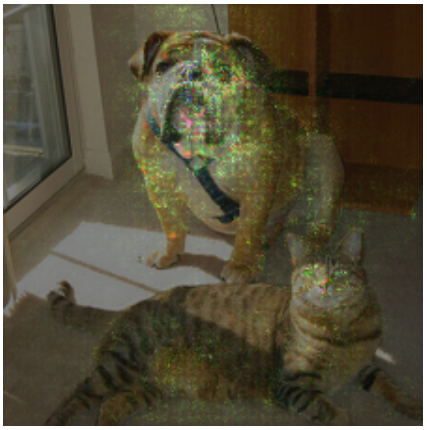
In [12]:
```
for label in ['boxer', 'tiger cat']:
    total = 0
    for i in range(20):
        prober = data + torch.randn(data.shape) * 0.2
        prober.requires_grad = True
        loss = torch.nn.functional.cross_entropy(
            net(prober[None]),
            torch.tensor([labels.index(label)]))
        loss.backward()

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

        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='
        ['overlay',
         overlay(renormalize.as_image(data, source='imagenet'),
                 renormalize.as_image((total / total.max() * 5).clamp(0, 1), s
    ]])
```

boxer

total grad**2

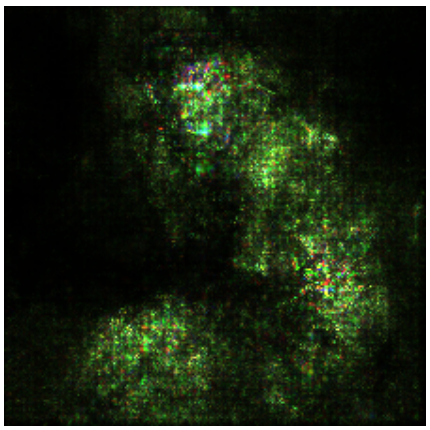


overlay

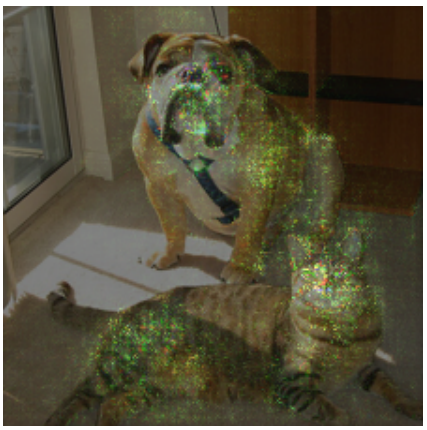


tiger cat

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 [13]:
```
if not os.path.isdir('imagenet_val_5k'):
    download_and_extract_archive('https://cs7150.baulab.info/2022-Fall/data/i
                                 'imagenet_val_5k')
ds = ImageFolderSet('imagenet_val_5k', shuffle=True, transform=Compose([
    Resize(256),
    CenterCrop(224),
    ToTensor(),
    renormalize.NORMALIZER['imagenet']
]))
```

```
Downloading https://cs7150.baulab.info/2022-Fall/data/imagenet_val_5k.zip
(https://cs7150.baulab.info/2022-Fall/data/imagenet_val_5k.zip) to imagenet
_val_5k/imagenet_val_5k.zip

100%|██████████| 50757954/50757954 [00:00<00:00, 64348863.47it/s]

Extracting imagenet_val_5k/imagenet_val_5k.zip to imagenet_val_5k
```

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 [14]:
```
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]:
```
# Type your solution here
top_filters = tr.output[0].amax(dim=[1,2]).topk(12).indices

for unit_num in top_filters:
    show(show.WRAP, [[f'neuron {unit_num}',
        overlay(im, rgb_heatmap(tr.output[0, unit_num]))]
    ])
```

neuron 115

neuron 391

neuron 58

neuron 321

neuron 62

neuron 65

neuron 13

neuron 138

neuron 262

neuron 190

neuron 239

neuron 217

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

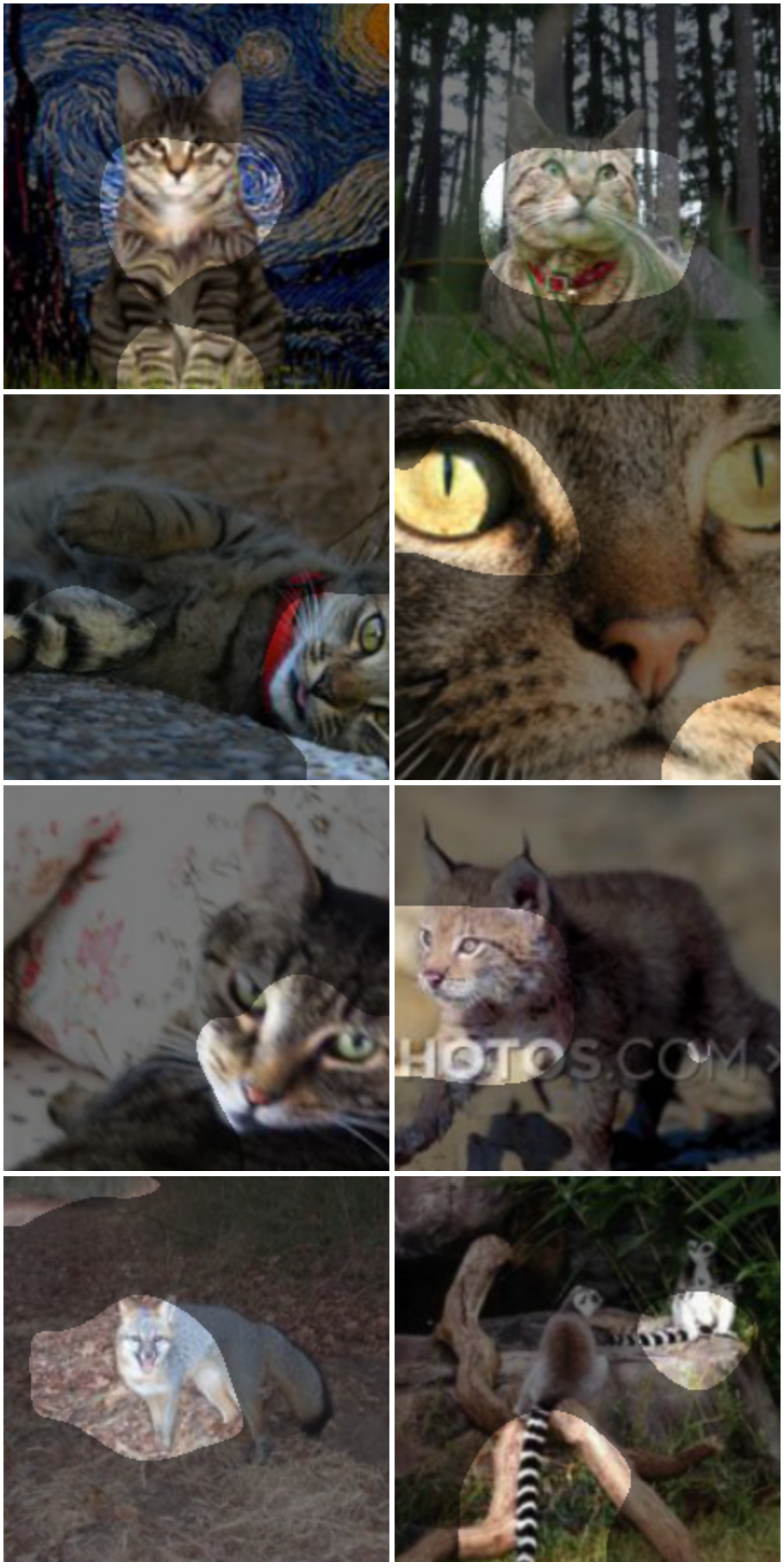
Choose one and run the network on all the data and sort to find the maximum-activating data. Let's see how the neuron you found to be top activating generalizes. We will trace the neuron activations of the entire dataset and visualise the top 12 images and display the regions where the chosen neurons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

```
  0%|          | 0/5001 [00:00<?, ?it/s]
```

layer4.1.conv1 neuron 58



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

We see that the neuron doesn't generalise very well. While it mostly gets activated for cats, we see that it gets activated for spuriously correlated patterns such as stripes and colors.

## 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.outp
```

```
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.

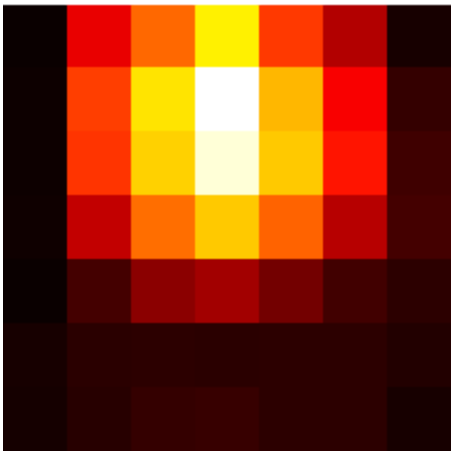
In [19]:
```
boxer_weights = net.fc.weight[labels.index('boxer')]
```

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))]
    ]]
)
```

original boxer

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

In [22]:
```
# Type your solution here
cat_weights = net.fc.weight[labels.index('tiger cat')]
cat_heatmap = torch.einsum('bcyx, c -> yx', tr.output, cat_weights)

show(show.TIGHT,
    [['original', im],
     ['cat', overlay(im, rgb_heatmap(cat_heatmap, im))]
    ]]
)
```

original cat

**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]:
```
im = resize_and_crop(PIL.Image.open('hungry-cat.jpg'), 224)
show(im)
data = renormalize.from_image(resize_and_crop(im, 224), target='imagenet')
```

In [24]:
```
with Trace(net, 'layer4') as tr:
    preds = net(data[None])
```

In [25]:
```
# Type your solution here
cat_weights = net.fc.weight[labels.index('tiger cat')]
cat_heatmap = torch.einsum('bcyx, c -> yx', tr.output, cat_weights)

show(show.TIGHT,
    [['original', im],
     ['tiger cat', overlay(im, rgb_heatmap(cat_heatmap, im))]
    ]]
)
```

original

tiger cat

In [26]:
```
# Type your solution here
cat_weights = net.fc.weight[labels.index('goldfish')]
cat_heatmap = torch.einsum('bcyx, c -> yx', tr.output, cat_weights)

show(show.TIGHT,
    [['original', im],
     ['goldfish', overlay(im, rgb_heatmap(cat_heatmap, im))]
    ]]
)
```