```
import numpy as np
import torch
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms
from torch import nn
from torchvision import utils
```

Load the datasets

```
# Load MNIST dataset
train_data = torchvision.datasets.MNIST(root="./",train=True,transform=transforms
test_data = torchvision.datasets.MNIST(root="./",train=False,transform=transforms
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: ce

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-u
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-u
100%|██████████| 9912422/9912422 [00:00<00:00, 15961007.53it/s]
Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: ce

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-u
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-u
100%|██████████| 28881/28881 [00:00<00:00, 491422.69it/s]
Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: ce

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ub
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ub
100%|██████████| 1648877/1648877 [00:00<00:00, 4414755.70it/s]
Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: ce

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ub
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ub
100%|██████████| 4542/4542 [00:00<00:00, 6477568.44it/s]
Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw
```

```
# Hyperparameters
learning_rate = 0.001
epochs = 10
batch_size = 200
no_of_eigen_val = 30 # number of principal components (for PCA)
```

```
train_loader = torch.utils.data.DataLoader(dataset=train_data,batch_size=batch_si
test_loader = torch.utils.data.DataLoader(dataset=test_data,batch_size=len(test_d
```

```
# Creating a DataLoader for a subset of test_data
test_sample_loader = torch.utils.data.DataLoader(dataset=test_data.data[9705:9715
```

```
train_dataset = train_data.data.reshape(train_data.data.shape[0],train_data.data.
test_dataset = test_data.data.reshape(test_data.data.shape[0],test_data.data.shap
test_dataset_sampled = test_dataset[np.arange(9705, 9715),:] # slicing & creating
```

Comparing PCA and autoencoders

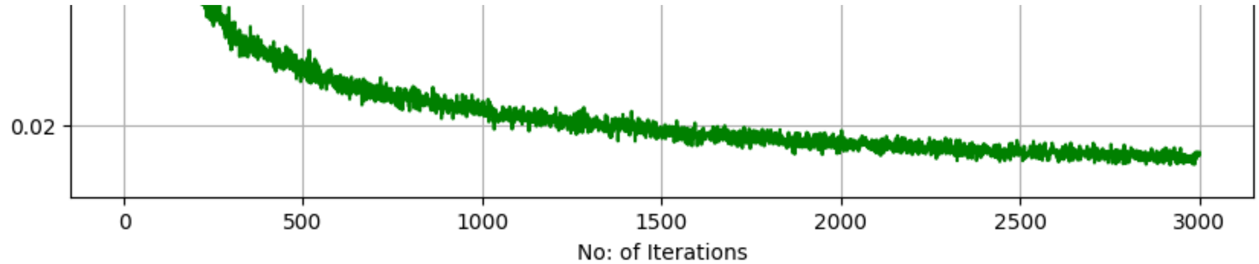
```
# i/p data is a 2D tensor of shape (num_datapts, 784)
def PCA(input_data, top_k_ev):
  # mean calculation and centering
  input_mean = torch.mean(input_data, 0)
  centered_data_matrix = input_data - input_mean

  # Covariance Matrix
  cov_matrix = torch.matmul(centered_data_matrix.T, centered_data_matrix) # dot p

  # Eigenvalues and Eigenvectors
  eigen_values, eigen_vectors = torch.linalg.eigh(cov_matrix) #  eigenvalues: vec
  sorted_eigen_vals, sorted_indices = torch.sort(eigen_values, descending=True) #

  # Select the top k eigenvectors and eigenvalues
  top_k_eigen_values, top_k_indices = sorted_eigen_vals[:top_k_ev], sorted_indice
  top_k_eigen_vectors = eigen_vectors[:,top_k_indices]

  assert top_k_eigen_vectors.shape == (784, top_k_ev) # checks the shape
  assert centered_data_matrix.shape == input_data.shape
  assert input_mean.shape == torch.Size([784])

  return top_k_eigen_vectors
```

```
pc = PCA(train_dataset.float(), no_of_eigen_val) # (784, top_k)


# reconstruct the original dataset (in its original high-dimensional space) using
def reconstruct_data(principal_components, dataset): # principal_components: matr

  # Create the projection matrix
  projection_matrix = torch.matmul(principal_components, principal_components.T)

  # project the dataset using the projection matrix
  projected_data = torch.matmul(dataset, projection_matrix)
  # each data point in dataset is projected back into the original 784-dimensiona

  assert projected_data.shape == (dataset.shape[0],principal_components.shape[0])
  return projected_data


reconstructed_test_data_sampled = reconstruct_data(pc, test_dataset_sampled.float


# Autoencoder neural network
class AutoEncoder(nn.Module):
  def __init__(self):
    super(AutoEncoder, self).__init__()
    self.encoder = nn.Sequential( # encoder part
        nn.Linear(784,512), # i/p has 784 dimensions - 28x28
        nn.ReLU(),
        nn.Linear(512,256),
        nn.ReLU(),
        nn.Linear(256,128),
        nn.ReLU(),
        nn.Linear(128,30),
        nn.ReLU())
    self.decoder = nn.Sequential(  # decoder part
        nn.Linear(30,128),
        nn.ReLU(),
        nn.Linear(128,256),
        nn.ReLU(),
        nn.Linear(256,784),
        nn.ReLU())

  # forward pass
  def forward(self,x): # shape of x (batch_size, 784))
    x = self.encoder(x)
    encoded_output = x
    x = self.decoder(x)
    return x, encoded_output


model1 = AutoEncoder()
criterion1 = nn.MSELoss() # loss function
```

```
optimizer1 = torch.optim.Adam(model1.parameters(), lr=learning_rate) # adam optim

# Training loop
training_loss=[]
for epoch in range(epochs):
  for images, labels in train_loader:
    images = images.reshape(images.shape[0], -1) # 2D to 1D vector
    outputs, _ = model1(images) # reconstructed images
    loss = criterion1(outputs, images) # reconstruction error
    training_loss.append(loss.item())

    optimizer1.zero_grad() # clears the gradients from the previous batch
    loss.backward() # backpropagation
    optimizer1.step() # update the model parameter

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
  #print("Epoch [{}/{}]: completed".format(epoch + 1, epochs))
```

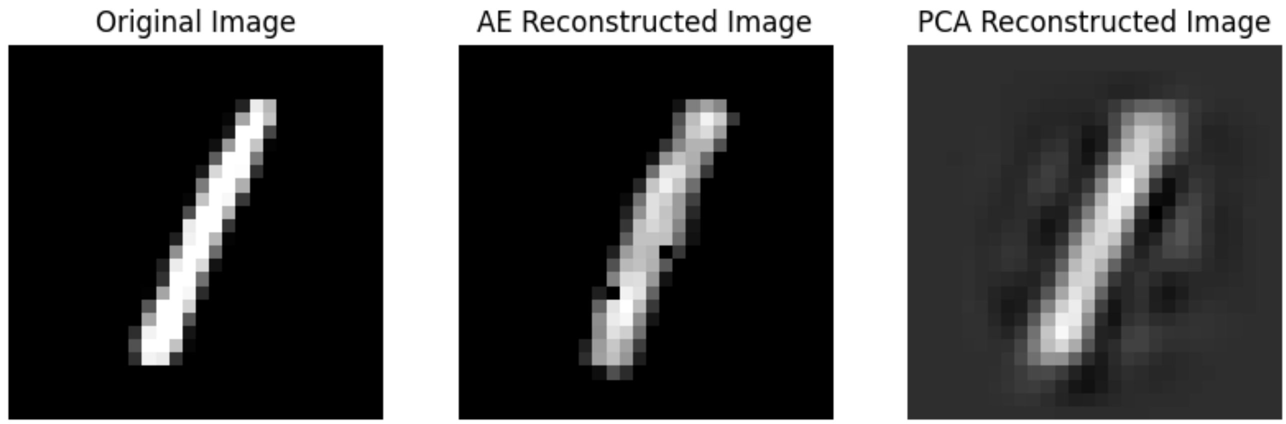
```
Epoch [1/10]: completed, Loss: 0.0329
Epoch [2/10]: completed, Loss: 0.0259
Epoch [3/10]: completed, Loss: 0.0221
Epoch [4/10]: completed, Loss: 0.0205
Epoch [5/10]: completed, Loss: 0.0188
Epoch [6/10]: completed, Loss: 0.0173
Epoch [7/10]: completed, Loss: 0.0166
Epoch [8/10]: completed, Loss: 0.0172
Epoch [9/10]: completed, Loss: 0.0154
Epoch [10/10]: completed, Loss: 0.0160
```

```
plt.figure(figsize=(10, 6))
plt.plot(range(1,len(training_loss)+1),training_loss, color='green')
plt.title("Training Loss vs Iterations")
plt.xlabel("No: of Iterations")
plt.ylabel("Training Loss")
plt.grid()
```

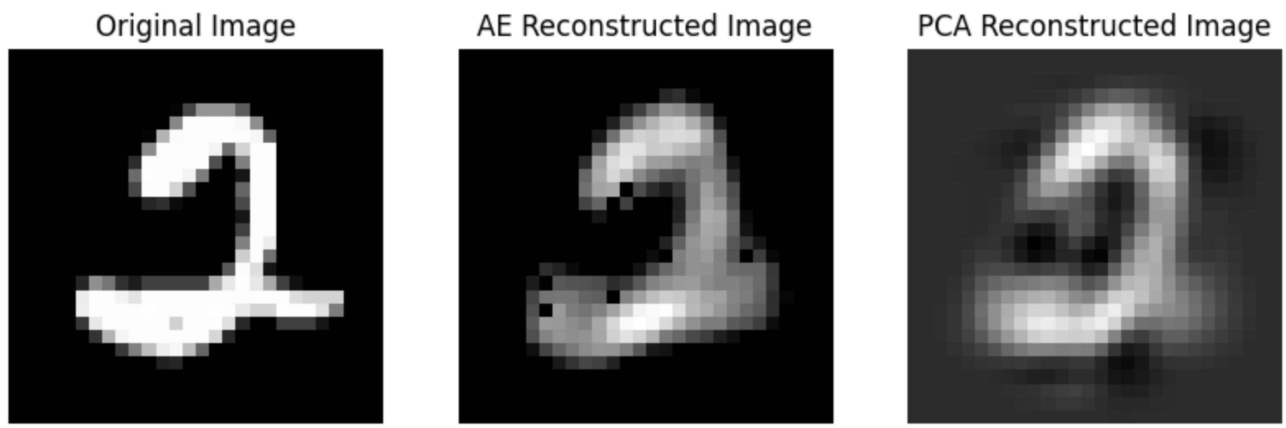



```
# Model evaluation
model1.eval() # evaluation mode
with torch.no_grad(): # disables gradient calculation

  # reshaping and passing test images through the AE
  for images in test_sample_loader:
    images = images.reshape(10, 28*28) # reshape from 2D to 1D
    outputs, _ = model1(images.float()) # reconstructed images

# visualization and plotting
plt.rcParams["figure.figsize"] = (10,6) # rcParams:set global parameters for plot
for i in range (5): # iterates over 10 images
  fig, (ax1, ax2, ax3) = plt.subplots(1,3)

  ax1.imshow(images[i].detach().numpy().reshape(28,28), cmap ='gray') # original
  ax1.set_title('Original Image')
  ax1.axis("off")

  ax2.imshow(outputs[i].detach().numpy().reshape(28,28), cmap ='gray') # reconctr
  ax2.set_title('AE Reconstructed Image')
  ax2.axis("off")

  ax3.imshow(reconstructed_test_data_sampled[i].reshape(28,28), cmap ='gray') # P
  ax3.set_title('PCA Reconstructed Image')
  ax3.axis("off")
  print("Reconstruction Error in AE:",np.dot(((images[i].detach().numpy()/255)-(o
                                          ((images[i].detach().numpy()/255)-(ou

  print("Reconstruction Error in PCA:",np.dot(((images[i].detach().numpy()/255)-(
                                            ((images[i].detach().numpy()/255)-(
  plt.show()
```

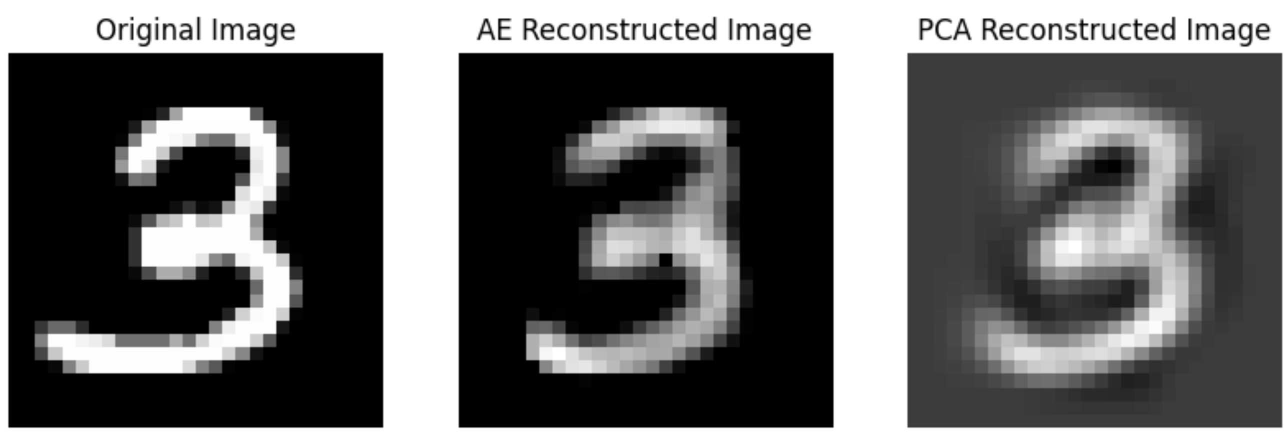
```
Reconstruction Error in AE: 5.114889141803459
Reconstruction Error in PCA: 4.906910216143052
```



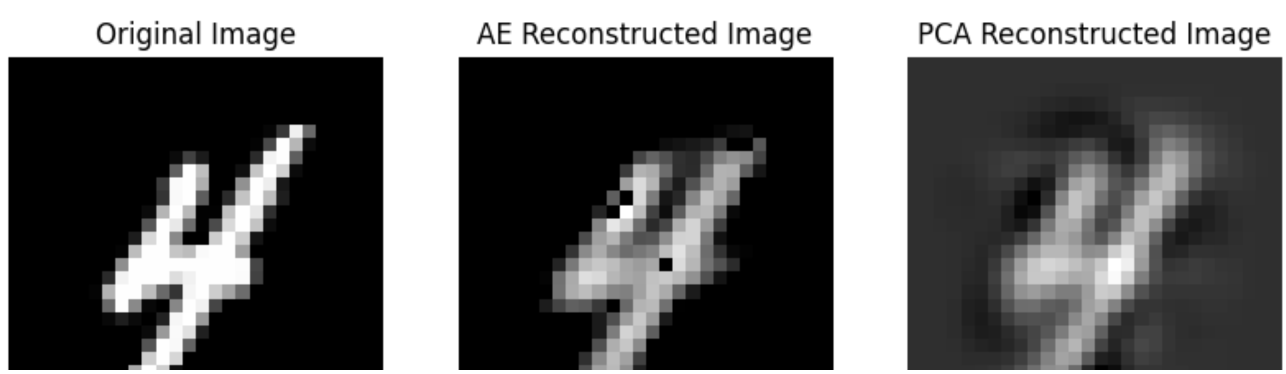

```
Reconstruction Error in AE: 22.499340382207624
Reconstruction Error in PCA: 16.862376030476067
```



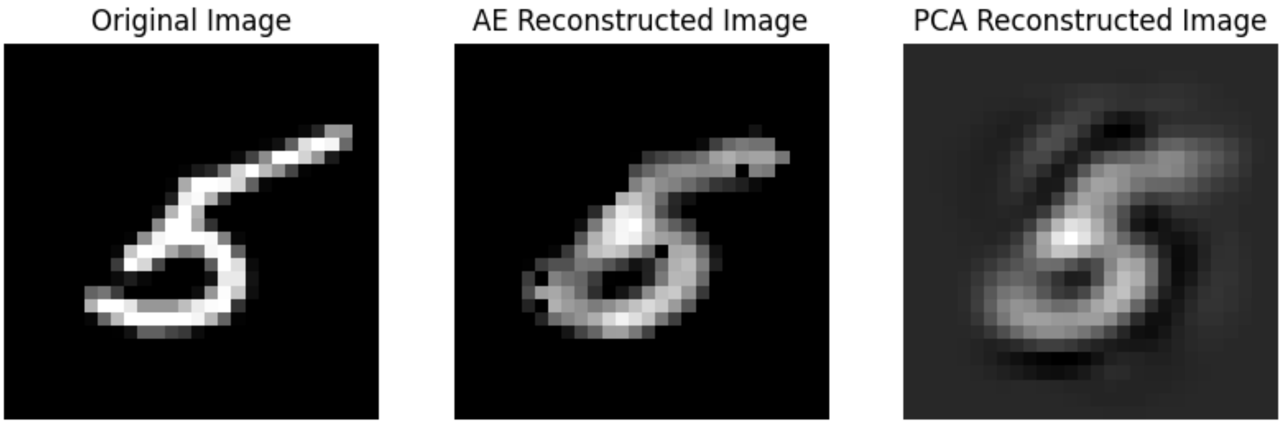

```
Reconstruction Error in AE: 15.27587834574187
Reconstruction Error in PCA: 16.0457551172948
```




```
Reconstruction Error in AE: 13.645043006066087
Reconstruction Error in PCA: 10.714796514097081
```



```
Reconstruction Error in AE: 14.066630936640893
Reconstruction Error in PCA: 14.848193155147694
```



Standard Autoencoder

```
# Under-complete AutoEncoder - one hidden layer
class Standard_AE(nn.Module):
  def __init__(self,hid):
    super(Standard_AE, self).__init__()
    self.hid = hid  # stores the no: of neurons in the hidden layer (hid)
    self.encoder = nn.Sequential(
        nn.Linear(784, self.hid),
        nn.ReLU()
        )
    self.decoder = nn.Sequential(
        nn.Linear(self.hid, 784),
        nn.ReLU()
        )

  def forward(self, x):
```

```
    x = self.encoder(x)
    encoded_output = x
    x = self.decoder(x)
    return x, encoded_output


# Hiddensize 64
model_hid_64 = Standard_AE(64)
criterion_hid64 = nn.MSELoss()
optimizer_hid64 = torch.optim.Adam(model_hid_64.parameters(), lr=learning_rate)

training_loss_hid64 = []

# Training loop for the specified number of epochs
for epoch in range(epochs):
  for images, labels in train_loader:
    images = images.reshape(images.shape[0], -1)
    outputs, _ = model_hid_64(images) # reconstructed images
    loss = criterion_hid64(outputs, images)
    training_loss_hid64.append(loss.item())

    optimizer_hid64.zero_grad()
    loss.backward()
    optimizer_hid64.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0182
Epoch [2/10]: completed, Loss: 0.0146
Epoch [3/10]: completed, Loss: 0.0134
Epoch [4/10]: completed, Loss: 0.0133
Epoch [5/10]: completed, Loss: 0.0125
Epoch [6/10]: completed, Loss: 0.0126
Epoch [7/10]: completed, Loss: 0.0125
Epoch [8/10]: completed, Loss: 0.0123
Epoch [9/10]: completed, Loss: 0.0112
Epoch [10/10]: completed, Loss: 0.0124
```

```
# Hiddensize 128
model_hid_128 = Standard_AE(128)
criterion_hid128=nn.MSELoss()
optimizer_hid128 = torch.optim.Adam(model_hid_128.parameters(), lr=learning_rate)

training_loss_hid128 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images=images.reshape(images.shape[0], -1)
    outputs, _ = model_hid_128(images)
    loss = criterion_hid128(outputs,images)
    training_loss_hid128.append(loss.item())
```

```
    optimizer_hid128.zero_grad()
    loss.backward()
    optimizer_hid128.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

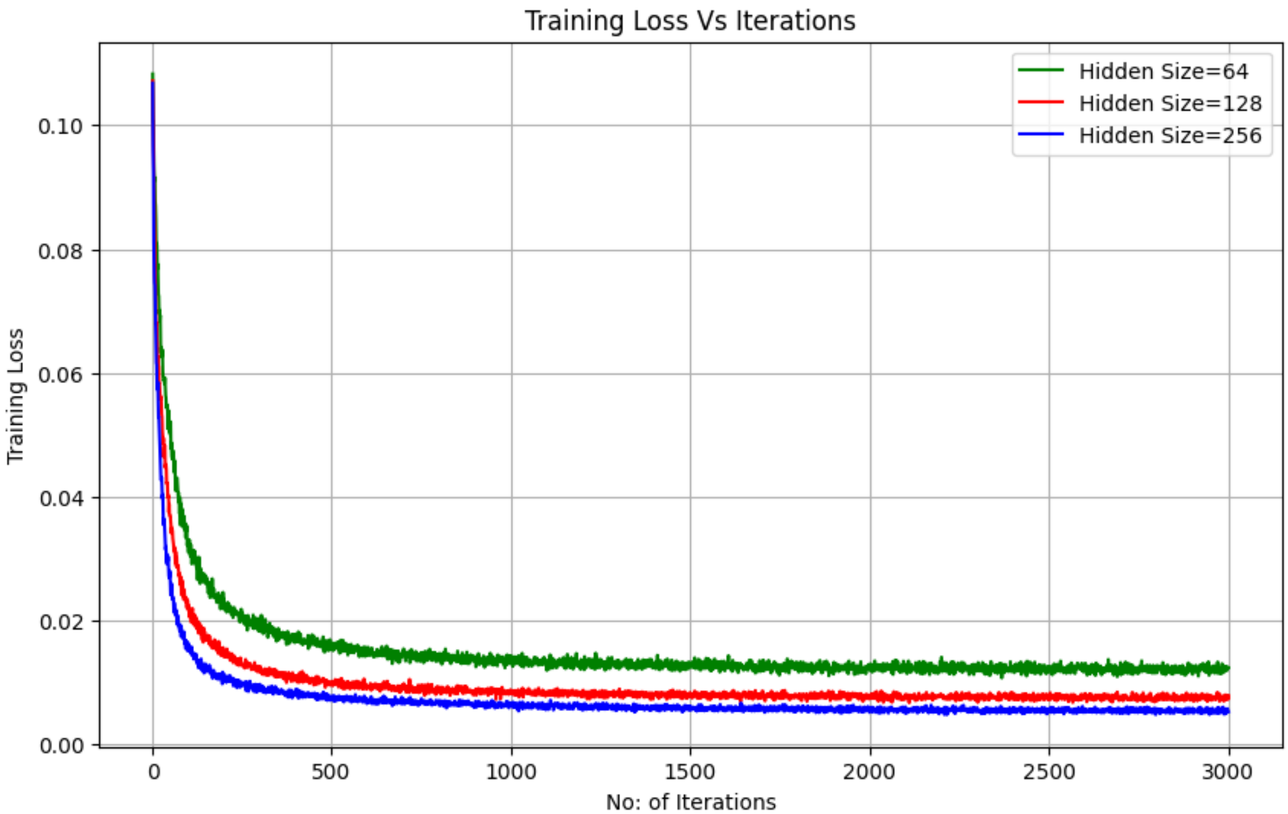
```
Epoch [1/10]: completed, Loss: 0.0122
Epoch [2/10]: completed, Loss: 0.0094
Epoch [3/10]: completed, Loss: 0.0085
Epoch [4/10]: completed, Loss: 0.0085
Epoch [5/10]: completed, Loss: 0.0078
Epoch [6/10]: completed, Loss: 0.0075
Epoch [7/10]: completed, Loss: 0.0081
Epoch [8/10]: completed, Loss: 0.0073
Epoch [9/10]: completed, Loss: 0.0074
Epoch [10/10]: completed, Loss: 0.0079
```

```
# Hiddensize 256
model_hid_256 = Standard_AE(256)
criterion_hid256=nn.MSELoss()
optimizer_hid256 = torch.optim.Adam(model_hid_256.parameters(),lr=learning_rate)

training_loss_hid256=[]

for epoch in range(epochs):
  for images,labels in train_loader:
    images=images.reshape(images.shape[0],-1)
    outputs, _ = model_hid_256(images)
    loss = criterion_hid256(outputs,images)
    training_loss_hid256.append(loss.item())

    optimizer_hid256.zero_grad()
    loss.backward()
    optimizer_hid256.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0097
Epoch [2/10]: completed, Loss: 0.0071
Epoch [3/10]: completed, Loss: 0.0062
Epoch [4/10]: completed, Loss: 0.0060
Epoch [5/10]: completed, Loss: 0.0057
Epoch [6/10]: completed, Loss: 0.0056
Epoch [7/10]: completed, Loss: 0.0050
Epoch [8/10]: completed, Loss: 0.0053
Epoch [9/10]: completed, Loss: 0.0053
Epoch [10/10]: completed, Loss: 0.0053
```

```
plt.plot(range(1,len(training_loss_hid64)+1),training_loss_hid64,label="Hidden Si
plt.plot(range(1,len(training_loss_hid64)+1),training_loss_hid128,label="Hidden S
```

```
plt.plot(range(1,len(training_loss_hid64)+1),training_loss_hid256,label="Hidden S
plt.legend()
plt.grid()
plt.title("Training Loss Vs Iterations")
plt.xlabel("No: of Iterations")
plt.ylabel("Training Loss")
plt.show()
```

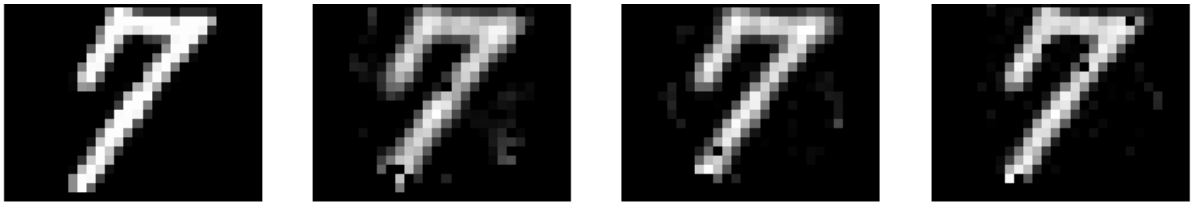



```
model_hid_64.eval()
with torch.no_grad():  # disable gradient calculation
  for images in test_sample_loader:
    # print(images.shape)
    images = images.reshape(10, 28*28)
    outputs_hid64, _ = model_hid_64(images.float()) # Forward pass through the mo
```

```
model_hid_128.eval()
with torch.no_grad():
  for images in test_sample_loader:
```

```
    for images in test_sample_loader:
      # print(images.shape)
      images = images.reshape(10,28*28)
      outputs_hid128, _ = model_hid_128(images.float())


model_hid_256.eval()
with torch.no_grad():
  for images in test_sample_loader:
    # print(images.shape)
    images = images.reshape(10,28*28)
    outputs_hid256, activations_hid256 = model_hid_256(images.float())


# Reconstruction from testset image
plt.rcParams["figure.figsize"] = (10,6)
i = 6
if i==6:
  fig, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)

  ax1.imshow(images[i].detach().numpy().reshape(28,28),cmap='gray')
  ax1.set_title('Original Image')
  ax1.axis("off")

  ax2.imshow(outputs_hid64[i].detach().numpy().reshape(28,28),cmap='gray')
  ax2.set_title('layer 64')
  ax2.axis("off")

  ax3.imshow(outputs_hid128[i].detach().numpy().reshape(28,28),cmap='gray')
  ax3.set_title('layer 128')
  ax3.axis("off")

  ax4.imshow(outputs_hid256[i].detach().numpy().reshape(28,28),cmap='gray')
  ax4.set_title('layer 256')
  ax4.axis("off")

  print("Reconstruction Error in hidden layer 64:",np.dot(((images[i].detach().nu
                                                          ((images[i].detach().numpy()/

  print("Reconstruction Error in hidden layer 128:",np.dot(((images[i].detach().n
                                                          ((images[i].detach().numpy()/

  print("Reconstruction Error in hidden layer 256:",np.dot(((images[i].detach().n
                                                          ((images[i].detach().numpy()/
```

```
Reconstruction Error in hidden layer 64: 11.198258444172145
Reconstruction Error in hidden layer 128: 7.300481213033256
Reconstruction Error in hidden layer 256: 5.585871585468324
```



```
# Reconstruction from non-digit images - Fashion MNIST o/p
test_data_fashion = torchvision.datasets.FashionMNIST(root="./",train=False,transfo
fashion_image_sample = test_data_fashion.data[9]
```

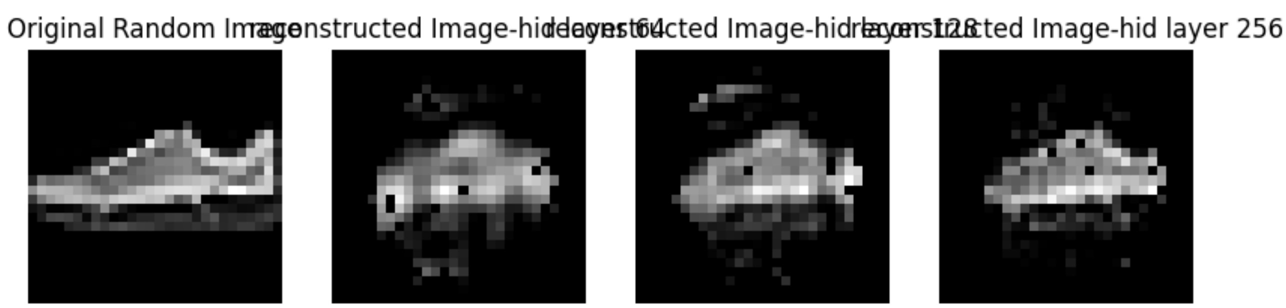
```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-i
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-i
100%|██████████| 26421880/26421880 [00:01<00:00, 13343461.45it/s]
Extracting ./FashionMNIST/raw/train-images-idx3-ubyte.gz to ./FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-l
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-l
100%|██████████| 29515/29515 [00:00<00:00, 211148.13it/s]
Extracting ./FashionMNIST/raw/train-labels-idx1-ubyte.gz to ./FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-in
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-in
100%|██████████| 4422102/4422102 [00:01<00:00, 3904504.94it/s]
Extracting ./FashionMNIST/raw/t10k-images-idx3-ubyte.gz to ./FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-la
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-la
100%|██████████| 5148/5148 [00:00<00:00, 15511693.24it/s]Extracting ./FashionM
```

```
model_hid_64.eval()
with torch.no_grad():
    images = fashion_image_sample.reshape(1,28*28)
    outputs_hid64,_ = model_hid_64(images.float())
```

```
model_hid_128.eval()
with torch.no_grad():
  images = fashion_image_sample.reshape(1,28*28)
  outputs_hid128,_ = model_hid_128(images.float())
```

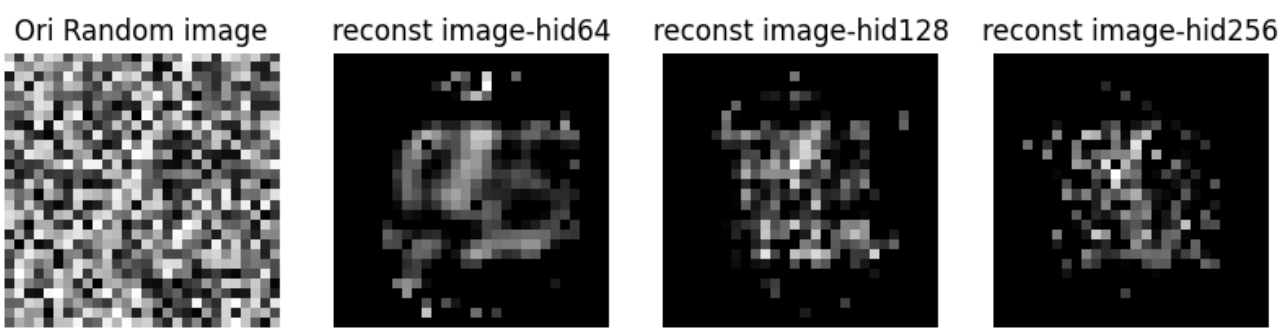
```
model_hid_256.eval()
with torch.no_grad():
  images = fashion_image_sample.reshape(1,28*28)
  outputs_hid256,_ = model_hid_256(images.float())
```

```
    outputs_hid256,_ = model_hid_256(images.float())


plt.rcParams["figure.figsize"] = (10,6)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)

ax1.imshow(fashion_image_sample.detach().numpy().reshape(28,28),cmap='gray')
ax1.set_title('Original Random Image')
ax1.axis("off")

ax2.imshow(outputs_hid64.detach().numpy().reshape(28,28),cmap='gray')
ax2.set_title('layer 64')
ax2.axis("off")

ax3.imshow(outputs_hid128.detach().numpy().reshape(28,28),cmap='gray')
ax3.set_title('layer 128')
ax3.axis("off")

ax4.imshow(outputs_hid256.detach().numpy().reshape(28,28),cmap='gray')
ax4.set_title('layer 256')
ax4.axis("off")
plt.show()
```




```
# random noise images
torch.manual_seed(0)
random_image = torch.randint(low=0, high=255,size=(1,28,28))


model_hid_64.eval()
with torch.no_grad():
    images = random_image.reshape(1,28*28)
    outputs_hid64,_ = model_hid_64(images.float())


model_hid_128.eval()
with torch.no_grad():
```

```
  images = random_image.reshape(1,28*28)
  outputs_hid128,_ = model_hid_128(images.float())


model_hid_256.eval()
with torch.no_grad():
  images = random_image.reshape(1,28*28)
  outputs_hid256,_ = model_hid_256(images.float())


plt.rcParams["figure.figsize"] = (10,6)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)

ax1.imshow(random_image.numpy().reshape(28,28),cmap='gray')
ax1.set_title('Ori Random image')
ax1.axis("off")

ax2.imshow(outputs_hid64.detach().numpy().reshape(28,28),cmap='gray')
ax2.set_title('reconst image-hid64')
ax2.axis("off")

ax3.imshow(outputs_hid128.detach().numpy().reshape(28,28),cmap='gray')
ax3.set_title('reconst image-hid128')
ax3.axis("off")

ax4.imshow(outputs_hid256.detach().numpy().reshape(28,28),cmap='gray')
ax4.set_title('reconst image-hid256')
ax4.axis("off")
plt.show()
```



Sparse Autoencoders

```
# over-complete autoencoder (AE) with sparsity regularization using L1 penalty on
```

```
class Sparse_AE(nn.Module):
  def __init__(self):
    super(Sparse_AE, self).__init__()
    self.encoder = nn.Sequential(
        nn.Linear(784, 1024), # Over-complete layer
        nn.ReLU())
    self.decoder = nn.Sequential(
        nn.Linear(1024, 784), # Reconstruction back to input dimension
        nn.ReLU())

  def forward(self,x):
    x = self.encoder(x) # encoded output
    encoded_output = x
    l1_norm = torch.norm(x, p=1) # calculate L1 norm of the activations
    x = self.decoder(x) # Get reconstructed output
    return x, l1_norm, encoded_output


sparsity_param  = 0.5*1e-6 # Sparsity regularization weight
model_sparse1 = Sparse_AE()
criterion_sparse1 = nn.MSELoss()
optimizer_sparse1 = torch.optim.Adam(model_sparse1.parameters(),lr=learning_rate)

training_loss_sparse1 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    outputs, l1_norm, _ = model_sparse1(images)
    loss = criterion_sparse1(outputs,images) + sparsity_param * l1_norm
    training_loss_sparse1.append(loss.item())

    optimizer_sparse1.zero_grad()
    loss.backward()
    optimizer_sparse1.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0129
Epoch [2/10]: completed, Loss: 0.0102
Epoch [3/10]: completed, Loss: 0.0091
Epoch [4/10]: completed, Loss: 0.0086
Epoch [5/10]: completed, Loss: 0.0076
Epoch [6/10]: completed, Loss: 0.0079
Epoch [7/10]: completed, Loss: 0.0071
Epoch [8/10]: completed, Loss: 0.0075
Epoch [9/10]: completed, Loss: 0.0065
Epoch [10/10]: completed, Loss: 0.0070
```

```
sparsity_param  = 1.5*1e-6
model_sparse2 = Sparse_AE()
```

```
criterion_sparse2 = nn.MSELoss()
optimizer_sparse2 = torch.optim.Adam(model_sparse2.parameters(),lr=learning_rate)

training_loss_sparse2=[]

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    outputs, l1_norm, _ = model_sparse2(images)
    loss = criterion_sparse2(outputs,images) + sparsity_param * l1_norm
    training_loss_sparse2.append(loss.item())

    optimizer_sparse2.zero_grad()
    loss.backward()
    optimizer_sparse2.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0171
Epoch [2/10]: completed, Loss: 0.0144
Epoch [3/10]: completed, Loss: 0.0123
Epoch [4/10]: completed, Loss: 0.0114
Epoch [5/10]: completed, Loss: 0.0110
Epoch [6/10]: completed, Loss: 0.0106
Epoch [7/10]: completed, Loss: 0.0093
Epoch [8/10]: completed, Loss: 0.0092
Epoch [9/10]: completed, Loss: 0.0088
Epoch [10/10]: completed, Loss: 0.0085
```

```
sparsity_param = 3*1e-6 # Sparsity regularization weight
model_sparse3 = Sparse_AE()
criterion_sparse3 = nn.MSELoss()
optimizer_sparse3 = torch.optim.Adam(model_sparse3.parameters(),lr=learning_rate)

training_loss_sparse3 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0], -1)
    outputs,l1_norm,_ = model_sparse3(images)
    loss = criterion_sparse3(outputs, images) + sparsity_param * l1_norm
    training_loss_sparse3.append(loss.item())

    optimizer_sparse3.zero_grad()
    loss.backward()
    optimizer_sparse3.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

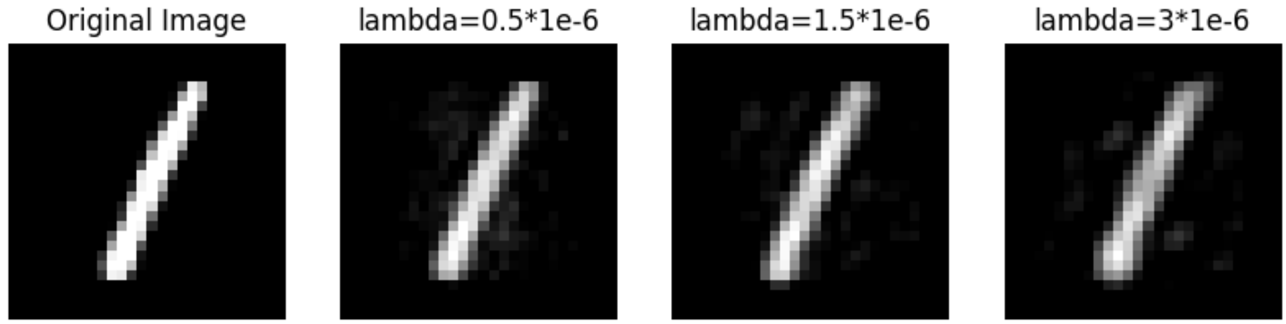
```
Epoch [1/10]: completed, Loss: 0.0249
Epoch [2/10]: completed, Loss: 0.0196
```

```
Epoch [3/10]: completed, Loss: 0.0187
Epoch [4/10]: completed, Loss: 0.0174
Epoch [5/10]: completed, Loss: 0.0162
Epoch [6/10]: completed, Loss: 0.0151
Epoch [7/10]: completed, Loss: 0.0142
Epoch [8/10]: completed, Loss: 0.0139
Epoch [9/10]: completed, Loss: 0.0142
Epoch [10/10]: completed, Loss: 0.0132
```

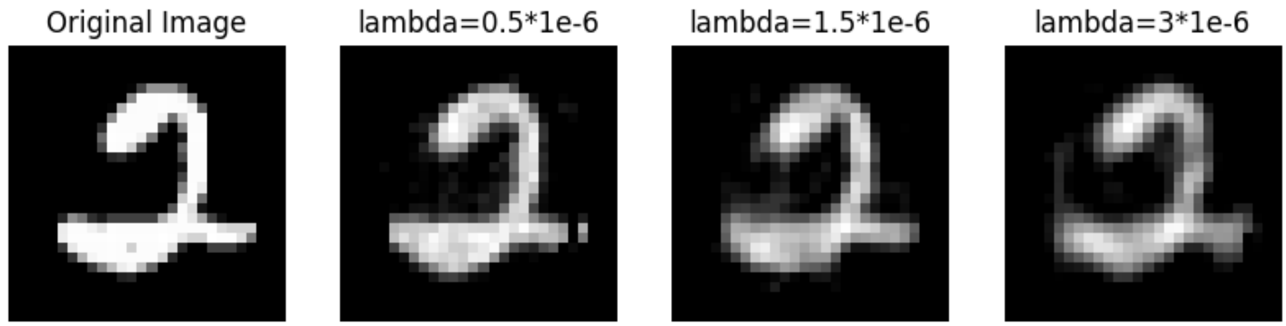
```
plt.plot(range(1,len(training_loss_sparse3)+1),training_loss_sparse1,label="lambd
plt.plot(range(1,len(training_loss_sparse3)+1),training_loss_sparse2,label="lambd
plt.plot(range(1,len(training_loss_sparse3)+1),training_loss_sparse3,label="lambd
plt.legend()
plt.grid()
plt.title("Training Loss Vs Iterations")
plt.xlabel("Number of Iterations")
plt.ylabel("Training Loss")
plt.show()
```

```
model_sparse1.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    outputs_sparse1, _ ,activation_sparse1 = model_sparse1(images.float())



model_sparse2.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    outputs_sparse2, _ , activation_sparse2 = model_sparse2(images.float())


model_sparse3.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    outputs_sparse3, _ , activation_sparse3 = model_sparse3(images.float())


plt.rcParams["figure.figsize"] = (10,6)

for i in range(5):
  fig, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)

  ax1.imshow(images[i].detach().numpy().reshape(28,28),cmap='gray')
  ax1.set_title('Original Image')
  ax1.axis("off")

  ax2.imshow(outputs_sparse1[i].detach().numpy().reshape(28,28),cmap='gray')
  ax2.set_title('lambda=0.5*1e-6')
  ax2.axis("off")

  ax3.imshow(outputs_sparse2[i].detach().numpy().reshape(28,28),cmap='gray')
  ax3.set_title('lambda=1.5*1e-6')
  ax3.axis("off")

  ax4.imshow(outputs_sparse3[i].detach().numpy().reshape(28,28),cmap='gray')
  ax4.set_title('lambda=3*1e-6')
  ax4.axis("off")

  plt.show()

  print("Reconstruction Error in SparseAE lambda = 0.6*1e-6 :",np.dot(((images[i].c
                                                                ((images[i].detac

  print("Reconstruction Error in SparseAE lambda = 1.5*1e-6:",np.dot(((images[i].de
                                                                  ((images[i].det
```

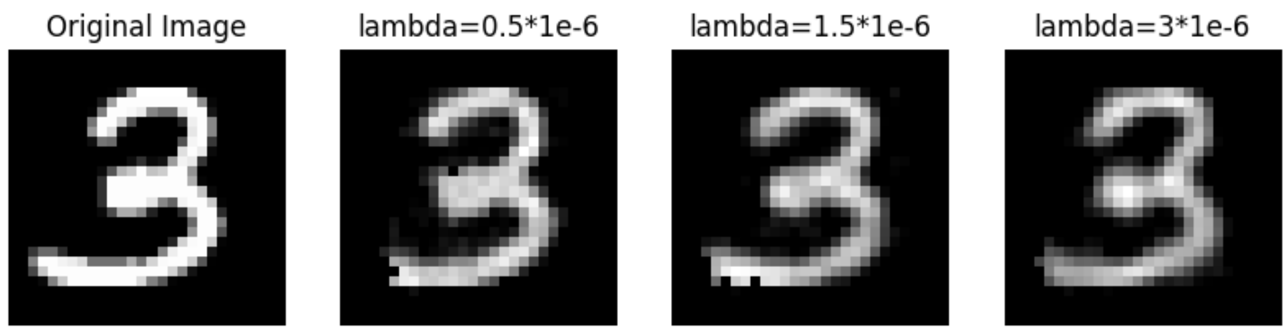
```
print("Reconstruction Error in SparseAE lambda = 3*1e-6:",np.dot(((images[i].deta
                                                                  ((images[i].det
```



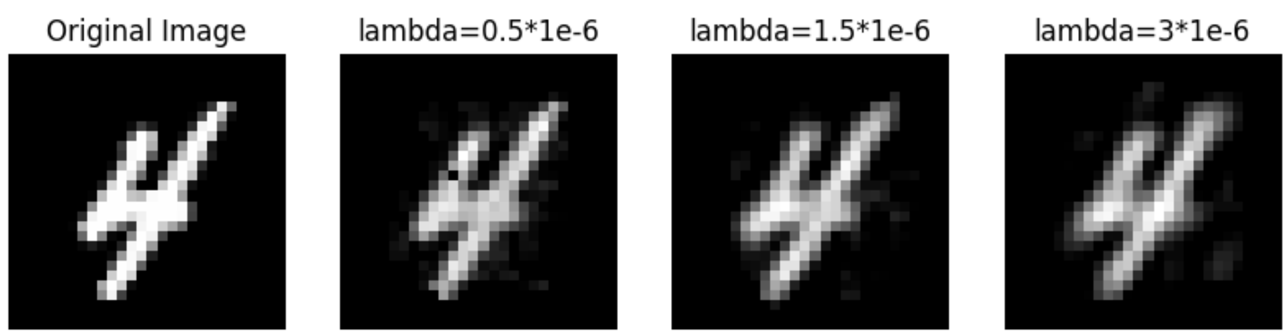

```
Reconstruction Error in SparseAE lambda = 0.6*1e-6 : 2.208584472857042
Reconstruction Error in SparseAE lambda = 1.5*1e-6: 1.9335901925147088
Reconstruction Error in SparseAE lambda = 3*1e-6: 3.6661419552609127
```




```
Reconstruction Error in SparseAE lambda = 0.6*1e-6 : 7.516961848370433
Reconstruction Error in SparseAE lambda = 1.5*1e-6: 7.161093606775397
Reconstruction Error in SparseAE lambda = 3*1e-6: 11.231844113122616
```

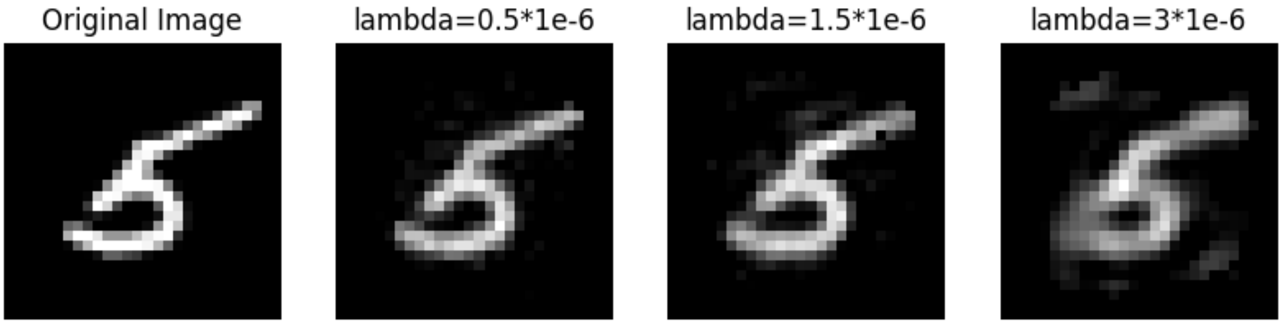



```
Reconstruction Error in SparseAE lambda = 0.6*1e-6 : 11.913395591403894
Reconstruction Error in SparseAE lambda = 1.5*1e-6: 9.469317564296931
Reconstruction Error in SparseAE lambda = 3*1e-6: 9.207602082149549
```




```
Reconstruction Error in SparseAE lambda = 0.6*1e-6 : 5.362101635436673
```

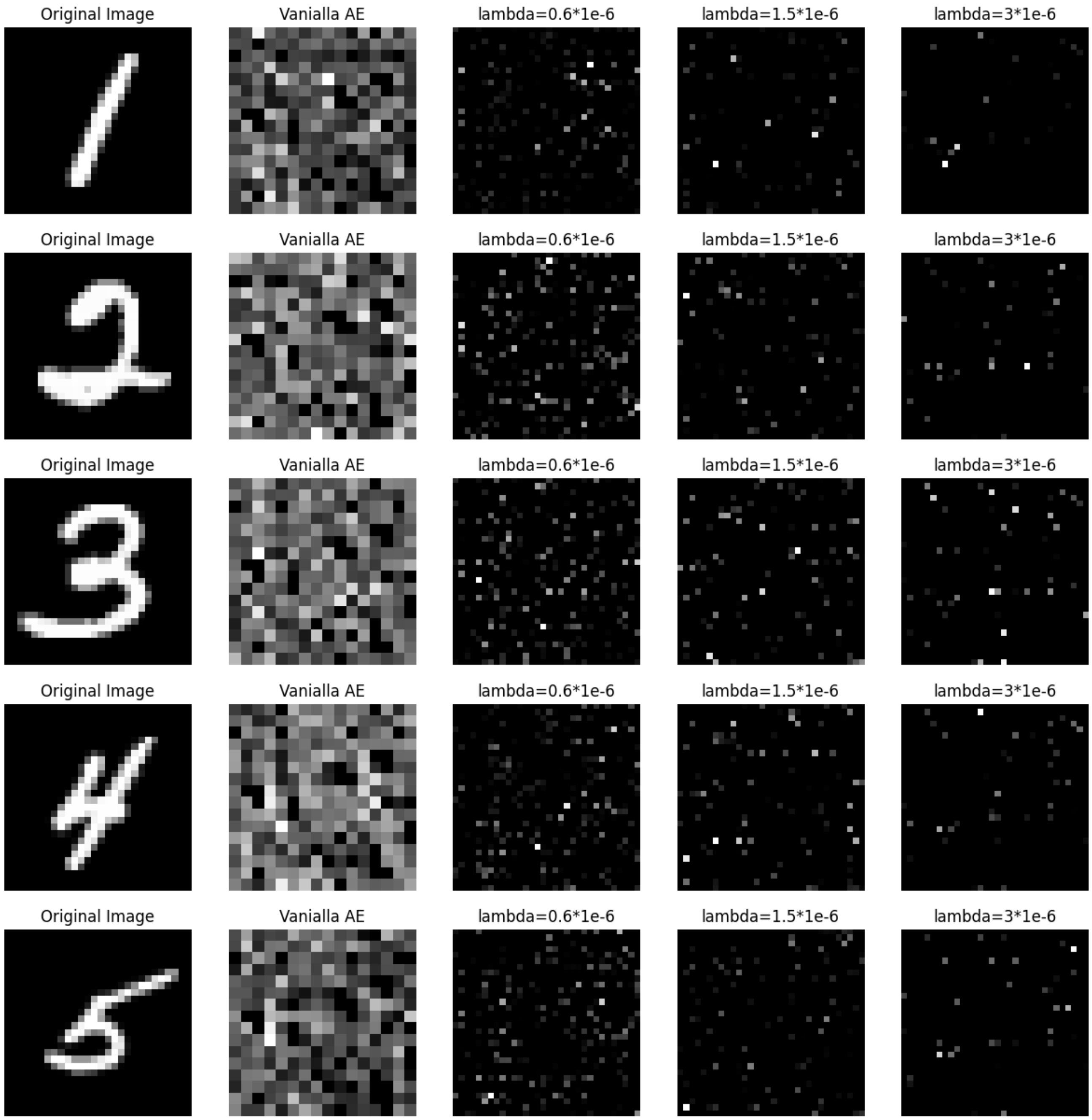
```
Reconstruction Error in SparseAE lambda = 1.5*1e-6: 4.371462238244634
Reconstruction Error in SparseAE lambda = 3*1e-6: 7.5787287819801605
```




```
Reconstruction Error in SparseAE lambda = 0.6*1e-6 : 4.129679657692563
Reconstruction Error in SparseAE lambda = 1.5*1e-6: 5.395042252176756
Reconstruction Error in SparseAE lambda = 3*1e-6: 12.234988018390908
```

```
# visualize the learned filters of the Sparse AE as images
plt.rcParams["figure.figsize"] = (15,6)
for i in range(5):
  fig, (ax1, ax2, ax3, ax4, ax5) = plt.subplots(1,5)
  ax1.imshow(images[i].detach().numpy().reshape(28,28),cmap='gray')
  ax1.set_title('Original Image')
  ax1.axis("off")

  ax2.imshow(np.array(activations_hid256.detach().numpy())[i].reshape(int(np.sqrt
  ax2.set_title('Vanialla AE')
  ax2.axis("off")

  ax3.imshow(np.array(activation_sparse1.detach().numpy())[i].reshape(int(np.sqrt
  ax3.set_title('lambda=0.6*1e-6')
  ax3.axis("off")

  ax4.imshow(np.array(activation_sparse2.detach().numpy())[i].reshape(int(np.sqrt
  ax4.set_title('lambda=1.5*1e-6')
  ax4.axis("off")

  ax5.imshow(np.array(activation_sparse3.detach().numpy())[i].reshape(int(np.sqrt
  ax5.set_title('lambda=3*1e-6')
  ax5.axis("off")

plt.show()
```

```
plt.show()
```


Original Image
Vanialla AE
lambda=0.6*1e-6
lambda=1.5*1e-6
lambda=3*1e-6

Denoising Autoencoders

```
class Denoising_AE(nn.Module):
  def __init__(self):
    super(Denoising_AE, self).__init__()
    self.encoder = nn.Sequential( # encoder
        nn.Linear(784, 256),
        nn.ReLU())
    self.decoder = nn.Sequential( # decoder
        nn.Linear(256, 784),
        nn.ReLU())

  def forward(self,x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
```

```
def add_noise(img, noise_val):
  noise = torch.randn(img.size()) * noise_val
  noisy_img = img + noise
  return noisy_img
```

```
# Hyperparameters
learning_rate = 0.0001
epochs = 10
batch_size = 200
```

```
model_denoise1 = Denoising_AE()
criterion_denoise1 = nn.MSELoss()
```

```
criterion_denoise1 = nn.MSELoss()
optimizer_denoise1 = torch.optim.Adam(model_denoise1.parameters(),lr=learning_rat

training_loss_denoise1 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    noisy_images = add_noise(images, 0.3)
    outputs = model_denoise1(noisy_images)
    loss = criterion_denoise1(outputs, images)
    training_loss_denoise1.append(loss.item())

    optimizer_denoise1.zero_grad()
    loss.backward()
    optimizer_denoise1.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0285
Epoch [2/10]: completed, Loss: 0.0188
Epoch [3/10]: completed, Loss: 0.0142
Epoch [4/10]: completed, Loss: 0.0137
Epoch [5/10]: completed, Loss: 0.0121
Epoch [6/10]: completed, Loss: 0.0116
Epoch [7/10]: completed, Loss: 0.0114
Epoch [8/10]: completed, Loss: 0.0113
Epoch [9/10]: completed, Loss: 0.0107
Epoch [10/10]: completed, Loss: 0.0108
```

```
model_denoise2 = Denoising_AE()
criterion_denoise2 = nn.MSELoss()
optimizer_denoise2 = torch.optim.Adam(model_denoise2.parameters(),lr=learning_rat

training_loss_denoise2 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    noisy_images = add_noise(images,0.5)
    outputs = model_denoise2(noisy_images)
    loss = criterion_denoise2(outputs, images)
    training_loss_denoise2.append(loss.item())

    optimizer_denoise2.zero_grad()
    loss.backward()
    optimizer_denoise2.step()
  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0349
Epoch [2/10]: completed, Loss: 0.0257
Epoch [3/10]: completed, Loss: 0.0215
```

```
Epoch [3/10]: completed, Loss: 0.0213
Epoch [4/10]: completed, Loss: 0.0194
Epoch [5/10]: completed, Loss: 0.0182
Epoch [6/10]: completed, Loss: 0.0187
Epoch [7/10]: completed, Loss: 0.0176
Epoch [8/10]: completed, Loss: 0.0172
Epoch [9/10]: completed, Loss: 0.0177
Epoch [10/10]: completed, Loss: 0.0178
```

```
model_denoise3 = Denoising_AE()
criterion_denoise3 = nn.MSELoss()
optimizer_denoise3 = torch.optim.Adam(model_denoise3.parameters(),lr=learning_rat

training_loss_denoise3 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    noisy_images = add_noise(images, 0.8)
    outputs = model_denoise3(noisy_images)
    loss = criterion_denoise3(outputs, images)
    training_loss_denoise3.append(loss.item())

    optimizer_denoise3.zero_grad()
    loss.backward()
    optimizer_denoise3.step()
  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

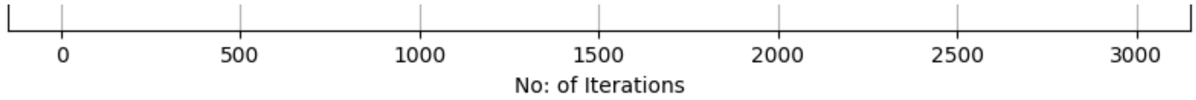
```
Epoch [1/10]: completed, Loss: 0.0437
Epoch [2/10]: completed, Loss: 0.0355
Epoch [3/10]: completed, Loss: 0.0312
Epoch [4/10]: completed, Loss: 0.0293
Epoch [5/10]: completed, Loss: 0.0276
Epoch [6/10]: completed, Loss: 0.0280
Epoch [7/10]: completed, Loss: 0.0281
Epoch [8/10]: completed, Loss: 0.0266
Epoch [9/10]: completed, Loss: 0.0260
Epoch [10/10]: completed, Loss: 0.0238
```

```
model_denoise4 = Denoising_AE()
criterion_denoise4 = nn.MSELoss()
optimizer_denoise4 = torch.optim.Adam(model_denoise4.parameters(),lr=learning_rat

training_loss_denoise4 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    images = images.reshape(images.shape[0],-1)
    noisy_images = add_noise(images, 0.9)
    outputs = model_denoise4(noisy_images)
    loss = criterion_denoise4(outputs, images)
    training_loss_denoise4.append(loss.item())
```

```
    training_loss_denoise4.append(loss.item())

    optimizer_denoise4.zero_grad()
    loss.backward()
    optimizer_denoise4.step()
  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0513
Epoch [2/10]: completed, Loss: 0.0390
Epoch [3/10]: completed, Loss: 0.0348
Epoch [4/10]: completed, Loss: 0.0324
Epoch [5/10]: completed, Loss: 0.0317
Epoch [6/10]: completed, Loss: 0.0287
Epoch [7/10]: completed, Loss: 0.0285
Epoch [8/10]: completed, Loss: 0.0289
Epoch [9/10]: completed, Loss: 0.0288
Epoch [10/10]: completed, Loss: 0.0276
```

```
plt.rcParams["figure.figsize"] = (10, 6)

plt.plot(range(1,len(training_loss_denoise1)+1),training_loss_denoise1,label="noi
plt.plot(range(1,len(training_loss_denoise1)+1),training_loss_denoise2,label="noi
plt.plot(range(1,len(training_loss_denoise1)+1),training_loss_denoise3,label="noi
plt.plot(range(1,len(training_loss_denoise1)+1),training_loss_denoise4,label="noi
plt.legend()
plt.grid()
plt.title("Training Loss Vs Iterations")
plt.xlabel("No: of Iterations")
plt.ylabel("Training Loss")
plt.show()
```




```
# With different noise factor 0.3,0.5,0.8,0.9 given to Vanilla AE
model_hid_256.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    noisy_images = add_noise(images,0.3)
    outputs_hid256_01, activations_hid256 = model_hid_256(noisy_images.float())
```

```
model_hid_256.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    noisy_images = add_noise(images,0.5)
    outputs_hid256_02, activations_hid256 = model_hid_256(noisy_images.float())
```

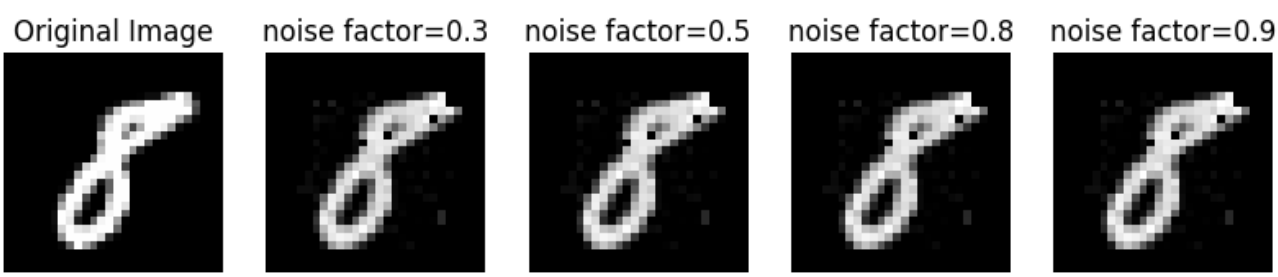
```
model_hid_256.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    noisy_images = add_noise(images, 0.8)
    outputs_hid256_03, activations_hid256 = model_hid_256(noisy_images.float())
```

```
model_hid_256.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,28*28)
    noisy_images = add_noise(images, 0.9)
    outputs_hid256_04, activations_hid256 = model_hid_256(noisy_images.float())
```

```
plt.rcParams["figure.figsize"] = (10,6)
i = 7
fig, (ax1, ax2,ax3,ax4,ax5) = plt.subplots(1,5)
```

```
ax1.imshow(images[i].detach().numpy().reshape(28,28),cmap='gray')
ax1.set_title('Original Image')
ax1.axis("off")

ax2.imshow(outputs_hid256_01[i].detach().numpy().reshape(28,28),cmap='gray')
ax2.set_title('noise factor=0.3')
ax2.axis("off")

ax3.imshow(outputs_hid256_02[i].detach().numpy().reshape(28,28),cmap='gray')
ax3.set_title('noise factor=0.5')
ax3.axis("off")

ax4.imshow(outputs_hid256_03[i].detach().numpy().reshape(28,28),cmap='gray')
ax4.set_title('noise factor=0.8')
ax4.axis("off")

ax5.imshow(outputs_hid256_04[i].detach().numpy().reshape(28,28),cmap='gray')
ax5.set_title('noise factor=0.9')
ax5.axis("off")

print("Reconstruction Error in VanillaAE with noise factor = 0.3 :",np.dot(((imag
                                                                   ((image

print("Reconstruction Error in VanillaAE with noise factor = 0.5 :",np.dot(((imag
                                                                   ((image

print("Reconstruction Error in VanillaAE with noise factor = 0.8 :",np.dot(((imag
                                                                   ((image

print("Reconstruction Error in VanillaAE with noise factor = 0.9 :",np.dot(((imag
                                                                   ((image
```

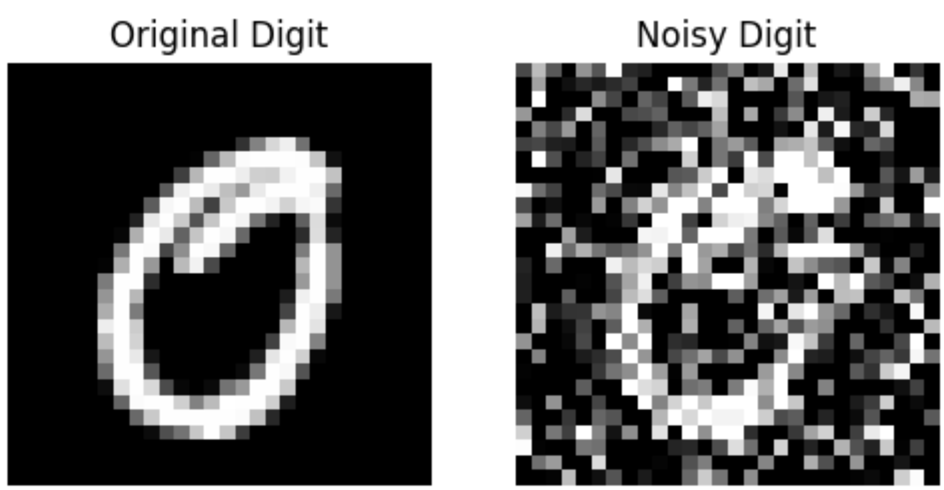
```
Reconstruction Error in VanillaAE with noise factor = 0.3 : 4.758285692172574
Reconstruction Error in VanillaAE with noise factor = 0.5 : 4.770937421211641!
Reconstruction Error in VanillaAE with noise factor = 0.8 : 4.758050556874504
Reconstruction Error in VanillaAE with noise factor = 0.9 : 4.75845746901802
```



Manifold learning

Manifold learning

```
# High-dimensional spaces are sparse: In a 784-dimensional space, valid digit ima
# Random noise disrupts correlations between pixels
# Perception in high dimensions: In high-dimensional spaces, distances between po


# Select a random digit
random_digit, _ = next(iter(train_loader))

# Original digit
original_digit = random_digit[0].squeeze().numpy()  # Shape (28, 28)

# Add random noise
noise_factor = 0.5
random_noise = np.random.normal(loc=0.0, scale=noise_factor, size=original_digit.
noisy_digit = np.clip(original_digit + random_noise, 0.0, 1.0)  # Keep values in

# Plotting the original and noisy digits
plt.figure(figsize=(6, 3))

# Original digit
plt.subplot(1, 2, 1)
plt.imshow(original_digit, cmap='gray')
plt.title('Original Digit')
plt.axis('off')

# Noisy digit
plt.subplot(1, 2, 2)
plt.imshow(noisy_digit, cmap='gray')
plt.title('Noisy Digit')
plt.axis('off')

plt.show()
```



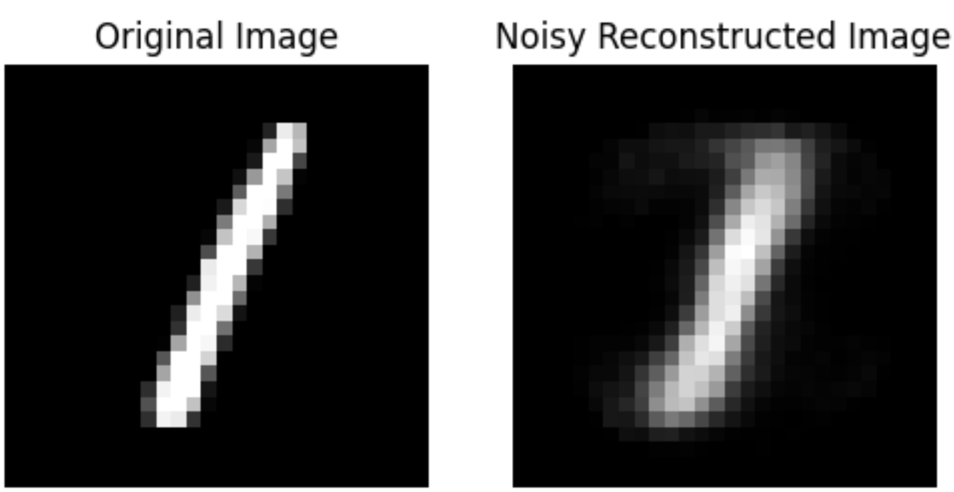
```
class AE_manifold(nn.Module):
    def __init__(self):
        super(AE_manifold, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(784, 64),
            nn.ReLU(),
            nn.Linear(64, 8)
        )
        self.decoder = nn.Sequential(
            nn.Linear(8, 64),
            nn.ReLU(),
            nn.Linear(64, 784),
            nn.Sigmoid()  # Sigmoid to ensure output is in range [0, 1]
        )

    def forward(self, x):
        latent_representation = self.encoder(x)  # Get latent representation
        reconstructed_image = self.decoder(latent_representation)  # Reconstruct
        return reconstructed_image, latent_representation  # Return both reconstr



# Initialize model, loss function, and optimizer
model = AE_manifold()
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
training_loss_manifold = []
for epoch in range(epochs):
    for images, labels in train_loader:
        images = images.reshape(images.shape[0], -1)  # 2D to 1D vector
        outputs, _ = model(images)  # Reconstructed images and latent representat
        loss = criterion(outputs, images)  # Reconstruction error
        training_loss_manifold.append(loss.item())

        optimizer.zero_grad()  # Clears the gradients from the previous batch
        loss.backward()  # Backpropagation
        optimizer.step()  # Update the model parameters

    print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.i
```

```
Epoch [1/10]: completed, Loss: 0.0731
Epoch [2/10]: completed, Loss: 0.0668
Epoch [3/10]: completed, Loss: 0.0650
Epoch [4/10]: completed, Loss: 0.0565
Epoch [5/10]: completed, Loss: 0.0487
Epoch [6/10]: completed, Loss: 0.0465
```

```
Epoch [7/10]: completed, Loss: 0.0438
Epoch [8/10]: completed, Loss: 0.0428
Epoch [9/10]: completed, Loss: 0.0397
Epoch [10/10]: completed, Loss: 0.0378
```

```
# # Autoencoder class
# class AE_manifold(nn.Module):
#   def __init__(self):
#     super(AE_manifold, self).__init__()
#     self.encoder = nn.Sequential(
#     nn.Linear(784, 64),
#     nn.ReLU(),
#     nn.Linear(64, 8)
#     )
#     self.decoder = nn.Sequential(
#     nn.Linear(8, 64),
#     nn.ReLU(),
#     nn.Linear(64, 784),
#     nn.Sigmoid()  # Sigmoid to ensure output is in range [0, 1]
#     )

#   def forward(self, x):
#     x = self.encoder(x)
#     x = self.decoder(x)
#     return x


# # Initialize model, loss function, and optimizer
# model = AE_manifold()
# criterion = nn.MSELoss()
# optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# # Training loop
# training_loss_manifold = []
# for epoch in range(epochs):
#     for images, labels in train_loader:
#         images = images.reshape(images.shape[0], -1)  # 2D to 1D vector
#         outputs = model(images)  # Reconstructed images (only one value returne
#         loss = criterion(outputs, images)  # Reconstruction error
#         training_loss_manifold.append(loss.item())

#         optimizer.zero_grad()  # Clears the gradients from the previous batch
#         loss.backward()  # Backpropagation
#         optimizer.step()  # Update the model parameters

#     print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss



# Testing on the test set
test_image = test_dataset[9705].unsqueeze(0)  # Take a test image and add a batch
```

```
test_image = test_dataset[9705].unsqueeze(0)  # Take a test image and add a batch
test_image = test_image.float() / 255.0  # Normalize

# Pass through encoder to get latent representation
with torch.no_grad():
    reconstructed_image, latent_representation = model(test_image)  # Now it shou

# Add noise to the latent representation
noise_factor = 0.5
noisy_latent = latent_representation + torch.randn(latent_representation.shape) *

# Reconstruct the image from the noisy latent representation
noisy_reconstructed_image = model.decoder(noisy_latent)

# Reshape images for plotting (28x28)
original_image = test_image.view(28, 28).detach().numpy()
noisy_image = noisy_reconstructed_image.view(28, 28).detach().numpy()



# Plot the original and noisy reconstructed images
plt.figure(figsize=(6, 3))
plt.subplot(1, 2, 1)
plt.imshow(original_image, cmap='gray')
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(noisy_image, cmap='gray')
plt.title('Noisy Reconstructed Image')
plt.axis('off')

plt.show()
```



Convolutional Autoencoders

Convolutional Autoencoders

```
class ConvolutionAE_unpooling(nn.Module):   # class definition
  def __init__(self):   # constructor
    super(ConvolutionAE_unpooling,self).__init__()

    # encoder module
    self.enc_conv1 = nn.Sequential(
        nn.Conv2d(1,8, kernel_size = 3, stride = 1, padding = 1), # 28x28x1 to 14
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2),return_indices = True)
    )
    self.enc_conv2 = nn.Sequential(
        nn.Conv2d(8,16, kernel_size = 3, stride = 1, padding = 1), # 14x14x8 to 7
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2), return_indices = True)
    )
    self.enc_conv3 = nn.Sequential(
        nn.Conv2d(16,16, kernel_size = 3, stride = 1, padding = 1), # 7x7x16 to 3
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2), return_indices = True)
    )

    # decoder module
    self.dec_conv1 = nn.Sequential(nn.Identity()) # 7x7x16 to 7x7x16
    self.dec_conv2 = nn.Sequential(
        nn.Conv2d(16,8, kernel_size = 3, stride = 1, padding = 1), # 14x14x16 to
        nn.ReLU()
    )
    self.dec_conv3 = nn.Sequential(
        nn.Conv2d(8,1, kernel_size = 3, stride = 1, padding = 1), # 28x28x8 to 28
        nn.ReLU()
    )

    # unpooling operation
    self.unpool = nn.MaxUnpool2d(kernel_size = (2,2))


  def forward(self, x):  # forward pass
    enc_input,indices1  = self.enc_conv1(x.float())  # 28x28x1 to 14x14x8
    enc_input,indices2  = self.enc_conv2(enc_input) #14x14x8 to 7x7x16
    enc_input,indices3  = self.enc_conv3(enc_input) #7x7x16 to 3x3x16

    reconst_img = self.unpool(enc_input,indices3,output_size=torch.Size([batch_si
    reconst_img = self.dec_conv1(reconst_img) #7x7x16 to 7x7x16
    reconst_img = self.unpool(reconst_img,indices2) #7x7x16 to 14x14x16
    reconst_img = self.dec_conv2(reconst_img)#14x14x16 to 14x14x8
    reconst_img = self.unpool(reconst_img,indices1)#14x14x8 to 28x28x8
    reconst_img = self.dec_conv3(reconst_img)#28x28x8 to 28x28x1
```

```
    return reconst_img, enc_input


model_conAE1 = ConvolutionAE_unpooling()
criterion_conAE1 = nn.MSELoss()
optimizer_conAE1 = torch.optim.Adam(model_conAE1.parameters(),lr=0.001)

training_loss_conAE1 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    outputs, _ = model_conAE1(images)
    loss = criterion_conAE1(outputs,images)
    training_loss_conAE1.append(loss.item())

    optimizer_conAE1.zero_grad()
    loss.backward()
    optimizer_conAE1.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0146
Epoch [2/10]: completed, Loss: 0.0103
Epoch [3/10]: completed, Loss: 0.0091
Epoch [4/10]: completed, Loss: 0.0079
Epoch [5/10]: completed, Loss: 0.0068
Epoch [6/10]: completed, Loss: 0.0069
Epoch [7/10]: completed, Loss: 0.0059
Epoch [8/10]: completed, Loss: 0.0061
Epoch [9/10]: completed, Loss: 0.0057
Epoch [10/10]: completed, Loss: 0.0055
```

```
class ConvolutionAE_deconv(nn.Module):
  def __init__(self):
    super(ConvolutionAE_deconv, self).__init__()

    # encoder
    self.enc_conv1 = nn.Sequential(
        nn.Conv2d(1,8, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2))
    )
    self.enc_conv2 = nn.Sequential(
        nn.Conv2d(8,16, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2))
    )
    self.enc_conv3 = nn.Sequential(
        nn.Conv2d(16,16, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU()
```

```
            nn.ReLU(),
            nn.MaxPool2d(kernel_size = (2,2))
        )


        # decoder
        self.dec_conv1 = nn.Sequential(
            nn.ConvTranspose2d(16,16, kernel_size = 3, stride = 2),
            nn.ReLU()
        )
        self.dec_conv2 = nn.Sequential(
            nn.ConvTranspose2d(16,8, kernel_size = 4, stride = 2, padding = 1),
            nn.ReLU()
        )
        self.dec_conv3 = nn.Sequential(
            nn.ConvTranspose2d(8,1, kernel_size = 4, stride = 2, padding = 1),
            nn.ReLU()
        )

    def forward(self,x):
        enc_input  = self.enc_conv1(x.float())
        enc_input  = self.enc_conv2(enc_input)
        enc_input  = self.enc_conv3(enc_input)

        reconst_img = self.dec_conv1(enc_input)
        reconst_img = self.dec_conv2(reconst_img)
        reconst_img = self.dec_conv3(reconst_img)

        return reconst_img, enc_input



model_conAE2 = ConvolutionAE_deconv()
criterion_conAE2 = nn.MSELoss()
optimizer_conAE2 = torch.optim.Adam(model_conAE2.parameters(),lr=0.001)

training_loss_conAE2=[]

for epoch in range(epochs):
  for images,labels in train_loader:
    outputs, _ = model_conAE2(images)
    loss = criterion_conAE2(outputs, images)
    training_loss_conAE2.append(loss.item())

    optimizer_conAE2.zero_grad()
    loss.backward()
    optimizer_conAE2.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0245
Epoch [2/10]: completed, Loss: 0.0183
```

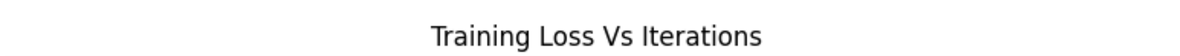
```
 Epoch [2/10]: completed, Loss: 0.0183
 Epoch [3/10]: completed, Loss: 0.0164
 Epoch [4/10]: completed, Loss: 0.0151
 Epoch [5/10]: completed, Loss: 0.0134
 Epoch [6/10]: completed, Loss: 0.0124
 Epoch [7/10]: completed, Loss: 0.0129
 Epoch [8/10]: completed, Loss: 0.0119
 Epoch [9/10]: completed, Loss: 0.0120
 Epoch [10/10]: completed, Loss: 0.0117
```

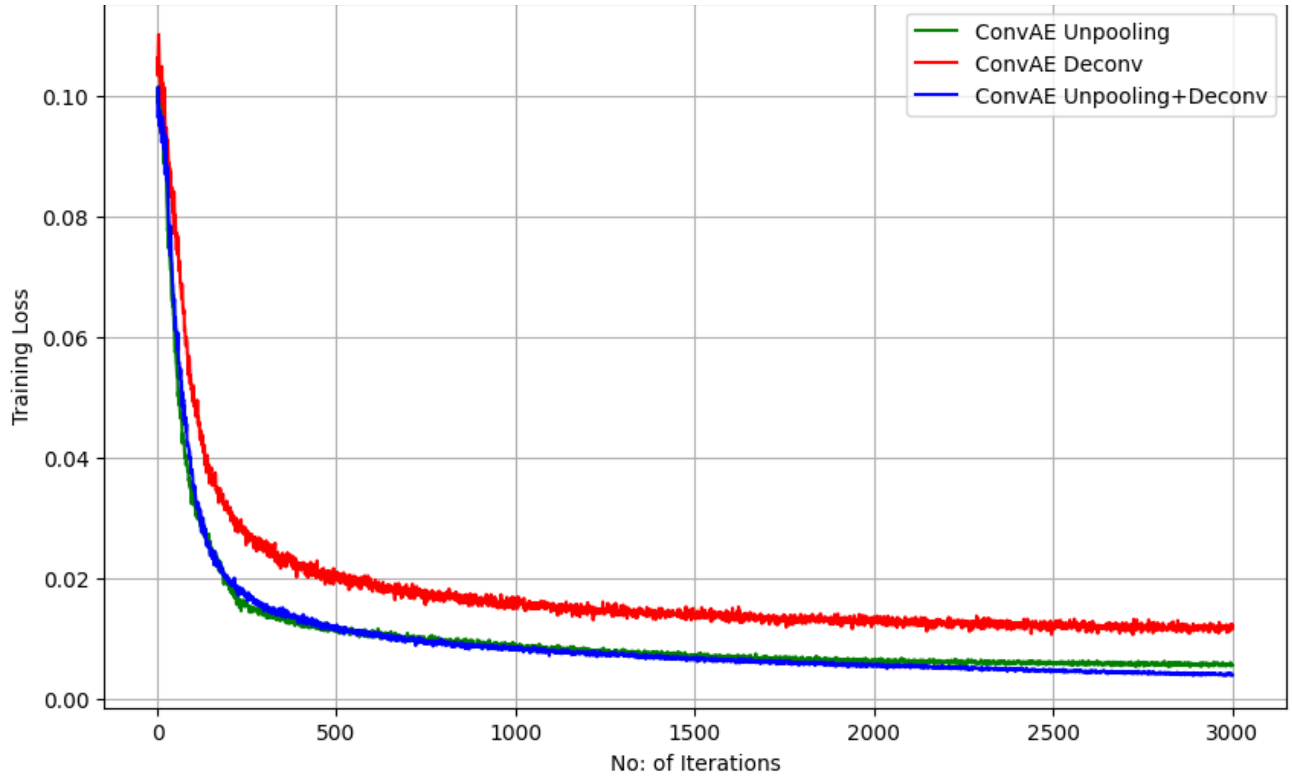
```
class ConvolutionAE_unpool_deconv(nn.Module):
  def __init__(self):
    super(ConvolutionAE_unpool_deconv,self).__init__()

    # encoder
    self.enc_conv1 = nn.Sequential(
        nn.Conv2d(1,8, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2),return_indices = True)
    )
    self.enc_conv2 = nn.Sequential(
        nn.Conv2d(8,16, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size = (2,2),return_indices = True)
    )
    self.enc_conv3 = nn.Sequential(
        nn.Conv2d(16,16, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU(),nn.MaxPool2d(kernel_size = (2,2),return_indices = True)
    )



    # decoder
    self.dec_conv1 = nn.Sequential(
        nn.ConvTranspose2d(16,16, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU()
    )
    self.dec_conv2 = nn.Sequential(
        nn.ConvTranspose2d(16,8, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU()
    )
    self.dec_conv3 = nn.Sequential(
        nn.ConvTranspose2d(8,1, kernel_size = 3, stride = 1, padding = 1),
        nn.ReLU()
    )
    # unpooling
    self.unpool = nn.MaxUnpool2d(kernel_size = (2,2))

  def forward(self,x):
    enc_input,indices1  = self.enc_conv1(x.float())
    enc_input,indices2  = self.enc_conv2(enc_input)
    enc_input,indices3  = self.enc_conv3(enc_input)
```

```
    reconst_img  = self.unpool(enc_input,indices3,output_size=torch.Size([batch_s
    reconst_img  = self.dec_conv1(reconst_img )
    reconst_img  = self.unpool(reconst_img ,indices2)
    reconst_img  = self.dec_conv2(reconst_img )
    reconst_img  = self.unpool(reconst_img ,indices1)
    reconst_img  = self.dec_conv3(reconst_img )

    return reconst_img, enc_input


model_conAE3 = ConvolutionAE_unpool_deconv()
criterion_conAE3 = nn.MSELoss()
optimizer_conAE3 = torch.optim.Adam(model_conAE3.parameters(),lr=0.001)

training_loss_conAE3 = []

for epoch in range(epochs):
  for images,labels in train_loader:
    outputs, _ = model_conAE3(images)
    loss = criterion_conAE3(outputs, images)
    training_loss_conAE3.append(loss.item())

    optimizer_conAE3.zero_grad()
    loss.backward()
    optimizer_conAE3.step()

  print("Epoch [{}/{}]: completed, Loss: {:.4f}".format(epoch+1, epochs, loss.ite
```

```
Epoch [1/10]: completed, Loss: 0.0150
Epoch [2/10]: completed, Loss: 0.0105
Epoch [3/10]: completed, Loss: 0.0085
Epoch [4/10]: completed, Loss: 0.0078
Epoch [5/10]: completed, Loss: 0.0065
Epoch [6/10]: completed, Loss: 0.0060
Epoch [7/10]: completed, Loss: 0.0053
Epoch [8/10]: completed, Loss: 0.0047
Epoch [9/10]: completed, Loss: 0.0041
Epoch [10/10]: completed, Loss: 0.0039
```

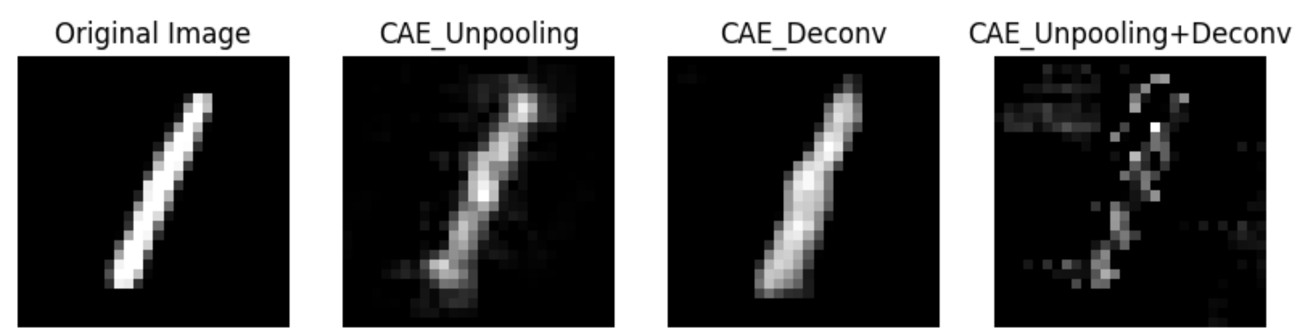
```
plt.plot(range(1,len(training_loss_conAE1)+1),training_loss_conAE1 ,label="ConvAE
plt.plot(range(1,len(training_loss_conAE1)+1),training_loss_conAE2 ,label="ConvAE
plt.plot(range(1,len(training_loss_conAE1 )+1),training_loss_conAE3 ,label="ConvA
plt.legend()
plt.grid()
plt.title("Training Loss Vs Iterations")
plt.xlabel("No: of Iterations")
plt.ylabel("Training Loss")
plt.show()
```



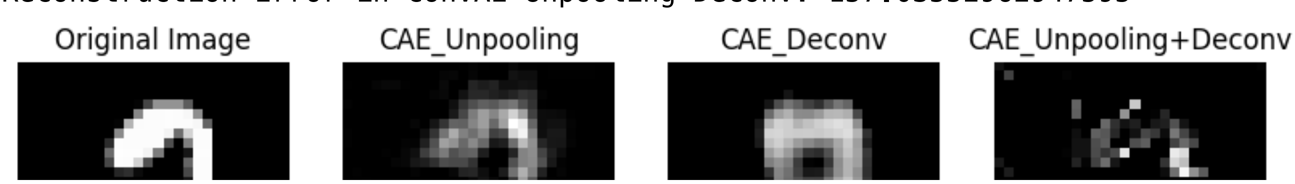


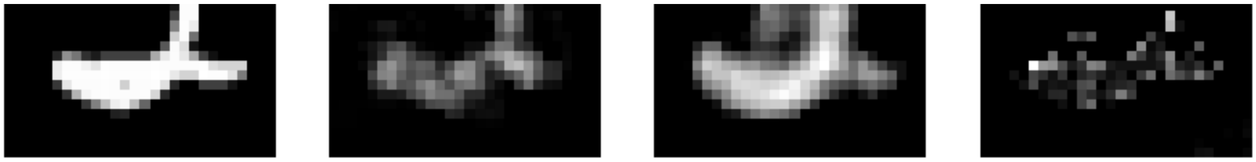

```
model_conAE1.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,1,28,28)
    outputs_conAE1, _ = model_conAE1(images.float())


model_conAE2.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,1,28,28)
    outputs_conAE2, _ = model_conAE2(images.float())


activation_conAE3 = []
model_conAE3.eval()
with torch.no_grad():
  for images in test_sample_loader:
    images = images.reshape(10,1,28,28)
    outputs_conAE3, _ = model_conAE3(images.float())
```

```
plt.rcParams["figure.figsize"] = (10,6)

for i in range(5):
  fig, (ax1, ax2, ax3, ax4) = plt.subplots(1,4)

  ax1.imshow(images[i].detach().numpy().reshape(28,28),cmap='gray')
  ax1.set_title('Original Image')
  ax1.axis("off")

  ax2.imshow(outputs_conAE1[i].detach().numpy().reshape(28,28),cmap='gray')
  ax2.set_title('CAE_Unpooling')
  ax2.axis("off")

  ax3.imshow(outputs_conAE2[i].detach().numpy().reshape(28,28),cmap='gray')
  ax3.set_title('CAE_Deconv')
  ax3.axis("off")

  ax4.imshow(outputs_conAE3[i].detach().numpy().reshape(28,28),cmap='gray')
  ax4.set_title('CAE_Unpooling+Deconv')
  ax4.axis("off")

  plt.show()

  print("Reconstruction Error in ConvAE Unpooling:",np.sum(np.dot(((images[i].det
                                                              ((images[i].detach().nu

  print("Reconstruction Error in ConvAE Deconv:",np.sum(np.dot(((images[i].detach
                                                              ((images[i].detach().n

  print("Reconstruction Error in ConvAE Unpooling+Deconv:",np.sum(np.dot(((images
                                                              ((images[i].detac
```

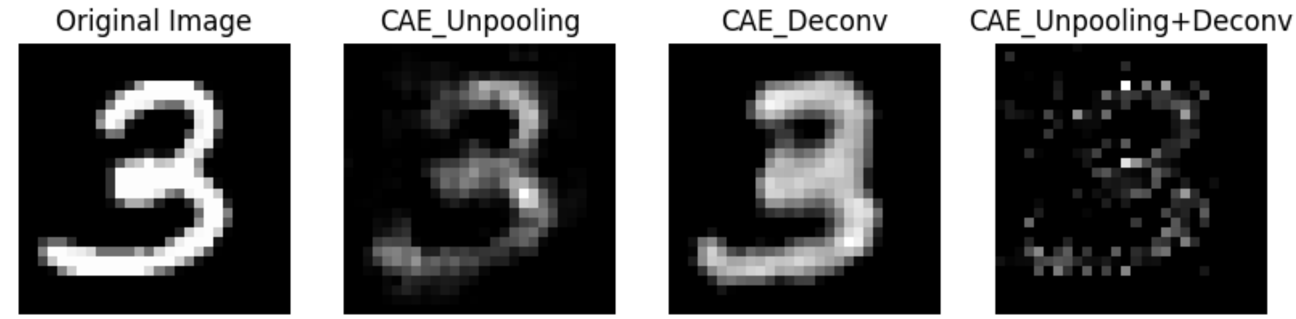



```
Reconstruction Error in ConvAE Unpooling: 41.09798739985503
Reconstruction Error in ConvAE Deconv: 11.099846611587598
Reconstruction Error in ConvAE Unpooling+Deconv: 157.03552962947595
```

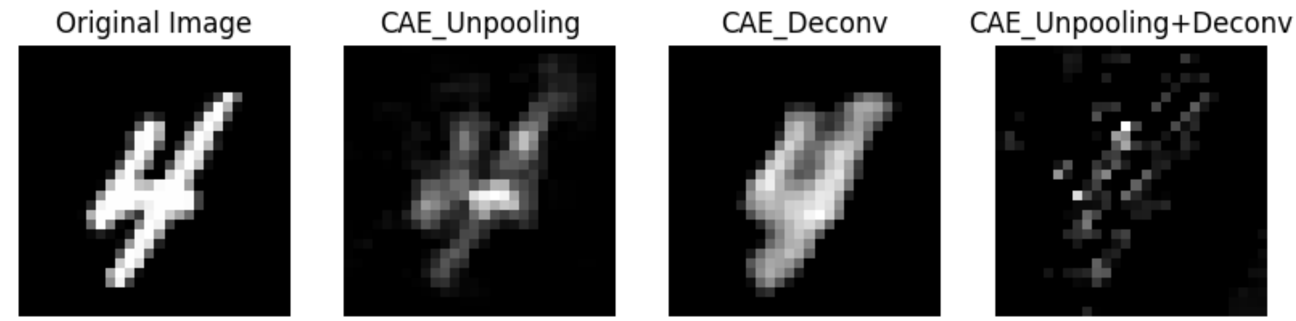




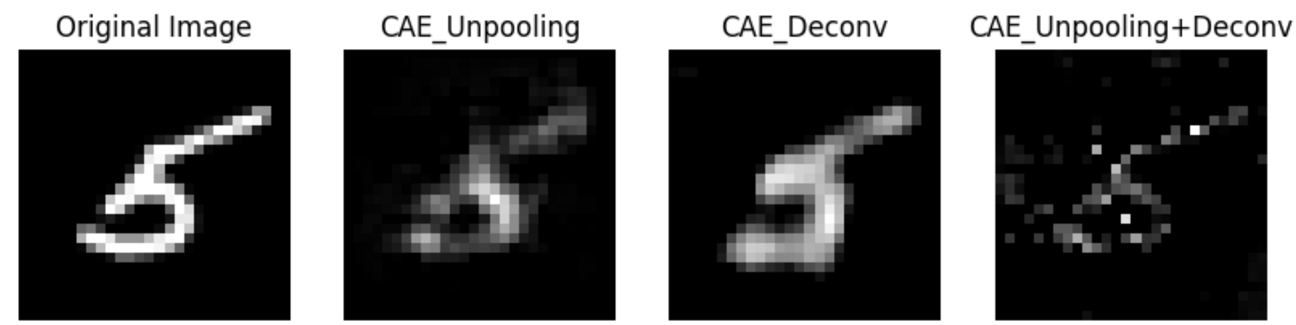
```
Reconstruction Error in ConvAE Unpooling: 249.00256747616373
Reconstruction Error in ConvAE Deconv: 49.661515593520456
Reconstruction Error in ConvAE Unpooling+Deconv: 906.3791487199521
```




```
Reconstruction Error in ConvAE Unpooling: 236.94265814368208
Reconstruction Error in ConvAE Deconv: 34.88963381365123
Reconstruction Error in ConvAE Unpooling+Deconv: 708.2955451104583
```




```
Reconstruction Error in ConvAE Unpooling: 191.92509100020084
Reconstruction Error in ConvAE Deconv: 7.73961747405001
Reconstruction Error in ConvAE Unpooling+Deconv: 538.5825585477866
```

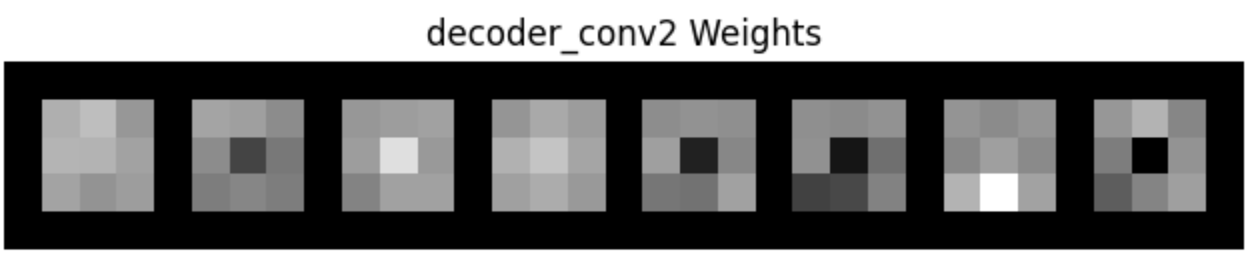


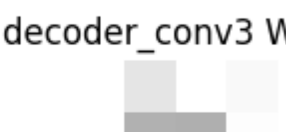

```
Reconstruction Error in ConvAE Unpooling: 79.80316303619999
Reconstruction Error in ConvAE Deconv: -4.719208042428017
Reconstruction Error in ConvAE Unpooling+Deconv: 256.2937023179323
```

```
model_conAE1.enc_conv1[0].weight.detach().numpy().squeeze().shape
```

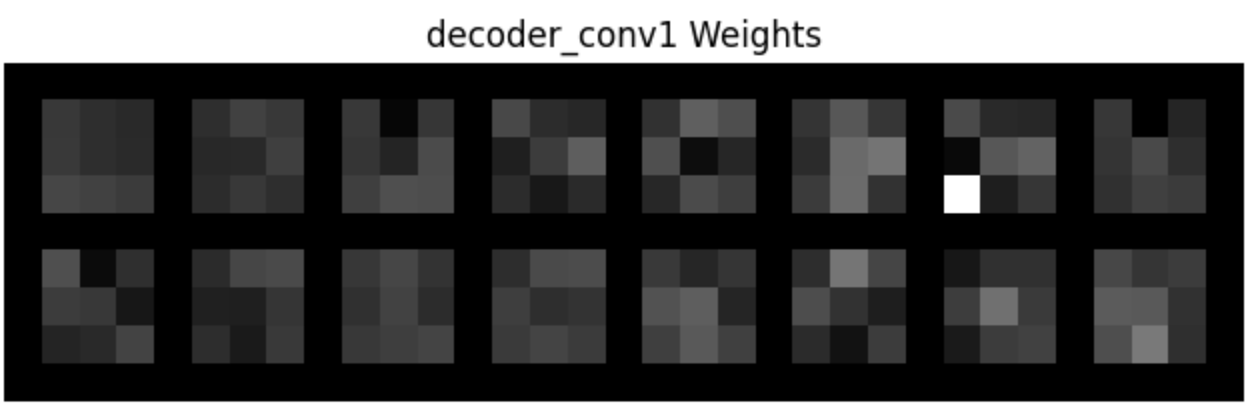
```
(8, 3, 3)
```

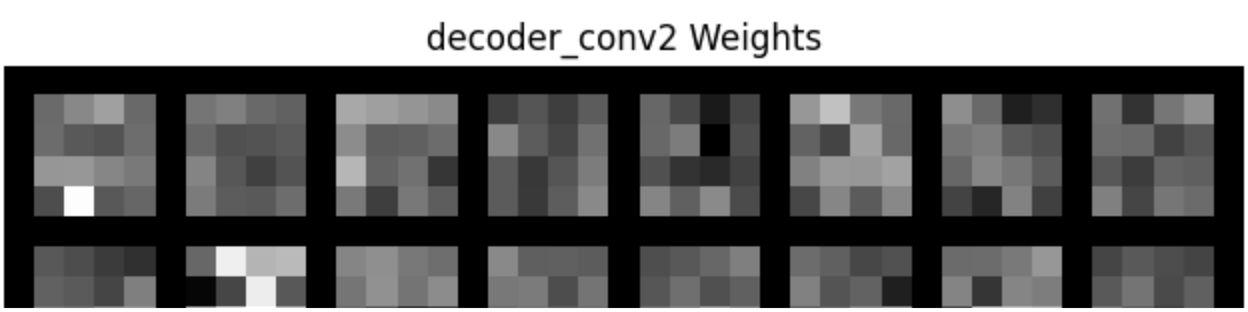
```
# Function for visualisation of weights
def visualize_tensor(tensor, ch=0, allkernels=False, nrow=8, padding=1):
  n,c,w,h = tensor.shape

  if allkernels: tensor = tensor.view(n*c, -1, w, h)
  elif c != 3: tensor = tensor[:,ch,:,:].unsqueeze(dim=1)

  rows = np.min((tensor.shape[0] // nrow + 1, 64))
  grid = utils.make_grid(tensor, nrow=nrow, normalize=True, padding=padding)
  plt.figure( figsize=(nrow,rows) )
  plt.imshow(grid.numpy().transpose((1, 2, 0)))


# Visualising decider weights for convolution Autoencoder with unpooling
filter = model_conAE1.dec_conv2[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv2 Weights')
plt.show()

filter = model_conAE1.dec_conv3[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv3 Weights')
plt.show()
```

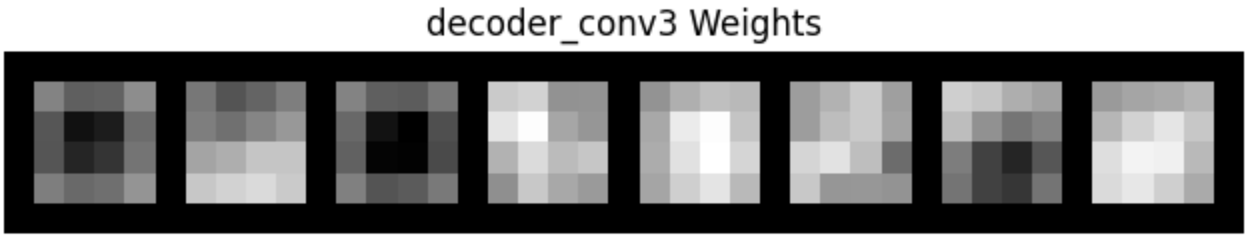




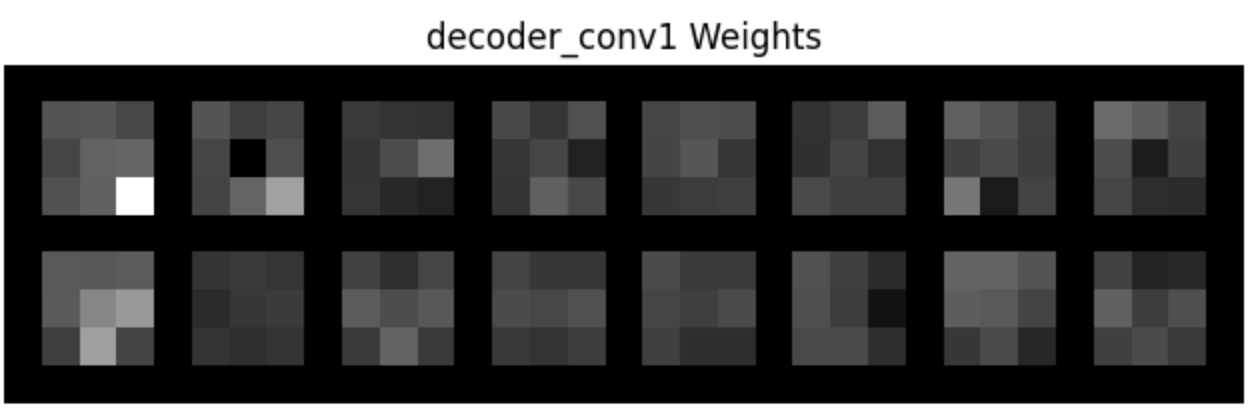
```
# Visulalising decoder weights for convolution Autoencoders with deconvolution

filter = model_conAE2.dec_conv1[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv1 Weights')
plt.show()

filter = model_conAE2.dec_conv2[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv2 Weights')
plt.show()

filter = model_conAE2.dec_conv3[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv3 Weights')
plt.show()
```



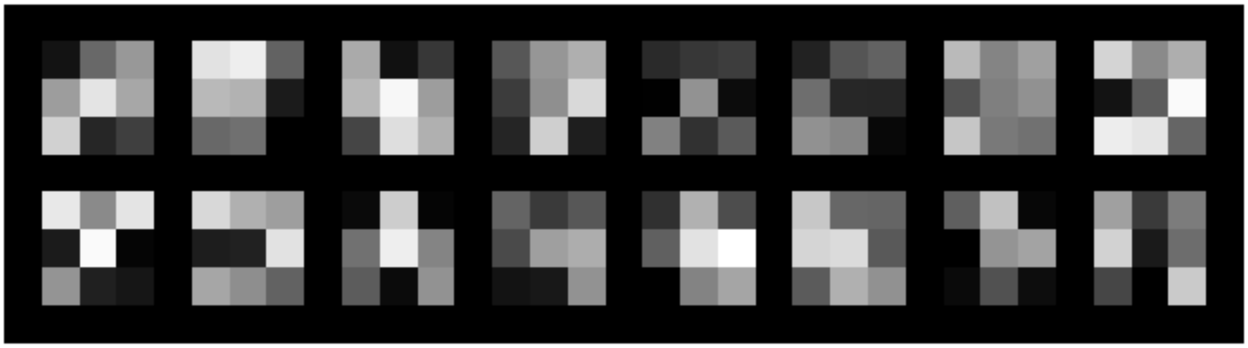





```
# Visulalising decoder weights for convolution Autoencoders with unpooling + decon

filter = model_conAE3.dec_conv1[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv1 Weights')
plt.show()

filter = model_conAE3.dec_conv2[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv2 Weights')
plt.show()

filter = model_conAE3.dec_conv3[0].weight.data.clone()
visualize_tensor(filter, ch=0, allkernels=False)

plt.axis('off')
plt.ioff()
plt.title('decoder_conv3 Weights')
plt.show()
```



decoder_conv2 Weights



decoder_conv3 Weights