1

1.a Geometric interpretation of gradient descent

Let C be the cost function we are aiming to minimize. Then the update rule for gradient descent takes the form

$$v \to v' = v - \eta \nabla C = v - \varepsilon \frac{\nabla C}{||\nabla C||}.$$

For the one-dimensional case this simplifies to

$$v \to v' = v - \varepsilon \cdot \mathrm{sgn}(C'(v)).$$

Geomtrically speaking, this implies we are always going ε to the left or the right. The direction depends on whether the cost function is increasing or decreasing at our current location.

1.b b

The use of mini-batches is justified by the following heuristic:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{1}$$

where n is the full size of the training data set and m is the size of the mini-batch. Let's compare $m = 1$ to $m = 20$. The validity of approximation (3) depends on m, with a higher m implying a better approximation (there are established concentration-inequalities to support this claim). For $m = 1$ the estimate of the gradient will frequently be worse than for $m = 20$. Consequently, each step with $m = 1$ will generally not decrease the cost as much as $m = 20$. In worst cases, we might have a terrible approximation and might increase the cost.

On the other hand, evaluating $m = 1$ is faster than $m = 20$, so we will be able to do more steps with the same time and resources.

2 Problem

2.a

Following the notation in the Nelson Book consider (BP2):

$$\delta^l = \left(\left(w^{l+1} \right)^T \delta^{l+1} \right) \odot \sigma' \left(z^l \right)$$

In component form, this is

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma' \left(z_j^l \right).$$

If we replace the activation function at a single neuron by f, this becomes for a fixed l, j

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f'\left(z_j^l\right).$$

Accordingly, $\delta^1, \ldots, \delta^{l-1}$, which depend on δ_j^l, will change. By (BP3) and (BP4), the step estimate for b_j^l, the biases and weights in all the layers before the change will have to be adjusted.

2.b

Let f the softmax activation function i.e.

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}},$$

Let y be the one-hot encoding of the correct label and C be the log-likelihood cost

$$C = -\sum_{i=1}^{c} y_i \cdot \log\left(a_i^L\right)$$

Then we have for the cost in the outputlayer: Consider first

$$\begin{aligned}
\frac{\partial - \log(a_j^L)}{\partial z_i^L} &= \frac{\partial}{\partial z_i^L} - \log\left(\frac{e^{z_j^L}}{\sum_k e^{z_k^L}}\right) \\
&= \frac{\partial}{\partial z_i^L}\left(-z_j^L + \log\left(\sum_k e^{z_k^L}\right)\right) \\
&= -\mathbb{1}(i=j) + \frac{\partial}{\partial z_i^L}\left(\log\left(\sum_k e^{z_k^L}\right)\right) \\
&= -\mathbb{1}(i=j) + \frac{1}{\sum_k e^{z_k^L}}\left(\frac{\partial}{\partial z_i^L}\left(\sum_k e^{z_k^L}\right)\right) \\
&= -\mathbb{1}(i=j) + \frac{e^{z_i^L}}{\sum_k e^{z_k^L}} \\
&= -\mathbb{1}(i=j) + a_i^L
\end{aligned}$$

Combining, we have

$$
\begin{aligned}
\delta_i^L &= \frac{\partial C}{\partial z_i^L} \\
&= \frac{\partial}{\partial z_i^L} - \sum_{j=1}^{c} y_j \cdot \log\left(a_j^L\right) \\
&= -\sum_{j=1}^{c} y_j \cdot \frac{\partial}{\partial z_i^L} \log\left(a_j^L\right) \\
&= \sum_{j=1}^{c} y_j \cdot \left(-\mathbb{1}(i=j) + a_i^L\right) \\
&= -y_i + a_i^L \sum_{j=1}^{c} y_j \\
&= a_i^L - y_i .
\end{aligned}
$$

2.c

If we replace the sigmoid layer with a linear identity layer, the neural network simplifies to a single big matrix multiplication from the input layer to the output layer. We will always have $\sigma'\left(z^L\right) = 1$. Accordingly, we have:

$$
\begin{aligned}
\delta^L &= \nabla_a C \\
\delta^l &= \left(w^{l+1}\right)^T \delta^{l+1} = \prod_{i=l+1}^{L-1} \left(w^i\right)^T \nabla_a C \\
\frac{\partial C}{\partial b_j^l} &= \delta_j^l = \left(w_{j\cdot}^{l+1}\right)^T \prod_{i=l+2}^{L-1} \left(w^i\right)^T \nabla_a C \\
\frac{\partial C}{\partial w_{jk}^l} &= a_k^{l-1}\delta_j^l = a_k^{l-1} \left(w_{j\cdot}^{l+1}\right)^T \prod_{i=l+2}^{L-1} \left(w^i\right)^T \nabla_a C
\end{aligned} \tag{2}
$$

The upgrade step of the gradient descent will be governed by the last two lines.

3 Problem

3.a Problem

Consider

$$- \left[y \ln a + (1-y)\ln(1-a)\right], \tag{3}$$

and

$$- \left[a \ln y + (1-a)\ln(1-y)\right]. \tag{4}$$

We know that $a = \sigma(z) \in (0,1)$, because σ only takes values in $(0,1)$. On the other hand, $y \in [0,1]$. We will use $a \in (0,1)$ in the following. For $y = 0$ we have in equation (3):

$$-[y \ln a + (1-y)\ln(1-a)] = \ln(1-a).$$

This makes sense. For $y = 1$ we have in equation (3):

$$-[y \ln a + (1-y)\ln(1-a)] = \ln a.$$

This makes sense. For $y = 0$ we have in equation (4):

$$-[a \ln y + (1-a)\ln(1-y)] = -[a \cdot (-\infty) + (1-a) \cdot 0] = \infty.$$

This makes no sense. For $y = 1$ we have in equation (4):

$$-[a \ln y + (1-a)\ln(1-y)] = -[a \cdot (0) + (1-a) \cdot (-\infty)] = \infty.$$

This makes no sense. Those problems do not arise in equation (3), because $a \in (0,1)$.

3.b Problem

Let σ be the sigmoid function and C the cross-entropy cost. By equations (3.7) and (3.8) in the Nielson book we have:

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y),$$
$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_x (\sigma(z) - y).$$

Note, that the derivation for those equations in the Nielson book is valid for general $y, a \in [0,1]$. Consequently, if we have $\sigma(z) = y$ for all training inputs, we have $\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial b} = 0$ for all j. This shows that $\sigma(z) = y$ is a critical point of the cost function. It remains to verify that it is a minimum. Note, that

$$\begin{aligned}\frac{\partial C}{\partial w_j \partial w_k} &= \frac{\partial C}{\partial w_j \partial z}\frac{\partial z}{\partial w_k} \\ &= \left(\frac{1}{n}\sum_x x_j \sigma'(z)\right)(a_k) > 0.\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial w_j \partial b} &= \frac{\partial C}{\partial w_j \partial z}\frac{\partial z}{\partial b} \\ &= \left(\frac{1}{n}\sum_x x_j \sigma'(z)\right) > 0.\end{aligned}$$

$$
\begin{aligned}
\frac{\partial C}{\partial b \partial b} &= \frac{\partial C}{\partial b \partial z}\frac{\partial z}{\partial b} \\
&= \left(\frac{1}{n}\sum_x \sigma'(z)\right) > 0.
\end{aligned}
$$

This concludes the proof. Furthermore, plugging in $\sigma(z) = y$ yields

this does not seem right

$$C = -\frac{1}{n}\sum_x [y \ln y + (1-y)\ln(1-y)].$$

3.c Problem

Let

$$
\begin{aligned}
W^2 &= \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix} = \begin{bmatrix} w^2_{1,1} & w^2_{1,2} \\ w^2_{2,1} & w^2_{2,2} \end{bmatrix}, \\
b^2 &= \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \begin{bmatrix} b^1_1 \\ b^1_2 \end{bmatrix}, \\
W^3 &= \begin{bmatrix} 0.4 & 0.5 \\ -0.45 & 0.55 \end{bmatrix} = \begin{bmatrix} w^3_{1,1} & w^3_{1,2} \\ w^3_{2,1} & w^3_{2,2} \end{bmatrix}, \\
b^3 &= \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \begin{bmatrix} b^2_1 \\ b^2_2 \end{bmatrix}.
\end{aligned}
$$

Then the output of the neural network for an input x is

$$\sigma(W^2(\sigma(W^1x + b^1)) + b^2).$$

We can calculate the $\delta^1, \delta^2, \delta^3$ with the central equations of the back-propagation algorithm:

$$
\begin{aligned}
\delta^L &= \nabla_a C \odot \sigma'\left(z^L\right) \\
\delta^l &= \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right)
\end{aligned}
$$

The cross-entropy function C is defined by

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)].$$

This yields

$$\nabla_a C = -\frac{1}{n}\sum_x \left[\frac{y}{a} - \frac{1-y}{1-a}\right].$$

$$
\begin{aligned}
\frac{\partial C}{\partial z} &= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial z} \\
&= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\sigma(z)(1-\sigma(z)) \\
&= -\frac{1}{n}\sum_x y(1-\sigma(z)) - \sigma(z)(1-y)
\end{aligned}
$$

```
In [ ]: import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn
from IPython.display import Image
from prob5_AE import AE
from prob6_CAE import CAE
from prob6_CAE_skip import CAE_skip, train_log, test, main_wrapper
from torchvision import datasets
```

Problem 3

Let

$$x = \begin{bmatrix} 0.05 \\ 0.1 \end{bmatrix},$$

$$y = \begin{bmatrix} 0.01 \\ 0.99 \end{bmatrix},$$

$$W^2 = \begin{bmatrix} 0.15 & 0.25 \\ 0.2 & 0.3 \end{bmatrix} = \begin{bmatrix} w^2_{1,1} & w^2_{1,2} \\ w^2_{2,1} & w^2_{2,2} \end{bmatrix},$$

$$b^2 = \begin{bmatrix} 0.35 \\ 0.35 \end{bmatrix} = \begin{bmatrix} b^1_1 \\ b^1_2 \end{bmatrix},$$

$$W^3 = \begin{bmatrix} 0.4 & 0.5 \\ -0.45 & 0.55 \end{bmatrix} = \begin{bmatrix} w^3_{1,1} & w^3_{1,2} \\ w^3_{2,1} & w^3_{2,2} \end{bmatrix},$$

$$b^3 = \begin{bmatrix} 0.6 \\ 0.6 \end{bmatrix} = \begin{bmatrix} b^2_1 \\ b^2_2 \end{bmatrix}.$$

```
In [ ]: # x matrix
x = np.array([[0.05], [0.1]])

# y matrix
y = np.array([[0.01], [0.99]])

# W2 matrix
W2 = np.array([[0.15, 0.25], [0.2, 0.3]])

# b2 matrix
b2 = np.array([[0.35], [0.35]])

# W3 matrix
W3 = np.array([[0.4, 0.5], [-0.45, 0.55]])

# b3 matrix
b3 = np.array([[0.6], [0.6]])
```

Then the output of the neural network for an input x is

$$\sigma(W^2(\sigma(W^1x + b^1)) + b^2).$$

The intermediate values are

In [ ]:
```
z2 = np.dot(W2, x) + b2
a2 = 1 / (1 + np.exp(-z2))
z3 = np.dot(W3, a2) + b3
a3 = 1 / (1 + np.exp(-z3))
```

We can calculate the $\delta^1, \delta^2, \delta^3$ with the central equations of the back-propagation algorithm:

$$\delta^L = \nabla_a C \odot \sigma'\left(z^L\right)$$
$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'\left(z^l\right)$$

For the cross entropy-loss in combination with the sigmoid activation function, we have

$$\begin{aligned}
\frac{\partial C}{\partial z} &= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\frac{\partial \sigma}{\partial z} \\
&= -\frac{1}{n}\sum_x \left(\frac{y}{\sigma(z)} - \frac{1-y}{1-\sigma(z)}\right)\sigma(z)(1-\sigma(z)) \\
&= -\frac{1}{n}\sum_x y(1-\sigma(z)) - \sigma(z)(1-y)
\end{aligned}$$

In [ ]:
```
delta3 = - (y*(1-a3)-a3*(1-y))
delta3
```

Out[ ]:
```
array([[ 0.74693192],
       [-0.33064048]])
```

Now for the intermediate values we have

In [ ]:
```
delta2 = np.dot(W3.T, delta3) * a2 * (1 - a2)
delta2
```

In [ ]:
```
delta1 = np.dot(W2.T, delta2) * x * (1 - x)
delta1
```

Out[ ]:
```
array([[0.00120696],
       [0.00367308]])
```

Problem 4

sample input data

In [ ]:
```
input = torch.randn(1,50,50)
```

In [ ]:
```
m = nn.Conv2d(1, 1, 4, stride=1, padding = 1)
output = m(input)
#dimensions of output
print(output.shape)
```

```
torch.Size([1, 49, 49])
```

In [ ]:
```
m = nn.Conv2d(1, 1, 8, stride=5, padding = 0)
output = m(input)
```

```
print(output.shape)
```

torch.Size([1, 9, 9])

In [ ]:
```
#max-pooling
m = nn.MaxPool2d(kernel_size= 10, stride=2, padding = 2)
output = m(input)
print(output.shape)
```

torch.Size([1, 23, 23])

In [ ]:
```
#max-pooling
m = nn.MaxPool2d(kernel_size= 2, stride=1, padding = 0)
output = m(input)
print(output.shape)
```

torch.Size([1, 49, 49])

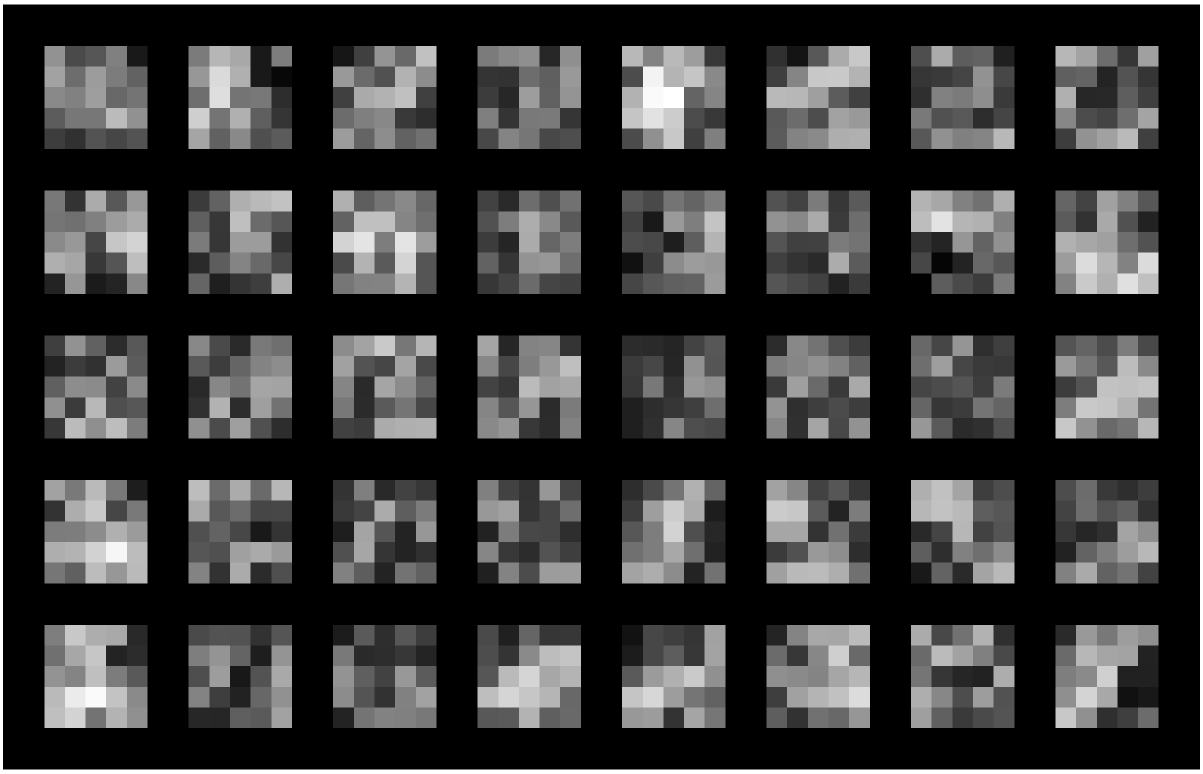
Problem 4.2

In [ ]:
```
mnist_testset = datasets.MNIST(root='./data', train=False, download=True, tr

ntest = 2000
test_data = (mnist_testset.data.to(dtype=torch.float32)[:ntest]/255).view(-1
test_labels = mnist_testset.targets.to(dtype=torch.long)[:ntest]
```

after runnign the code we have

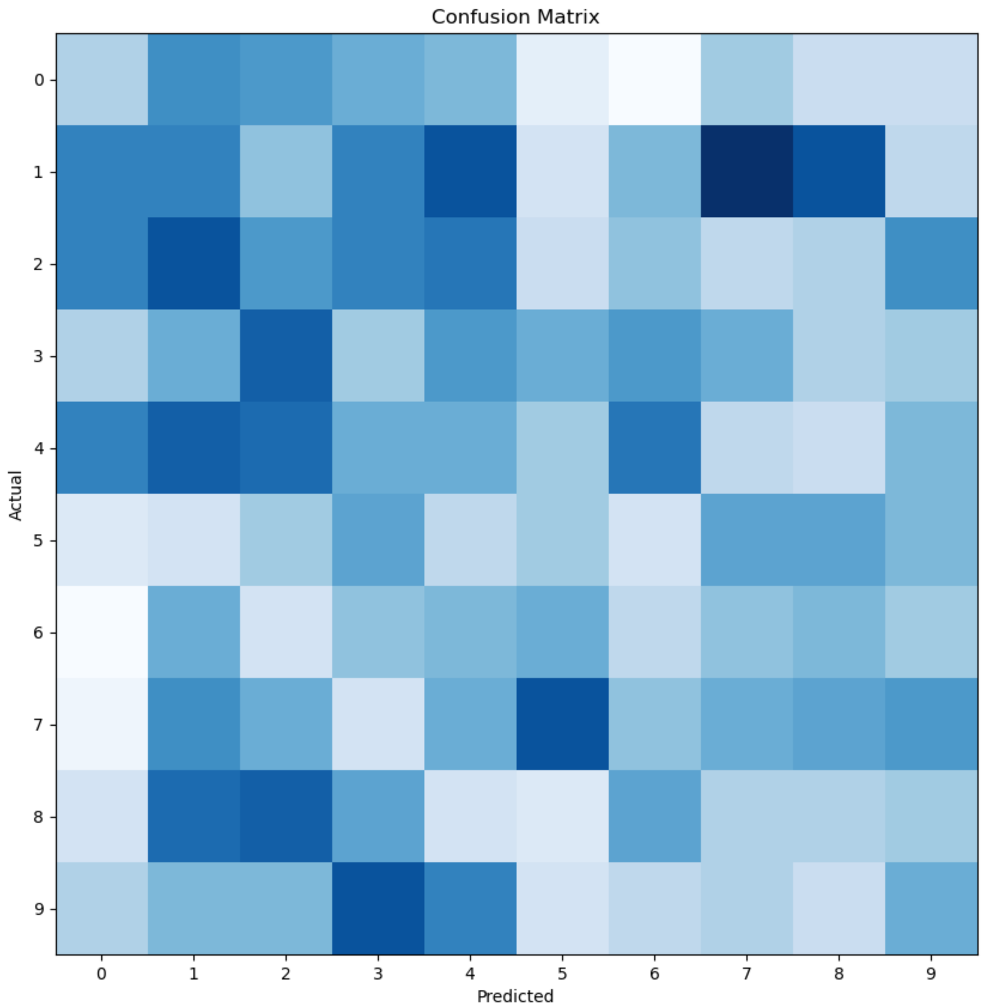
In [ ]:
```
#show prob4_CNN.png
Image(filename='prob4_CNN.png', width=800)
```

Out[ ]:



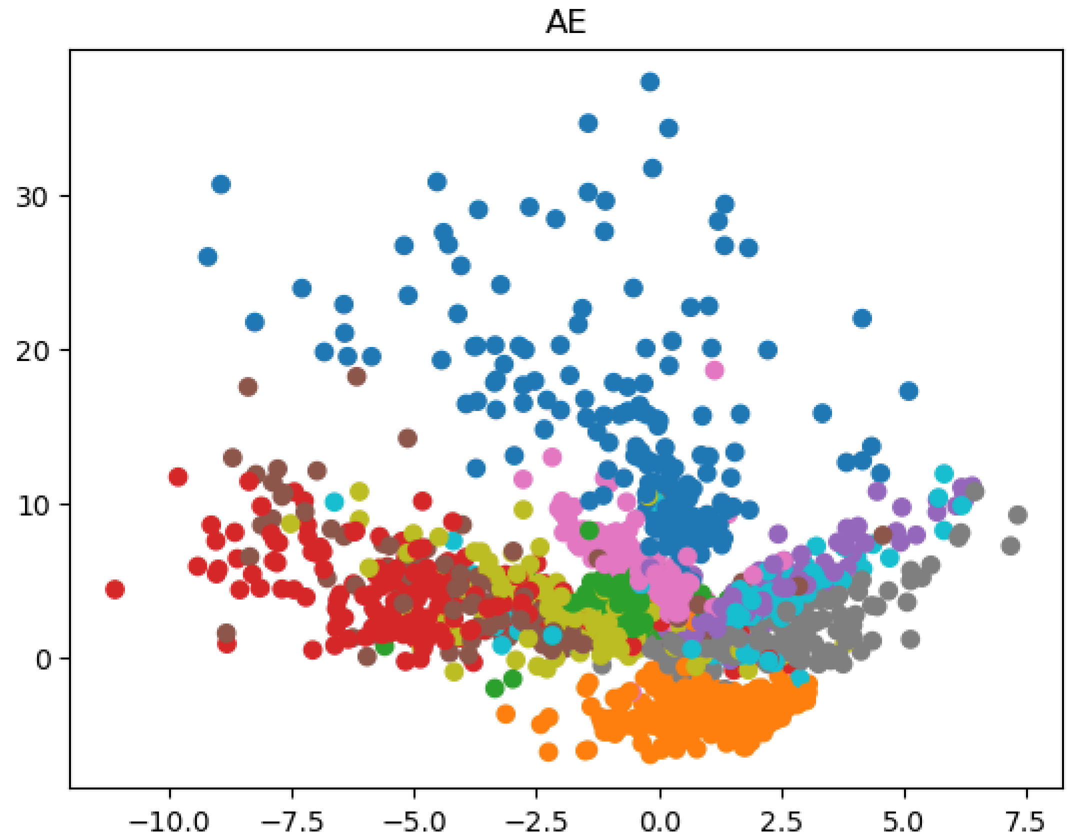
In [ ]:
```
Image(filename='prob4_confusion_matrix.png', width=800)
```

Out[ ]:



1 and 7 are confused, which makes sense. 9 and 3 are confused, which makes sense.

In [ ]:
```
test_data_flat = test_data.reshape(-1, 28*28)

model = AE()
model.load_state_dict(torch.load('mnist_ae.pt'))

embedded_ae = model.encoder(test_data_flat)
embedded_ae = embedded_ae.detach().numpy()

plt.figure()
plt.scatter(embedded_ae[:,0], embedded_ae[:,1], c=test_labels, cmap='tab10')
plt.title('AE')
plt.show()
```



We see that 5 and 3 are harder to seperate which agrees with our domain knowledge. 7 is similar to 9.

For Problem 5.4,

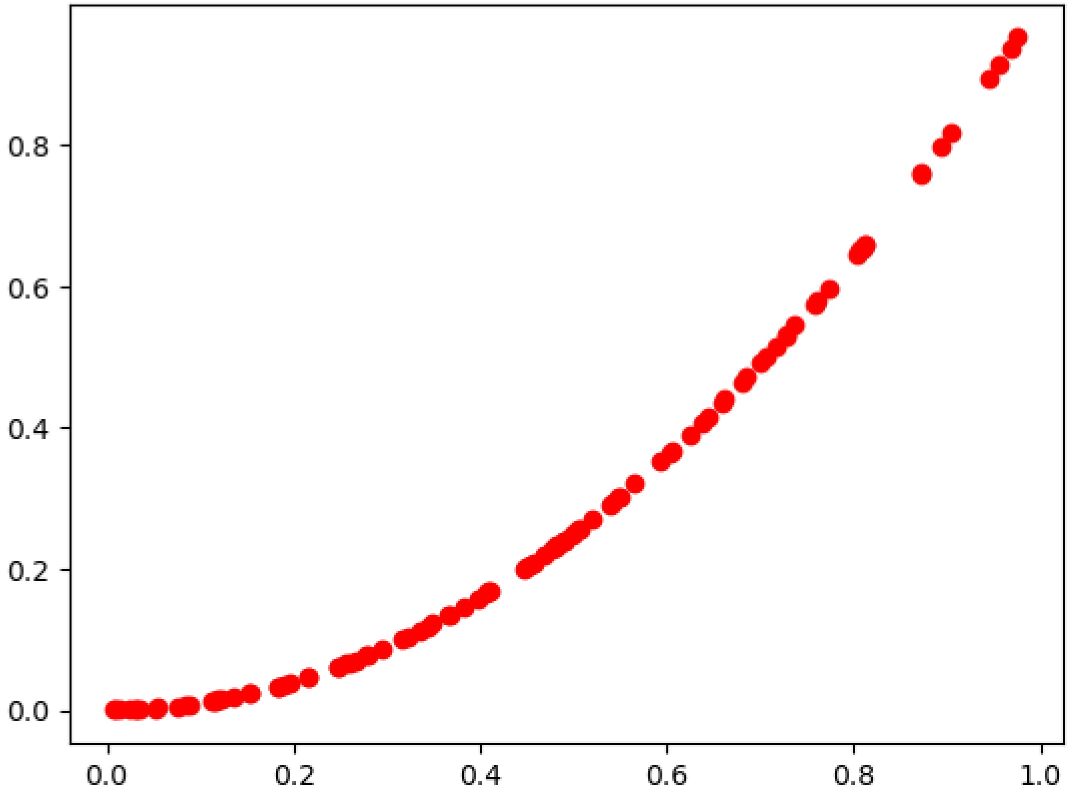
Problem 5

Problem 5.1 1) In a classical auto encoder, bow tie architercutre the middle layer is called the bottlenec. The dimension of the bottleneck is chosen to be drastically different from the input and output layer. This way it is impossible for the NN to simply copy the input to the output. 2) By introducing regularisation/corruption, we introduce a preference for some solution compared to others. Typically this prohibits the NN from simply copying the input to the output. 3) For an input sample X, we can perturb it to \tidle$X$. We train the NN to compare the output of X to \tidle$X$. This way the NN learns to ignore the perturbations and focus on the important features of the input.

Problem 5.2 We can add gaussian noise to a sample to corrupt it.

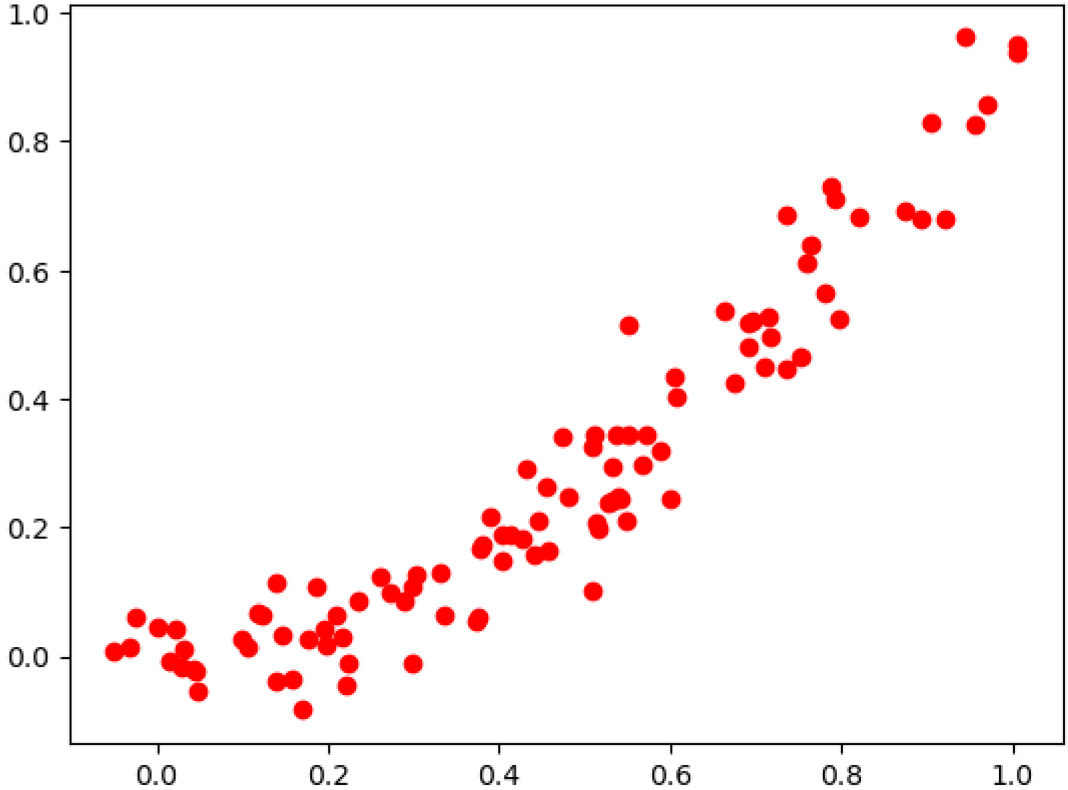
In [ ]:
```
#generate 1000 random numbers uniformly distributed between 0 and 1
x = np.random.rand(100)
y = x**2

plt.plot(x,y, 'ro')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f965a888d60>]
```



In [ ]:
```
#add gaussian noise to x and y
x_corrupt = x + np.random.randn(100)*0.05
y_corrupt = y + np.random.randn(100)*0.05

plt.plot(x_corrupt,y_corrupt, 'ro')
```

Out[ ]: [<matplotlib.lines.Line2D at 0x7f965a9e4400>]



Problem 5.3 It can mathematically be shown that an autoencoder with a single linear layer is equivalent to PCA. This is because PCA is a linear transformation that tries to find

the directions of maximum variance in the data. The autoencoder with a single linear layer tries to find a subspace which preserves as much information as possible, s.t. the reconstruction loss is reduced.

Problem 5.4

Regular autoencoder

In [ ]:
```
model = AE()
model.load_state_dict(torch.load('mnist_ae.pt'))
```

Out[ ]: <All keys matched successfully>

In [ ]:
```
test_data_flat = test_data.reshape(-1, 28*28)
embedded_ae = model.encoder(test_data_flat)
decoded = model.decoder(embedded_ae)
```

Problem 6

Convolutional auto encoder without skip connections

In [ ]:
```
model_cae = CAE()
model_cae.load_state_dict(torch.load('mnist_cae.pt'))
```

Out[ ]: <All keys matched successfully>

In [ ]:
```
embedded_cae = model_cae.encoder(test_data)
decoded_cae = model_cae.decoder(embedded_cae)
```

Convolutional auto encoder with skip connections

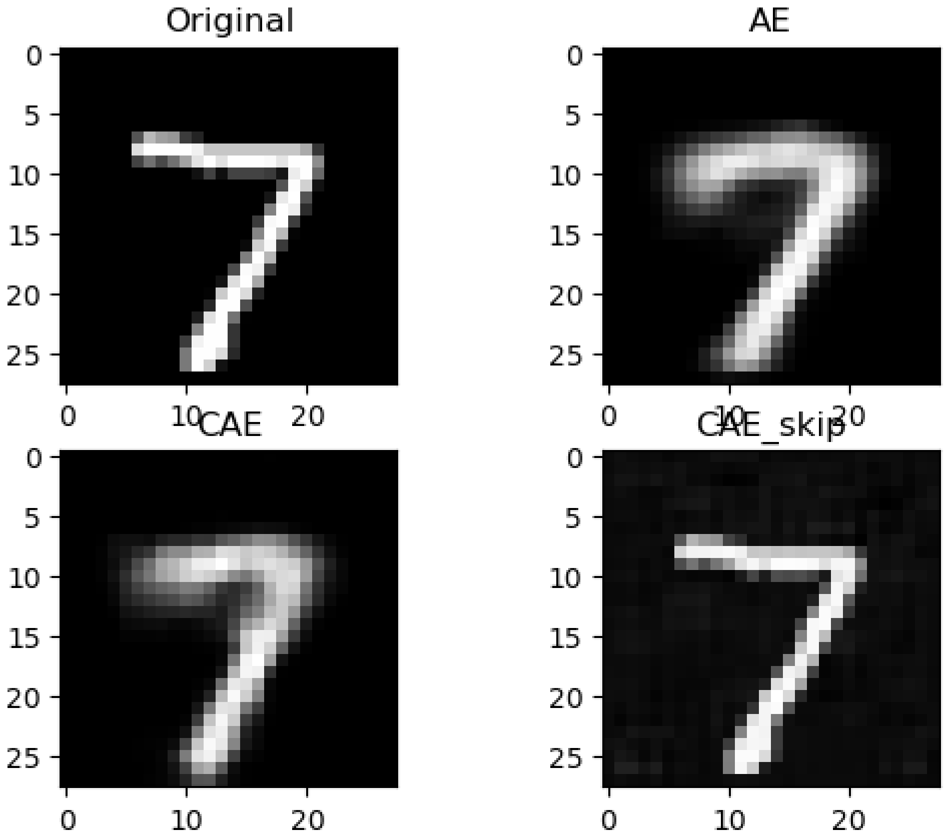
In [ ]:
```
model_cae_skip = CAE_skip()
model_cae_skip.load_state_dict(torch.load('mnist_cae_skip.pt'))
```

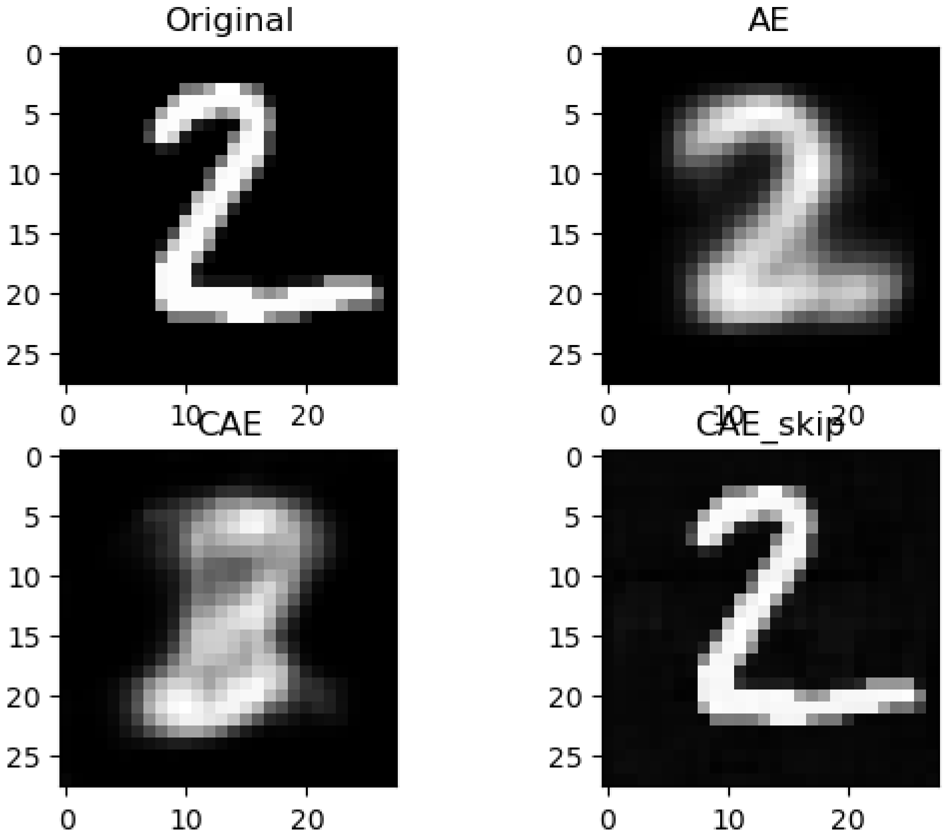
Out[ ]: <All keys matched successfully>

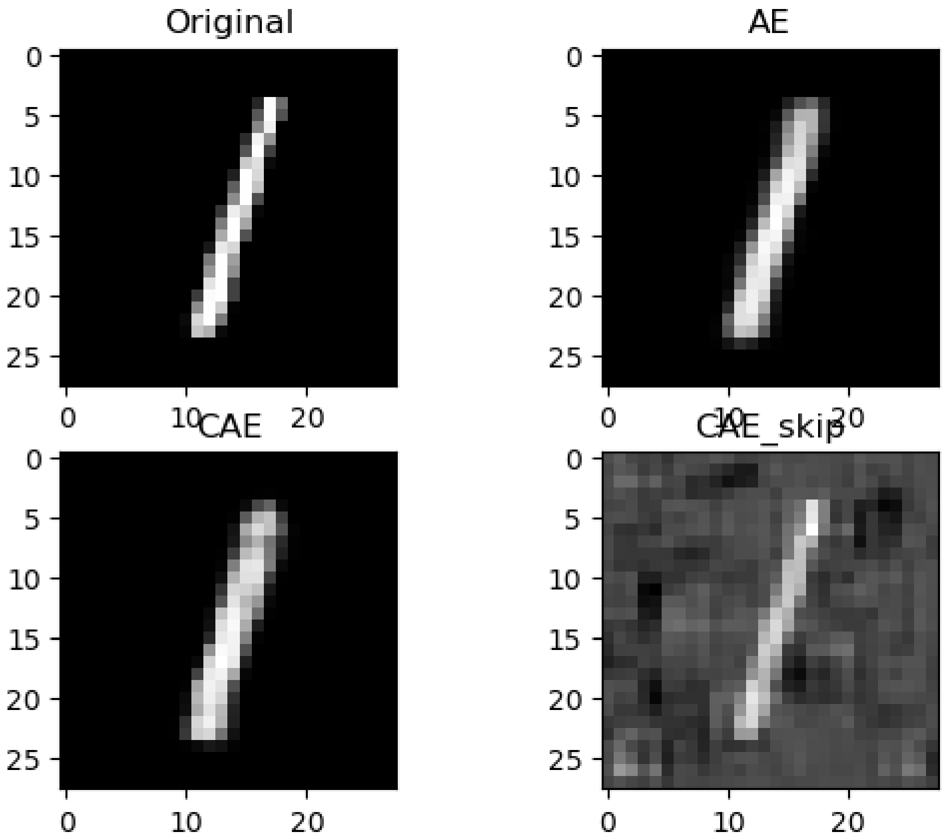
In [ ]:
```
embedded_cae_skip, x4, x1 = model_cae_skip.encoder(test_data)
decoded_cae_skip = model_cae_skip.decoder(embedded_cae, x4, x1)
```

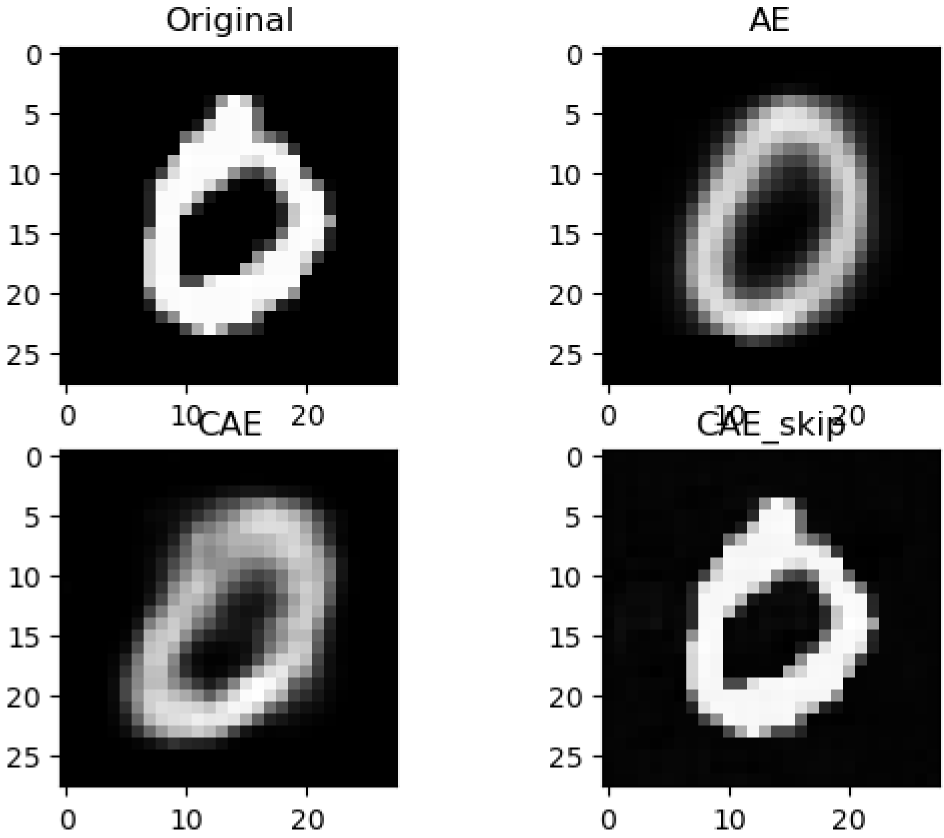
In [ ]:
```
#plot test_data and reconstructed images
for i in range(4):
        #original image
        input_i = test_data[i].detach().numpy()
        #ae
        output_i_ae = decoded[i].detach().numpy()
        #cae
        output_i_cae = decoded_cae[i].detach().numpy()
        #cae_skip
        output_i_cae_skip = decoded_cae_skip[i].detach().numpy()

        input_i = input_i.reshape(28,28)
        output_i_ae = output_i_ae.reshape(28,28)
        output_i_cae = output_i_cae.reshape(28,28)
        output_i_cae_skip = output_i_cae_skip.reshape(28,28)

        #plot data and output next to each other
        plt.figure()
        plt.subplot(2,2,1)
```

```
        plt.imshow(input_i, cmap='gray')
        plt.title('Original')
        plt.subplot(2,2,2)
        plt.imshow(output_i_ae, cmap='gray')
        plt.title('AE')
        plt.subplot(2,2,3)
        plt.imshow(output_i_cae, cmap='gray')
        plt.title('CAE')
        plt.subplot(2,2,4)
        plt.imshow(output_i_cae_skip, cmap='gray')
        plt.title('CAE_skip')
        plt.show()
```



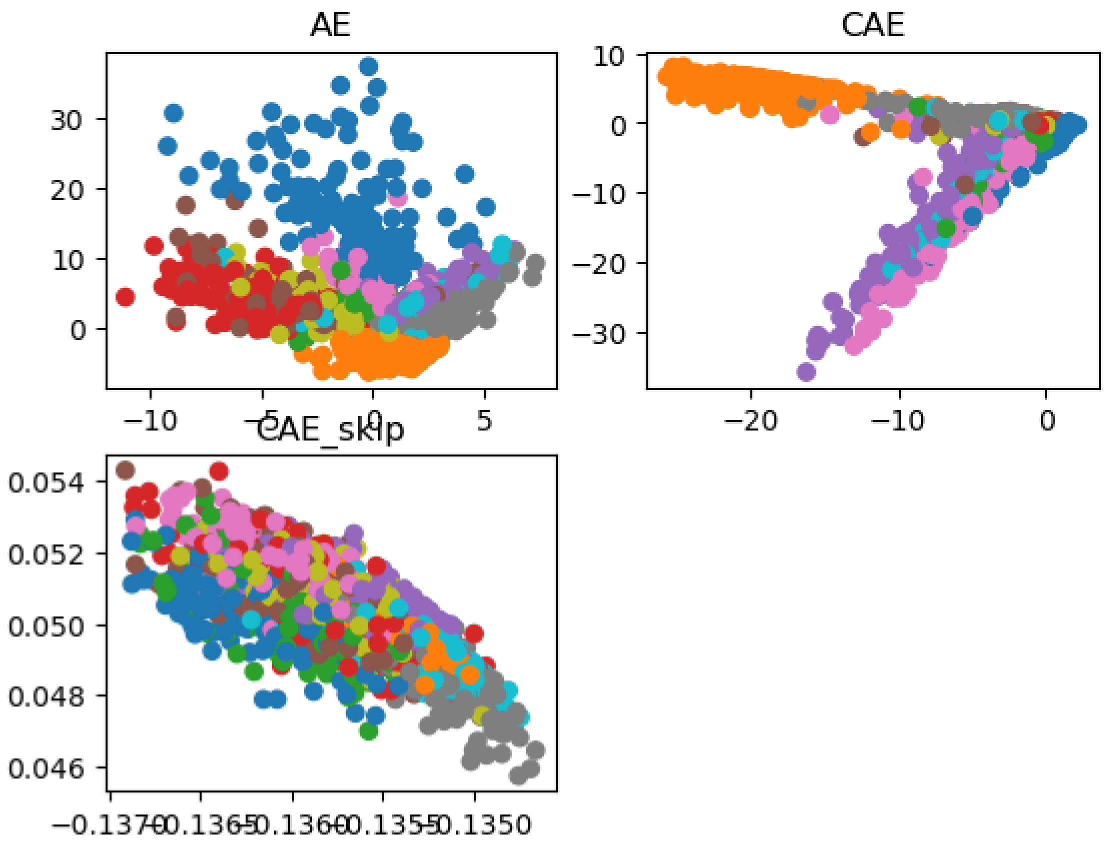

Original
AE
CAE
CAE_skip


Original
AE
CAE
CAE_skip



In [ ]:
```
embedded_ae = embedded_ae.detach().numpy()
embedded_cae = embedded_cae.detach().numpy()
embedded_cae_skip = embedded_cae_skip.detach().numpy()
```

In [ ]:
```
#plot embedded, embedded_cae, embedded_cae_skip
plt.figure()
plt.subplot(2,2,1)
plt.scatter(embedded_ae[:,0], embedded_ae[:,1], c=test_labels, cmap='tab10')
plt.title('AE')
plt.subplot(2,2,2)
plt.scatter(embedded_cae[:,0], embedded_cae[:,1], c=test_labels, cmap='tab10
plt.title('CAE')
plt.subplot(2,2,3)
plt.scatter(embedded_cae_skip[:,0], embedded_cae_skip[:,1], c=test_labels, c
plt.title('CAE_skip')
plt.show()
```



TODO write We see that 5 and 3 are harder to seperate which agrees with our domain knowledge. 7 is similar to 9.

With Corruption

In [ ]:
```
args =  dict(batch_size=64,
        test_batch_size=1000,
        epochs=10,
        momentum=0.5,
        no_cuda=False,
        seed=1,
        log_interval=10,
        save_model=True)

kwargs = {'num_workers': 1, 'pin_memory': True} if not args['no_cuda'] else
```

In [ ]:
```
# Download the MNIST dataset
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, tr
mnist_testset = datasets.MNIST(root='./data', train=False, download=True, tr

# training data
ntrain = 60000
train_data = (mnist_trainset.data.to(dtype=torch.float32)[:ntrain]/255).view
train_labels = mnist_trainset.targets.to(dtype=torch.long)[:ntrain]

    # testing data
ntest = 2000
test_data = (mnist_testset.data.to(dtype=torch.float32)[:ntest]/255).view(-1
test_labels = mnist_testset.targets.to(dtype=torch.long)[:ntest]

    # Load into torch datasets
train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)

train_loader = torch.utils.data.DataLoader(
```

```
    train_dataset, batch_size=args['batch_size'], drop_last=True, shuffle=Tr
)

test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=args['test_batch_size'], drop_last=True, shuffl
)
```

In [ ]:
```
#add gaussian noise with variance 0.3 to train_loader
train_data_corrupt = train_data + torch.randn(train_data.shape)*0.3

train_dataset_corrupt = torch.utils.data.TensorDataset(train_data_corrupt, t
train_loader_corrupt = torch.utils.data.DataLoader(
    train_dataset_corrupt, batch_size=args['batch_size'], drop_last=True, sh
)

#add gaussian noise to test_loader
test_data_corrupt = test_data + torch.randn(test_data.shape)*0.3

test_dataset_corrupt = torch.utils.data.TensorDataset(test_data_corrupt, tes
test_loader_corrupt = torch.utils.data.DataLoader(
    test_dataset_corrupt, batch_size=args['test_batch_size'], drop_last=True
)
```

Training the autoencoder with corruption

Training the convolutional autoencoder with corruption

In [ ]:
```
#main_wrapper(train_loader_corrupt, test_loader_corrupt, CAE(), file_name =

mnist_cae_corrupt = CAE()
mnist_cae_corrupt.load_state_dict(torch.load('mnist_cae_corrupt.pt'))
```

Out[ ]:
```
<All keys matched successfully>
```

TTraining the convolutional autoencoder with skip with corruption

In [ ]:
```
#main_wrapper(train_loader_corrupt, test_loader_corrupt, CAE_skip(), file_na

mnist_cae_skip_corrupt = CAE_skip()
mnist_cae_skip_corrupt.load_state_dict(torch.load('mnist_cae_skip_corrupt.pt
```

```
Train Epoch:   1        | Train set: Average loss: 0.0136
Test set: Average loss: 1.0454

Train Epoch:   2        | Train set: Average loss: 0.0009
Test set: Average loss: 0.3776

Train Epoch:   3        | Train set: Average loss: 0.0003
Test set: Average loss: 0.1074

Train Epoch:   4        | Train set: Average loss: 0.0001
Test set: Average loss: 0.0284

Train Epoch:   5        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0089

Train Epoch:   6        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0036

Train Epoch:   7        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0019

Train Epoch:   8        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0011

Train Epoch:   9        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0006

Train Epoch:  10        | Train set: Average loss: 0.0000
Test set: Average loss: 0.0004
```

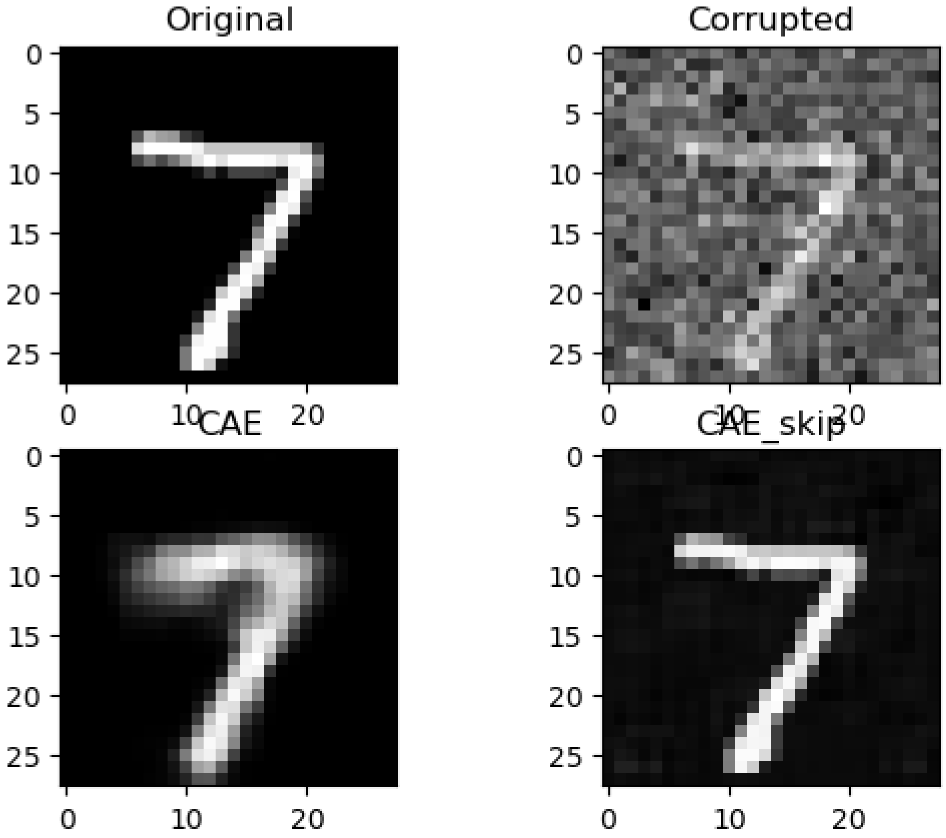
Out[ ]:
```
<All keys matched successfully>
```

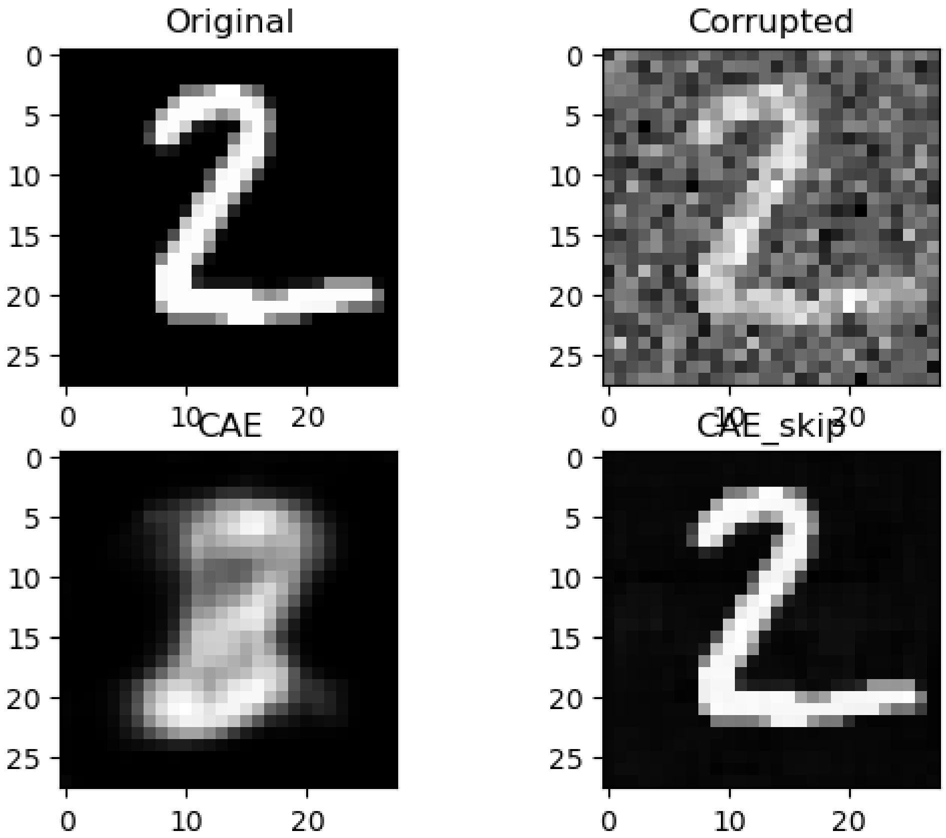
In [ ]:
```
embedded_cae_corrupt = mnist_cae_corrupt.encoder(test_data_corrupt)
decoded_cae_corrupt = mnist_cae_corrupt.decoder(embedded_cae_corrupt)
```

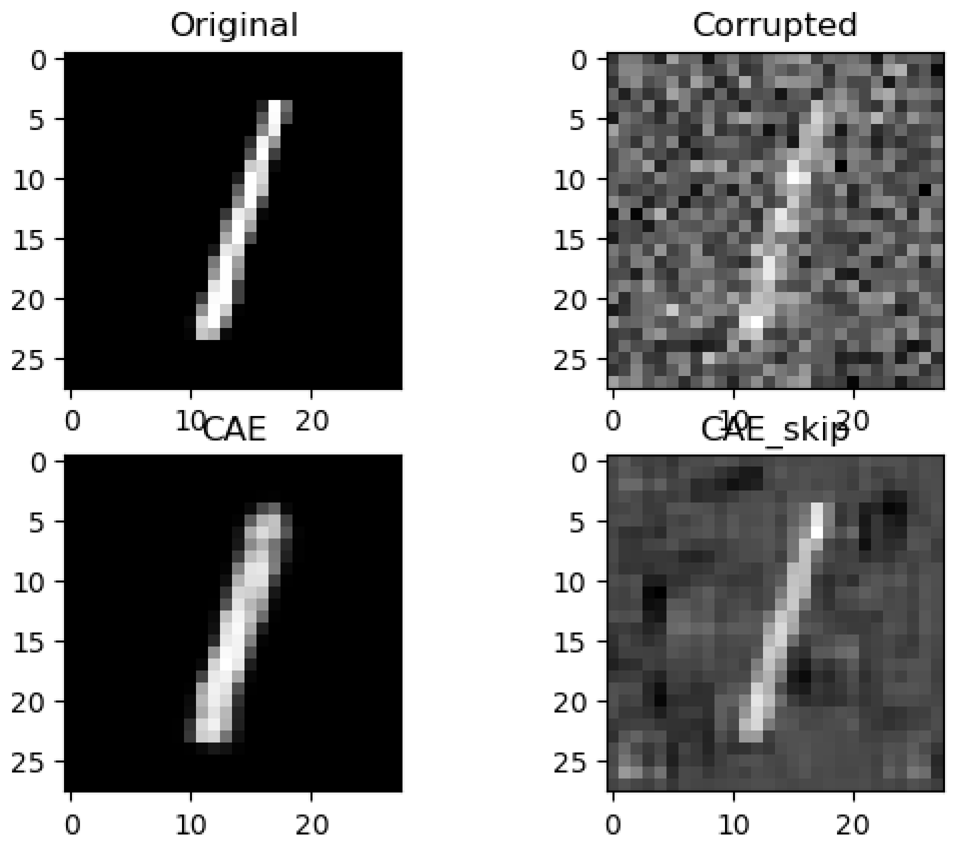
In [ ]:
```
embedded_cae_skip_corrupt, x4, x1 = mnist_cae_skip_corrupt.encoder(test_data
decoded_cae_skip_corrupt = mnist_cae_skip_corrupt.decoder(embedded_cae_skip_
```

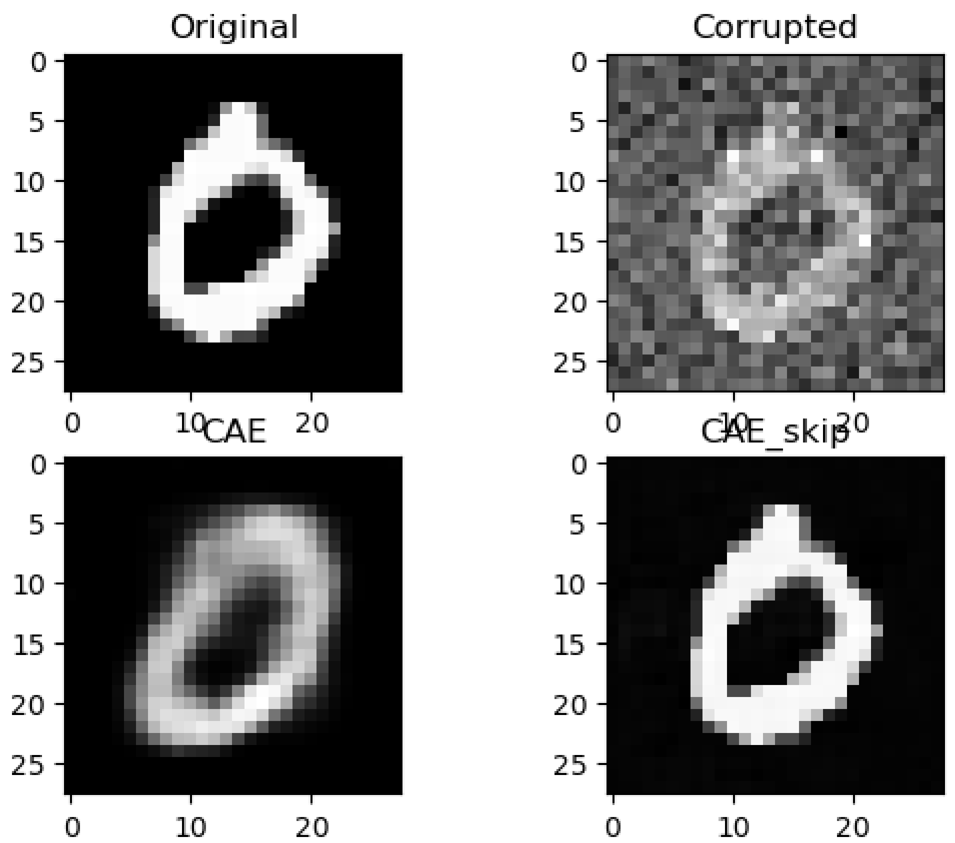
In [ ]:
```
#plot test_data and reconstructed images
for i in range(4):
        #original image
        input_i = test_data[i].detach().numpy()
        #corrupted image
        input_i_corrupt = test_data_corrupt[i].detach().numpy()
        #cae
        output_i_cae = decoded_cae[i].detach().numpy()
        #cae_skip
        output_i_cae_skip = decoded_cae_skip[i].detach().numpy()

        input_i = input_i.reshape(28,28)
        input_i_corrupt = input_i_corrupt.reshape(28,28)
        output_i_cae = output_i_cae.reshape(28,28)
        output_i_cae_skip = output_i_cae_skip.reshape(28,28)

        #plot data and output next to each other
        plt.figure()
        plt.subplot(2,2,1)
        plt.imshow(input_i, cmap='gray')
        plt.title('Original')
        plt.subplot(2,2,2)
        plt.imshow(input_i_corrupt, cmap='gray')
        plt.title('Corrupted')
```

```
    plt.subplot(2,2,3)
    plt.imshow(output_i_cae, cmap='gray')
    plt.title('CAE')
    plt.subplot(2,2,4)
    plt.imshow(output_i_cae_skip, cmap='gray')
    plt.title('CAE_skip')
    plt.show()
```







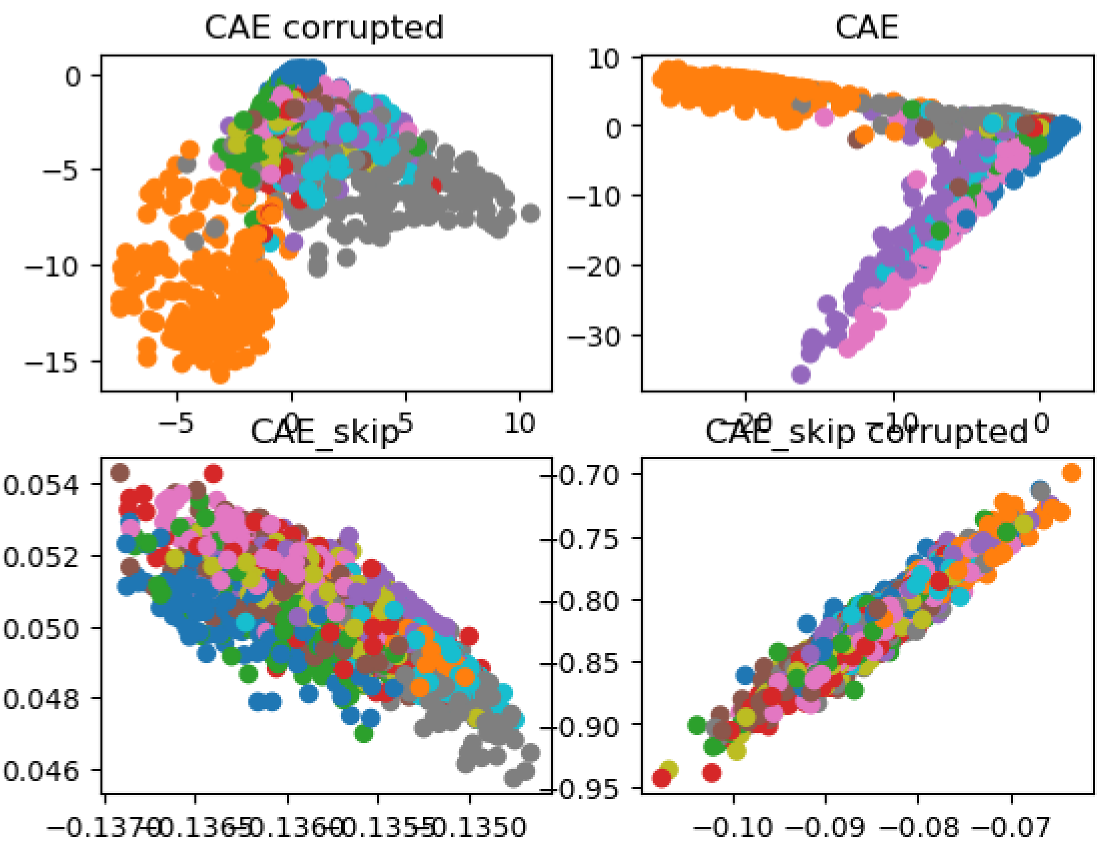


For most pictures, the CAE with skip conncetions performs a lot better!

In [ ]:
```
embedded_cae_corrupt = embedded_cae_corrupt.detach().numpy()
embedded_cae_skip_corrupt = embedded_cae_skip_corrupt.detach().numpy()
```

In [ ]:
```
#plot embedded, embedded_cae, embedded_cae_skip
plt.figure()
plt.subplot(2,2,1)
```

```
plt.scatter(embedded_cae_corrupt[:,0], embedded_cae_corrupt[:,1], c=test_lab
plt.title('CAE corrupted')
plt.subplot(2,2,2)
plt.scatter(embedded_cae[:,0], embedded_cae[:,1], c=test_labels, cmap='tab10
plt.title('CAE')
plt.subplot(2,2,3)
plt.scatter(embedded_cae_skip[:,0], embedded_cae_skip[:,1], c=test_labels, c
plt.title('CAE_skip')
plt.subplot(2,2,4)
plt.scatter(embedded_cae_skip_corrupt[:,0], embedded_cae_skip_corrupt[:,1],
plt.title('CAE_skip corrupted')
plt.show()
```



The emdedding of the CAE with skip connections is a lot more compact than the other two.