Übung 3. Analyzing latent space of variational autoencoders

In this exercise you will get a hands on the Variational Autoencoder (VAE) neural networks, an unsupervized deep learning technique. VAE consists of three main blocks:

- Encoder, which takes input and encodes into a latent space representation
- Latent space - bottleneck of VAE, usually modelled as a multinomial Gaussian distribution
- Decoder, which takes a point in the latent space and decodes it, as close as possible, to the input

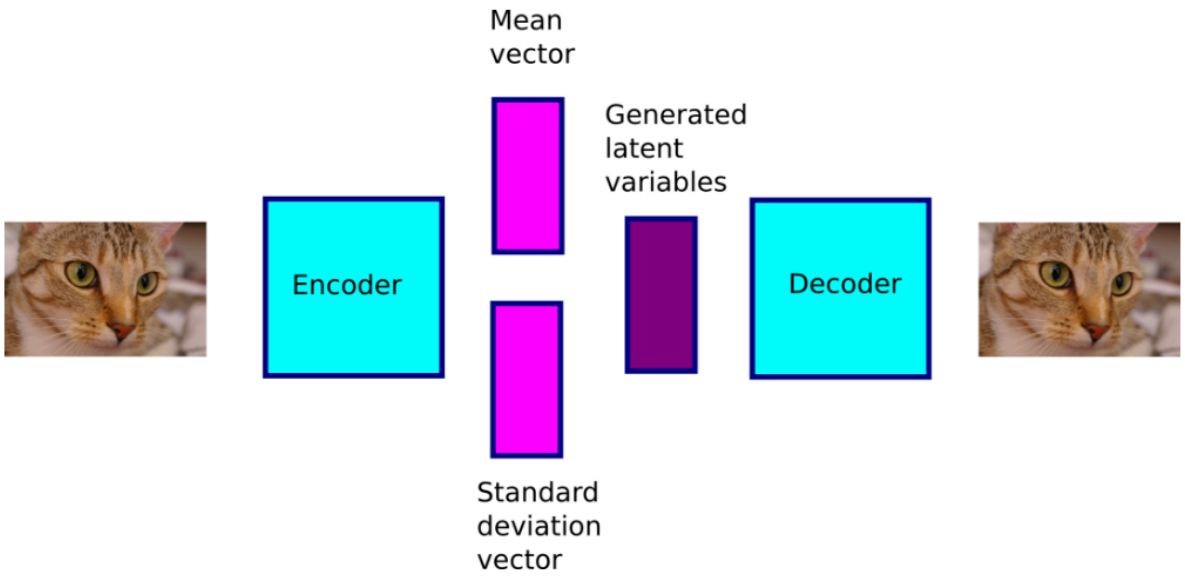
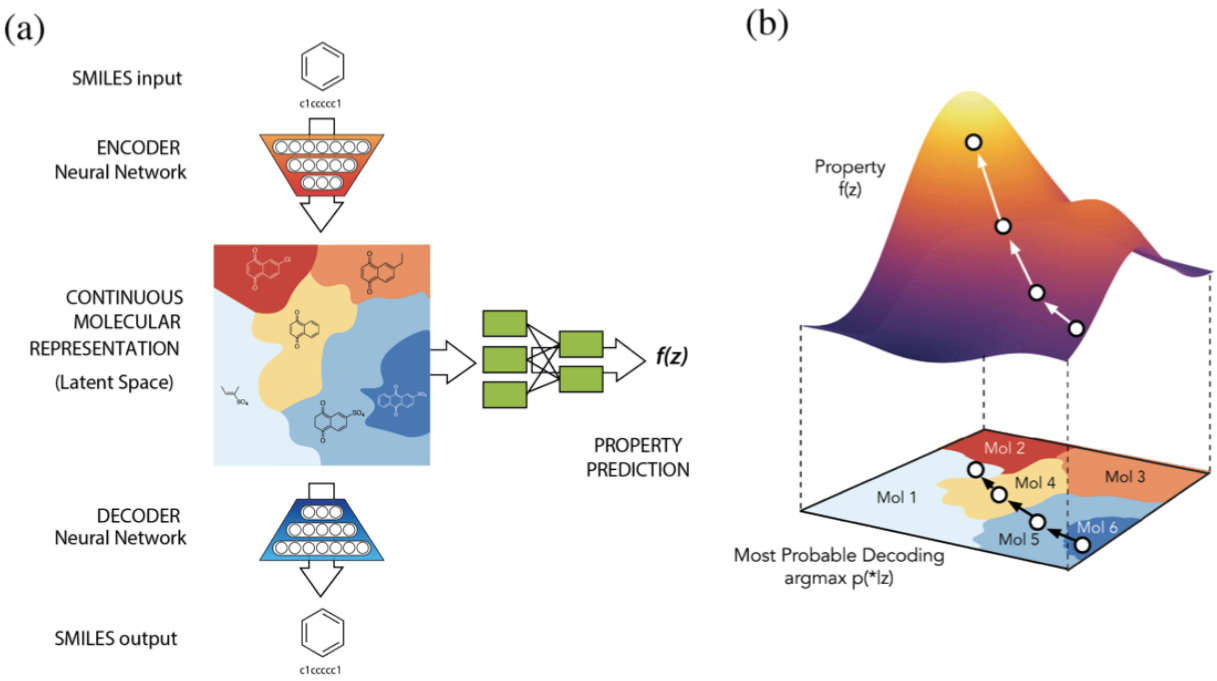


Figure 1: *Image credit: B. Irving Autoencoder tutorial https://github.com/benjaminirving/mlseminars-autoencoders}*

Image credit: B. Irving Autoencoder tutorial https://github.com/benjaminirving/mlseminars-autoencoders}

VAE latent space is a feature space whose structure can be better controlled and interpreted. As such, the dimensions of this feature space may encode important properties of the data, such that changes in this feature space would correspond to interpretable changes in the reconstructed output. In their work, Gomez-Bombarelli et al., use VAE to generate molecule representations from a latent space, whose dimensions encode important molecular properties (e.g., stability, toxicity).



(Image credit) Gomez-Bombarelli "Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules" arxiv. 2017

```
student = 'Maximillian Max MÜstermann' # CHANGE TO YOUR NAME
%reload_ext watermark
%watermark -a "$student" -d -v -m -p torch,torchvision,numpy,scikit-learn
```

```
Maximillian Max MÜstermann 2019-12-07

CPython 3.6.9
IPython 7.10.1

torch 1.3.1+cpu
torchvision 0.4.2+cpu
numpy 1.17.4
sklearn 0.22

compiler   : GCC 5.4.0 20160609
system     : Linux
release    : 4.4.0-170-generic
machine    : x86_64
processor  : x86_64
CPU cores  : 8
```

```
interpreter: 64bit
```

Setting up packages

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid

import numpy as np

from tqdm.notebook import tqdm

from sklearn.manifold import TSNE

%matplotlib inline
import matplotlib.pyplot as plt
```

Download MNIST dataset and set up dataloaders

```
transforms = transforms.Compose([transforms.ToTensor()])
train_dataset = datasets.MNIST(
    './data',
    train=True,
    download=True,
    transform=transforms)

test_dataset = datasets.MNIST(
    './data',
    train=False,
    download=True,
    transform=transforms
)
```

```
train_iterator = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
test_iterator = DataLoader(test_dataset, batch_size=BATCH_SIZE)
```

Visualize one image

```
plt.imshow(train_dataset[0][0].squeeze(), cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fea71eb0cc0>
```

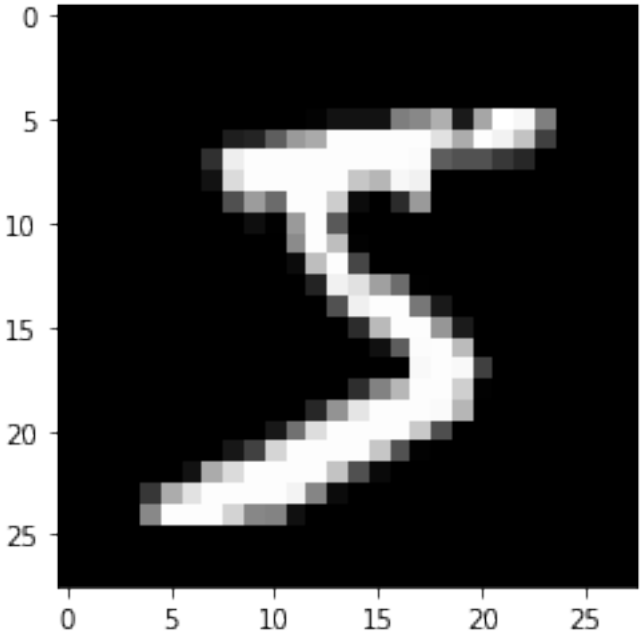


Figure 2: *png*

VAE pytorch implementation

```
class Encoder(nn.Module):
    ''' This the encoder part of VAE
    '''
    def __init__(self, input_dim, hidden_dim, z_dim):
        '''
        Args:
            input_dim: A integer indicating the size of input (in case of MNIST
            hidden_dim: A integer indicating the size of hidden dimension.
            z_dim: A integer indicating the latent dimension.
        '''
```

```
        super().__init__()

        self.linear = nn.Linear(input_dim, hidden_dim)
        self.mu = nn.Linear(hidden_dim, z_dim)
        self.var = nn.Linear(hidden_dim, z_dim)

    def forward(self, x):
        # x is of shape [batch_size, input_dim]

        hidden = F.relu(self.linear(x))
        # hidden is of shape [batch_size, hidden_dim]
        z_mu = self.mu(hidden)
        # z_mu is of shape [batch_size, latent_dim]
        z_var = self.var(hidden)
        # z_var is of shape [batch_size, latent_dim]

        return z_mu, z_var


class Decoder(nn.Module):
    ''' This the decoder part of VAE
    '''
    def __init__(self, z_dim, hidden_dim, output_dim):
        '''
        Args:
            z_dim: A integer indicating the latent size.
            hidden_dim: A integer indicating the size of hidden dimension.
            output_dim: A integer indicating the output dimension (in case of MN
        '''
        super().__init__()

        self.linear = nn.Linear(z_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # x is of shape [batch_size, latent_dim]

        hidden = F.relu(self.linear(x))
```

```
        # hidden is of shape [batch_size, hidden_dim]

        #predicted = torch.sigmoid(self.out(hidden))
        predicted = self.out(hidden)
        # predicted is of shape [batch_size, output_dim]
        return predicted


class VAE(nn.Module):
    def __init__(self, enc, dec):
        ''' This the VAE, which takes a encoder and decoder.
        '''
        super().__init__()

        self.enc = enc
        self.dec = dec

    def forward(self, x):
        # encode
        z_mu, z_var = self.enc(x)

        # sample from the distribution having latent parameters z_mu, z_var
        # reparameterize
        std = torch.exp(z_var / 2)
        eps = torch.randn_like(std)
        x_sample = eps.mul(std).add_(z_mu)

        # decode
        predicted = self.dec(x_sample)
        return predicted, z_mu, z_var
```

Training a VAE to reconstruct MNIST images for 2 epochs

```
# Uncomment if you have access to GPU
# device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device = 'cpu'

BATCH_SIZE = 64         # number of data points in each batch
N_EPOCHS = 2            # times to run the model on complete data
```

```
INPUT_DIM = 28 * 28      # size of each input
HIDDEN_DIM = 256         # hidden dimension
LATENT_DIM = 2          # latent vector dimension
lr = 1e-3                # learning rate

# encoder
encoder = Encoder(INPUT_DIM, HIDDEN_DIM, LATENT_DIM)

# decoder
decoder = Decoder(LATENT_DIM, HIDDEN_DIM, INPUT_DIM)

# vae
model = VAE(encoder, decoder).to(device)

# optimizer
optimizer = optim.Adam(model.parameters(), lr=lr)


def train():
    # set the train mode
    model.train()

    # loss of the epoch
    train_loss = 0

    tr_bar = tqdm(
        enumerate(train_iterator), total=len(train_iterator),
        desc='(train) mini batches', leave=False)

    for i, (x, _) in tr_bar:
        # reshape the data into [batch_size, 784]
        x = x.view(-1, 28 * 28)
        x = x.to(device)

        # update the gradients to zero
        optimizer.zero_grad()

        # forward pass
```

```
        x_sample, z_mu, z_var = model(x)

        # reconstruction loss
        recon_loss = F.binary_cross_entropy_with_logits(x_sample, x, reduction='

        # kl divergence loss
        kl_loss = 0.5 * torch.sum(torch.exp(z_var) + z_mu**2 - 1.0 - z_var)

        # total loss
        loss = recon_loss + kl_loss

        # backward pass
        loss.backward()
        train_loss += loss.item()

        # update the weights
        optimizer.step()

    return train_loss


def test():
    # set the evaluation mode
    model.eval()

    # test loss for the data
    test_loss = 0

    # we don't need to track the gradients, since we are not updating the parame
    with torch.no_grad():

        te_bar = tqdm(
            enumerate(test_iterator), total=len(test_iterator),
            desc='(test) mini batches', leave=False)

        for i, (x, _) in te_bar:
            # reshape the data
            x = x.view(-1, 28 * 28)
```

```
            x = x.to(device)

            # forward pass
            x_sample, z_mu, z_var = model(x)

            # reconstruction loss
            recon_loss = F.binary_cross_entropy_with_logits(x_sample, x, reducti

            # kl divergence loss
            kl_loss = 0.5 * torch.sum(torch.exp(z_var) + z_mu**2 - 1.0 - z_var)

            # total loss
            loss = recon_loss + kl_loss
            test_loss += loss.item()

    return test_loss

epochs_bar = tqdm(
    range(1, N_EPOCHS+1), total=N_EPOCHS
)

for e in epochs_bar:

    train_loss = train()
    test_loss = test()

    train_loss /= len(train_dataset)
    test_loss /= len(test_dataset)

    epochs_bar.set_description('Epoch {}, Train Loss: {:.2f}, Test Loss: {:.2f}'
        e, train_loss, test_loss))
```

```
HBox(children=(FloatProgress(value=0.0, max=2.0), HTML(value='')))
```

```
HBox(children=(FloatProgress(value=0.0, description='(train) mini batches', max=93
```

```
HBox(children=(FloatProgress(value=0.0, description='(test) mini batches', max=157
```

```
HBox(children=(FloatProgress(value=0.0, description='(train) mini batches', max=93
```

```
HBox(children=(FloatProgress(value=0.0, description='(test) mini batches', max=157
```

Sample and generate an image from VAE

```
z = torch.randn(1, LATENT_DIM).to(device)
reconstructed_img = torch.sigmoid(model.dec(z))
img = reconstructed_img.view(28, 28).data.cpu()

print(z.shape)
print(img.shape)
plt.figure()
plt.imshow(img, cmap='gray')
plt.show()
```

```
torch.Size([1, 2])
torch.Size([28, 28])
```

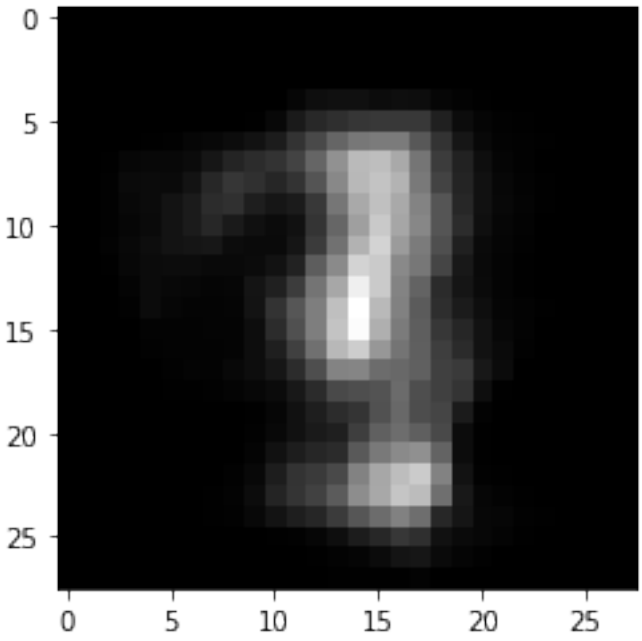


Figure 3: *png*

Assessing image reconstruction quality

How good can a VAE reconstruct images that it had never seen before?

```
def img_to_numpy(img):
    to_np = img.numpy().transpose((1, 2, 0))

    return to_np

te_img, te_labels = next(iter(test_iterator))
te_grid = make_grid(te_img)

rec_imgs, rec_z_mu, rec_z_std = model(te_img.view(-1, 1, 28*28).to(device))
rec_grid = make_grid(rec_imgs.view(-1, 1, 28, 28).data.cpu())

plt.figure(figsize=(12, 10))
plt.subplot(1, 2, 1)
plt.axis('off')
plt.imshow(img_to_numpy(te_grid))
plt.title("Test images")
```

```
plt.subplot(1, 2, 2)
plt.axis('off')
plt.imshow(img_to_numpy(rec_grid))
plt.title("Reconstructed test images")
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
```

```
Text(0.5, 1.0, 'Reconstructed test images')
```

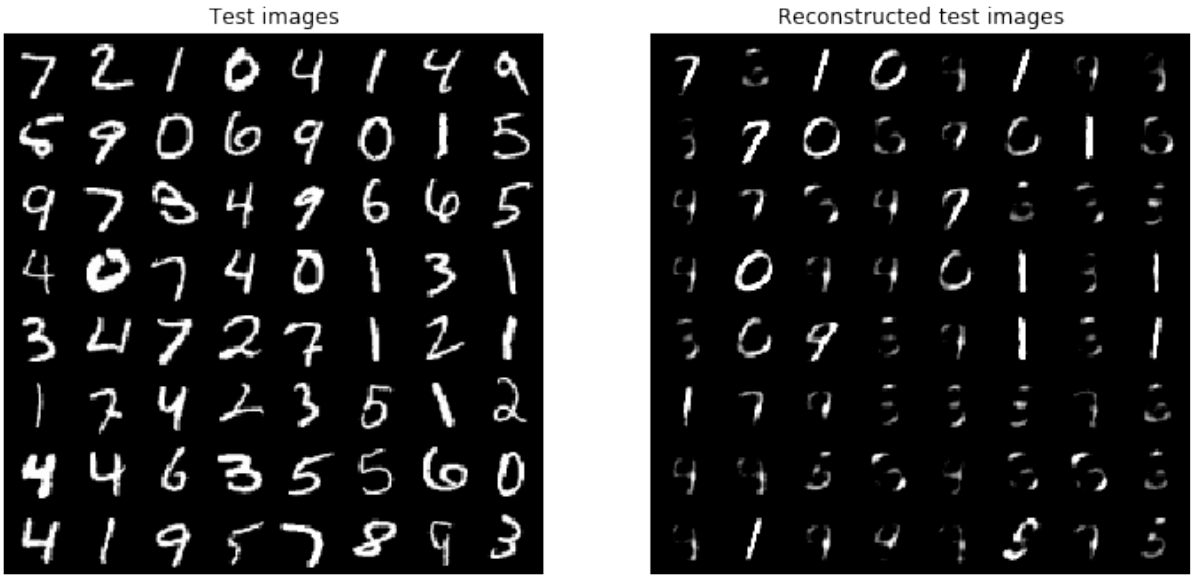


Figure 4: *png*

Question 1: Explain (with your own words and understanding) why a VAE is bad at reconstructing test images (with the current training settings)

Your answer:

Visualization of VAE latent space

In the plot below you can see the latent space of our VAE; it has two dimensions (`LATENT_DIM`). All the generated images by the VAE are *parameterized* with this latent space. You can see where each *digit* lives within this latent space. One point represents an image, and its color tells us which digit it is (i.e., 0 or 9).

```
latent_loader = DataLoader(test_dataset, batch_size=5000)
latent_img, latent_labels = next(iter(latent_loader))
latent_rec_imgs, latent_rec_z_mu, latent_rec_z_std = model(latent_img.view(-1, 1

plt.figure(figsize=(12, 10))
plt.scatter(latent_rec_z_mu.data.cpu()[:, :, 0].numpy().ravel(),
            latent_rec_z_mu.data.cpu()[:, :, 1].numpy().ravel(),
            c=latent_labels.numpy().ravel())
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7fea5c4f0390>
```

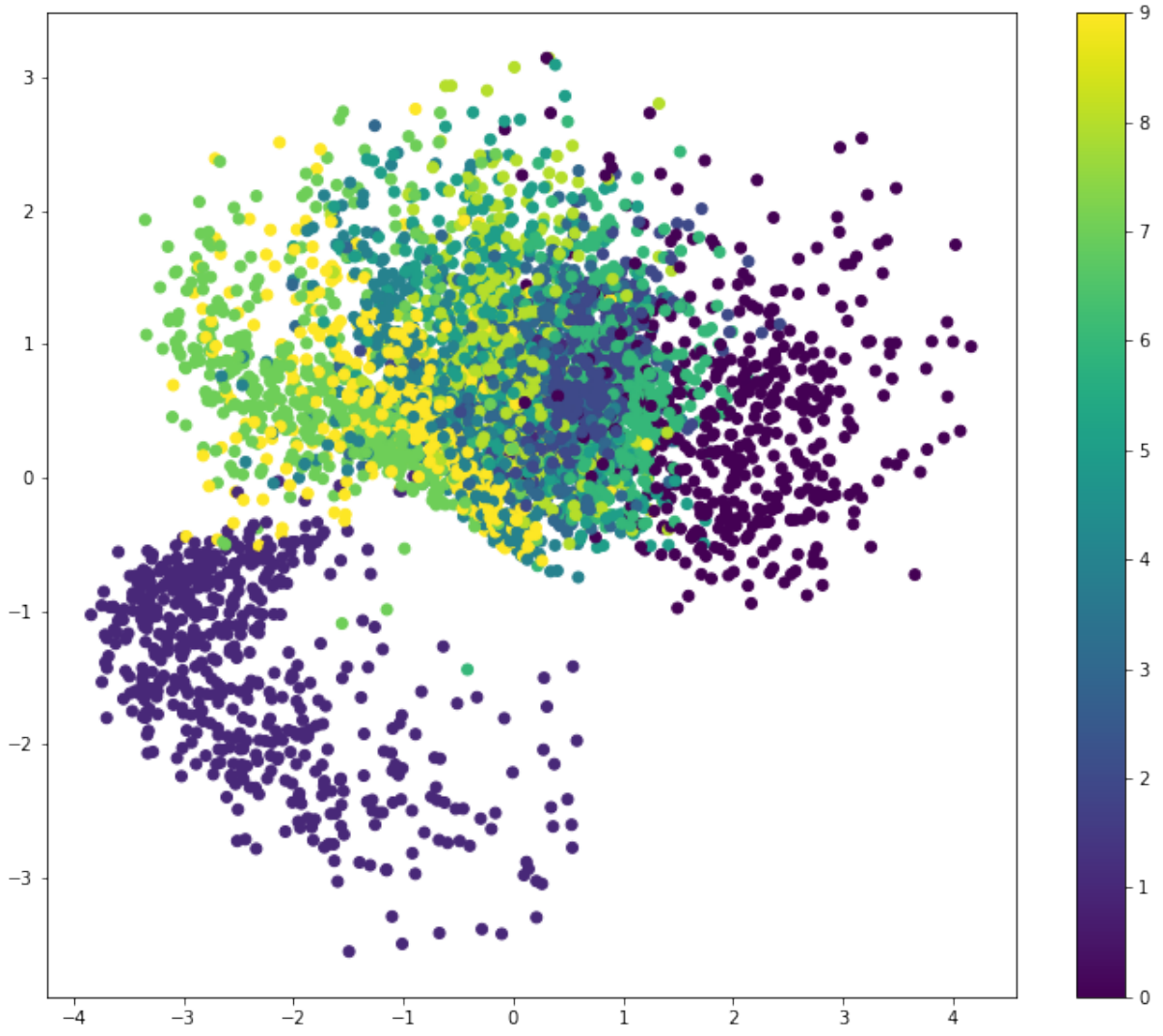


Figure 5: *png*

Question 2: Explain (with your own words and understanding) why so many digits are overlapping, and how does it affect the image reconstruction quality?

Your answer:

Latent space canvas

In this visualization we sample uniformly points (images) from the latent spacce and visualize them.

```
nx = ny = 20
x_values = np.linspace(-4, 4, nx)
y_values = np.linspace(-4, 4, ny)

canvas = np.empty((28*ny, 28*nx))
for i, yi in enumerate(x_values):
    for j, xi in enumerate(y_values):
        z_mu = torch.Tensor([[xi, yi]])
        x_mean = torch.sigmoid(model.dec(z_mu)).data.numpy()
        canvas[(nx-i-1)*28:(nx-i)*28, j*28:(j+1)*28] = x_mean.reshape(28, 28)

plt.figure(figsize=(10, 10))
plt.imshow(canvas, origin="upper", cmap="gray")
axis_ticks = np.linspace(0, 20*28, 10)
axis_labels = ['{:.1g}'.format(x) for x in np.linspace(-4, 4, 10)]
plt.xticks(axis_ticks, axis_labels)
plt.yticks(axis_ticks, axis_labels)
plt.tight_layout()
```

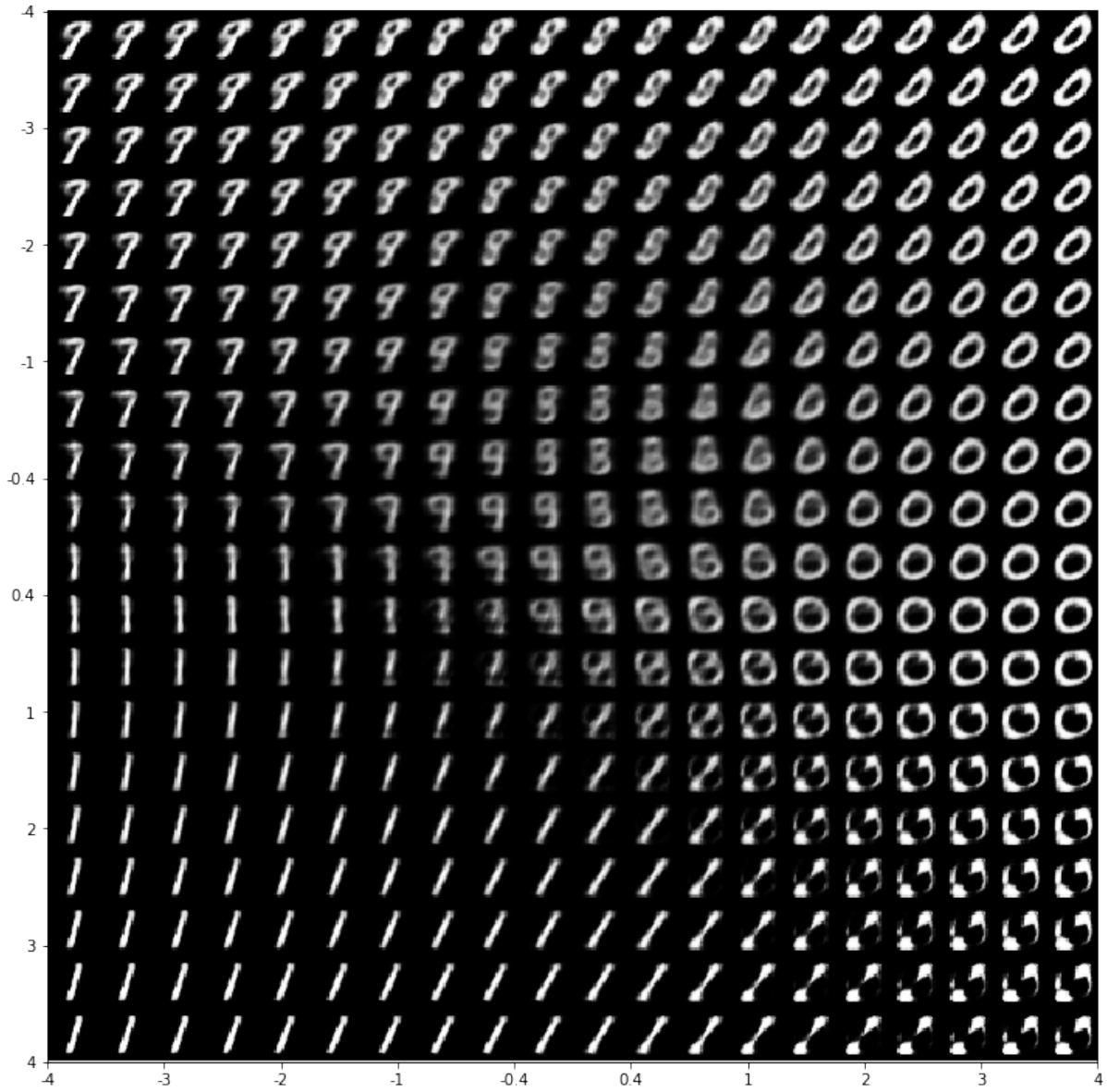


Figure 6: *png*

Question 3: Explain (with your own words and understanding) why some digits seem to have properties of several digits at the same time (i.e., 6 that looks both as 8 and 0)?

Your answer:

Task 1: Train VAE models for more than 2 epochs. Compare visualizations of latent space for different VAE models. Analyze with your own words and understanding.

You can for instance compare latent spaces of these three models:

1. VAE trained for 2 epochs
2. VAE trained for 10 epochs
3. VAE trained for 50 epochs

Always argument and explain your findings. Do not simply *execute* tasks. The goal is to learn!

Bonus task 2: Train VAE models with more dimensions. Use a dimensionality reduction technique to visualize the structure of latent space in 2D

This is not necessary for your grade. However, this is highly recommended for all interested

Below we use *t-SNE* (https://lvdmaaten.github.io/tsne/) to *embed* the latent space of our VAE into a 2-dimensional Euclidean space. Notice that we go from 2 dimensions into 2 dimensions. Your task is to train a VAE with more dimensions (i.e., `LATENT_DIM = 50`), and use *t-SNE* transform to embed the latent space (`mu_z`) into a two-dimensional space.

```
tsne = TSNE(n_components=2, n_jobs=-1)
lat_space = latent_rec_z_mu.squeeze().data.cpu() # extract latent space from you
latent_embedded = tsne.fit_transform(lat_space)
```

```
plt.scatter(latent_embedded[:, 0], latent_embedded[:, 1], c=latent_labels.numpy(
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar at 0x7fea5c97c630>
```

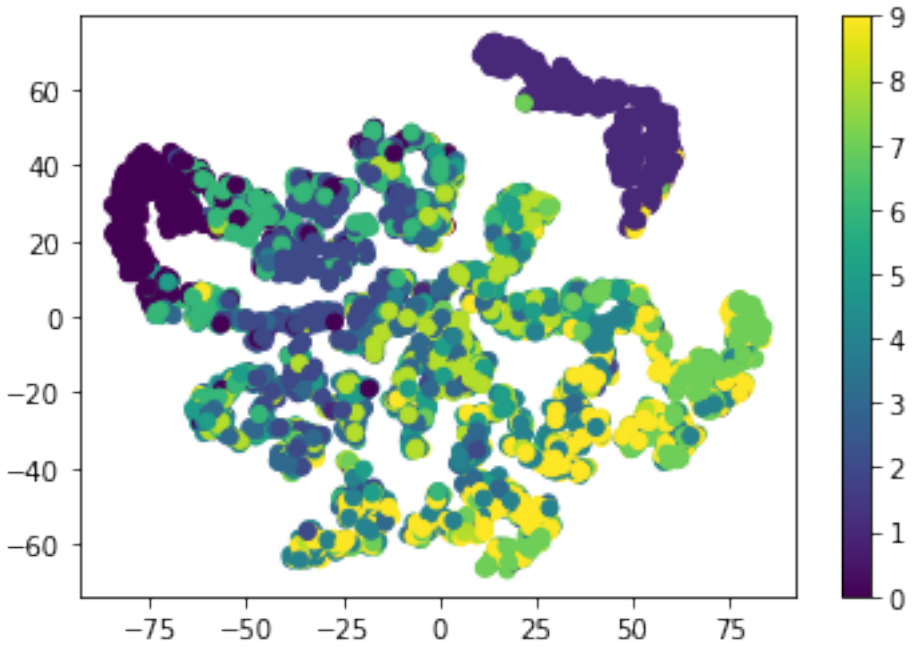


Figure 7: *png*

Your code below