1 $p_{j|i} = \dfrac{\exp(-\|x_i - x_j\|^2 / 2\sigma^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / 2\sigma^2)}$, $p_{i|i} = 0$, $p_{ij} = \dfrac{p_{i|j} + p_{j|i}}{2}$

- $e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}} = \sqrt[2\sigma^2]{\dfrac{1}{e^{\|x_i - x_j\|^2}}}$ $\Rightarrow$ consider an increase in σ to be a change of scale in the probabilities, the neighbour graph plots should be identical but with a darker shade of gray when $\sigma = 5$

$\Rightarrow$ high similarity should result in darker groups $\Rightarrow$ we should see 3 blocks in the neighbour graph plots

$\Rightarrow$ only two of them have darker blocks along the diagonal (b) & f))

$\Rightarrow$ since $\sigma = 5$ should be darker $\Rightarrow$ $\begin{cases} b) \longrightarrow \sigma = 5 \\ f) \longrightarrow \sigma = 2 \end{cases}$

2 if $k < D$, we usually have less degrees of freedom to capture all of the relevant features in the input data.
So when applying this bottleneck in the reconstruction network, the most prominent features are learned and the rest are discarded. Since we lose some amount of information, altough we are able to reconstruct most of the information, the data which was discarded can't be reconstructed, therefore we incur some reconstruction loss in our model.

- if there exists correlated data in our input we can model the full information even with the bottleneck layer.
Consider two parts of the input x_1 and x_2 where we have $x_2 = \lambda x_1$ with some arbitrary $\lambda \in \mathbb{R}$: the autoencoder, even with less neurons will be able to reconstruct the original information since two data points can be reconstructed with a learnt feature and different weights.
So, as long as the number of correlated data points $\geq D - k$ the autoencoder should be able to have zero loss

formally: $f(x) = X W_1 W_2$ and $X \in \mathbb{R}^D$, $W_1 \in \mathbb{R}^{D \times k}$, $W_2 \in \mathbb{R}^{k \times D}$
so as long as X can be fully represented in $\mathbb{R}^k$ with no overlap between points the autoencoder will have no loss.

code

January 16, 2022

0.1 Exporting the results to PDF

Once you complete the assignments, export the entire notebook as PDF and attach it to your homework solutions. The best way of doing that is 1. Run all the cells of the notebook. 2. Export/download the notebook as PDF (File -> Download as -> PDF via LaTeX (.pdf)). 3. Concatenate your solutions for other tasks with the output of Step 2. On linux, you can use `pdfunite`, there are similar tools for other platforms, too. You can only upload a single PDF file to Moodle.

Make sure you are using `nbconvert` version 5.5 or later by running `jupyter nbconvert --version`. Older versions clip lines that exceed page width, which makes your code harder to grade.

1 Matrix Factorization

[ ]:
```
import time
import scipy.sparse as sp
import numpy as np
import warnings
from scipy.sparse.linalg import svds
from sklearn.linear_model import Ridge

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

1.1 Restaurant recommendation

The goal of this task is to recommend restaurants to users based on the rating data in the Yelp dataset. For this, we try to predict the rating a user will give to a restaurant they have not yet rated based on a latent factor model.

Specifically, the objective function (loss) we wanted to optimize is:

$$\mathcal{L} = \min_{P,Q} \sum_{(u,i)\in S} (R_{ui} - \mathbf{q}_u \mathbf{p}_i^T)^2 + \lambda \sum_i \|\mathbf{p}_i\|^2 + \lambda \sum_u \|\mathbf{q}_u\|^2$$

where S is the set of (u, i) pairs for which the rating R_{ui} given by user u to restaurant i is known. Here we have also introduced two regularization terms to help us with overfitting where λ is hyper-parameter that control the strength of the regularization.

The task it to solve the matrix factorization via alternating least squares *and* stochastic gradient descent (non-batched, you may omit the bias).

Hint 1: Using the closed form solution for regression might lead to singular values. To avoid this issue perform the regression step with an existing package such as scikit-learn. It is advisable to use ridge regression to account for regularization.

Hint 2: If you are using the scikit-learn package remember to set `fit_intercept = False` to only learn the coefficients of the linear regression.

1.1.1 Load and Preprocess the Data (nothing to do here)

[ ]:
```
ratings = np.load("exercise_11_matrix_factorization_ratings.npy")
```

[ ]:
```
# We have triplets of (user, restaurant, rating).
ratings
```

[ ]:
```
array([[101968,   1880,      1],
       [101968,    284,      5],
       [101968,   1378,      2],
       …,
       [ 72452,   2100,      4],
       [ 72452,   2050,      5],
       [ 74861,   3979,      5]])
```

Now we transform the data into a matrix of dimension [N, D], where N is the number of users and D is the number of restaurants in the dataset. We store the data as a sparse matrix to avoid out-of-memory issues.

[ ]:
```
n_users = np.max(ratings[:,0] + 1)
n_restaurants = np.max(ratings[:,1] + 1)
R = sp.coo_matrix((ratings[:,2], (ratings[:,0], ratings[:,1])), shape=(n_users,␣
 ↪n_restaurants)).tocsr()
R
```

[ ]:
```
<337867x5899 sparse matrix of type '<class 'numpy.int64'>'
	with 929606 stored elements in Compressed Sparse Row format>
```

To avoid the cold start problem, in the preprocessing step, we recursively remove all users and restaurants with 10 or less ratings.

Then, we randomly select 200 data points for the validation and test sets, respectively.

After this, we subtract the mean rating for each users to account for this global effect.

Note: Some entries might become zero in this process – but these entries are different than the 'unknown' zeros in the matrix. We store the indices for which we the rating data available in a separate variable.

```
[ ]: def cold_start_preprocessing(matrix, min_entries):
         """
         Recursively removes rows and columns from the input matrix which have less␣
      ↪than min_entries nonzero entries.

         Parameters
         ----------
         matrix      : sp.spmatrix, shape [N, D]
                       The input matrix to be preprocessed.
         min_entries : int
                       Minimum number of nonzero elements per row and column.

         Returns
         -------
         matrix      : sp.spmatrix, shape [N', D']
                       The pre-processed matrix, where N' <= N and D' <= D

         """
         print("Shape before: {}".format(matrix.shape))

         shape = (-1, -1)
         while matrix.shape != shape:
             shape = matrix.shape
             nnz = matrix > 0
             row_ixs = nnz.sum(1).A1 > min_entries
             matrix = matrix[row_ixs]
             nnz = matrix > 0
             col_ixs = nnz.sum(0).A1 > min_entries
             matrix = matrix[:, col_ixs]
         print("Shape after: {}".format(matrix.shape))
         nnz = matrix>0
         assert (nnz.sum(0).A1 > min_entries).all()
         assert (nnz.sum(1).A1 > min_entries).all()
         return matrix
```

1.1.2 Task 1: Implement a function that subtracts the mean user rating from the sparse rating matrix

```
[ ]: def shift_user_mean(matrix):
         """
         Subtract the mean rating per user from the non-zero elements in the input␣
      ↪matrix.

         Parameters
         ----------
         matrix : sp.spmatrix, shape [N, D]
                  Input sparse matrix.
```

```
    Returns
    -------
    matrix : sp.spmatrix, shape [N, D]
             The modified input matrix.

    user_means : np.array, shape [N, 1]
                 The mean rating per user that can be used to recover the␣
↪absolute ratings from the mean-shifted ones.

    """
    ## BEGIN SOLUTION
    user_means = np.mean(matrix, axis=1)
    matrix = matrix.astype(np.float64) - user_means
    ## END SOLUTION
    assert np.all(np.isclose(matrix.mean(1), 0))
    return matrix, user_means
```

1.1.3 Split the data into a train, validation and test set (nothing to do here)

```
[ ]: def split_data(matrix, n_validation, n_test):
    """
    Extract validation and test entries from the input matrix.

    Parameters
    ----------
    matrix          : sp.spmatrix, shape [N, D]
                      The input data matrix.
    n_validation    : int
                      The number of validation entries to extract.
    n_test          : int
                      The number of test entries to extract.

    Returns
    -------
    matrix_split    : sp.spmatrix, shape [N, D]
                      A copy of the input matrix in which the validation and␣
↪test entries have been set to zero.

    val_idx         : tuple, shape [2, n_validation]
                      The indices of the validation entries.

    test_idx        : tuple, shape [2, n_test]
                      The indices of the test entries.

    val_values      : np.array, shape [n_validation, ]
                      The values of the input matrix at the validation indices.
```

```
    test_values     : np.array, shape [n_test, ]
                      The values of the input matrix at the test indices.

    """

    matrix_cp = matrix.copy()
    non_zero_idx = np.argwhere(matrix_cp)
    ixs = np.random.permutation(non_zero_idx)
    val_idx = tuple(ixs[:n_validation].T)
    test_idx = tuple(ixs[n_validation:n_validation + n_test].T)

    val_values = matrix_cp[val_idx].A1
    test_values = matrix_cp[test_idx].A1

    matrix_cp[val_idx] = matrix_cp[test_idx] = 0
    matrix_cp.eliminate_zeros()

    return matrix_cp, val_idx, test_idx, val_values, test_values
```

[ ]:
```
R = cold_start_preprocessing(R, 20)
```

```
Shape before: (337867, 5899)
Shape after: (3529, 2072)
```

[ ]:
```
n_validation = 200
n_test = 200
# Split data
R_train, val_idx, test_idx, val_values, test_values = split_data(R,␣
 ↪n_validation, n_test)
```

[ ]:
```
# Remove user means.
nonzero_indices = np.argwhere(R_train)
R_shifted, user_means = shift_user_mean(R_train)
# Apply the same shift to the validation and test data.
val_values_shifted = val_values - user_means[np.array(val_idx).T[:,0]].A1
test_values_shifted = test_values - user_means[np.array(test_idx).T[:,0]].A1
```

1.1.4 Compute the loss function (nothing to do here)

[ ]:
```
def loss(values, ixs, Q, P, reg_lambda):
    """
    Compute the loss of the latent factor model (at indices ixs).
    Parameters
    ----------
    values : np.array, shape [n_ixs,]
        The array with the ground-truth values.
```

```
    ixs : tuple, shape [2, n_ixs]
        The indices at which we want to evaluate the loss (usually the nonzero␣
↪indices of the unshifted data matrix).
    Q : np.array, shape [N, k]
        The matrix Q of a latent factor model.
    P : np.array, shape [k, D]
        The matrix P of a latent factor model.
    reg_lambda : float
        The regularization strength

    Returns
    -------
    loss : float
           The loss of the latent factor model.
    """

    mean_sse_loss = np.sum((values - (Q.dot(P))[ixs])**2)
    regularization_loss =  reg_lambda * (np.sum(np.linalg.norm(P, axis=0)**2) +␣
↪np.sum(np.linalg.norm(Q, axis=1) ** 2))

    return mean_sse_loss + regularization_loss
```

1.2 Alternating optimization

In the first step, we will approach the problem via alternating optimization, as learned in the lecture. That is, during each iteration you first update Q while having P fixed and then vice versa.

1.2.1 Task 2: Implement a function that initializes the latent factors Q and P

[ ]:
```
def initialize_Q_P(matrix, k, init='random'):
    """
    Initialize the matrices Q and P for a latent factor model.

    Parameters
    ----------
    matrix : sp.spmatrix, shape [N, D]
             The matrix to be factorized.
    k      : int
             The number of latent dimensions.
    init   : str in ['svd', 'random'], default: 'random'
             The initialization strategy. 'svd' means that we use SVD to␣
↪initialize P and Q,
             'random' means we initialize the entries in P and Q randomly in␣
↪the interval [0, 1).

    Returns
    -------
```

```
    Q : np.array, shape [N, k]
        The initialized matrix Q of a latent factor model.

    P : np.array, shape [k, D]
        The initialized matrix P of a latent factor model.
    """
    np.random.seed(0)

    ## BEGIN SOLUTION
    N = matrix.shape[0]
    D = matrix.shape[1]

    if (init=="random"):
        Q = np.random.rand(N, k)
        P = np.random.rand(k, D)

    elif (init=="svd"):
        u, s, vt = svds(matrix, k)
        Q = u @ s
        P = vt

    ## END SOLUTION

    assert Q.shape == (matrix.shape[0], k)
    assert P.shape == (k, matrix.shape[1])
    return Q, P
```

1.2.2 Task 3: Implement the alternating optimization approach and stochastic gradient approach

```
[ ]: def latent_factor_alternating_optimization(R, non_zero_idx, k, val_idx,␣
     ↪val_values,
                                                reg_lambda, max_steps=100,␣
     ↪init='random',
                                                log_every=1, patience=5,␣
     ↪eval_every=1, optimizer='sgd', lr=1e-2):
         """
         Perform matrix factorization using alternating optimization. Training is␣
     ↪done via patience,
         i.e. we stop training after we observe no improvement on the validation␣
     ↪loss for a certain
         amount of training steps. We then return the best values for Q and P␣
     ↪oberved during training.

         Parameters
         ----------
```

```
    R                  : sp.spmatrix, shape [N, D]
                         The input matrix to be factorized.

    non_zero_idx       : np.array, shape [nnz, 2]
                         The indices of the non-zero entries of the un-shifted␣
↪matrix to be factorized.
                         nnz refers to the number of non-zero entries. Note that␣
↪this may be different
                         from the number of non-zero entries in the input matrix␣
↪M, e.g. in the case
                         that all ratings by a user have the same value.

    k                  : int
                         The latent factor dimension.

    val_idx            : tuple, shape [2, n_validation]
                         Tuple of the validation set indices.
                         n_validation refers to the size of the validation set.

    val_values         : np.array, shape [n_validation, ]
                         The values in the validation set.

    reg_lambda         : float
                         The regularization strength.

    max_steps          : int, optional, default: 100
                         Maximum number of training steps. Note that we will␣
↪stop early if we observe
                         no improvement on the validation error for a specified␣
↪number of steps
                         (see "patience" for details).

    init               : str in ['random', 'svd'], default 'random'
                         The initialization strategy for P and Q. See function␣
↪initialize_Q_P for details.

    log_every          : int, optional, default: 1
                         Log the training status every X iterations.

    patience           : int, optional, default: 5
                         Stop training after we observe no improvement of the␣
↪validation loss for X evaluation
                         iterations (see eval_every for details). After we stop␣
↪training, we restore the best
                         observed values for Q and P (based on the validation␣
↪loss) and return them.
```

```
    eval_every        : int, optional, default: 1
                        Evaluate the training and validation loss every X steps.
↪ If we observe no improvement
                        of the validation error, we decrease our patience by 1,␣
↪else we reset it to *patience*.

    optimizer         : str, optional, default: sgd
                        If `sgd` stochastic gradient descent shall be used.␣
↪Otherwise, use alternating least squares.

    Returns
    -------
    best_Q            : np.array, shape [N, k]
                        Best value for Q (based on validation loss) observed␣
↪during training

    best_P            : np.array, shape [k, D]
                        Best value for P (based on validation loss) observed␣
↪during training

    validation_losses : list of floats
                        Validation loss for every evaluation iteration, can be␣
↪used for plotting the validation
                        loss over time.

    train_losses      : list of floats
                        Training loss for every evaluation iteration, can be␣
↪used for plotting the training
                        loss over time.

    converged_after   : int
                        it - patience*eval_every, where it is the iteration in␣
↪which patience hits 0,
                        or -1 if we hit max_steps before converging.

    """

    ## BEGIN SOLUTION

    Q, P = initialize_Q_P(R, k, init)
    nnzs = tuple(non_zero_idx.T)
    values = R[nnzs].A1

    model = Ridge(alpha=reg_lambda, fit_intercept=False)
    curr_patience = patience
```

```python
    train_losses = []
    validation_losses = []
    prev_loss = None

    bestq = Q
    bestp = P

    for step in range(max_steps):

        if (optimizer == "sgd"):
            for idx in non_zero_idx:
                u, i = idx[0], idx[1]
                error = R[u,i] - Q[u,:]@P[:,i]

                Q[u,:] += 2*lr*(error*P[:,i] - reg_lambda*Q[u,:])
                P[:,i] += 2*lr*(error*Q[u,:] - reg_lambda*P[:,i])

        else:

            P = model.fit(Q, R).coef_.T
            Q = model.fit(P.T, R.T).coef_

        train_loss = loss(values, nnzs, Q, P, reg_lambda)
        train_losses.append(train_loss)

        if step % eval_every == 0:
            val_loss = loss(val_values, val_idx, Q, P, reg_lambda)
            validation_losses.append(val_loss)

            if (prev_loss == None):
                prev_loss = val_loss
                bestq = Q
                bestp = P

            elif (val_loss >= prev_loss):
                curr_patience -= 1

            else:
                curr_patience = patience
                prev_loss = val_loss
                bestq = Q
                bestp = P

        if step % log_every == 0:
            print(f"Iteration {step}, training loss {train_loss}, validation␣
↪loss: {val_loss}")
```

```
        if curr_patience == 0:
            return bestq, bestp, validation_losses, train_losses, step -␣
↪patience*eval_every


    ## END SOLUTION

    return bestq, bestp, validation_losses, train_losses, -1
```

1.2.3 Train the latent factor (nothing to do here)

[ ]:
```
warnings.filterwarnings("ignore")

Q_sgd, P_sgd, val_loss_sgd, train_loss_sgd, converged_sgd =␣
↪latent_factor_alternating_optimization(
    R_shifted, nonzero_indices, k=100, val_idx=val_idx,␣
↪val_values=val_values_shifted,
    reg_lambda=1e-4, init='random', max_steps=100, patience=10,␣
↪optimizer='sgd', lr=1e-2
)
```

```
Iteration 0, training loss 1913079.2151871456, validation loss:
1926.4579207727056
Iteration 1, training loss 614026.4001069352, validation loss:
1049.7005575426556
Iteration 2, training loss 146449.63059935515, validation loss:
527.2866655889877
Iteration 3, training loss 61226.65438399374, validation loss: 458.5507202563196
Iteration 4, training loss 36834.42086907077, validation loss:
462.27078078076204
Iteration 5, training loss 27495.456497314935, validation loss:
467.8327806711508
Iteration 6, training loss 21207.66994654611, validation loss: 470.1966562465473
Iteration 7, training loss 16862.178702675774, validation loss:
473.2846946000175
Iteration 8, training loss 13880.212494956544, validation loss:
477.34503044557704
Iteration 9, training loss 11709.370453727981, validation loss:
481.5108125202826
Iteration 10, training loss 10043.317429016663, validation loss:
485.3808170460918
Iteration 11, training loss 8724.422148330637, validation loss:
488.8881773071899
Iteration 12, training loss 7661.006172825707, validation loss:
492.0178068031721
Iteration 13, training loss 6789.678948837059, validation loss: 494.73962891658
```

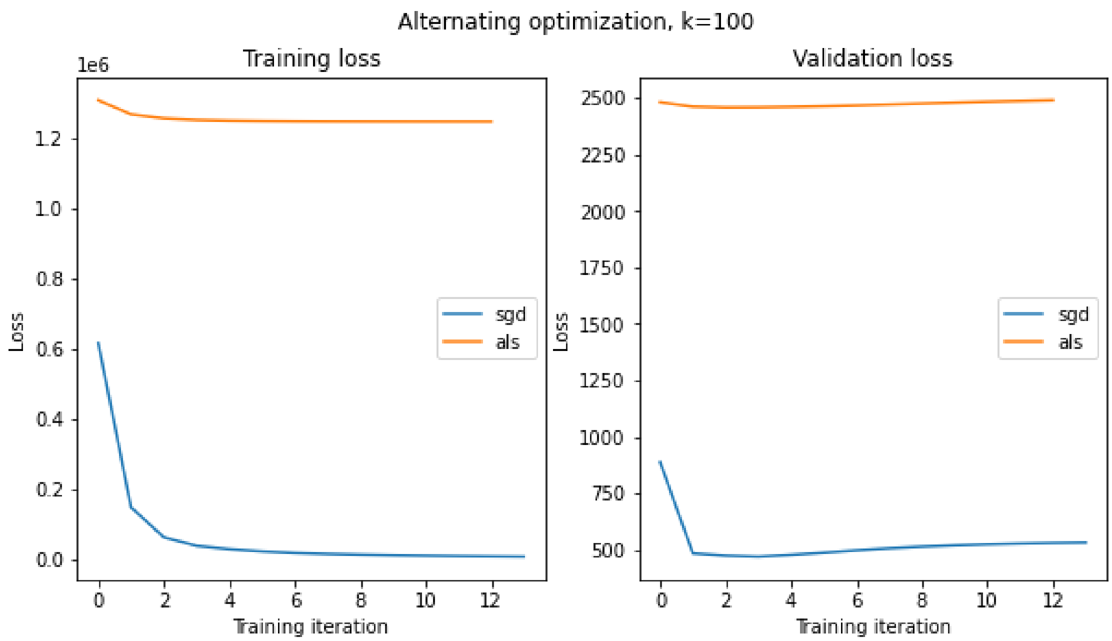
[ ]:
```
warnings.filterwarnings("ignore")

Q_als, P_als, val_loss_als, train_loss_als, converged_als =␣
↪latent_factor_alternating_optimization(
    R_shifted, nonzero_indices, k=100, val_idx=val_idx,␣
↪val_values=val_values_shifted,
    reg_lambda=1e-4, init='random', max_steps=100, patience=10, optimizer='als'
)
```

```
Iteration 0, training loss 1560767.4818099355, validation loss:
2604.1472874894207
Iteration 1, training loss 1306142.5095206571, validation loss:
2482.395308523441
Iteration 2, training loss 1265965.9163291007, validation loss:
2463.1023686092108
Iteration 3, training loss 1254591.5464836305, validation loss:
2459.3640057803314
Iteration 4, training loss 1250151.0745797595, validation loss:
2460.109722872749
Iteration 5, training loss 1248086.083126195, validation loss: 2462.135578901375
Iteration 6, training loss 1247022.1927940175, validation loss:
2464.9837162351287
Iteration 7, training loss 1246430.0720933112, validation loss:
2468.5432983716278
Iteration 8, training loss 1246077.5687334493, validation loss:
2472.5469834416435
Iteration 9, training loss 1245855.046172327, validation loss:
2476.6872921667295
Iteration 10, training loss 1245707.243764587, validation loss:
2480.727499803756
Iteration 11, training loss 1245604.4233051357, validation loss:
2484.5220201200186
Iteration 12, training loss 1245529.5686815053, validation loss:
2487.997125606604
Iteration 13, training loss 1245472.4297945707, validation loss:
2491.1274942152954
```

1.2.4 Plot the validation and training losses over for each iteration (nothing to do here)

[ ]:
```
fig, ax = plt.subplots(1, 2, figsize=[10, 5])
fig.suptitle("Alternating optimization, k=100")

ax[0].plot(train_loss_sgd[1::], label='sgd')
ax[0].plot(train_loss_als[1::], label='als')
ax[0].set_title('Training loss')
ax[0].set_xlabel("Training iteration")
```

```
ax[0].set_ylabel("Loss")
ax[0].legend()


ax[1].plot(val_loss_sgd[1::], label='sgd')
ax[1].plot(val_loss_als[1::], label='als')
ax[1].set_title('Validation loss')
ax[1].set_xlabel("Training iteration")
ax[1].set_ylabel("Loss")
ax[1].legend()

plt.show()
```



2 Autoencoder and t-SNE

Hereinafter, we will implement an autoencoder and analyze its latent space via interpolations and t-SNE. For this, we will use the famous Fashion-MNIST dataset.

[ ]:
```
from typing import List

from matplotlib.offsetbox import AnnotationBbox, OffsetImage
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import torchvision
```

```python
from torchvision.datasets import FashionMNIST
import torch
from torch import nn
import torch.nn.functional as F
from torch.optim.lr_scheduler import ExponentialLR
from copy import deepcopy
```

Hint: If you run into memory issues simply reduce the `batch_size`

[ ]:
```python
train_dataset = FashionMNIST(root='data', download=True, train=True,␣
↪transform=torchvision.transforms.ToTensor())
test_dataset = FashionMNIST(root='data', download=True, train=False,␣
↪transform=torchvision.transforms.ToTensor())
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=1024,␣
↪shuffle=True,
                                           num_workers=2, pin_memory=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1024,␣
↪shuffle=False,
                                          num_workers=2, pin_memory=True)
```

2.0.1 Task 4: Define decoder network

Feel free to choose any architecture you like. Our model was this:

```
Autoencoder(
  (encode): Sequential(
    (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): Conv2d(4, 16, kernel_size=(3, 3), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): LeakyReLU(negative_slope=0.01)
    (5): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): LeakyReLU(negative_slope=0.01)
    (10): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
    (11): LeakyReLU(negative_slope=0.01)
  )
  (decode): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2), output_padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.01)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ConvTranspose2d(16, 16, kernel_size=(3, 3), stride=(2, 2), output_padding=(1, 1))
    (6): LeakyReLU(negative_slope=0.01)
    (7): ConvTranspose2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (8): ConvTranspose2d(16, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ConvTranspose2d(4, 1, kernel_size=(3, 3), stride=(1, 1))
    (10): Sigmoid()
  )
)
```

[ ]:
```
class Autoencoder(nn.Module):
    ## BEGIN SOLUTION

    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(4, 16, kernel_size=(3, 3), stride=(1, 1)),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,␣
↪ceil_mode=False),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,␣
↪track_running_stats=True),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1)),
            nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,␣
↪ceil_mode=False),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1)),
            nn.LeakyReLU(negative_slope=0.01)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(1, 1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.ConvTranspose2d(32, 16, kernel_size=(
                3, 3), stride=(2, 2), output_padding=(1, 1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(16, eps=1e-05, momentum=0.1,
                           affine=True, track_running_stats=True),
            nn.ConvTranspose2d(16, 16, kernel_size=(
                3, 3), stride=(2, 2), output_padding=(1, 1)),
            nn.LeakyReLU(negative_slope=0.01),
            nn.ConvTranspose2d(16, 16, kernel_size=(
                3, 3), stride=(1, 1), padding=(1, 1)),
            nn.ConvTranspose2d(16, 4, kernel_size=(
                3, 3), stride=(1, 1), padding=(1, 1)),
            nn.ConvTranspose2d(4, 1, kernel_size=(3, 3), stride=(1, 1)),
            nn.Sigmoid()
        )
```

```python
        self.decoder.apply(self.initialize_weight)
        self.encoder.apply(self.initialize_weight)

    def initialize_weight(self, module):
        if isinstance(module, (nn.ConvTranspose2d, nn.Conv2d)):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
        elif isinstance(module, nn.BatchNorm2d):
            nn.init.constant_(module.weight, 1)
            nn.init.constant_(module.bias, 0)

    def encode(self, x):
        return self.encoder.forward(x)

    def decode(self, x):
        return self.decoder.forward(x)

    ## END SOLUTION
    def forward(self, x):
        z = self.encode(x)
        x_approx = self.decode(z)

        assert x.shape == x_approx.shape
        return x_approx

print(Autoencoder())
```

```
Autoencoder(
  (encoder): Sequential(
    (0): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
    (2): Conv2d(4, 16, kernel_size=(3, 3), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (4): LeakyReLU(negative_slope=0.01)
    (5): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (6): LeakyReLU(negative_slope=0.01)
    (7): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (9): LeakyReLU(negative_slope=0.01)
    (10): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
    (11): LeakyReLU(negative_slope=0.01)
  )
  (decoder): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): LeakyReLU(negative_slope=0.01)
```

```
    (2): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(2, 2),
output_padding=(1, 1))
    (3): LeakyReLU(negative_slope=0.01)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ConvTranspose2d(16, 16, kernel_size=(3, 3), stride=(2, 2),
output_padding=(1, 1))
    (6): LeakyReLU(negative_slope=0.01)
    (7): ConvTranspose2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (8): ConvTranspose2d(16, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
    (9): ConvTranspose2d(4, 1, kernel_size=(3, 3), stride=(1, 1))
    (10): Sigmoid()
  )
)
```

We see that our model transform the image from $28 \cdot 28 = 784$ dimensional space down into a $32 \cdot 3 \cdot 3 = 288$ dimensional space. However, note that the latent space also must contain some spatial information that the decoder needs for decoding.

[ ]:
```python
x = test_dataset[0][0][None, ...]
z = Autoencoder().encode(x)

print(x.shape)
print(z.shape)
print(Autoencoder().decode(z).shape)
```

```
torch.Size([1, 1, 28, 28])
torch.Size([1, 32, 3, 3])
torch.Size([1, 1, 28, 28])
```

2.1 Task 5: Train the autoencoder

Of course, we must train the autoencoder if we want to analyze it later on.

[ ]:
```python
device = 0 if torch.cuda.is_available() else 'cpu'
model = Autoencoder().to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
scheduler = ExponentialLR(optimizer, gamma=0.999)

log_every_batch = 20

# defining patience to speed up training
patience = 10
curr_patience = 10
best_loss = None
best_weights = None
```

```python
for epoch in range(50):
    model.train()
    train_loss_trace = []
    for batch, (x, _) in enumerate(train_loader):
        ## BEGIN SOLUTION

        x = x.to(device)
        x_approx = model(x)
        loss = F.mse_loss(x_approx,x)

        loss.backward()
        optimizer.step()

        ## END SOLUTION
        train_loss_trace.append(loss.detach().item())
        if batch % log_every_batch == 0:
            print(f'Training: Epoch {epoch} batch {batch} - loss {loss}')

    model.eval()
    test_loss_trace = []
    for batch, (x, _) in enumerate(test_loader):
        x = x.to(device)
        x_approx = model(x)
        loss = F.mse_loss(x_approx, x)
        test_loss_trace.append(loss.detach().item())

        if (best_loss == None or loss <= best_loss):
            best_weights = deepcopy(model.state_dict())
            best_loss = loss
            curr_patience = patience

        else:
            curr_patience -= 1

        if batch % log_every_batch == 0:
            print(f'Test: Epoch {epoch} batch {batch} loss {loss}')

    print(f'Epoch {epoch} finished - average train loss {np.
 ↪mean(train_loss_trace)}, '
          f'average test loss {np.mean(test_loss_trace)}')

    if (curr_patience == 0):
      break

model.load_state_dict(best_weights)
```

```
Training: Epoch 0 batch 0 - loss 0.22105960547924042
Training: Epoch 0 batch 20 - loss 0.2440939098596573
Training: Epoch 0 batch 40 - loss 0.187652587890625
Test: Epoch 0 batch 0 loss 0.15293912589550018
Epoch 0 finished - average train loss 0.18025842005923642, average test loss
0.15142675042152404
Training: Epoch 1 batch 0 - loss 0.15250302851200104
Training: Epoch 1 batch 20 - loss 0.22972092032432556
Training: Epoch 1 batch 40 - loss 0.10326273739337921
Test: Epoch 1 batch 0 loss 0.1264612376689911
Epoch 1 finished - average train loss 0.14353578146231377, average test loss
0.12592412382364274
Training: Epoch 2 batch 0 - loss 0.14086087048053741
Training: Epoch 2 batch 20 - loss 0.12498904019594193
Training: Epoch 2 batch 40 - loss 0.12210098654031754
Test: Epoch 2 batch 0 loss 0.15943506360054016
Epoch 2 finished - average train loss 0.12632550867432255, average test loss
0.15964883267879487
Training: Epoch 3 batch 0 - loss 0.12337891012430191
Training: Epoch 3 batch 20 - loss 0.12574464082717896
Training: Epoch 3 batch 40 - loss 0.1139763593673706
Test: Epoch 3 batch 0 loss 0.10677484422922134
Epoch 3 finished - average train loss 0.11739583629167687, average test loss
0.10619214922189713
Training: Epoch 4 batch 0 - loss 0.11164586246013641
Training: Epoch 4 batch 20 - loss 0.13948604464530945
Training: Epoch 4 batch 40 - loss 0.141514852643013
Test: Epoch 4 batch 0 loss 0.16125158965587616
Epoch 4 finished - average train loss 0.14126976831989774, average test loss
0.15959856361150743
Training: Epoch 5 batch 0 - loss 0.18048477172851562
Training: Epoch 5 batch 20 - loss 0.16199259459972382
Training: Epoch 5 batch 40 - loss 0.15570378303527832
Test: Epoch 5 batch 0 loss 0.14271414279937744
Epoch 5 finished - average train loss 0.16010575981463415, average test loss
0.14148640483617783
Training: Epoch 6 batch 0 - loss 0.1500980257987976
Training: Epoch 6 batch 20 - loss 0.13786426186561584
Training: Epoch 6 batch 40 - loss 0.13549210131168365
Test: Epoch 6 batch 0 loss 0.1426888257265091
Epoch 6 finished - average train loss 0.1390710455381264, average test loss
0.1416544049978256
Training: Epoch 7 batch 0 - loss 0.13467301428318024
Training: Epoch 7 batch 20 - loss 0.13653507828712463
Training: Epoch 7 batch 40 - loss 0.13422717154026031
Test: Epoch 7 batch 0 loss 0.1363920420408249
Epoch 7 finished - average train loss 0.1344745836015475, average test loss
0.1353606879711151
```

```
Training: Epoch 8 batch 0 - loss 0.12869319319725037
Training: Epoch 8 batch 20 - loss 0.12171107530593872
Training: Epoch 8 batch 40 - loss 0.12050461024045944
Test: Epoch 8 batch 0 loss 0.1164807602763176
Epoch 8 finished - average train loss 0.12128467150663926, average test loss
0.11584777384996414
Training: Epoch 9 batch 0 - loss 0.11493651568889618
Training: Epoch 9 batch 20 - loss 0.11140702664852142
Training: Epoch 9 batch 40 - loss 0.11507546156644821
Test: Epoch 9 batch 0 loss 0.11392009258270264
Epoch 9 finished - average train loss 0.11368163788722734, average test loss
0.11338231489062309
Training: Epoch 10 batch 0 - loss 0.11618899554014206
Training: Epoch 10 batch 20 - loss 0.11657048016786575
Training: Epoch 10 batch 40 - loss 0.11569216102361679
Test: Epoch 10 batch 0 loss 0.11510530859231949
Epoch 10 finished - average train loss 0.11605353186191139, average test loss
0.11463154032826424
Training: Epoch 11 batch 0 - loss 0.11789952218532562
Training: Epoch 11 batch 20 - loss 0.11462733894586563
Training: Epoch 11 batch 40 - loss 0.11028798669576645
Test: Epoch 11 batch 0 loss 0.11055335402488708
Epoch 11 finished - average train loss 0.1132542059078055, average test loss
0.1106224425137043
Training: Epoch 12 batch 0 - loss 0.109492726624012
Training: Epoch 12 batch 20 - loss 0.10828885436058044
Training: Epoch 12 batch 40 - loss 0.10915444046258926
Test: Epoch 12 batch 0 loss 0.1097826436161995
Epoch 12 finished - average train loss 0.10942580765586789, average test loss
0.1098979465663433
Training: Epoch 13 batch 0 - loss 0.10987796634435654
Training: Epoch 13 batch 20 - loss 0.10837055742740631
Training: Epoch 13 batch 40 - loss 0.1076211929321289
Test: Epoch 13 batch 0 loss 0.10726995021104813
Epoch 13 finished - average train loss 0.10720554335137546, average test loss
0.10741868019104003
Training: Epoch 14 batch 0 - loss 0.1049274131655693
Training: Epoch 14 batch 20 - loss 0.1054273247718811
Training: Epoch 14 batch 40 - loss 0.10391013324260712
Test: Epoch 14 batch 0 loss 0.10466161370277405
Epoch 14 finished - average train loss 0.10471335141840628, average test loss
0.10472820028662681
Training: Epoch 15 batch 0 - loss 0.10422267019748688
Training: Epoch 15 batch 20 - loss 0.10467132925987244
Training: Epoch 15 batch 40 - loss 0.10411117225885391
Test: Epoch 15 batch 0 loss 0.1045895591378212
Epoch 15 finished - average train loss 0.10441087287361339, average test loss
0.10446633547544479
```

```
Training: Epoch 16 batch 0 - loss 0.10514512658119202
Training: Epoch 16 batch 20 - loss 0.10193284600973129
Training: Epoch 16 batch 40 - loss 0.10394955426454544
Test: Epoch 16 batch 0 loss 0.10486168414354324
Epoch 16 finished - average train loss 0.10381966639878386, average test loss
0.10514436438679695
Training: Epoch 17 batch 0 - loss 0.10477977991104126
Training: Epoch 17 batch 20 - loss 0.10611610114574432
Training: Epoch 17 batch 40 - loss 0.10511588305234909
Test: Epoch 17 batch 0 loss 0.1050676703453064
Epoch 17 finished - average train loss 0.1046742166755563, average test loss
0.1057390384376049
Training: Epoch 18 batch 0 - loss 0.10390881448984146
Training: Epoch 18 batch 20 - loss 0.10164777934551239
Training: Epoch 18 batch 40 - loss 0.10072976350784302
Test: Epoch 18 batch 0 loss 0.10410483181476593
Epoch 18 finished - average train loss 0.10276390081745083, average test loss
0.1038704313337803
Training: Epoch 19 batch 0 - loss 0.10397558659315109
Training: Epoch 19 batch 20 - loss 0.10580790042877197
Training: Epoch 19 batch 40 - loss 0.10557638108730316
Test: Epoch 19 batch 0 loss 0.10402453690767288
Epoch 19 finished - average train loss 0.10531063302088592, average test loss
0.1035535104572773
Training: Epoch 20 batch 0 - loss 0.10446595400571823
Training: Epoch 20 batch 20 - loss 0.10018998384475708
Training: Epoch 20 batch 40 - loss 0.10007140040397644
Test: Epoch 20 batch 0 loss 0.09790080785751343
Epoch 20 finished - average train loss 0.10002775401887247, average test loss
0.09840730354189872
Training: Epoch 21 batch 0 - loss 0.09668239206075668
Training: Epoch 21 batch 20 - loss 0.10204508155584335
Training: Epoch 21 batch 40 - loss 0.09947426617145538
Test: Epoch 21 batch 0 loss 0.09800439327955246
Epoch 21 finished - average train loss 0.09889460948564238, average test loss
0.09834456443786621
Training: Epoch 22 batch 0 - loss 0.09791015088558197
Training: Epoch 22 batch 20 - loss 0.09620548784732819
Training: Epoch 22 batch 40 - loss 0.09634800255298615
Test: Epoch 22 batch 0 loss 0.09812179207801819
Epoch 22 finished - average train loss 0.09687149208986152, average test loss
0.09782659560441971
Training: Epoch 23 batch 0 - loss 0.09793732315301895
Training: Epoch 23 batch 20 - loss 0.09892825037240982
Training: Epoch 23 batch 40 - loss 0.09935104846954346
Test: Epoch 23 batch 0 loss 0.09817545861005783
Epoch 23 finished - average train loss 0.09853253463062189, average test loss
0.09781977087259293
```

```
Training: Epoch 24 batch 0 - loss 0.09767583757638931
Training: Epoch 24 batch 20 - loss 0.09640657901763916
Training: Epoch 24 batch 40 - loss 0.09565369784832001
Test: Epoch 24 batch 0 loss 0.10063514113426208
Epoch 24 finished - average train loss 0.0973279729990636, average test loss
0.10029345899820327
Training: Epoch 25 batch 0 - loss 0.09735416620969772
Training: Epoch 25 batch 20 - loss 0.12745942175388336
Training: Epoch 25 batch 40 - loss 0.12764322757720947
Test: Epoch 25 batch 0 loss 0.09820715337991714
Epoch 25 finished - average train loss 0.12034376584372278, average test loss
0.09801411405205726
Training: Epoch 26 batch 0 - loss 0.09704458713531494
Training: Epoch 26 batch 20 - loss 0.09619958698749542
Training: Epoch 26 batch 40 - loss 0.09628253430128098
Test: Epoch 26 batch 0 loss 0.09825792163610458
Epoch 26 finished - average train loss 0.09629689371686871, average test loss
0.09797044917941093
Training: Epoch 27 batch 0 - loss 0.09648235142230988
Training: Epoch 27 batch 20 - loss 0.09925365447998047
Training: Epoch 27 batch 40 - loss 0.10211771726608276
Test: Epoch 27 batch 0 loss 0.10179444402456284
Epoch 27 finished - average train loss 0.10075451117956032, average test loss
0.1012374371290207
Training: Epoch 28 batch 0 - loss 0.10162962973117828
Training: Epoch 28 batch 20 - loss 0.10248631238937378
Training: Epoch 28 batch 40 - loss 0.10211378335952759
Test: Epoch 28 batch 0 loss 0.09956511855125427
Epoch 28 finished - average train loss 0.10062064268326355, average test loss
0.09935281574726104
Training: Epoch 29 batch 0 - loss 0.09721613675355911
Training: Epoch 29 batch 20 - loss 0.09993073344230652
Training: Epoch 29 batch 40 - loss 0.09966491907835007
Test: Epoch 29 batch 0 loss 0.09885279834270477
Epoch 29 finished - average train loss 0.09903087820542061, average test loss
0.09868963062763214
Training: Epoch 30 batch 0 - loss 0.09846356511116028
Training: Epoch 30 batch 20 - loss 0.09863436967134476
Training: Epoch 30 batch 40 - loss 0.09660438448190689
Test: Epoch 30 batch 0 loss 0.09694404900074005
Epoch 30 finished - average train loss 0.09756204927876844, average test loss
0.09684852734208108
Training: Epoch 31 batch 0 - loss 0.09696362167596817
Training: Epoch 31 batch 20 - loss 0.09686282277107239
Training: Epoch 31 batch 40 - loss 0.09796374291181564
Test: Epoch 31 batch 0 loss 0.09584121406078339
Epoch 31 finished - average train loss 0.09628970011816186, average test loss
0.09566466212272644
```

```
Training: Epoch 32 batch 0 - loss 0.09545659273862839
Training: Epoch 32 batch 20 - loss 0.09567335993051529
Training: Epoch 32 batch 40 - loss 0.0935555174946785
Test: Epoch 32 batch 0 loss 0.09500280767679214
Epoch 32 finished - average train loss 0.0953685723371425, average test loss
0.09488878920674323
Training: Epoch 33 batch 0 - loss 0.09536005556583405
Training: Epoch 33 batch 20 - loss 0.09432274848222733
Training: Epoch 33 batch 40 - loss 0.09282709658145905
Test: Epoch 33 batch 0 loss 0.09461606293916702
Epoch 33 finished - average train loss 0.09444944434246774, average test loss
0.09468849077820778
Training: Epoch 34 batch 0 - loss 0.0934622511267662
Training: Epoch 34 batch 20 - loss 0.09465780854225159
Training: Epoch 34 batch 40 - loss 0.09434635937213898
Test: Epoch 34 batch 0 loss 0.0954214334487915
Epoch 34 finished - average train loss 0.09458328998189862, average test loss
0.09560337886214257
Training: Epoch 35 batch 0 - loss 0.09390491992235184
Training: Epoch 35 batch 20 - loss 0.09593982249498367
Training: Epoch 35 batch 40 - loss 0.09475421160459518
Test: Epoch 35 batch 0 loss 0.09580021351575851
Epoch 35 finished - average train loss 0.09527153231329837, average test loss
0.09600725248456002
Training: Epoch 36 batch 0 - loss 0.0963989868760109
Training: Epoch 36 batch 20 - loss 0.09440147876739502
Training: Epoch 36 batch 40 - loss 0.09522144496440887
Test: Epoch 36 batch 0 loss 0.09579554200172424
Epoch 36 finished - average train loss 0.09533228838847856, average test loss
0.09606835320591926
Training: Epoch 37 batch 0 - loss 0.09472830593585968
Training: Epoch 37 batch 20 - loss 0.09508220851421356
Training: Epoch 37 batch 40 - loss 0.09575612843036652
Test: Epoch 37 batch 0 loss 0.09585006535053253
Epoch 37 finished - average train loss 0.0954828105740628, average test loss
0.09619750902056694
Training: Epoch 38 batch 0 - loss 0.09589995443820953
Training: Epoch 38 batch 20 - loss 0.09606580436229706
Training: Epoch 38 batch 40 - loss 0.09313579648733139
Test: Epoch 38 batch 0 loss 0.09501387178897858
Epoch 38 finished - average train loss 0.09510596298565299, average test loss
0.09535959139466285
Training: Epoch 39 batch 0 - loss 0.09505201876163483
Training: Epoch 39 batch 20 - loss 0.09335309267044067
Training: Epoch 39 batch 40 - loss 0.09613507241010666
Test: Epoch 39 batch 0 loss 0.09395071119070053
Epoch 39 finished - average train loss 0.09412857599682727, average test loss
0.09425866529345513
```

```
Training: Epoch 40 batch 0 - loss 0.09449775516986847
Training: Epoch 40 batch 20 - loss 0.09250371158123016
Training: Epoch 40 batch 40 - loss 0.09448319673538208
Test: Epoch 40 batch 0 loss 0.09388478100299835
Epoch 40 finished - average train loss 0.09371074604786049, average test loss
0.09418363198637962
Training: Epoch 41 batch 0 - loss 0.09663590788841248
Training: Epoch 41 batch 20 - loss 0.09768622368574142
Training: Epoch 41 batch 40 - loss 0.09485302120447159
Test: Epoch 41 batch 0 loss 0.09551290422677994
Epoch 41 finished - average train loss 0.09480705756252095, average test loss
0.09570447206497193
Training: Epoch 42 batch 0 - loss 0.09647510200738907
Training: Epoch 42 batch 20 - loss 0.09504137188196182
Training: Epoch 42 batch 40 - loss 0.09597695618867874
Test: Epoch 42 batch 0 loss 0.09808594733476639
Epoch 42 finished - average train loss 0.09687906200602903, average test loss
0.09819813743233681
Training: Epoch 43 batch 0 - loss 0.10084281861782074
Training: Epoch 43 batch 20 - loss 0.10199412703514099
Training: Epoch 43 batch 40 - loss 0.09803386777639389
Test: Epoch 43 batch 0 loss 0.098978191614151
Epoch 43 finished - average train loss 0.09898704290390015, average test loss
0.09900245815515518
Training: Epoch 44 batch 0 - loss 0.0995921716094017
Training: Epoch 44 batch 20 - loss 0.0993729829788208
Training: Epoch 44 batch 40 - loss 0.1004076674580574
Test: Epoch 44 batch 0 loss 0.09869487583637238
Epoch 44 finished - average train loss 0.09924743930667133, average test loss
0.09873597100377082
Training: Epoch 45 batch 0 - loss 0.09930220246315002
Training: Epoch 45 batch 20 - loss 0.09656540304422379
Training: Epoch 45 batch 40 - loss 0.09864326566457748
Test: Epoch 45 batch 0 loss 0.09620997309684753
Epoch 45 finished - average train loss 0.09714691707138289, average test loss
0.09639898985624314
Training: Epoch 46 batch 0 - loss 0.09557969868183136
Training: Epoch 46 batch 20 - loss 0.09750839322805405
Training: Epoch 46 batch 40 - loss 0.09351161122322083
Test: Epoch 46 batch 0 loss 0.09636478126049042
Epoch 46 finished - average train loss 0.09567569423530062, average test loss
0.09655911549925804
Training: Epoch 47 batch 0 - loss 0.09510691463947296
Training: Epoch 47 batch 20 - loss 0.09780419617891312
Training: Epoch 47 batch 40 - loss 0.10113314539194107
Test: Epoch 47 batch 0 loss 0.11261187493801117
Epoch 47 finished - average train loss 0.0995436770431066, average test loss
0.11313542276620865
```

```
Training: Epoch 48 batch 0 - loss 0.1116500124335289
Training: Epoch 48 batch 20 - loss 0.13702082633972168
Training: Epoch 48 batch 40 - loss 0.15179632604122162
Test: Epoch 48 batch 0 loss 0.15388794243335724
Epoch 48 finished - average train loss 0.13885430511781724, average test loss
0.15440610498189927
Training: Epoch 49 batch 0 - loss 0.15091174840927124
Training: Epoch 49 batch 20 - loss 0.15236112475395203
Training: Epoch 49 batch 40 - loss 0.11062931269407272
Test: Epoch 49 batch 0 loss 0.10247856378555298
Epoch 49 finished - average train loss 0.13321462172572895, average test loss
0.10279674902558326
```

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

[ ]:
```
model.eval()
with torch.no_grad():
    latent = []
    for batch, (x, _) in enumerate(test_loader):
        latent.append(model.encode(x.to(device)).cpu())
    latent = torch.cat(latent)
```

2.2 PCA and t-SNE (nothing to do here)

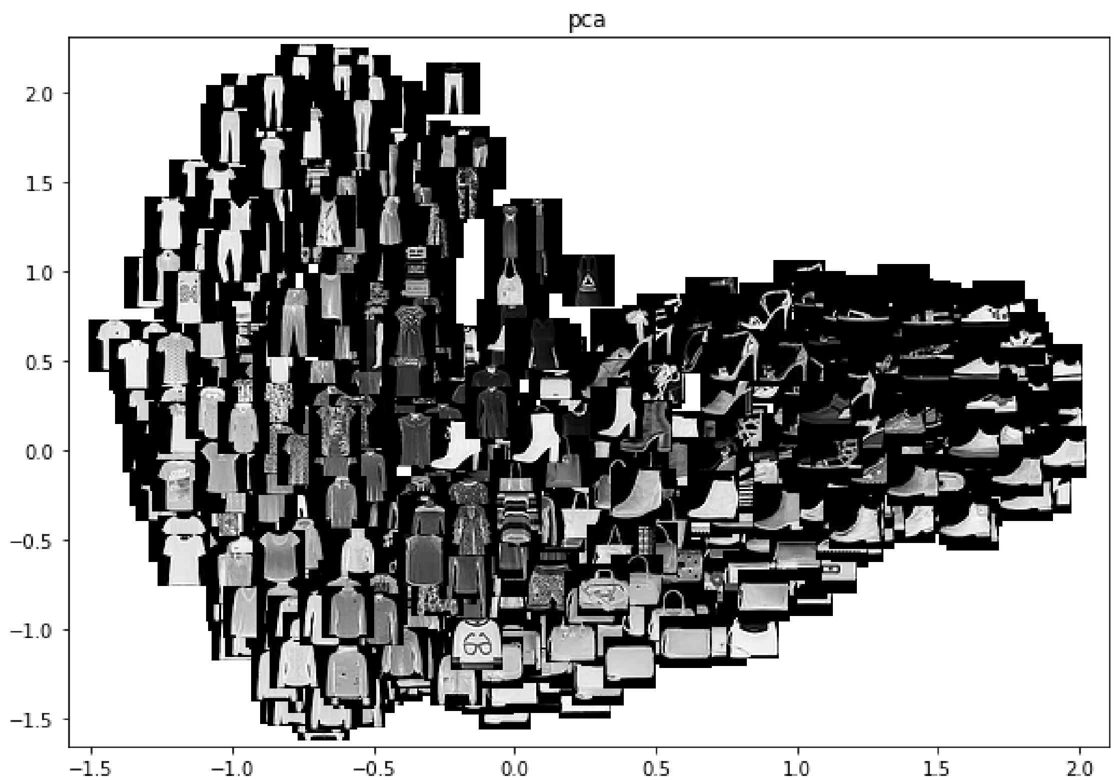
Next, we are going to look at some random images and their embeddings. Since 7x7 is still too large to visialize further dimensionality reduction techniques are required.

It is not uncommand that a neural network designer wants to understand whats going on in the latent space and therefore uses techniques such as t-SNE.

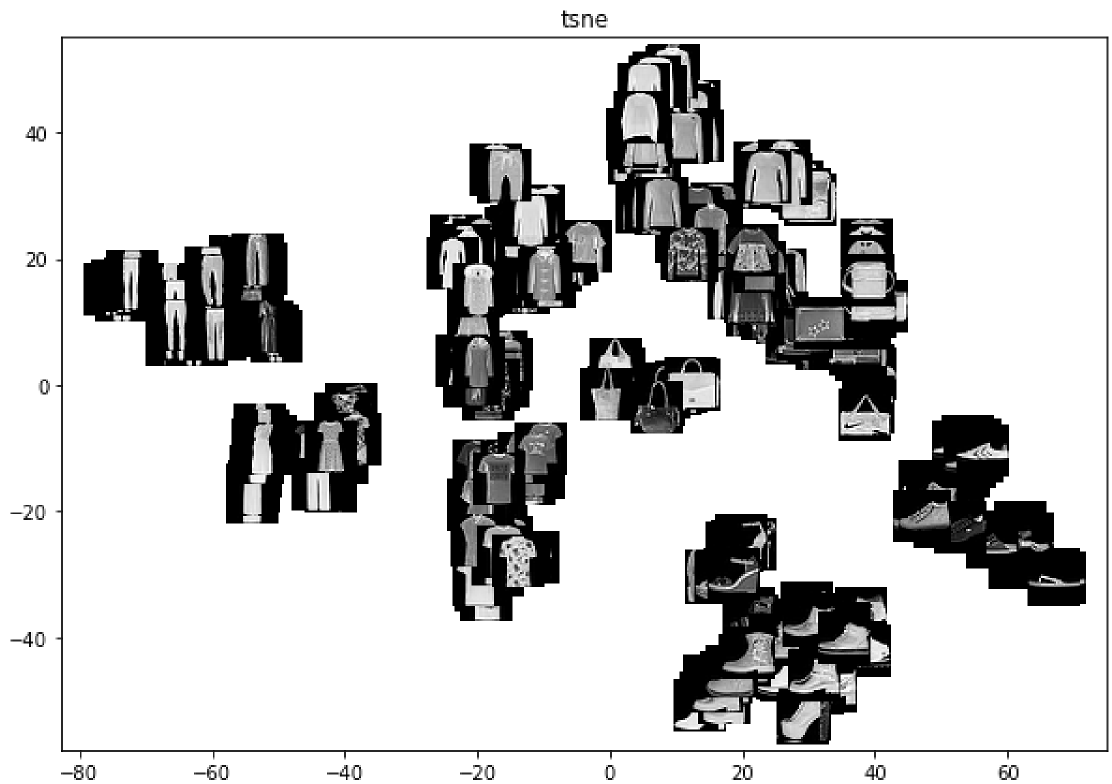
[ ]:
```
def plot_latent(test_dataset: torch.utils.data.Dataset, z_test: torch.Tensor,␣
↪count: int,
                technique: str, perplexity: float = 30):
    """
    Fit t-SNE or PCA and plots the latent space. Moreover, we then display the␣
↪correspeondig image.

    Parameters
    ----------
    test_dataset  : torch.utils.data.DataSet
                    Dataset containing raw images to display.
    z_test        : torch.Tensor
                    The transformed images.
    count         : int
                    Number of random images to sample
    technique     : str
                    Either "pca" or "tsne". Otherwise, a ValueError is thrown.
    perplexity    : float, optional, default: 30.0
```

```
                    Perplexity is t-SNE is used.

    """
    indices = np.random.choice(len(z_test), count, replace=False)
    inputs = z_test[indices]
    fig, ax = plt.subplots(figsize=(10, 7))
    ax.set_title(technique)
    if technique == 'pca':
        coords = PCA(n_components=2).fit_transform(inputs.reshape(count, -1))
    elif technique == 'tsne':
        coords = TSNE(n_components=2, perplexity=perplexity).
 ↪fit_transform(inputs.reshape(count, -1))
    else:
        raise ValueError()

    for idx, (x, y) in zip(indices, coords):
        im = OffsetImage(test_dataset[idx][0].squeeze().numpy(), zoom=1,␣
 ↪cmap='gray')
        ab = AnnotationBbox(im, (x, y), xycoords='data', frameon=False)
        ax.add_artist(ab)
    ax.update_datalim(coords)
    ax.autoscale()
    plt.show()
```

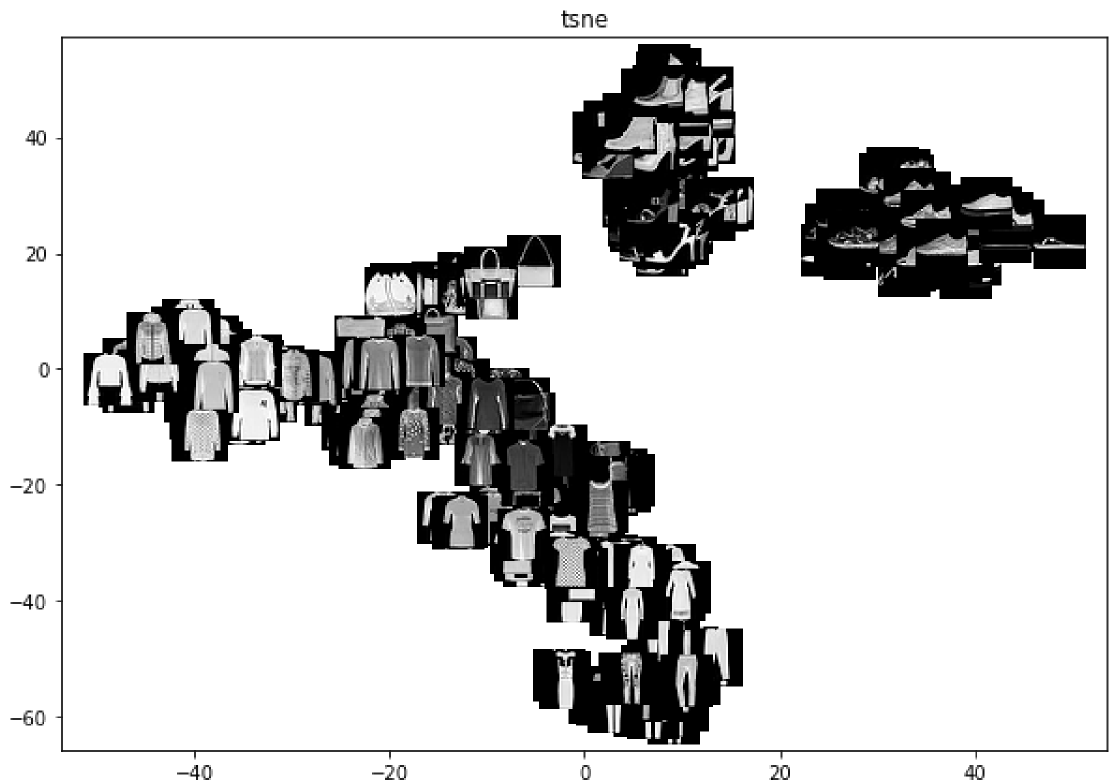
[ ]:
```
plot_latent(test_dataset, latent, 1000, 'pca')
```



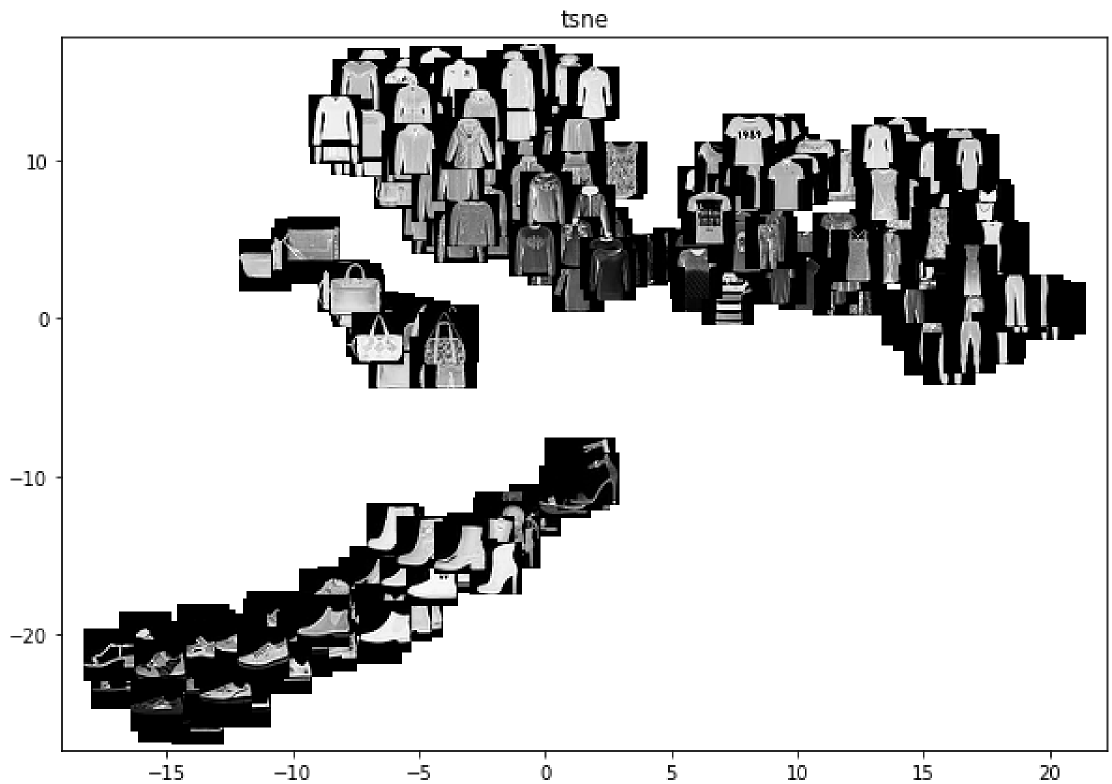
[ ]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=5)
```



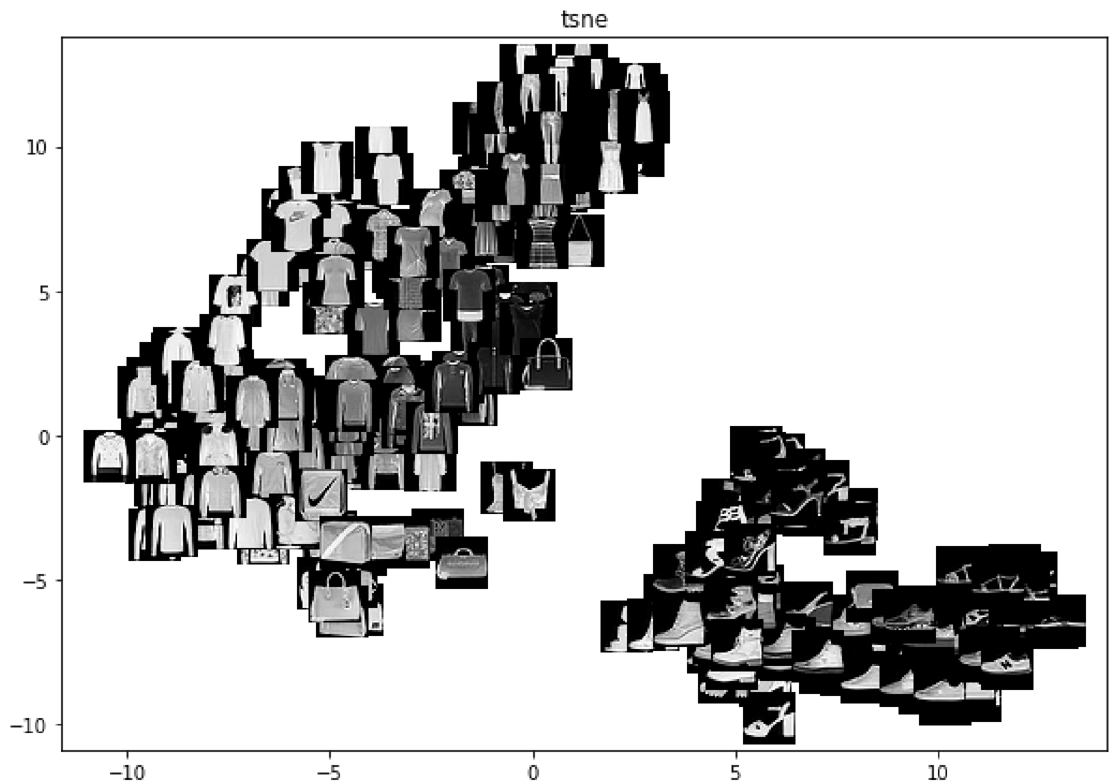

```
[ ]: plot_latent(test_dataset, latent, 300, 'tsne', perplexity=10)
```



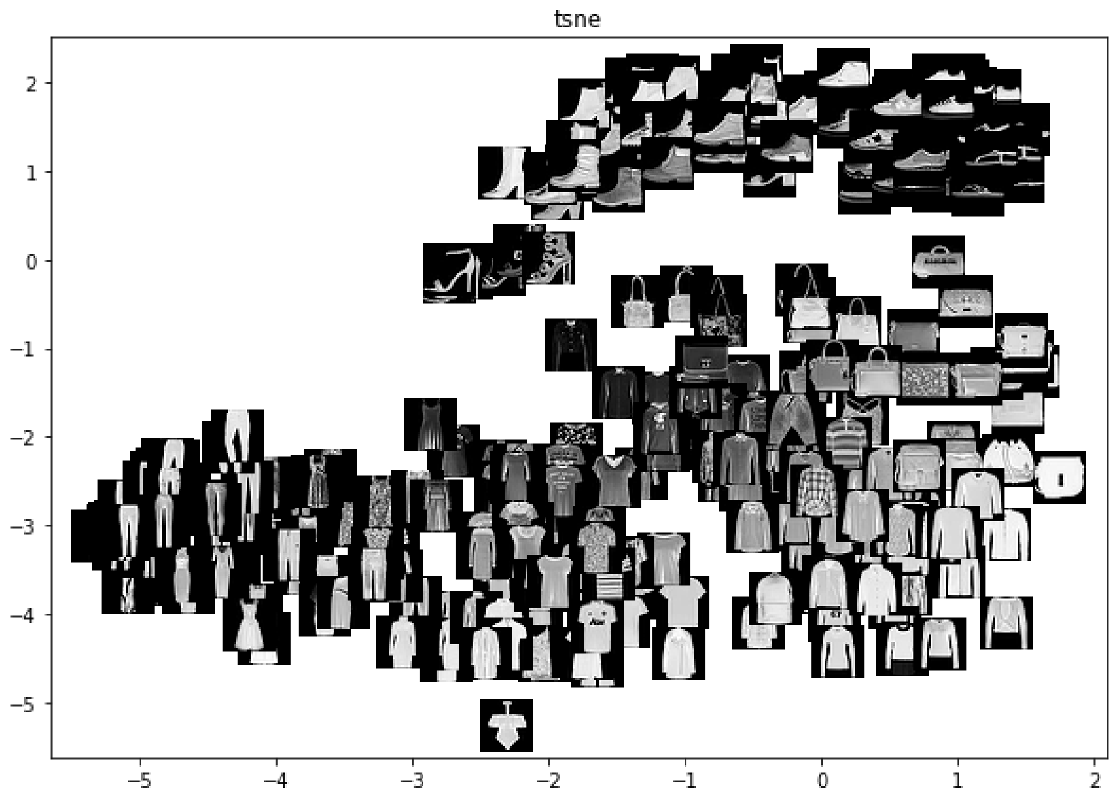
[ ]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=30)
```




```
[ ]: plot_latent(test_dataset, latent, 300, 'tsne', perplexity=50)
```




```
[ ]: plot_latent(test_dataset, latent, 300, 'tsne', perplexity=150)
```



2.3 Task 6: Linear Interpolation on the latent space

If the latent space has learned something meanigfull, we can leverage this for further analysis/downstream tasks. Anyways, we were wondering all along how the interpolation between a shoe and a pullover might look like.

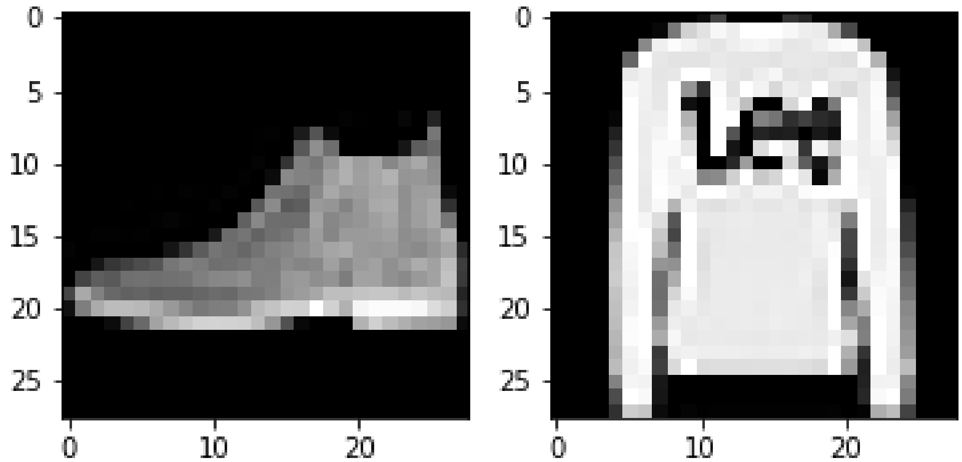
For this we encode two images $z_i = f_{enc}(x_i)$ and $z_j = f_{enc}(x_j)$. Then we linearly interpolate k equidistant locations on the line between z_i and z_j. Those locations are then be decooded by the decoder network $f_{dec}(\dots)$.

[ ]:
```
def interpolate_between(model: Autoencoder, test_dataset: torch.utils.data.
↪Dataset, idx_i: int, idx_j: int, n = 12):
    """
    Plot original images and the reconstruction of the linear interpolation in␣
↪the respective latent space embedding.

    Parameters
    ----------
    model          : Autoencoder
                     The (trained) autoencoder.
    test_dataset   : torch.utils.data.Dataset
                     Test images.
```

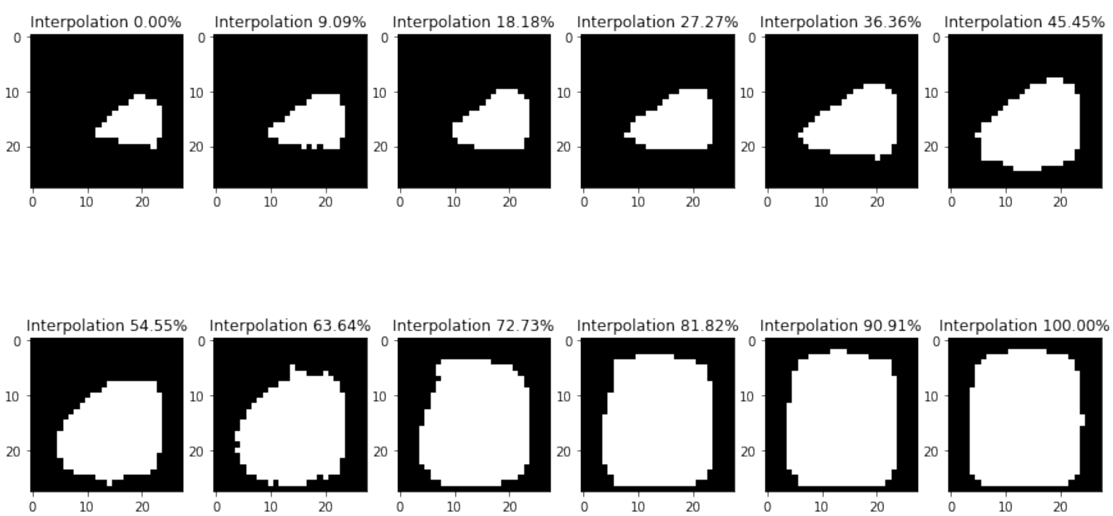
```
    idx_i          : int
                     Id for first image.
    idx_j          : int
                     Id for second image.
    n              : n, optional, default: 1
                     Number of intermediate interoplations (including original␣
↪reconstructions).

    """
    fig, ax = plt.subplots(1, 2, figsize=[6, 4])
    fig.suptitle("Original images")
    ax[0].imshow(test_dataset[idx_i][0][0].numpy(), cmap='gray')
    ax[1].imshow(test_dataset[idx_j][0][0].numpy(), cmap='gray')

    # Get embedding
    z_i = model.encode(test_dataset[idx_i][0].to(device)[None, ...])[0]
    z_j = model.encode(test_dataset[idx_j][0].to(device)[None, ...])[0]

    fig, ax = plt.subplots(2, n//2, figsize=[15, 8])
    ax = [sub for row in ax for sub in row]
    fig.suptitle("Reconstruction after interpolation in latent space")

    with torch.no_grad():
        ## BEGIN SOLUTION

        for i in range(0, n):
            interp = i/(n-1)
            flag = (z_i*(1-interp)+z_j*interp)
            fig_interpolation = model.decode(flag.to(device)[None, ...])
            ax[i].set_title(f"Interpolation {interp*100:.2f}%")
            ax[i].imshow(fig_interpolation[0][0].numpy(), cmap='gray')

        ## END SOLUTION

    plt.show()
```

[ ]:
```
interpolate_between(model, test_dataset, 0, 1)
```

Original images

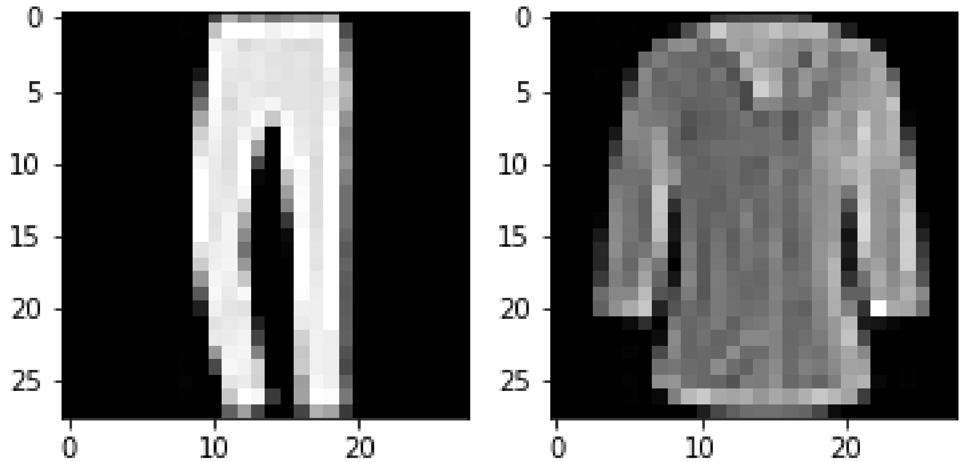


Reconstruction after interpolation in latent space

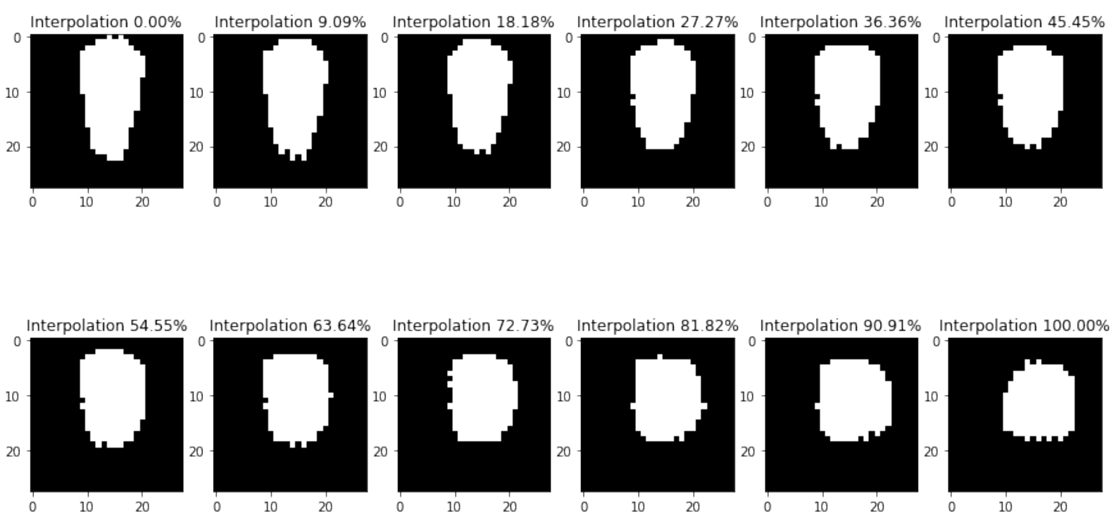



```
[ ]: interpolate_between(model, test_dataset, 2, 4)
```

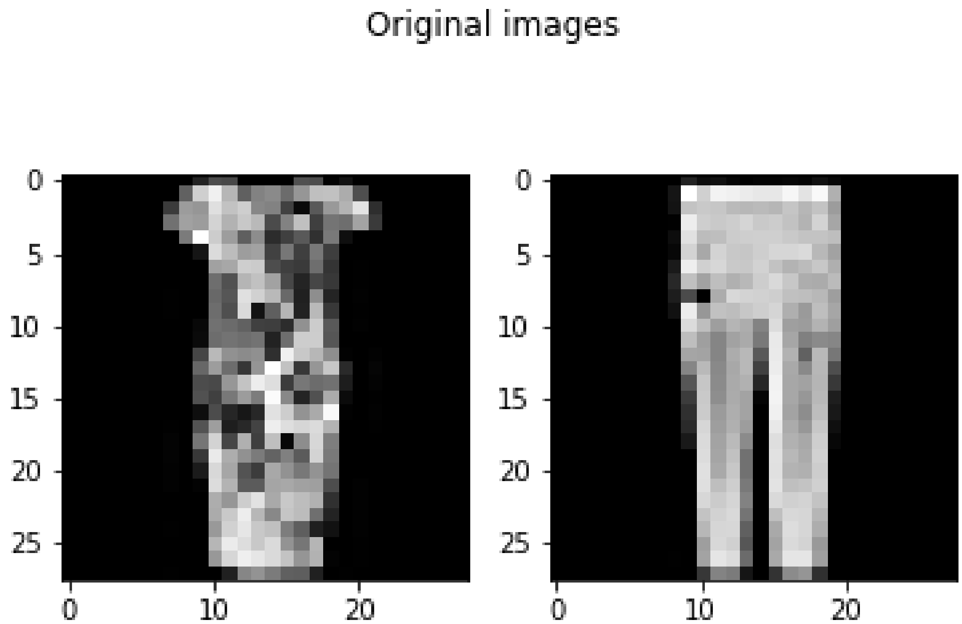
Original images

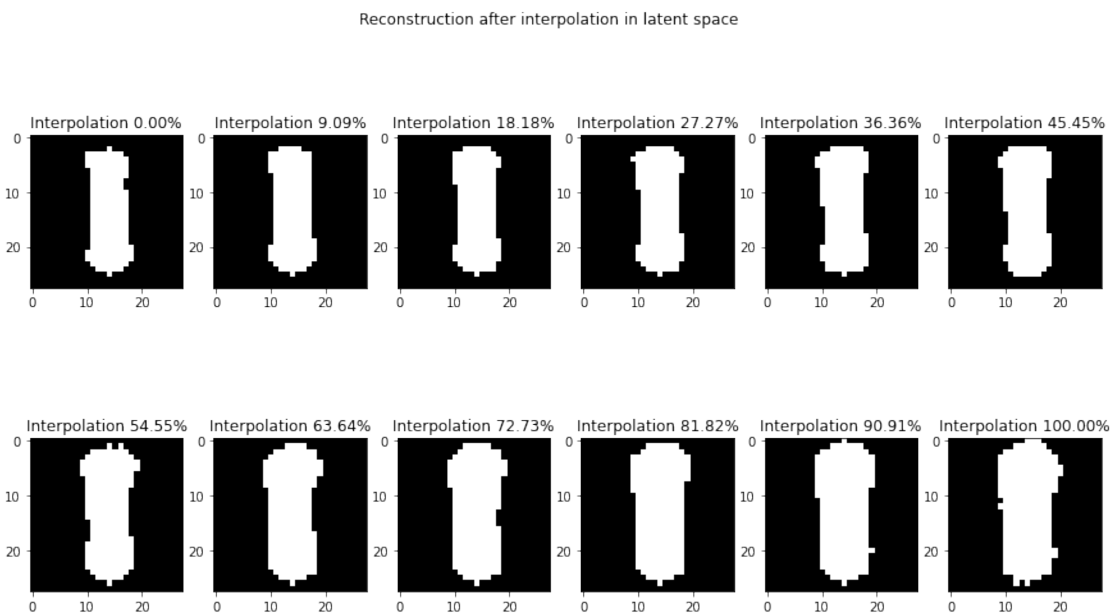


Reconstruction after interpolation in latent space



[ ]:
```
interpolate_between(model, test_dataset, 100, 200)
```


Original images


Reconstruction after interpolation in latent space
Interpolation 0.00%
Interpolation 9.09%
Interpolation 18.18%
Interpolation 27.27%
Interpolation 36.36%
Interpolation 45.45%
Interpolation 54.55%
Interpolation 63.64%
Interpolation 72.73%
Interpolation 81.82%
Interpolation 90.91%
Interpolation 100.00%