exercise_11_notebook

January 17, 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

[1]:
```
import time
import scipy.sparse as sp
import numpy as np
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)

[2]:
```
ratings = np.load("exercise_11_matrix_factorization_ratings.npy")
```

[3]:
```
# We have triplets of (user, restaurant, rating).
ratings
```

[3]:
```
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.

[4]:
```
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
```

[4]:
```
<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.

[5]:
```
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

[6]:
```
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.

    """

    # TODO: Compute the modified matrix and user_means

    ## BEGIN SOLUTION
    N, D = matrix.shape
    markers = matrix != 0
    user_means = matrix.sum(1) / markers.sum(1)
    nonz = matrix.nonzero()
    matrix = matrix.asfptype()
    matrix[nonz] -= np.ravel(user_means)[nonz[0]]

    ## 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)

[7]:
```
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
```

[8]:
```
R = cold_start_preprocessing(R, 20)
```

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

[9]:
```
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)
```

[10]:
```
# 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)

[11]:
```
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

[12]:
```
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)

    # TODO: Compute Q and P

    ## BEGIN SOLUTION
    if init=='random':
        Q = np.random.rand(matrix.shape[0], k)
        P = np.random.rand(k, matrix.shape[1])
    elif init=='svd':
        matrix = matrix.asfptype()

        U, S, V = svds(matrix, k=k)
        Q = U @ np.diagflat(S)
        P = V


    ## 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

[13]:
```
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.
```

```
    """

    # TODO: Compute best_Q, best_P, validation_losses, train_losses and converged_after


    ## BEGIN SOLUTION

    N, D = R.shape
    ridge_model = Ridge(alpha=reg_lambda,fit_intercept=False)
    idx = tuple(nonzero_indices.T)


    def get_indices_where_col(col_num):
        a = np.argwhere(idx[1]==col_num)
        ii = (np.ravel(idx[0][a]), np.ravel(idx[1][a]))
        return ii

    def get_indices_where_row(row_num):
        a = np.argwhere(idx[0]==row_num)
        ii = (np.ravel(idx[0][a]), np.ravel(idx[1][a]))
        return ii

    best_Q = None
    best_P = None
    validation_losses = []
    train_losses = []


    current_patience = None
    best_val = None

    #[N,k] , [k, D]
    Q, P = initialize_Q_P(R, k, init)
    converged_after = None

    #traininig iteration
    for it in range(max_steps):
        if optimizer == 'sgd':
            for el_idx in range(len(idx[0])):
                u = idx[0][el_idx]
                i = idx[1][el_idx]
                q_u = Q[u, :]
                p_i = P[:, i]
                rat = R[u, i]
                pred = q_u.dot(p_i)
                e_ui = rat - pred
```

```
            if np.isnan(e_ui):
                print('nan val!!')
                print(it)
                print(el_idx)
                print('rating {}, pred {}'.format(rat, pred))
                print('________')
            P[:, i] = p_i + lr*e_ui*q_u - lr * reg_lambda * p_i
            p_i_n = P[:, i]
            Q[u, :] = q_u + lr*e_ui*p_i_n - lr * reg_lambda * q_u

    else:
        #optimizing P
        for i in range(D):
            valued_rows, current_col = get_indices_where_col(i)
            Q_sel = Q[valued_rows, :]
            ridge_model.fit(Q_sel, np.ravel(R[(valued_rows, current_col)]))
            P[:, i] = ridge_model.coef_
        #optimizing Q
        for u in range(N):
            current_row, valued_cols = get_indices_where_row(u)
            P_sel = P[:, valued_cols].T
            ridge_model.fit(P_sel, np.ravel(R[(current_row, valued_cols)]))
            Q[u, :] = ridge_model.coef_



    if it % eval_every == 0:
        #calculate losses

        tr_loss = loss(np.ravel(R[idx]), idx, Q, P, reg_lambda)
        val_loss = loss(val_values, val_idx, Q, P, reg_lambda)
        validation_losses.append(val_loss)
        train_losses.append(tr_loss)

        if best_val is None or val_loss < best_val:
            best_val = val_loss
            best_Q = Q
            best_P = P
            current_patience = 0
        else:
            current_patience += 1
            if patience - current_patience == 0:
                converged_after = it - patience * eval_every
                break


    if it % log_every == 0:
```

```
            print("Iteration {}, training loss: {}, validation loss: {}".
↪format(it, tr_loss, val_loss))

    if converged_after is None:
        converged_after = -1


    ## END SOLUTION

    return best_Q, best_P, validation_losses, train_losses, converged_after
```

1.2.3 Train the latent factor (nothing to do here)

[14]:
```
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: 311711.90806194756, validation loss:
377.59913021730375
Iteration 1, training loss: 189420.51428835813, validation loss:
347.18212499715315
Iteration 2, training loss: 135311.91373975735, validation loss:
339.0707717998836
Iteration 3, training loss: 104092.80204879717, validation loss:
338.5379207518771
Iteration 4, training loss: 83485.22617698503, validation loss:
341.2923467896907
Iteration 5, training loss: 68768.80053508203, validation loss: 345.675521454483
Iteration 6, training loss: 57725.95553067324, validation loss:
350.8936939162965
Iteration 7, training loss: 49159.225562153166, validation loss:
356.5048597722811
Iteration 8, training loss: 42352.56134188428, validation loss:
362.2406875920693
Iteration 9, training loss: 36844.76475138914, validation loss:
367.93271363154525
Iteration 10, training loss: 32322.42600952386, validation loss:
373.47544872488305
Iteration 11, training loss: 28564.042352134904, validation loss:
378.80455118419053
Iteration 12, training loss: 25408.312096772774, validation loss:
383.88264745998833
```

[15]:
```
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: 2236.3437701254443, validation loss:
1582.973273489629
Iteration 1, training loss: 510.86467120143885, validation loss:
1223.175809571189
Iteration 2, training loss: 199.14735767507023, validation loss:
1445.9220599062735
Iteration 3, training loss: 98.52769521134294, validation loss:
1410.1010034256426
Iteration 4, training loss: 56.22009901446679, validation loss:
1197.7684085794108
Iteration 5, training loss: 35.56784151007595, validation loss:
1123.9126803377553
Iteration 6, training loss: 24.516242853972862, validation loss:
1331.5232826624176
Iteration 7, training loss: 18.29418418841616, validation loss:
1264.9008000779256
Iteration 8, training loss: 14.604172678696447, validation loss:
1367.48630372723
Iteration 9, training loss: 12.312454787050935, validation loss:
1376.3103907352006
Iteration 10, training loss: 10.844051501846316, validation loss:
1373.0486051727626
Iteration 11, training loss: 9.877731359080652, validation loss:
1354.9353548813995
Iteration 12, training loss: 9.223838655819732, validation loss:
1320.9191848116936
Iteration 13, training loss: 8.771038748291991, validation loss:
1282.858279728616
Iteration 14, training loss: 8.449721644623006, validation loss:
1251.1318813597525
```

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

[16]:
```
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.

[17]:
```
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
```

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

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

[18]:
```
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()
  )
)
```

[19]:
```
class Autoencoder(nn.Module):
    ## BEGIN SOLUTION
    def __init__(self):
        super().__init__()
        self.encode = nn.Sequential(
            nn.Conv2d(1, 8, kernel_size=3),
            nn.ReLU(),
            nn.Conv2d(8, 16, kernel_size=3),
            nn.ReLU(),
            nn.BatchNorm2d(16),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(16, 32, kernel_size=3),
            nn.ReLU(),
            nn.BatchNorm2d(32),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(32, 32, kernel_size=3),
            nn.ReLU()
        )

        self.decode = nn.Sequential(
            nn.ConvTranspose2d(32, 32, kernel_size=3),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2,␣
↪output_padding=(1, 1)),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 8, kernel_size=3, stride=2,␣
↪output_padding=(1, 1)),
            nn.ReLU(),
            nn.ConvTranspose2d(8, 8, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(8, 4, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(4, 1, kernel_size=3),

            nn.Sigmoid()
        )

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

[20]:
```
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.

[21]:
```
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

for epoch in range(50):
    model.train()
    train_loss_trace = []
    for batch, (x, _) in enumerate(train_loader):
        # TODO: The autoendocer shall be trained on the mse loss
        ## BEGIN SOLUTION
        x = x.to(device)
        optimizer.zero_grad()
        x_rec = model(x)
        loss = F.mse_loss(x_rec, x)
        loss.backward()
        optimizer.step()
        #todo scheduler 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 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)}')
```

```
Training: Epoch 0 batch 0 - loss 0.21741913259029388
Training: Epoch 0 batch 20 - loss 0.20758292078971863
Training: Epoch 0 batch 40 - loss 0.17231076955795288
Test: Epoch 0 batch 0 loss 0.12655405700206757
Epoch 0 finished - average train loss 0.18630958436909367, average test loss
0.12541355490684508
Training: Epoch 1 batch 0 - loss 0.12424914538860321
Training: Epoch 1 batch 20 - loss 0.10054537653923035
Training: Epoch 1 batch 40 - loss 0.08330623805522919
Test: Epoch 1 batch 0 loss 0.07595664262771606
Epoch 1 finished - average train loss 0.09521898501000162, average test loss
0.07506146281957626
Training: Epoch 2 batch 0 - loss 0.07372911274433136
Training: Epoch 2 batch 20 - loss 0.06678086519241333
Training: Epoch 2 batch 40 - loss 0.06010272726416588
Test: Epoch 2 batch 0 loss 0.05124456062912941
Epoch 2 finished - average train loss 0.06401032575611341, average test loss
0.050722284242510796
Training: Epoch 3 batch 0 - loss 0.05112357437610626
Training: Epoch 3 batch 20 - loss 0.04135793447494507
Training: Epoch 3 batch 40 - loss 0.035728100687265396
Test: Epoch 3 batch 0 loss 0.04107818007469177
Epoch 3 finished - average train loss 0.040220767701581374, average test loss
0.04073378220200539
Training: Epoch 4 batch 0 - loss 0.033855948597192764
Training: Epoch 4 batch 20 - loss 0.03168313950300217
Training: Epoch 4 batch 40 - loss 0.028821704909205437
Test: Epoch 4 batch 0 loss 0.027387049049139023
Epoch 4 finished - average train loss 0.029968589323304467, average test loss
0.027195652201771735
Training: Epoch 5 batch 0 - loss 0.026921018958091736
Training: Epoch 5 batch 20 - loss 0.025060325860977173
Training: Epoch 5 batch 40 - loss 0.023119419813156128
Test: Epoch 5 batch 0 loss 0.0227628406137228
Epoch 5 finished - average train loss 0.024134662633730194, average test loss
0.022596355900168417
Training: Epoch 6 batch 0 - loss 0.022177137434482574
Training: Epoch 6 batch 20 - loss 0.021624205633997917
Training: Epoch 6 batch 40 - loss 0.021758079528808594
Test: Epoch 6 batch 0 loss 0.021625587716698647
```

```
Epoch 6 finished - average train loss 0.021631384078981512, average test loss
0.021430008113384247
Training: Epoch 7 batch 0 - loss 0.02069477178156376
Training: Epoch 7 batch 20 - loss 0.02028995379805565
Training: Epoch 7 batch 40 - loss 0.02039993368089199
Test: Epoch 7 batch 0 loss 0.020407086238265038
Epoch 7 finished - average train loss 0.02031078489528874, average test loss
0.020210618153214455
Training: Epoch 8 batch 0 - loss 0.019527381286025047
Training: Epoch 8 batch 20 - loss 0.02128937467932701
Training: Epoch 8 batch 40 - loss 0.019114509224891663
Test: Epoch 8 batch 0 loss 0.021221749484539032
Epoch 8 finished - average train loss 0.019478082120165986, average test loss
0.021029322966933252
Training: Epoch 9 batch 0 - loss 0.020525677129626274
Training: Epoch 9 batch 20 - loss 0.019902756437659264
Training: Epoch 9 batch 40 - loss 0.017939679324626923
Test: Epoch 9 batch 0 loss 0.01886356808245182
Epoch 9 finished - average train loss 0.018887959073408175, average test loss
0.018672334030270576
Training: Epoch 10 batch 0 - loss 0.017900794744491577
Training: Epoch 10 batch 20 - loss 0.017817053943872452
Training: Epoch 10 batch 40 - loss 0.018155131489038467
Test: Epoch 10 batch 0 loss 0.018061863258481026
Epoch 10 finished - average train loss 0.01813997978628692, average test loss
0.017872826755046846
Training: Epoch 11 batch 0 - loss 0.018146667629480362
Training: Epoch 11 batch 20 - loss 0.017956402152776718
Training: Epoch 11 batch 40 - loss 0.01691889390349388
Test: Epoch 11 batch 0 loss 0.01769210770726204
Epoch 11 finished - average train loss 0.017597110721014313, average test loss
0.01748859416693449
Training: Epoch 12 batch 0 - loss 0.017248397693037987
Training: Epoch 12 batch 20 - loss 0.01739320531487465
Training: Epoch 12 batch 40 - loss 0.017179155722260475
Test: Epoch 12 batch 0 loss 0.017322702333331108
Epoch 12 finished - average train loss 0.01727218859655372, average test loss
0.017125485464930534
Training: Epoch 13 batch 0 - loss 0.016983861103653908
Training: Epoch 13 batch 20 - loss 0.01703847199678421
Training: Epoch 13 batch 40 - loss 0.017099129036068916
Test: Epoch 13 batch 0 loss 0.016880784183740616
Epoch 13 finished - average train loss 0.016886776755169287, average test loss
0.016694404371082782
Training: Epoch 14 batch 0 - loss 0.01617017202079296
Training: Epoch 14 batch 20 - loss 0.016608769074082375
Training: Epoch 14 batch 40 - loss 0.016378503292798996
Test: Epoch 14 batch 0 loss 0.01814572699368
```

```
Epoch 14 finished - average train loss 0.016765832048604042, average test loss
0.01796536464244127
Training: Epoch 15 batch 0 - loss 0.01815171167254448
Training: Epoch 15 batch 20 - loss 0.015885422006249428
Training: Epoch 15 batch 40 - loss 0.016315940767526627
Test: Epoch 15 batch 0 loss 0.016712777316570282
Epoch 15 finished - average train loss 0.016484199817908014, average test loss
0.016513942368328573
Training: Epoch 16 batch 0 - loss 0.016287555918097496
Training: Epoch 16 batch 20 - loss 0.015768392011523247
Training: Epoch 16 batch 40 - loss 0.0156294796615839
Test: Epoch 16 batch 0 loss 0.01617983542382717
Epoch 16 finished - average train loss 0.015951740713316506, average test loss
0.015994628332555293
Training: Epoch 17 batch 0 - loss 0.016248760744929314
Training: Epoch 17 batch 20 - loss 0.016604376956820488
Training: Epoch 17 batch 40 - loss 0.015796877443790436
Test: Epoch 17 batch 0 loss 0.015891898423433304
Epoch 17 finished - average train loss 0.015783734114493353, average test loss
0.015702625922858714
Training: Epoch 18 batch 0 - loss 0.015459835529327393
Training: Epoch 18 batch 20 - loss 0.015525146387517452
Training: Epoch 18 batch 40 - loss 0.015442867763340473
Test: Epoch 18 batch 0 loss 0.015773683786392212
Epoch 18 finished - average train loss 0.01566363457527201, average test loss
0.015579796303063631
Training: Epoch 19 batch 0 - loss 0.015285849571228027
Training: Epoch 19 batch 20 - loss 0.015371238812804222
Training: Epoch 19 batch 40 - loss 0.01492735929787159
Test: Epoch 19 batch 0 loss 0.015716740861535072
Epoch 19 finished - average train loss 0.015207965195305267, average test loss
0.015547777060419321
Training: Epoch 20 batch 0 - loss 0.0153794065117836
Training: Epoch 20 batch 20 - loss 0.016584016382694244
Training: Epoch 20 batch 40 - loss 0.015331069938838482
Test: Epoch 20 batch 0 loss 0.015942810103297234
Epoch 20 finished - average train loss 0.015331654651564056, average test loss
0.015769756864756346
Training: Epoch 21 batch 0 - loss 0.015171502716839314
Training: Epoch 21 batch 20 - loss 0.014756261371076107
Training: Epoch 21 batch 40 - loss 0.01565905287861824
Test: Epoch 21 batch 0 loss 0.015291917137801647
Epoch 21 finished - average train loss 0.015184066195230363, average test loss
0.015125763416290284
Training: Epoch 22 batch 0 - loss 0.015117743983864784
Training: Epoch 22 batch 20 - loss 0.014604704454541206
Training: Epoch 22 batch 40 - loss 0.014144329354166985
Test: Epoch 22 batch 0 loss 0.015185613185167313
```

```
Epoch 22 finished - average train loss 0.014767511684636948, average test loss
0.015013848803937435
Training: Epoch 23 batch 0 - loss 0.01501588523387909
Training: Epoch 23 batch 20 - loss 0.014704450964927673
Training: Epoch 23 batch 40 - loss 0.015102923847734928
Test: Epoch 23 batch 0 loss 0.01522471010684967
Epoch 23 finished - average train loss 0.014927585861819276, average test loss
0.015046348888427019
Training: Epoch 24 batch 0 - loss 0.014661173336207867
Training: Epoch 24 batch 20 - loss 0.015241795219480991
Training: Epoch 24 batch 40 - loss 0.01408536545932293
Test: Epoch 24 batch 0 loss 0.014706019312143326
Epoch 24 finished - average train loss 0.014486243853629645, average test loss
0.014534307178109884
Training: Epoch 25 batch 0 - loss 0.014169649221003056
Training: Epoch 25 batch 20 - loss 0.014448419213294983
Training: Epoch 25 batch 40 - loss 0.014571625739336014
Test: Epoch 25 batch 0 loss 0.014617666602134705
Epoch 25 finished - average train loss 0.014568087723800692, average test loss
0.014448403008282184
Training: Epoch 26 batch 0 - loss 0.014017725363373756
Training: Epoch 26 batch 20 - loss 0.015774236992001534
Training: Epoch 26 batch 40 - loss 0.014168597757816315
Test: Epoch 26 batch 0 loss 0.014643759466707706
Epoch 26 finished - average train loss 0.014380469649903855, average test loss
0.014485487155616284
Training: Epoch 27 batch 0 - loss 0.014237545430660248
Training: Epoch 27 batch 20 - loss 0.014598982408642769
Training: Epoch 27 batch 40 - loss 0.014260836876928806
Test: Epoch 27 batch 0 loss 0.014679904095828533
Epoch 27 finished - average train loss 0.014451804002589089, average test loss
0.01451482977718115
Training: Epoch 28 batch 0 - loss 0.0144872535020113
Training: Epoch 28 batch 20 - loss 0.01424217876046896
Training: Epoch 28 batch 40 - loss 0.014297445304691792
Test: Epoch 28 batch 0 loss 0.014220429584383965
Epoch 28 finished - average train loss 0.01428222724126052, average test loss
0.014055287465453148
Training: Epoch 29 batch 0 - loss 0.013910742476582527
Training: Epoch 29 batch 20 - loss 0.013251343742012978
Training: Epoch 29 batch 40 - loss 0.013694829307496548
Test: Epoch 29 batch 0 loss 0.014583117328584194
Epoch 29 finished - average train loss 0.013993008614722954, average test loss
0.014403394982218742
Training: Epoch 30 batch 0 - loss 0.013970805332064629
Training: Epoch 30 batch 20 - loss 0.01463969424366951
Training: Epoch 30 batch 40 - loss 0.014800305478274822
Test: Epoch 30 batch 0 loss 0.014208230189979076
```

```
Epoch 30 finished - average train loss 0.014077466849427102, average test loss
0.014041206892579794
Training: Epoch 31 batch 0 - loss 0.013647401705384254
Training: Epoch 31 batch 20 - loss 0.013943706639111042
Training: Epoch 31 batch 40 - loss 0.01377937477082014
Test: Epoch 31 batch 0 loss 0.014066381379961967
Epoch 31 finished - average train loss 0.013896416825384407, average test loss
0.013889675866812468
Training: Epoch 32 batch 0 - loss 0.013540788553655148
Training: Epoch 32 batch 20 - loss 0.013570941984653473
Training: Epoch 32 batch 40 - loss 0.013743393123149872
Test: Epoch 32 batch 0 loss 0.0137792294844985
Epoch 32 finished - average train loss 0.013779716061080917, average test loss
0.013613554649055004
Training: Epoch 33 batch 0 - loss 0.013524893671274185
Training: Epoch 33 batch 20 - loss 0.013301614671945572
Training: Epoch 33 batch 40 - loss 0.013164469040930271
Test: Epoch 33 batch 0 loss 0.01388320978730917
Epoch 33 finished - average train loss 0.013675403007763928, average test loss
0.013713352382183075
Training: Epoch 34 batch 0 - loss 0.013526546768844128
Training: Epoch 34 batch 20 - loss 0.014669832773506641
Training: Epoch 34 batch 40 - loss 0.01327430922538042
Test: Epoch 34 batch 0 loss 0.013655789196491241
Epoch 34 finished - average train loss 0.013909282352207071, average test loss
0.013488291949033736
Training: Epoch 35 batch 0 - loss 0.013267402537167072
Training: Epoch 35 batch 20 - loss 0.013768922537565231
Training: Epoch 35 batch 40 - loss 0.013067063875496387
Test: Epoch 35 batch 0 loss 0.013636820949614048
Epoch 35 finished - average train loss 0.013460281415510986, average test loss
0.013479210063815117
Training: Epoch 36 batch 0 - loss 0.013860629871487617
Training: Epoch 36 batch 20 - loss 0.014098548330366611
Training: Epoch 36 batch 40 - loss 0.013346029445528984
Test: Epoch 36 batch 0 loss 0.013680893927812576
Epoch 36 finished - average train loss 0.013375382902005972, average test loss
0.013525768741965294
Training: Epoch 37 batch 0 - loss 0.013578996062278748
Training: Epoch 37 batch 20 - loss 0.01327001303434372
Training: Epoch 37 batch 40 - loss 0.013134011998772621
Test: Epoch 37 batch 0 loss 0.013353460468351841
Epoch 37 finished - average train loss 0.01334819472315958, average test loss
0.013196100853383541
Training: Epoch 38 batch 0 - loss 0.013072915375232697
Training: Epoch 38 batch 20 - loss 0.013167573139071465
Training: Epoch 38 batch 40 - loss 0.013351282104849815
Test: Epoch 38 batch 0 loss 0.014525748789310455
```

```
Epoch 38 finished - average train loss 0.013519325656658513, average test loss
0.01437969459220767
Training: Epoch 39 batch 0 - loss 0.0138158043846488
Training: Epoch 39 batch 20 - loss 0.013864445500075817
Training: Epoch 39 batch 40 - loss 0.012940951623022556
Test: Epoch 39 batch 0 loss 0.013742247596383095
Epoch 39 finished - average train loss 0.01328435800653898, average test loss
0.013559303060173989
Training: Epoch 40 batch 0 - loss 0.013260041363537312
Training: Epoch 40 batch 20 - loss 0.013399504125118256
Training: Epoch 40 batch 40 - loss 0.013626559637486935
Test: Epoch 40 batch 0 loss 0.01332086231559515
Epoch 40 finished - average train loss 0.013227991396719116, average test loss
0.013158757612109185
Training: Epoch 41 batch 0 - loss 0.013153934851288795
Training: Epoch 41 batch 20 - loss 0.01340360939502716
Training: Epoch 41 batch 40 - loss 0.013144062831997871
Test: Epoch 41 batch 0 loss 0.013057606294751167
Epoch 41 finished - average train loss 0.013171459153547125, average test loss
0.012894051801413297
Training: Epoch 42 batch 0 - loss 0.012963246554136276
Training: Epoch 42 batch 20 - loss 0.012899958528578281
Training: Epoch 42 batch 40 - loss 0.012529270723462105
Test: Epoch 42 batch 0 loss 0.01411441620439291
Epoch 42 finished - average train loss 0.012868477726134203, average test loss
0.013923975639045238
Training: Epoch 43 batch 0 - loss 0.013324226252734661
Training: Epoch 43 batch 20 - loss 0.013113335706293583
Training: Epoch 43 batch 40 - loss 0.012835823930799961
Test: Epoch 43 batch 0 loss 0.013428299687802792
Epoch 43 finished - average train loss 0.012981899319437601, average test loss
0.013247021473944187
Training: Epoch 44 batch 0 - loss 0.01296902447938919
Training: Epoch 44 batch 20 - loss 0.012646644376218319
Training: Epoch 44 batch 40 - loss 0.012765967287123203
Test: Epoch 44 batch 0 loss 0.012830900028347969
Epoch 44 finished - average train loss 0.0126929582428124, average test loss
0.012672989815473556
Training: Epoch 45 batch 0 - loss 0.012244723737239838
Training: Epoch 45 batch 20 - loss 0.012891614809632301
Training: Epoch 45 batch 40 - loss 0.012920539826154709
Test: Epoch 45 batch 0 loss 0.01283462718129158
Epoch 45 finished - average train loss 0.012834506646051244, average test loss
0.01267293319106102
Training: Epoch 46 batch 0 - loss 0.012639001943171024
Training: Epoch 46 batch 20 - loss 0.012696929275989532
Training: Epoch 46 batch 40 - loss 0.013237426057457924
Test: Epoch 46 batch 0 loss 0.012760729528963566
```

```
Epoch 46 finished - average train loss 0.012735675799392037, average test loss
0.012604620307683945
Training: Epoch 47 batch 0 - loss 0.012097585946321487
Training: Epoch 47 batch 20 - loss 0.012690001167356968
Training: Epoch 47 batch 40 - loss 0.012260207906365395
Test: Epoch 47 batch 0 loss 0.012943107634782791
Epoch 47 finished - average train loss 0.012621868480691465, average test loss
0.012780253868550062
Training: Epoch 48 batch 0 - loss 0.012523876503109932
Training: Epoch 48 batch 20 - loss 0.01259747426956892
Training: Epoch 48 batch 40 - loss 0.012891542166471481
Test: Epoch 48 batch 0 loss 0.012743325904011726
Epoch 48 finished - average train loss 0.01274415070854001, average test loss
0.012584624998271465
Training: Epoch 49 batch 0 - loss 0.012498521246016026
Training: Epoch 49 batch 20 - loss 0.012713146395981312
Training: Epoch 49 batch 40 - loss 0.013341298326849937
Test: Epoch 49 batch 0 loss 0.01267041452229023
Epoch 49 finished - average train loss 0.012781107103673079, average test loss
0.012513034325093032
```

[22]:
```python
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)

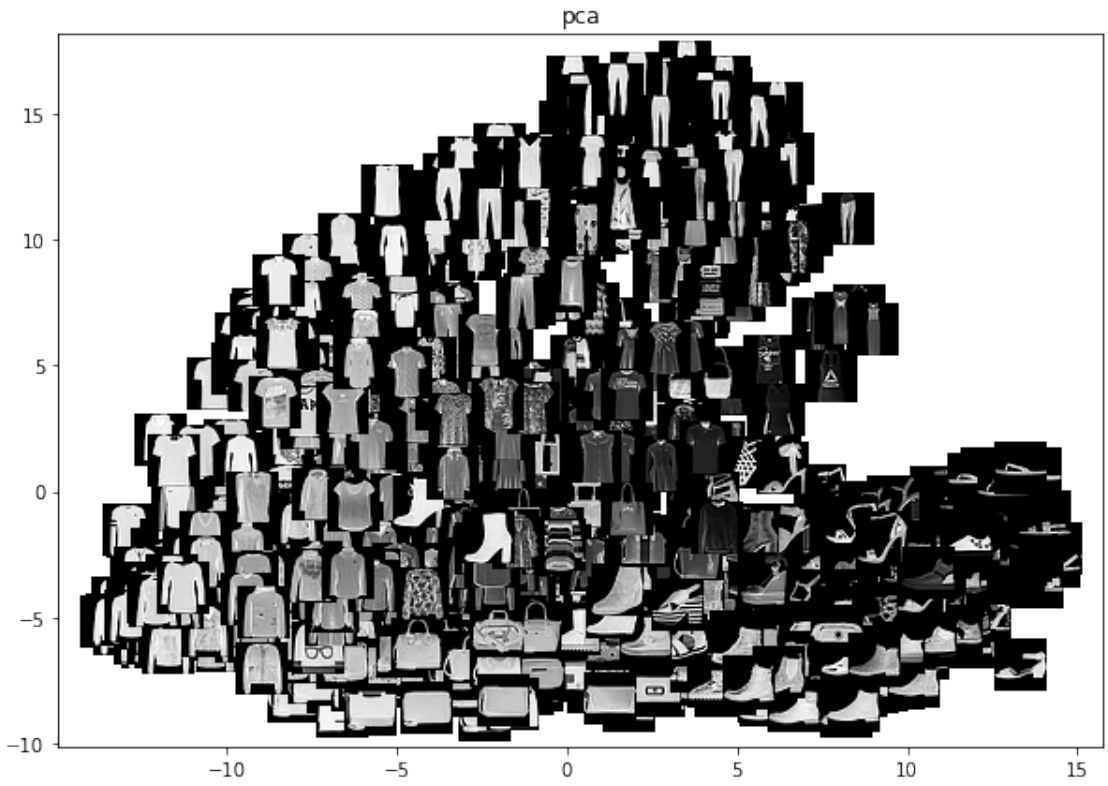
Next, we are going to look at some random images and their embeddings. Since 7x7 is still too large to visialize further dimensionality reduction techniques are required.

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.

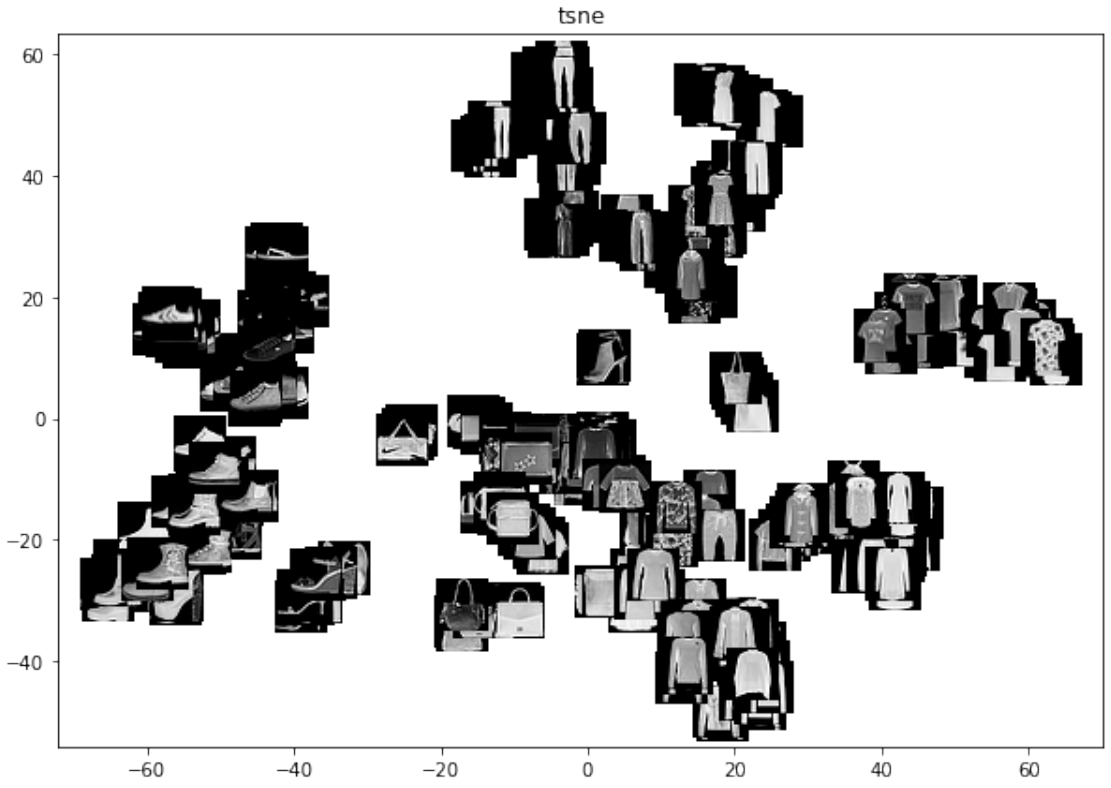
[23]:
```python
def plot_latent(test_dataset: torch.utils.data.Dataset, z_test: torch.Tensor,␣
↪count: int,
                technique: str, perplexity: float = 30):
    """
    Fit t-SNE or PCA and plots the latent space. Moreover, we then display the␣
↪correspeondig image.

    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()
```

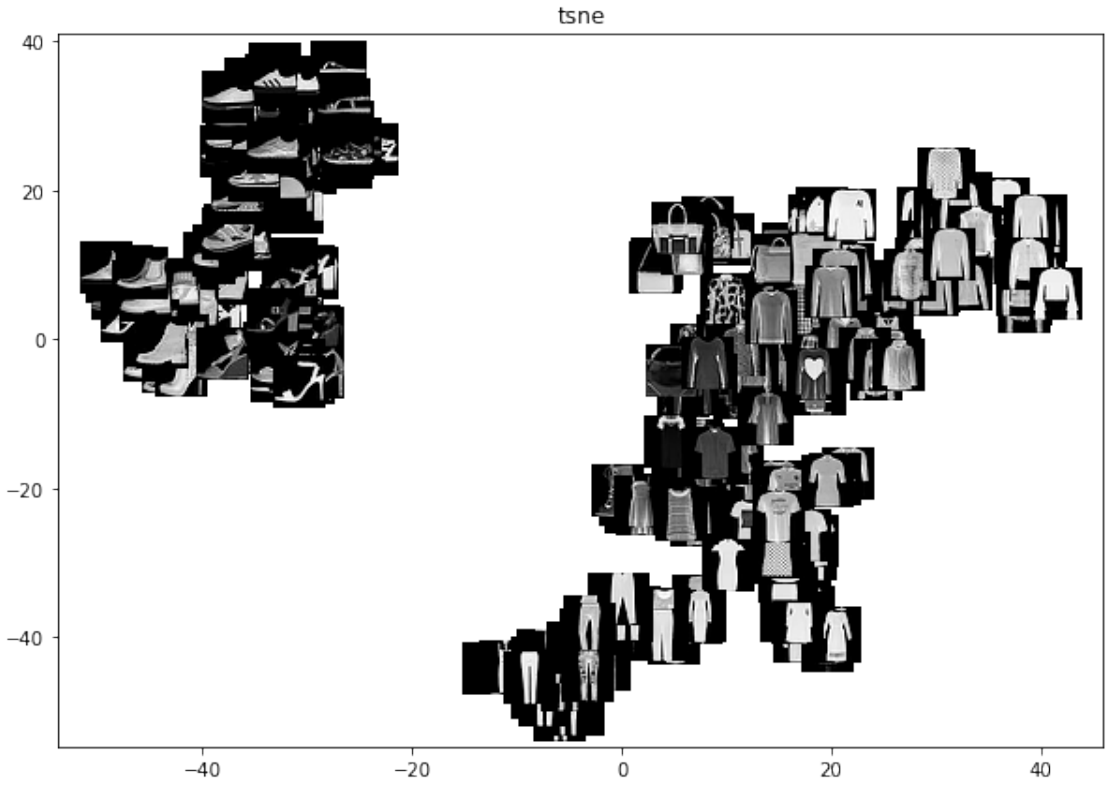
[24]:
```
plot_latent(test_dataset, latent, 1000, 'pca')
```



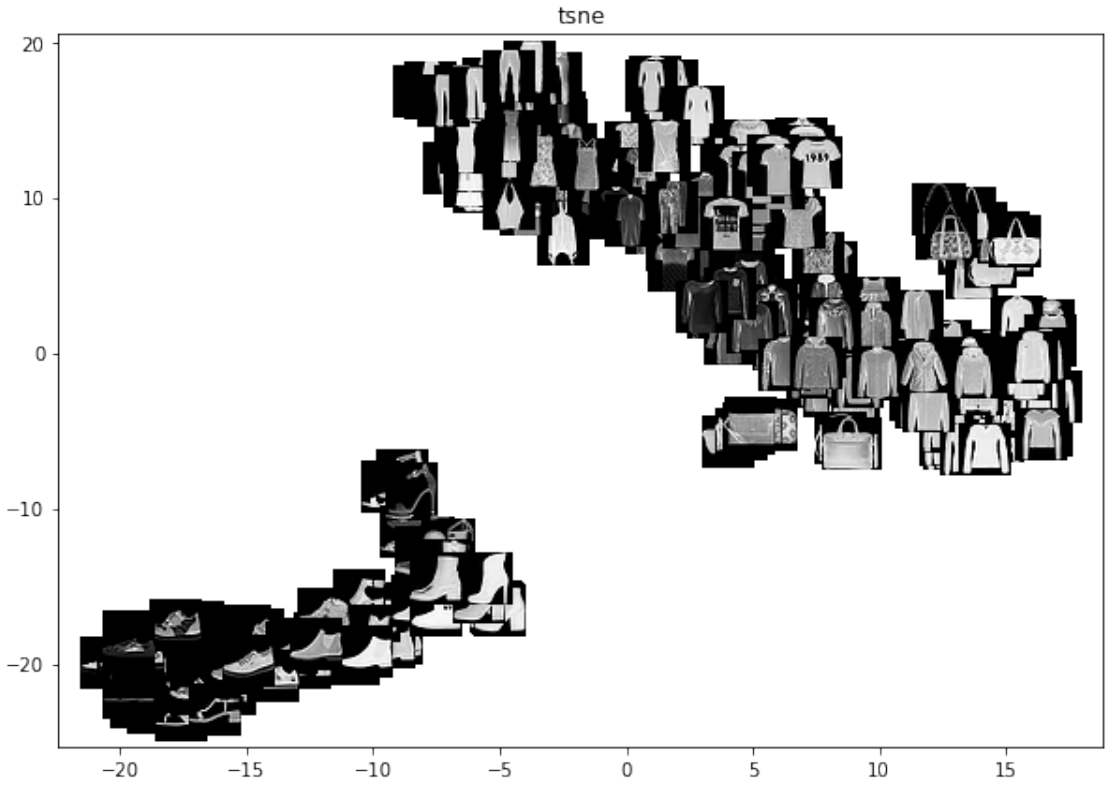
[25]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=5)
```



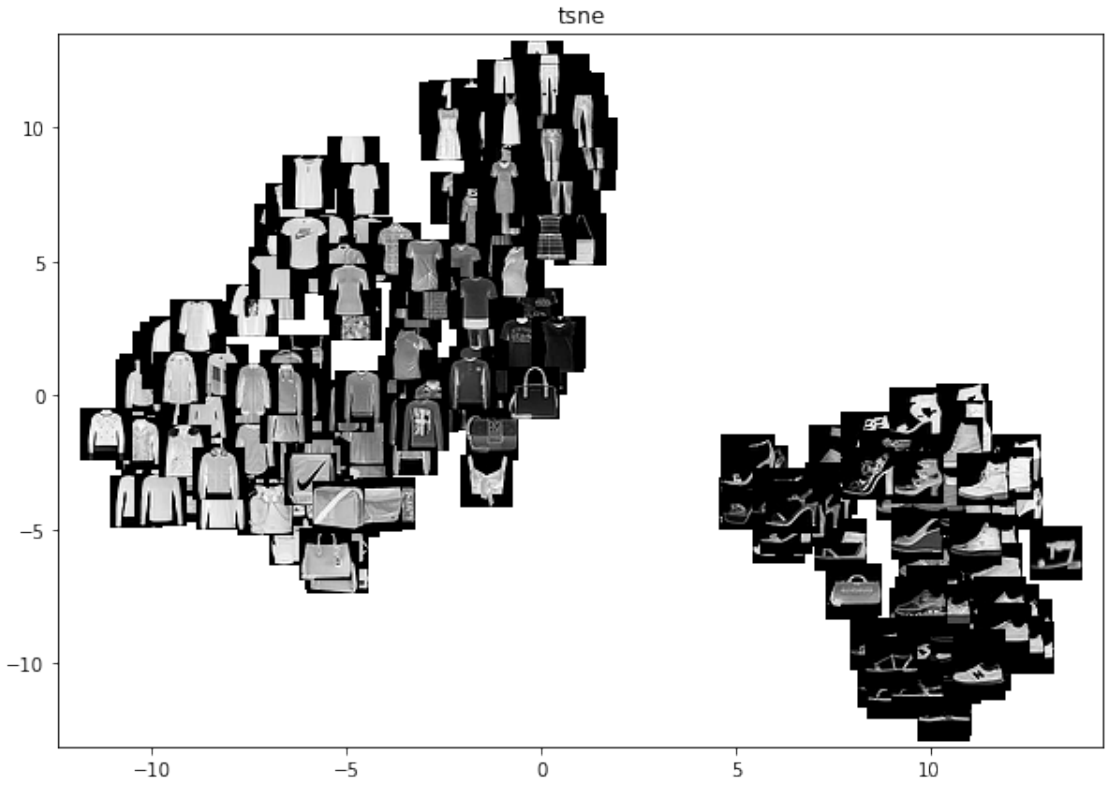
[26]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=10)
```



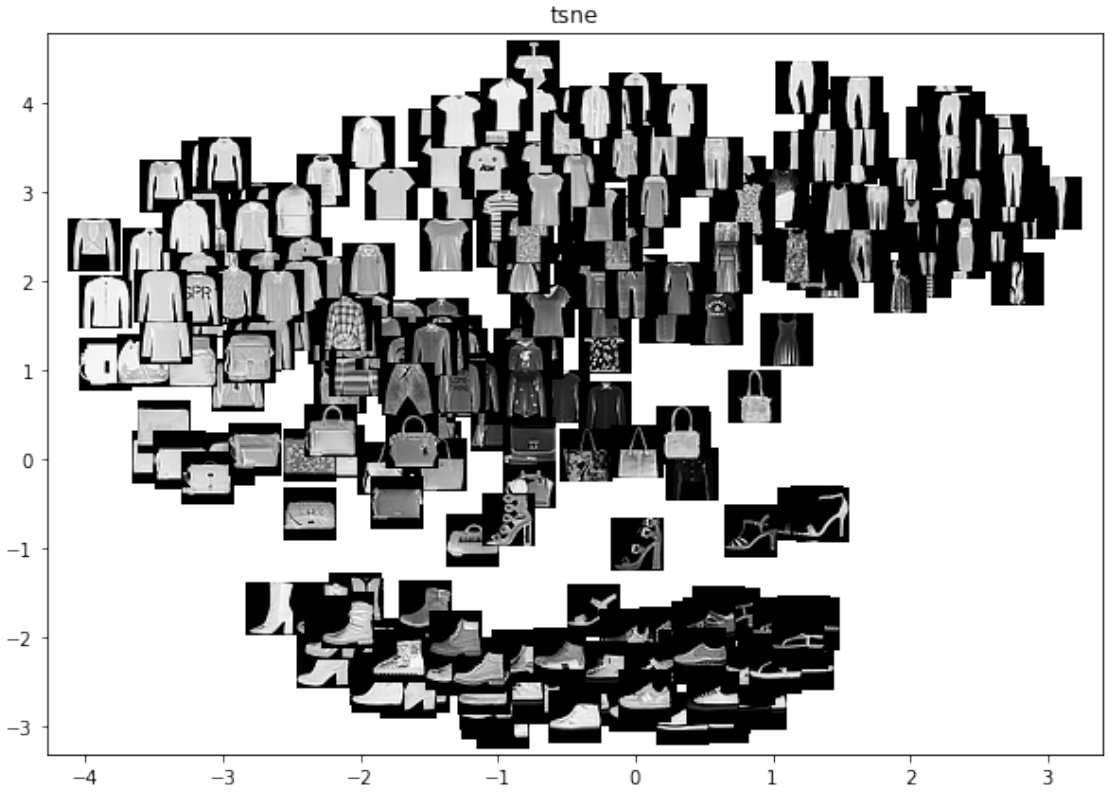
[27]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=30)
```



[28]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=50)
```



[29]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=150)
```



2.3 Task 6: Linear Interpolation on the latent space

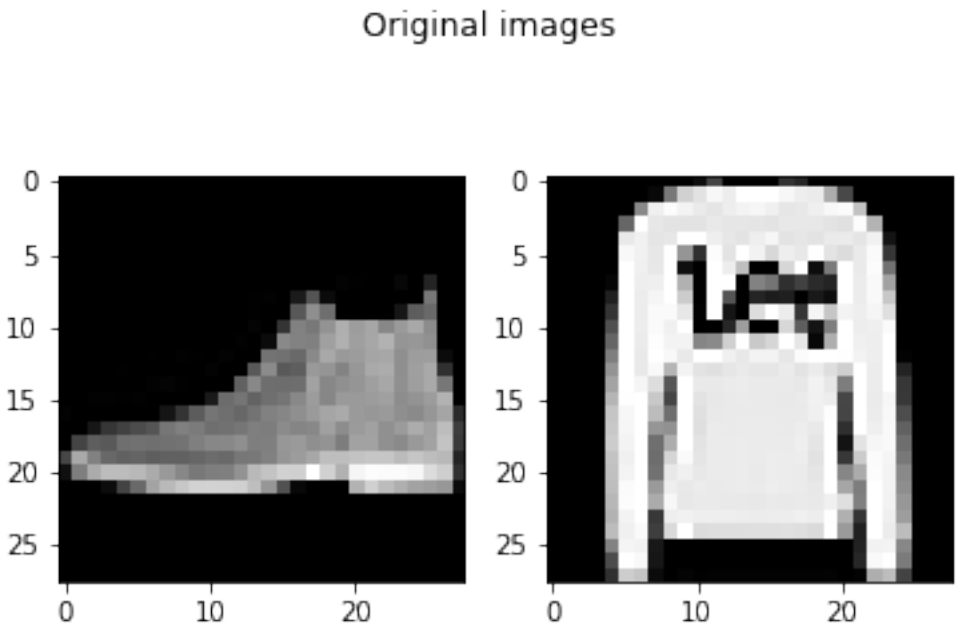
If the latent space has learned something meanigfull, we can leverage this for further analysis/downstream tasks. Anyways, we were wondering all along how the interpolation between a shoe and a pullover might look like.

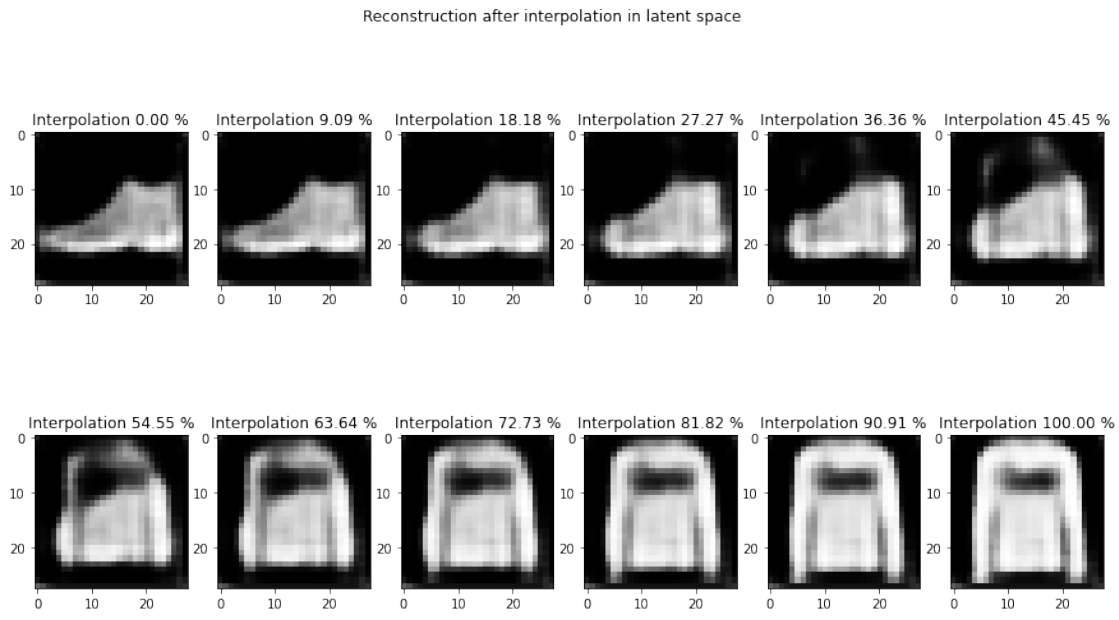
For this we encode two images $z_i = f_{enc}(x_i)$ and $z_j = f_{enc}(x_j)$. Then we linearily interpolate k equidistant locations on the line between z_i and z_j. Those locations are then be decooded by the decoder network $f_{dec}(\dots)$.

[30]:
```
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.
```

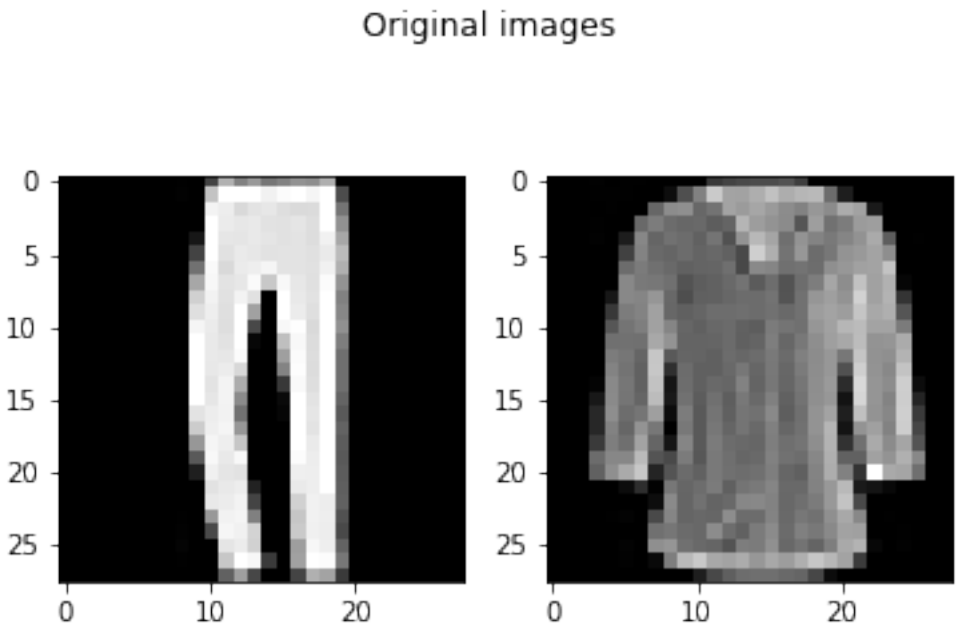
```
    idx_i          : int
                     Id for first image.
    idx_j          : int
                     Id for second image.
    n              : n, optional, default: 1
                     Number of intermediate interoplations (including original␣
↪reconstructions).

    """
    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():
        # TODO: Linearily interpolate between `z_i` and `z_j` in `n`␣
↪equidistant steps.
        # Then decode the embedding and plot the image and add the percentage␣
↪as a title.
        ## BEGIN SOLUTION
        alpha_vals = np.linspace(0., 1., num=n, endpoint=True)
        for i, alpha in enumerate(alpha_vals):
            a = torch.tensor(alpha)
            z_inter = (1.0 - a) * z_i + a * z_j
            im_inter = model.decode(torch.unsqueeze(z_inter, 0))
            ax[i].imshow(im_inter[0][0].numpy(), cmap='gray')
            ax[i].set_title("Interpolation {0:.2f} %".format(alpha*100))

        ## END SOLUTION

    plt.show()
```

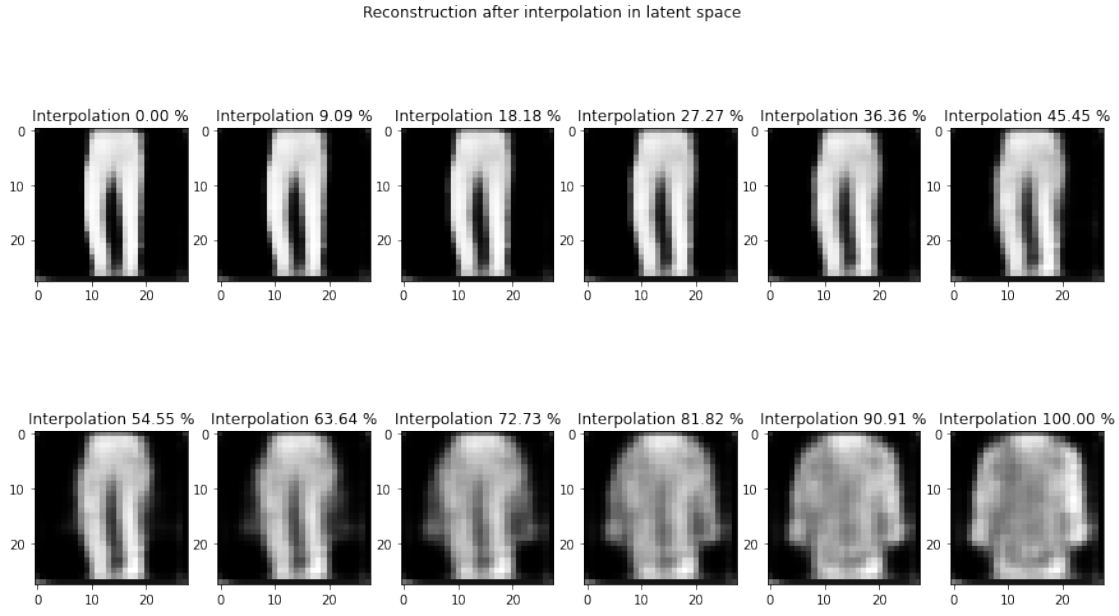
[31]:
```
interpolate_between(model, test_dataset, 0, 1)
```



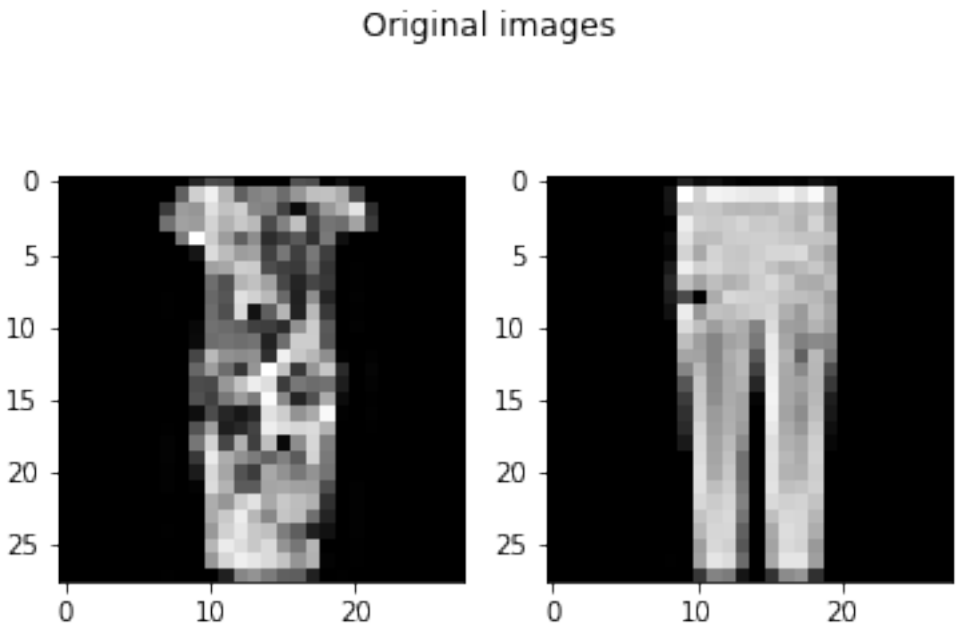


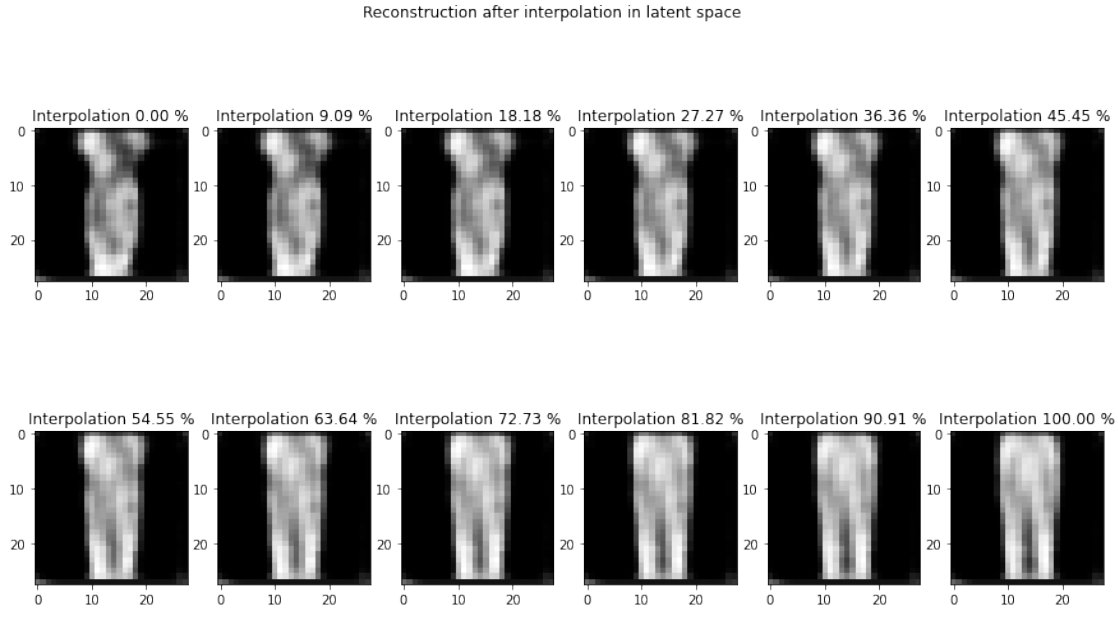
[32]:
```
interpolate_between(model, test_dataset, 2, 4)
```





[33]:
```
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 %

[ ]: