Problem 11: first row a.b.c.d. belongs to $\sigma=2$, because for large distance instances i and j their similarity is lower than the second row e.f.g.h. in photo we can see the first row's color is darker than the second row.

d.h are not correct because for $P_{0,12}$ should be equal to $P_{12,0}$

b.f are not correct because 2 points which are near to each other are darker than the 2 points which are far from each other, it should be lighter.

c.d are not correct, because for first cluster it should be 4 points but there are only 3 points.

So the correct one is (a). for $\sigma=2$ and (e) for $\sigma=5$

problem 2:

$$f(X) = \sigma(X \cdot W_1) W_2$$
$$= X W_1 W_2 \qquad X \in R^{N \times D}$$
$$W_1 \in R^{D \times K}$$
$$W_2 \in R^{K \times D}$$

if $K < D$

there is always error when doing upsampling from K to D dimension.

so the reconstruction error can't be 0.

if $\mathrm{rank}(X) \leq K$, then it could be possible.

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.

Matrix Factorization

In [27]:
```
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
```

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.

Load and Preprocess the Data (nothing to do here)

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

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

Out[3]:
```
array([[101968,   1880,      1],
       [101968,    284,      5],
       [101968,   1378,      2],
       ...,
       [ 72452,   2100,      4],
       [ 72452,   2050,      5],
       [ 74861,   3979,      5]], dtype=int64)
```

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.

In [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_r
R
```

Out[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.

In [5]:
```
def cold_start_preprocessing(matrix, min_entries):
    """
    Recursively removes rows and columns from the input matrix which have less than

    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   # A1 is used to convert the matrix
        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
```

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

In [6]:
```
def shift_user_mean(matrix):
    """
    Subtract the mean rating per user from the non-zero elements in the input matri

    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

    """

    # TODO: Compute the modified matrix and user_means

    ## BEGIN SOLUTION
    nnz_mask = (matrix > 0) # represents the non-zero elements in the matrix
    user_means = matrix.sum(1) / nnz_mask.sum(1) #sum of all ratings per user divid
```

```
    subtract_mask = sp.csr_matrix(user_means).multiply(nnz_mask)   #transform the u
    matrix = matrix-subtract_mask
    ## END SOLUTION

    assert np.all(np.isclose(matrix.mean(1), 0))
    return matrix, user_means
```

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

In [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 e

    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
```

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

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

In [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
```

In [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 - np.ravel(user_means[np.array(val_idx).T[:,0]])
test_values_shifted = test_values - np.ravel(user_means[np.array(test_idx).T[:,0]])
```

Compute the loss function (nothing to do here)

In [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 indi
    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.

    return mean_sse_loss + regularization_loss
```

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.

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

In [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
             'random' means we initialize the entries in P and Q randomly in the in

    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 == 'svd':
        u, s, v = svds(matrix, k)   # s is the 1 dimensional array of singular valu
        Q = u.dot(s)
        P = v
    elif init == 'random':
        Q = np.random.rand(matrix.shape[0], k)
        P = np.random.rand(k, matrix.shape[1])
    else:
        raise ValueError("init not recognized")
    ## END SOLUTION

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

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

In [28]:
```
from scipy.sparse import csr_matrix
```

In [29]:
```python
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, o
    """
    Perform matrix factorization using alternating optimization. Training is done v
    i.e. we stop training after we observe no improvement on the validation loss fo
    amount of training steps. We then return the best values for Q and P oberved du

    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 matri
                         nnz refers to the number of non-zero entries. Note that thi
                         from the number of non-zero entries in the input matrix M,
                         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 ea
                         no improvement on the validation error for a specified numb
                         (see "patience" for details).

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

    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 valida
                         iterations (see eval_every for details). After we stop trai
                         observed values for Q and P (based on the validation loss)

    eval_every         : int, optional, default: 1
                         Evaluate the training and validation loss every X steps. If
                         of the validation error, we decrease our patience by 1, els

    optimizer          : str, optional, default: sgd
                         If `sgd` stochastic gradient descent shall be used. Otherwi
```

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

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

    validation_losses : list of floats
                         Validation loss for every evaluation iteration, can be used
                         loss over time.

    train_losses       : list of floats
                         Training loss for every evaluation iteration, can be used f
                         loss over time.

    converged_after    : int
                         it - patience*eval_every, where it is the iteration in whic
                         or -1 if we hit max_steps before converging.

    """

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

    # Build a sparse mask of non-zero entries from non_zero_idx
    nnz_mask = sp.coo_matrix((np.ones(len(non_zero_idx)),
                              (non_zero_idx[:, 0], non_zero_idx[:, 1])),
                             shape=R.shape, dtype="uint8").tocsr()
    # Precompute per-column and per-row nonzero indices using LIL format
    cols = nnz_mask.T.tolil().rows  # list of lists for each column (each list cont
    rows = nnz_mask.tolil().rows    # list of lists for each row (each list contain

    # Create a Ridge regressor instance for ALS updates
    reg = Ridge(alpha=reg_lambda, fit_intercept=False)

    # Initialize Q and P using the specified strategy
    Q, P = initialize_Q_P(R, k, init)

    # Precompute training indices in the expected tuple format for the loss functio
    # (array_of_row_indices, array_of_col_indices)
    train_idx = tuple(non_zero_idx.T)

    # Initialize containers for losses and early stopping variables
    train_losses = []
    validation_losses = []
    best_val_loss = np.inf
    best_Q = Q.copy()
    best_P = P.copy()
    current_patience = patience
    converged_after = -1
    times = []
    bef = None

    # Ensure that R is in CSR format for efficient row access
    R = csr_matrix(R)
```

```
    for it in range(max_steps):
        # Timing per iteration (optional)
        if bef is not None:
            times.append(time.time() - bef)
        bef = time.time()

        # Evaluate loss every eval_every iterations
        if it % eval_every == 0:
            # Compute training loss on non-zero entries
            # Here, R[train_idx].A1 converts the nonzero entries to a 1D array
            train_loss = loss(R[train_idx].A1, train_idx, Q, P, reg_lambda)
            train_losses.append(train_loss)

            # Compute validation loss using the provided validation indices and val
            val_loss = loss(val_values, val_idx, Q, P, reg_lambda)
            validation_losses.append(val_loss)

            # Update best model if validation loss improves; otherwise, reduce pati
            if val_loss < best_val_loss:
                best_val_loss = val_loss
                best_Q = Q.copy()
                best_P = P.copy()
                current_patience = patience  # reset patience
            else:
                current_patience -= 1

            if log_every and it % log_every == 0:
                print("Iteration {}: training loss = {:.3f}, validation loss = {:.3

            # Early stopping check
            if current_patience <= 0:
                converged_after = it - patience * eval_every
                print("Early stopping at iteration {}, converged after {} iteration
                break

        # Update latent factors depending on the chosen optimizer
        if optimizer == 'sgd':
            # SGD update: shuffle indices and update one nonzero element at a time.
            sgd_indices = np.arange(len(train_idx[0]))
            np.random.shuffle(sgd_indices)
            for idx in sgd_indices:
                u = train_idx[0][idx]
                i = train_idx[1][idx]
                prediction = Q[u, :].dot(P[:, i])
                # Note: R[u,i] is a 1x1 matrix in sparse format; use float() to ext
                e = float(R[u, i]) - prediction  # error
                # Update latent factors using gradient ascent (or descent with the
                Q[u, :] += lr * (e * P[:, i] - reg_lambda * Q[u, :])
                P[:, i] += lr * (e * Q[u, :] - reg_lambda * P[:, i])
        else:
            # ALS update using Ridge regression:
            # First, fix Q and update P (for each item/column)
            for i in range(R.shape[1]):
                nnz_rows = cols[i]  # List of row indices where column i is nonzero
                if len(nnz_rows) == 0:
                    continue
```

```python
                # Extract the corresponding ratings as a 1D array
                y = np.squeeze(R[nnz_rows, i].toarray())
                # Fit a ridge regression model with predictors Q[nnz_rows, :]
                res = reg.fit(Q[nnz_rows, :], y)
                P[:, i] = res.coef_

            # Next, fix P and update Q (for each user/row)
            for u in range(R.shape[0]):
                nnz_cols = rows[u]  # List of column indices where row u is nonzero
                if len(nnz_cols) == 0:
                    continue
                y = np.squeeze(R[u, nnz_cols].toarray())
                res = reg.fit(P[:, nnz_cols].T, y)
                Q[u, :] = res.coef_

    if times:
        print("Converged after {} iterations, average time per iteration: {:.3f}s".
    else:
        print("No timing information collected.")
    ## END SOLUTION

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

Train the latent factor (nothing to do here)

In [15]:
```python
Q_sgd, P_sgd, val_loss_sgd, train_loss_sgd, converged_sgd = latent_factor_alternati
    R_shifted, nonzero_indices, k=100, val_idx=val_idx, val_values=val_values_shift
    reg_lambda=1e-4, init='random', max_steps=100, patience=10, optimizer='sgd', lr
)
```

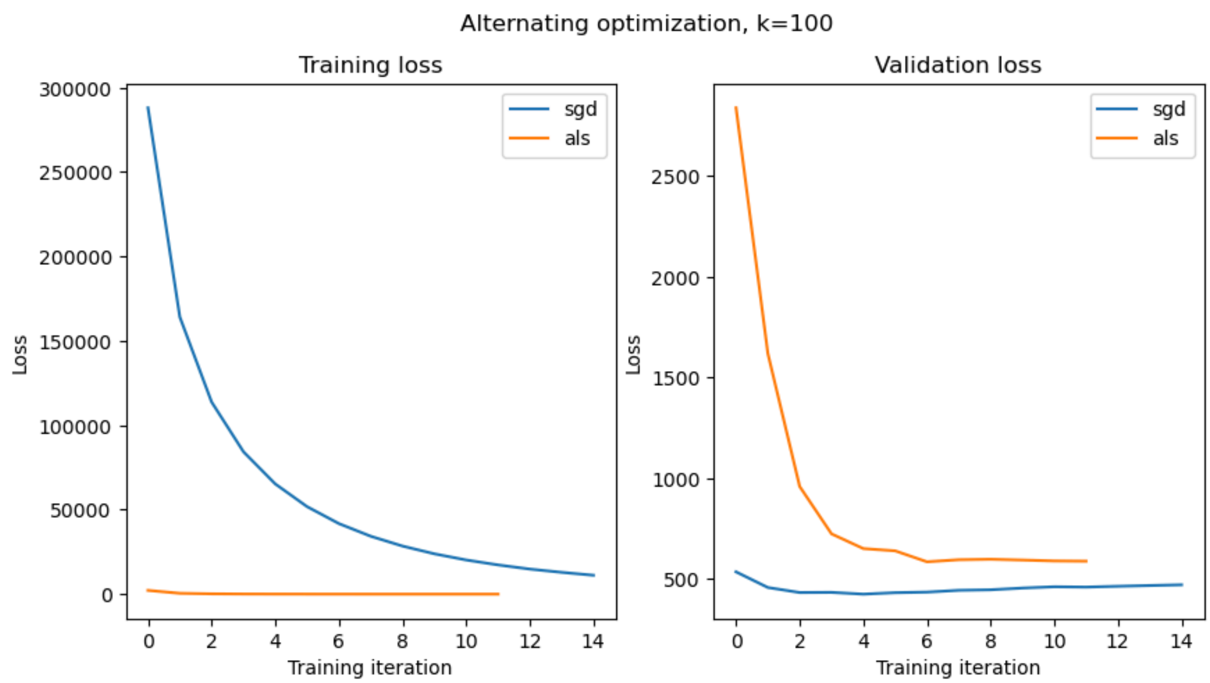
```
Iteration 0: training loss = 96808126.324, validation loss = 123875.680
Iteration 1: training loss = 288085.688, validation loss = 535.421
Iteration 2: training loss = 164209.860, validation loss = 457.681
Iteration 3: training loss = 113798.318, validation loss = 432.778
Iteration 4: training loss = 84373.145, validation loss = 433.530
Iteration 5: training loss = 65291.349, validation loss = 424.737
Iteration 6: training loss = 51833.586, validation loss = 432.020
Iteration 7: training loss = 41798.256, validation loss = 434.836
Iteration 8: training loss = 34304.399, validation loss = 443.813
Iteration 9: training loss = 28483.637, validation loss = 446.300
Iteration 10: training loss = 23895.490, validation loss = 455.048
Iteration 11: training loss = 20247.941, validation loss = 461.358
Iteration 12: training loss = 17321.208, validation loss = 459.781
Iteration 13: training loss = 14852.815, validation loss = 463.825
Iteration 14: training loss = 12924.490, validation loss = 467.573
Iteration 15: training loss = 11195.546, validation loss = 471.131
Early stopping at iteration 15, converged after 5 iterations
Converged after 5 iterations, average time per iteration: 3.013s
```

In [16]:
```python
Q_als, P_als, val_loss_als, train_loss_als, converged_als = latent_factor_alternati
    R_shifted, nonzero_indices, k=100, val_idx=val_idx, val_values=val_values_shift
    reg_lambda=1e-4, init='random', max_steps=100, patience=5, optimizer='als'
)
```

```
Iteration 0: training loss = 96808126.324, validation loss = 123875.680
Iteration 1: training loss = 2233.972, validation loss = 2839.390
Iteration 2: training loss = 513.910, validation loss = 1619.397
Iteration 3: training loss = 196.343, validation loss = 959.220
Iteration 4: training loss = 96.200, validation loss = 724.161
Iteration 5: training loss = 54.610, validation loss = 650.903
Iteration 6: training loss = 34.542, validation loss = 639.788
Iteration 7: training loss = 23.874, validation loss = 585.334
Iteration 8: training loss = 17.810, validation loss = 595.625
Iteration 9: training loss = 14.222, validation loss = 598.023
Iteration 10: training loss = 12.003, validation loss = 594.053
Iteration 11: training loss = 10.579, validation loss = 589.631
Iteration 12: training loss = 9.642, validation loss = 588.454
Early stopping at iteration 12, converged after 7 iterations
Converged after 7 iterations, average time per iteration: 4.893s
```

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

In [17]:
```
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()
```



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.

In [25]:
```
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`

In [26]:
```
train_dataset = FashionMNIST(root='data', download=True, train=True, transform=torc
test_dataset = FashionMNIST(root='data', download=True, train=False, transform=torc
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=1024, shuffle=
                                           num_workers=2, pin_memory=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=1024, shuffle=Fa
                                          num_workers=2, pin_memory=True)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-
idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-images-
idx3-ubyte.gz to data\FashionMNIST\raw\train-images-idx3-ubyte.gz
```

```
100%|██████████| 26.4M/26.4M [00:00<00:00, 48.6MB/s]
Extracting data\FashionMNIST\raw\train-images-idx3-ubyte.gz to data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-labels-idx1-ubyte.gz to data\FashionMNIST\raw\train-labels-idx1-ubyte.gz
100%|██████████| 29.5k/29.5k [00:00<00:00, 1.40MB/s]
Extracting data\FashionMNIST\raw\train-labels-idx1-ubyte.gz to data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz
100%|██████████| 4.42M/4.42M [00:00<00:00, 24.2MB/s]
Extracting data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz to data\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz
100%|██████████| 5.15k/5.15k [00:00<?, ?B/s]
Extracting data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz to data\FashionMNIST\raw
```

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()
  )
)
```

In [32]:

```python
class Autoencoder(nn.Module):
    ## BEGIN SOLUTION
    def __init__(self):
        super().__init__()
        self.encode = nn.Sequential(
            nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(in_channels=4, out_channels=16, kernel_size=3, stride=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.LeakyReLU(negative_slope=0.01),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, stride=1),
            nn.LeakyReLU(negative_slope=0.01)
        )
        self.decode = nn.Sequential(
            nn.ConvTranspose2d(in_channels=32, out_channels=32, kernel_size=3, stri
            nn.LeakyReLU(negative_slope=0.01),
            nn.ConvTranspose2d(in_channels=32, out_channels=16, kernel_size=3, stri
            nn.LeakyReLU(negative_slope=0.01),
            nn.BatchNorm2d(16),
            nn.ConvTranspose2d(in_channels=16, out_channels=16, kernel_size=3, stri
            nn.LeakyReLU(negative_slope=0.01),
            nn.ConvTranspose2d(in_channels=16, out_channels=16, kernel_size=3, stri
            nn.ConvTranspose2d(in_channels=16, out_channels=4, kernel_size=3, strid
            nn.ConvTranspose2d(in_channels=4, out_channels=1, kernel_size=3, stride
            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, 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=T
rue)
    (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=T
rue)
    (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.

In [33]:

```
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])
```

Task 5: Train the autoencoder

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

In [34]:
```
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) # weig
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_approx = model(x)
        loss = F.mse_loss(x_approx, x)  # use the mean squared error loss
        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 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 0.16332513093948364
Training: Epoch 0 batch 20 - loss 0.07702259719371796
Training: Epoch 0 batch 40 - loss 0.046009503304958344
Test: Epoch 0 batch 0 loss 0.05424020066857338
Epoch 0 finished - average train loss 0.07444179316951056, average test loss 0.05377
799160778522
Training: Epoch 1 batch 0 - loss 0.034740861505270004
Training: Epoch 1 batch 20 - loss 0.029188092797994614
Training: Epoch 1 batch 40 - loss 0.02634264901280403
Test: Epoch 1 batch 0 loss 0.024677013978362083
Epoch 1 finished - average train loss 0.02830266283225205, average test loss 0.02443
3193355798723
Training: Epoch 2 batch 0 - loss 0.02467944100499153
Training: Epoch 2 batch 20 - loss 0.023356642574071884
Training: Epoch 2 batch 40 - loss 0.021212579682469368
Test: Epoch 2 batch 0 loss 0.02174491062760353
Epoch 2 finished - average train loss 0.02257301497383643, average test loss 0.02153
863701969385
Training: Epoch 3 batch 0 - loss 0.021204497665166855
Training: Epoch 3 batch 20 - loss 0.020755626261234283
Training: Epoch 3 batch 40 - loss 0.019263967871665955
Test: Epoch 3 batch 0 loss 0.020022915676236153
Epoch 3 finished - average train loss 0.020118014129289125, average test loss 0.0198
1538496911526
Training: Epoch 4 batch 0 - loss 0.020288849249482155
Training: Epoch 4 batch 20 - loss 0.018774626776576042
Training: Epoch 4 batch 40 - loss 0.017693696543574333
Test: Epoch 4 batch 0 loss 0.018866803497076035
Epoch 4 finished - average train loss 0.018633040047045482, average test loss 0.0186
4192318171263
Training: Epoch 5 batch 0 - loss 0.018362239003181458
Training: Epoch 5 batch 20 - loss 0.017524991184473038
Training: Epoch 5 batch 40 - loss 0.017268827185034752
Test: Epoch 5 batch 0 loss 0.017725510522723198
Epoch 5 finished - average train loss 0.017661373872878187, average test loss 0.0175
39930529892445
Training: Epoch 6 batch 0 - loss 0.01701536402106285
Training: Epoch 6 batch 20 - loss 0.016717921942472458
Training: Epoch 6 batch 40 - loss 0.01660574972629547
Test: Epoch 6 batch 0 loss 0.017155200242996216
Epoch 6 finished - average train loss 0.01679033890240273, average test loss 0.01696
257423609495
Training: Epoch 7 batch 0 - loss 0.016436509788036346
Training: Epoch 7 batch 20 - loss 0.015376505441963673
Training: Epoch 7 batch 40 - loss 0.015487338416278362
Test: Epoch 7 batch 0 loss 0.01612367480993271
Epoch 7 finished - average train loss 0.016002490706110404, average test loss 0.0159
70653668046
Training: Epoch 8 batch 0 - loss 0.015585355460643768
Training: Epoch 8 batch 20 - loss 0.015307411551475525
Training: Epoch 8 batch 40 - loss 0.01493903435766697
Test: Epoch 8 batch 0 loss 0.015602289699018002
Epoch 8 finished - average train loss 0.015366997222526598, average test loss 0.0154
38100416213274
Training: Epoch 9 batch 0 - loss 0.014677392318844795
Training: Epoch 9 batch 20 - loss 0.014951184391975403
```

```
Training: Epoch 9 batch 40 - loss 0.014636107720434666
Test: Epoch 9 batch 0 loss 0.015006846748292446
Epoch 9 finished - average train loss 0.014915093558572106, average test loss 0.0148
66883121430873
Training: Epoch 10 batch 0 - loss 0.014897612854838371
Training: Epoch 10 batch 20 - loss 0.014539618976414204
Training: Epoch 10 batch 40 - loss 0.014263765886425972
Test: Epoch 10 batch 0 loss 0.014664363116025925
Epoch 10 finished - average train loss 0.014357844635970512, average test loss 0.014
516695961356163
Training: Epoch 11 batch 0 - loss 0.014133739285171032
Training: Epoch 11 batch 20 - loss 0.013830744661390781
Training: Epoch 11 batch 40 - loss 0.01400463841855526
Test: Epoch 11 batch 0 loss 0.013989139348268509
Epoch 11 finished - average train loss 0.013926606451682115, average test loss 0.013
869798742234707
Training: Epoch 12 batch 0 - loss 0.013612659648060799
Training: Epoch 12 batch 20 - loss 0.013439959846436977
Training: Epoch 12 batch 40 - loss 0.013481802307069302
Test: Epoch 12 batch 0 loss 0.014212798327207565
Epoch 12 finished - average train loss 0.013643492360488844, average test loss 0.014
107572287321091
Training: Epoch 13 batch 0 - loss 0.013737405650317669
Training: Epoch 13 batch 20 - loss 0.013696666806936264
Training: Epoch 13 batch 40 - loss 0.013009854592382908
Test: Epoch 13 batch 0 loss 0.013348404318094254
Epoch 13 finished - average train loss 0.01344069562284118, average test loss 0.0132
2788642719388
Training: Epoch 14 batch 0 - loss 0.01314406469464302
Training: Epoch 14 batch 20 - loss 0.013288733549416065
Training: Epoch 14 batch 40 - loss 0.01324933860450983
Test: Epoch 14 batch 0 loss 0.013626034371554852
Epoch 14 finished - average train loss 0.01307654622310804, average test loss 0.0135
21607406437397
Training: Epoch 15 batch 0 - loss 0.013130522333085537
Training: Epoch 15 batch 20 - loss 0.012910023331642151
Training: Epoch 15 batch 40 - loss 0.01238768920302391
Test: Epoch 15 batch 0 loss 0.012939715757966042
Epoch 15 finished - average train loss 0.012765351067281376, average test loss 0.012
82679894939065
Training: Epoch 16 batch 0 - loss 0.012488340027630329
Training: Epoch 16 batch 20 - loss 0.01271215733140707
Training: Epoch 16 batch 40 - loss 0.012102817185223103
Test: Epoch 16 batch 0 loss 0.01272273063659668
Epoch 16 finished - average train loss 0.01255596500142651, average test loss 0.0126
1558597907424
Training: Epoch 17 batch 0 - loss 0.012682215310633183
Training: Epoch 17 batch 20 - loss 0.012278897687792778
Training: Epoch 17 batch 40 - loss 0.012239453382790089
Test: Epoch 17 batch 0 loss 0.012921199202537537
Epoch 17 finished - average train loss 0.012289256804575354, average test loss 0.012
804200220853091
Training: Epoch 18 batch 0 - loss 0.012089014984667301
Training: Epoch 18 batch 20 - loss 0.0122118154540658
Training: Epoch 18 batch 40 - loss 0.012484940700232983
Test: Epoch 18 batch 0 loss 0.012460680678486824
```

```
Epoch 18 finished - average train loss 0.012092187574480549, average test loss 0.012
353301979601383
Training: Epoch 19 batch 0 - loss 0.012193476781249046
Training: Epoch 19 batch 20 - loss 0.012568363919854164
Training: Epoch 19 batch 40 - loss 0.011695049703121185
Test: Epoch 19 batch 0 loss 0.0118959816172719
Epoch 19 finished - average train loss 0.011873520879169642, average test loss 0.011
798912286758423
Training: Epoch 20 batch 0 - loss 0.011729513294994831
Training: Epoch 20 batch 20 - loss 0.01189006119966507
Training: Epoch 20 batch 40 - loss 0.012110292911529541
Test: Epoch 20 batch 0 loss 0.011768973432481289
Epoch 20 finished - average train loss 0.011859171581849202, average test loss 0.011
666670627892017
Training: Epoch 21 batch 0 - loss 0.011566086672246456
Training: Epoch 21 batch 20 - loss 0.01139517966657877
Training: Epoch 21 batch 40 - loss 0.011498126201331615
Test: Epoch 21 batch 0 loss 0.012786075472831726
Epoch 21 finished - average train loss 0.011559399176325839, average test loss 0.012
665333412587642
Training: Epoch 22 batch 0 - loss 0.01296939142048359
Training: Epoch 22 batch 20 - loss 0.011646687984466553
Training: Epoch 22 batch 40 - loss 0.011553688906133175
Test: Epoch 22 batch 0 loss 0.011370765045285225
Epoch 22 finished - average train loss 0.011472421066867093, average test loss 0.011
26386970281601
Training: Epoch 23 batch 0 - loss 0.011309162713587284
Training: Epoch 23 batch 20 - loss 0.01118531171232462
Training: Epoch 23 batch 40 - loss 0.01118093729019165
Test: Epoch 23 batch 0 loss 0.011394556611776352
Epoch 23 finished - average train loss 0.011334089451800968, average test loss 0.011
306302808225154
Training: Epoch 24 batch 0 - loss 0.010896682739257812
Training: Epoch 24 batch 20 - loss 0.010997184552252293
Training: Epoch 24 batch 40 - loss 0.010981237515807152
Test: Epoch 24 batch 0 loss 0.01108752004802227
Epoch 24 finished - average train loss 0.011146038012989497, average test loss 0.010
990981385111809
Training: Epoch 25 batch 0 - loss 0.010861345566809177
Training: Epoch 25 batch 20 - loss 0.011222340166568756
Training: Epoch 25 batch 40 - loss 0.01115289144217968
Test: Epoch 25 batch 0 loss 0.011359281837940216
Epoch 25 finished - average train loss 0.01103632120510279, average test loss 0.0112
63496428728103
Training: Epoch 26 batch 0 - loss 0.011453206650912762
Training: Epoch 26 batch 20 - loss 0.010815705172717571
Training: Epoch 26 batch 40 - loss 0.01077176257967949
Test: Epoch 26 batch 0 loss 0.010944569483399391
Epoch 26 finished - average train loss 0.010890498110172103, average test loss 0.010
849197581410407
Training: Epoch 27 batch 0 - loss 0.010548166930675507
Training: Epoch 27 batch 20 - loss 0.010749094188213348
Training: Epoch 27 batch 40 - loss 0.010779944248497486
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[34], line 19
     17 x_approx = model(x)
     18 loss = F.mse_loss(x_approx, x)  # use the mean squared error loss
---> 19 loss.backward()
     20 optimizer.step()
     22 ## END SOLUTION

File g:\anaconda_app\envs\ml_env\lib\site-packages\torch\_tensor.py:521, in Tensor.b
ackward(self, gradient, retain_graph, create_graph, inputs)
    511 if has_torch_function_unary(self):
    512     return handle_torch_function(
    513         Tensor.backward,
    514         (self,),
   (...)
    519         inputs=inputs,
    520     )
--> 521 torch.autograd.backward(
    522     self, gradient, retain_graph, create_graph, inputs=inputs
    523 )

File g:\anaconda_app\envs\ml_env\lib\site-packages\torch\autograd\__init__.py:289, i
n backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables, input
s)
    284     retain_graph = create_graph
    286 # The reason we repeat the same comment below is that
    287 # some Python versions print out the first line of a multi-line function
    288 # calls in the traceback and some print out the last line
--> 289 _engine_run_backward(
    290     tensors,
    291     grad_tensors_,
    292     retain_graph,
    293     create_graph,
    294     inputs,
    295     allow_unreachable=True,
    296     accumulate_grad=True,
    297 )

File g:\anaconda_app\envs\ml_env\lib\site-packages\torch\autograd\graph.py:769, in _
engine_run_backward(t_outputs, *args, **kwargs)
    767     unregister_hooks = _register_logging_hooks_on_whole_graph(t_outputs)
    768 try:
--> 769     return Variable._execution_engine.run_backward(  # Calls into the C++ en
gine to run the backward pass
    770         t_outputs, *args, **kwargs
    771     )  # Calls into the C++ engine to run the backward pass
    772 finally:
    773     if attach_logging_hooks:

KeyboardInterrupt: 
```

In [35]:

```
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)
```

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.

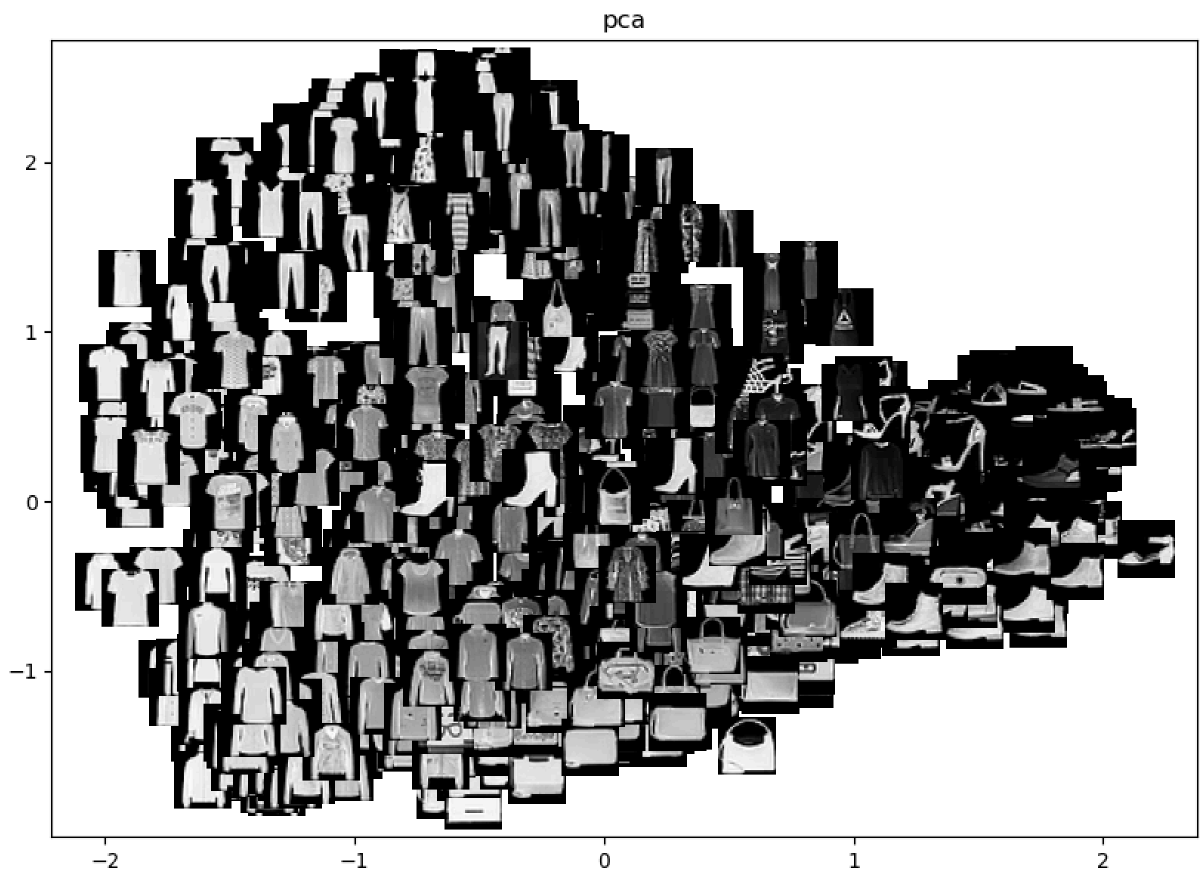
It is not uncommand that a neural network designer wants to understand whats going on in the latent space and therefore uses techniques such as t-SNE.

In [36]:

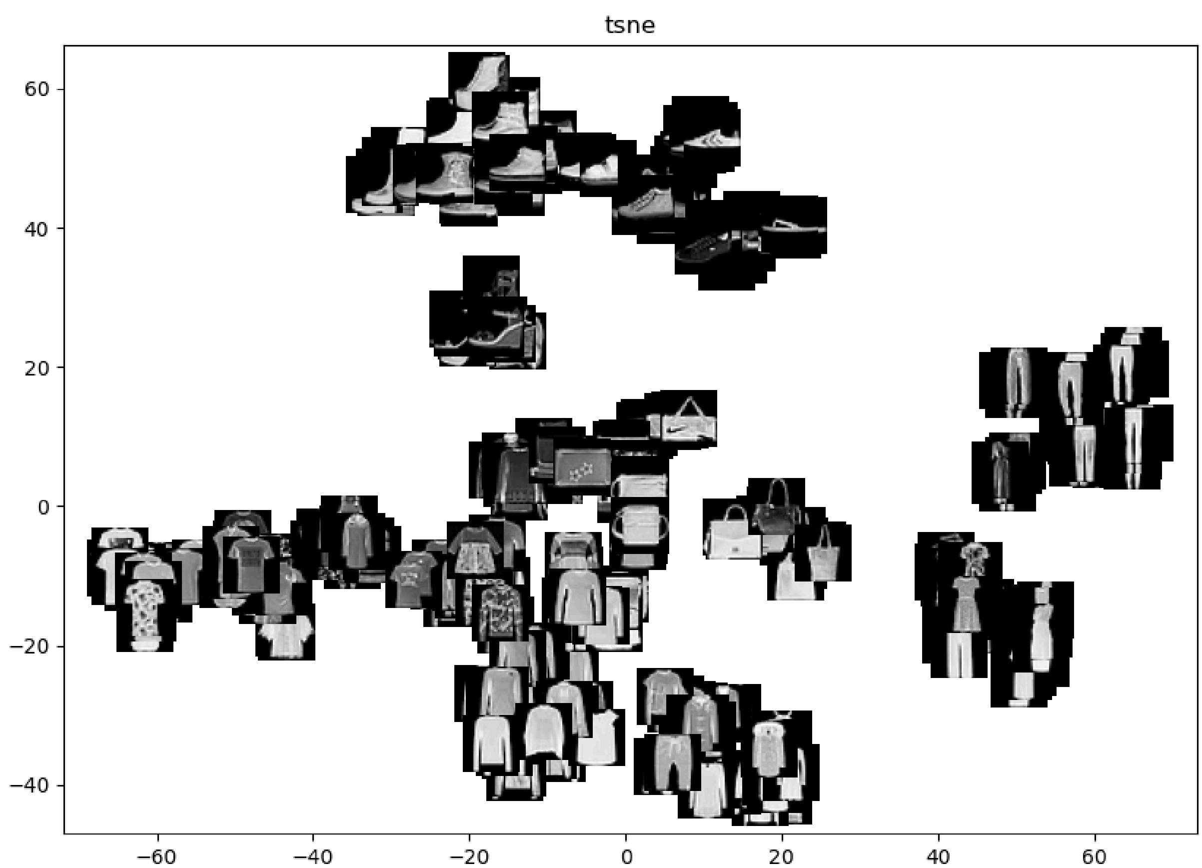
```
def plot_latent(test_dataset: torch.utils.data.Dataset, z_test: torch.Tensor, count
                technique: str, perplexity: float = 30):
    """
    Fit t-SNE or PCA and plots the latent space. Moreover, we then display the corr

    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.r
    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()
```

In [37]:

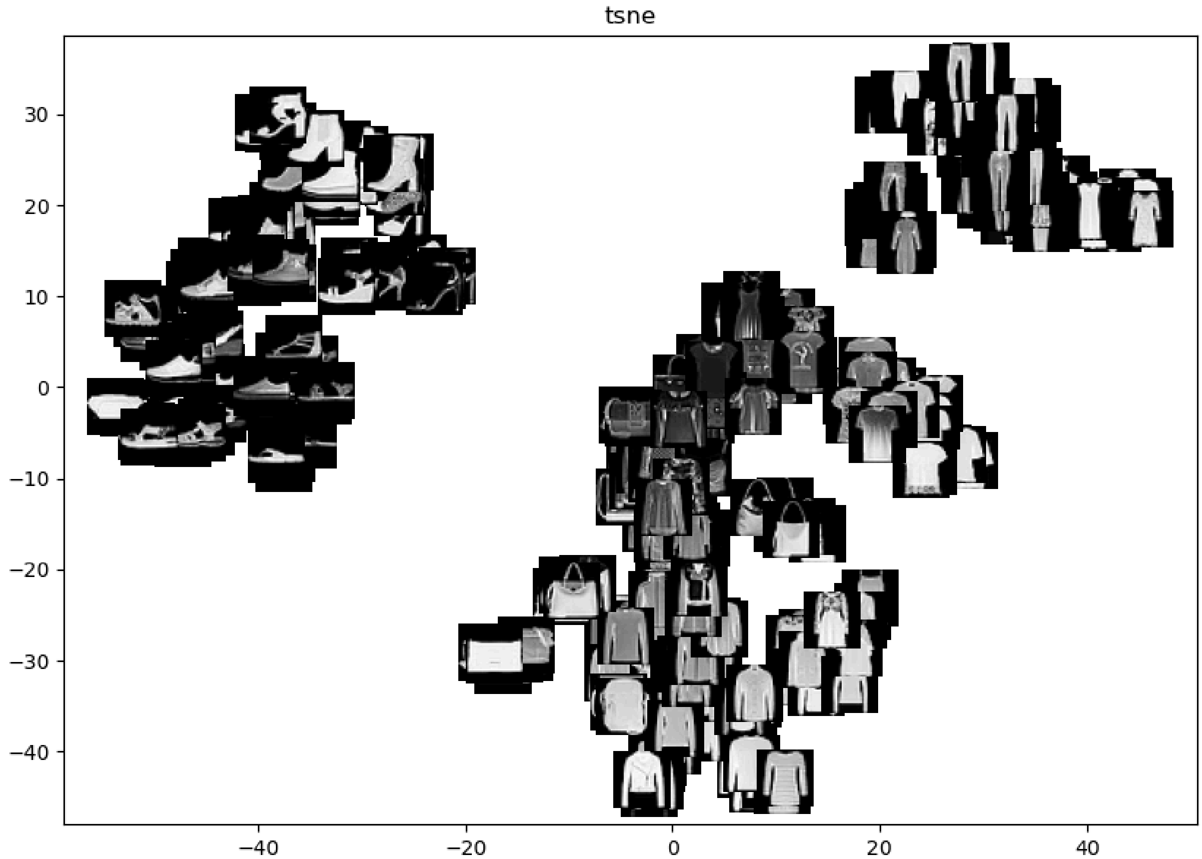
```
plot_latent(test_dataset, latent, 1000, 'pca')
```



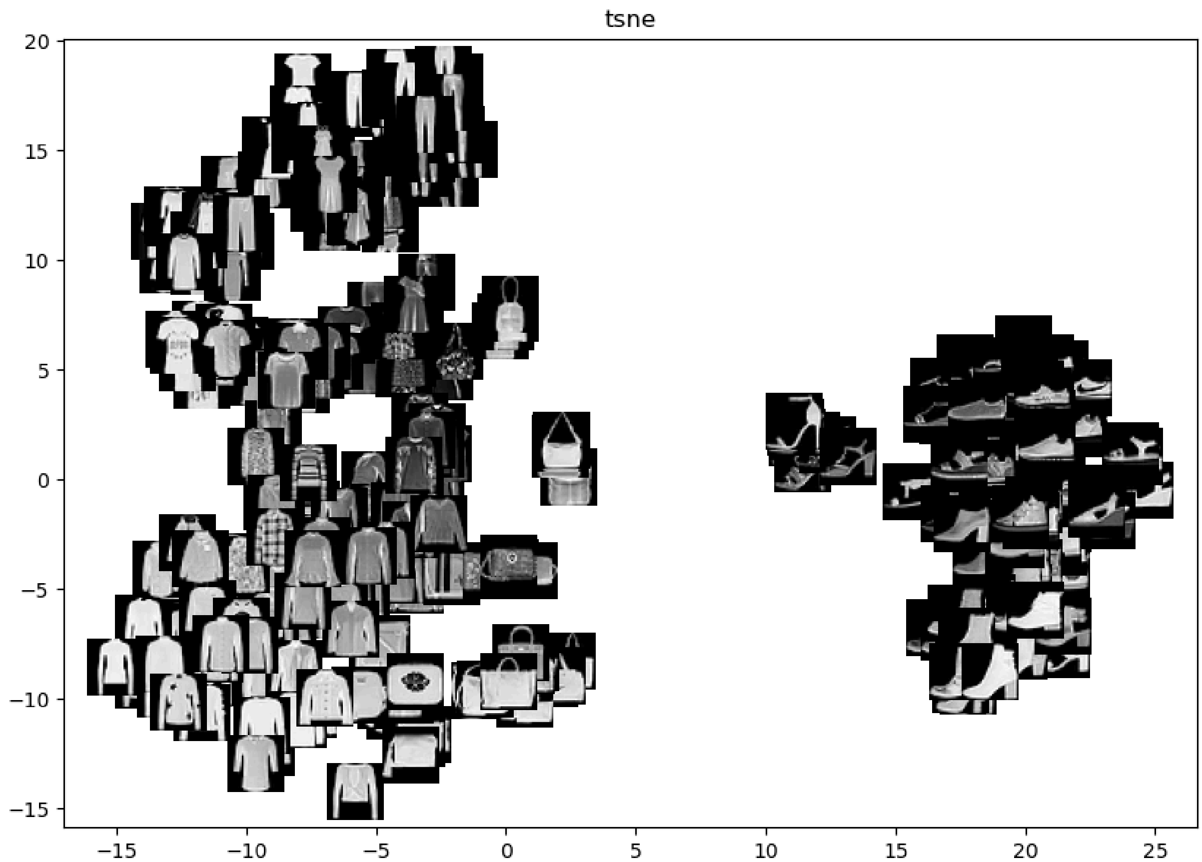

```
In [38]: plot_latent(test_dataset, latent, 300, 'tsne', perplexity=5)
```



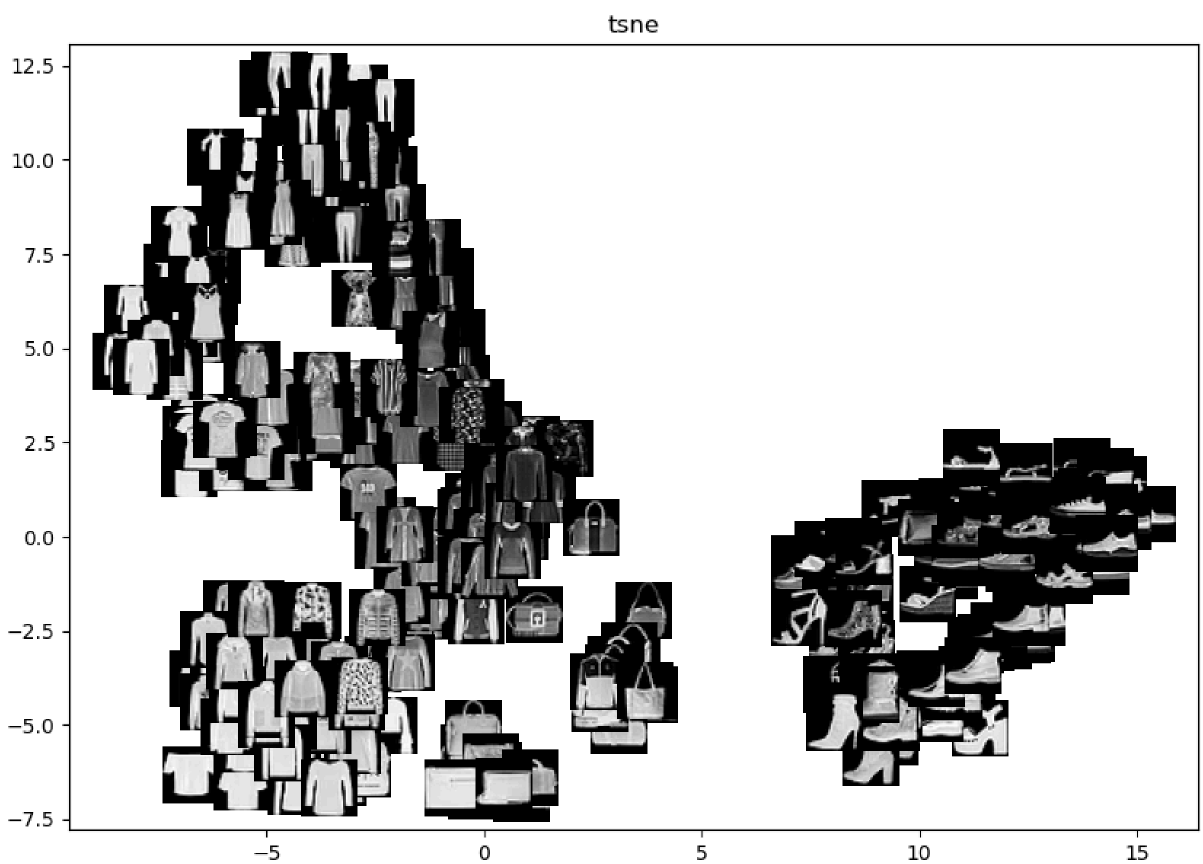
In [39]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=10)
```



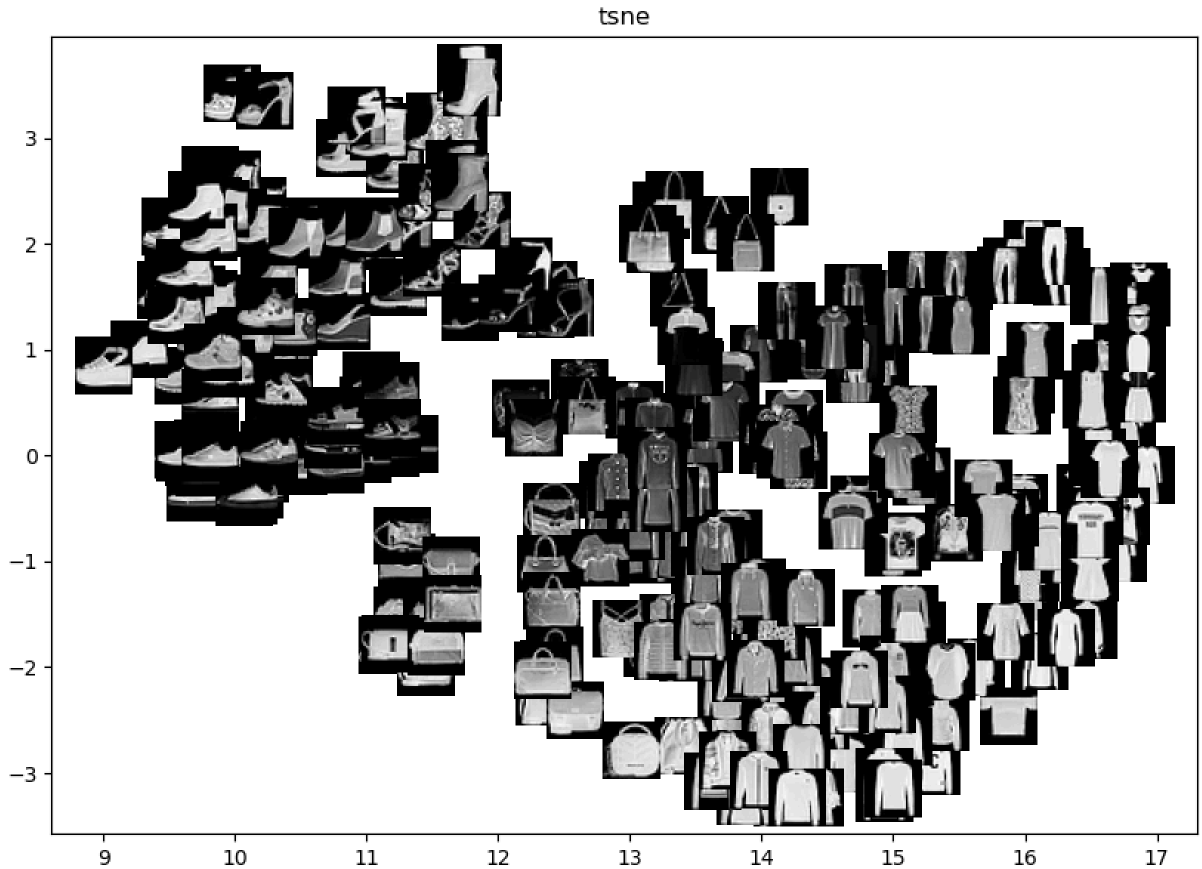
In [40]:
```
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=30)
```




```
In [41]: plot_latent(test_dataset, latent, 300, 'tsne', perplexity=50)
```



In [42]:
```python
plot_latent(test_dataset, latent, 300, 'tsne', perplexity=150)
```



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 $f_{dec}(\ldots)$.

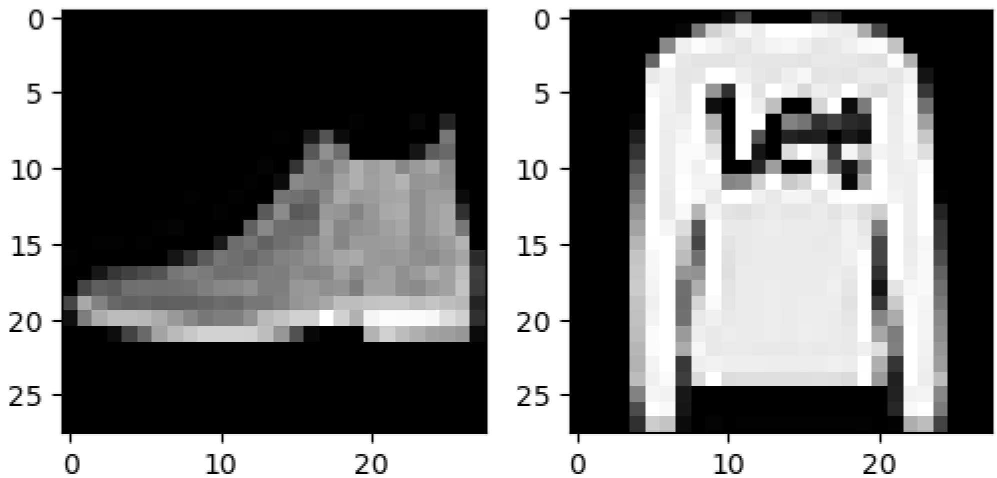
In [ ]:
```python
def interpolate_between(model: Autoencoder, test_dataset: torch.utils.data.Dataset,
    """
    Plot original images and the reconstruction of the linear interpolation in the

    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 recon

    """
    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] # test_data
    z_j = model.encode(test_dataset[idx_j][0].to(device)[None, ...])[0] #None is us

    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 st
        # Then decode the embedding and plot the image and add the percentage as a
        ## BEGIN SOLUTION
        # Generate n equidistant interpolation factors between 0 and 1.
        alphas = np.linspace(0, 1, n)
        for i, alpha in enumerate(alphas):
            # this interpolation is to transform the latent representation of the i
            z_interp = (1 - alpha) * z_i + alpha * z_j
            # Add a batch dimension for decoding (from shape [latent_dim] to [1, la
            z_interp = z_interp.unsqueeze(0)
            # Decode the interpolated latent representation:
            x_recon = model.decode(z_interp)
            # remove the batch dimension
            img = x_recon[0].cpu().numpy().squeeze()
            # Plot the image and add the interpolation percentage as title
            ax[i].imshow(img, cmap='gray')
            ax[i].set_title(f"{alpha*100:.0f}%")
        ## END SOLUTION

    plt.show()
```

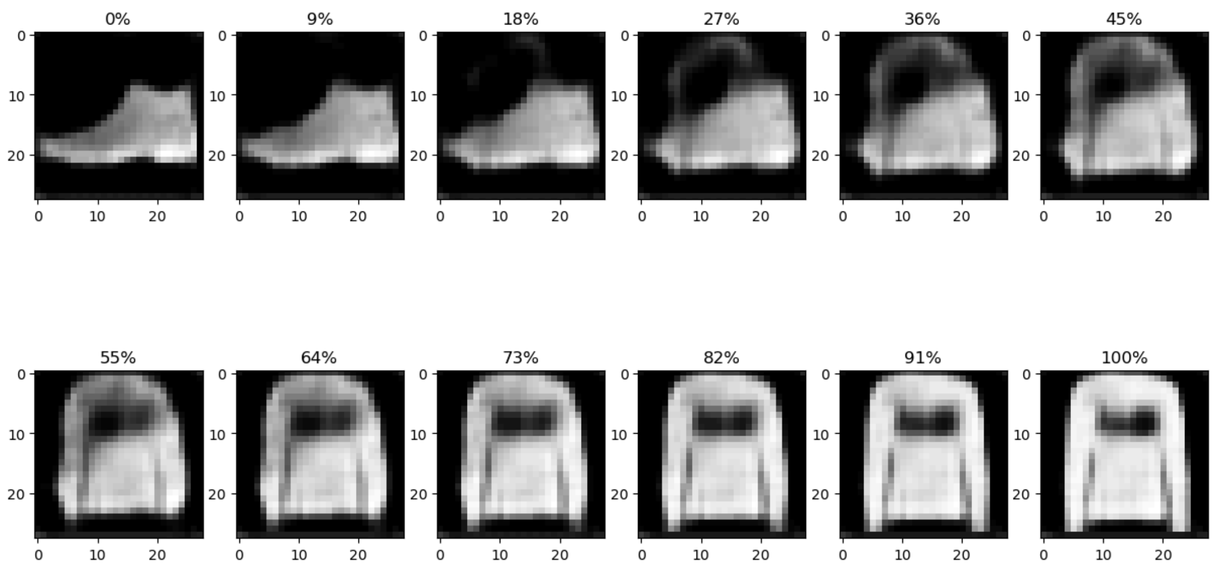
In [46]:

```
interpolate_between(model, test_dataset, 0, 1)
```

Original images

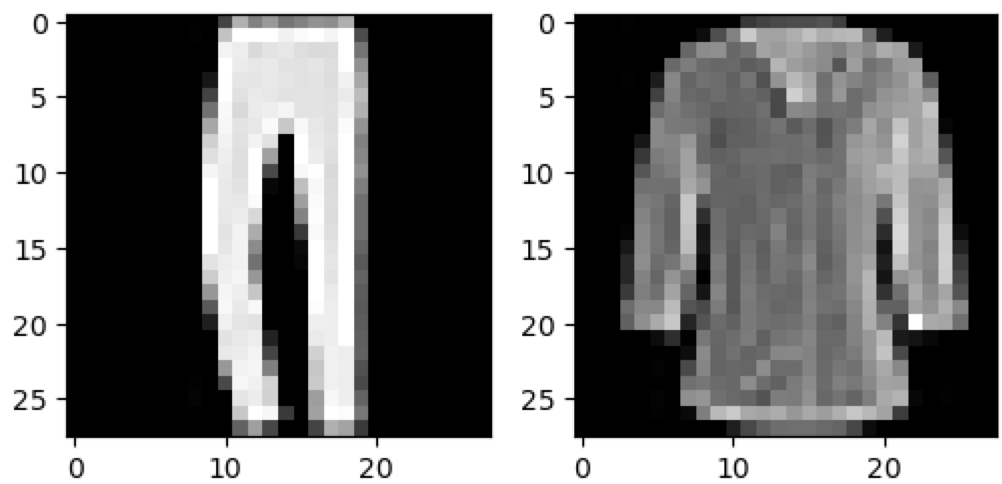


Reconstruction after interpolation in latent space

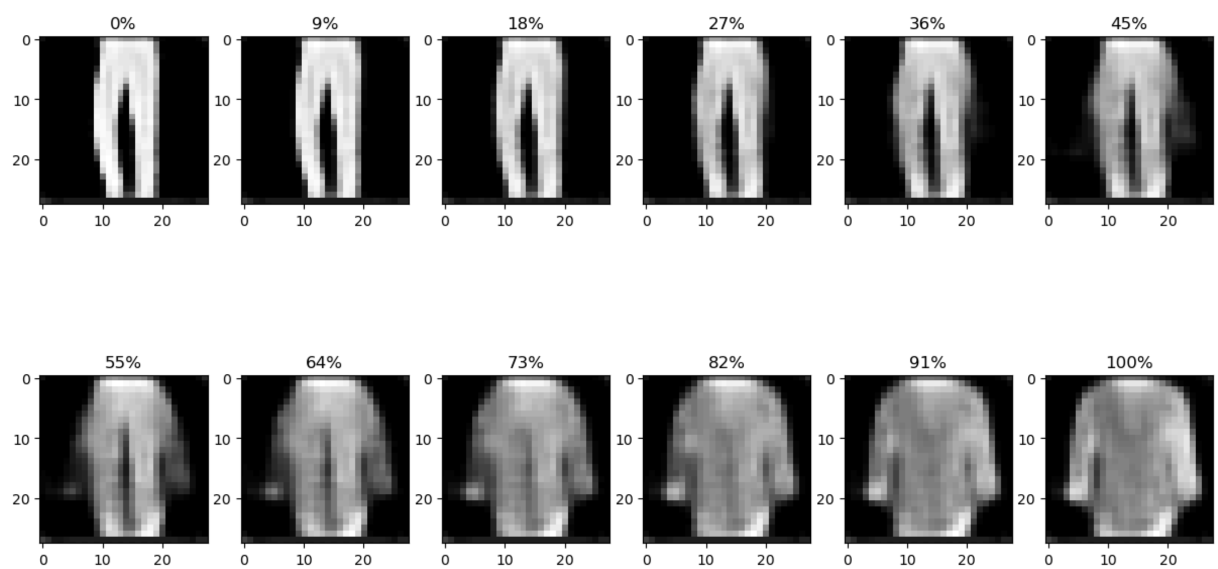


In [47]:
```
interpolate_between(model, test_dataset, 2, 4)
```

Original images

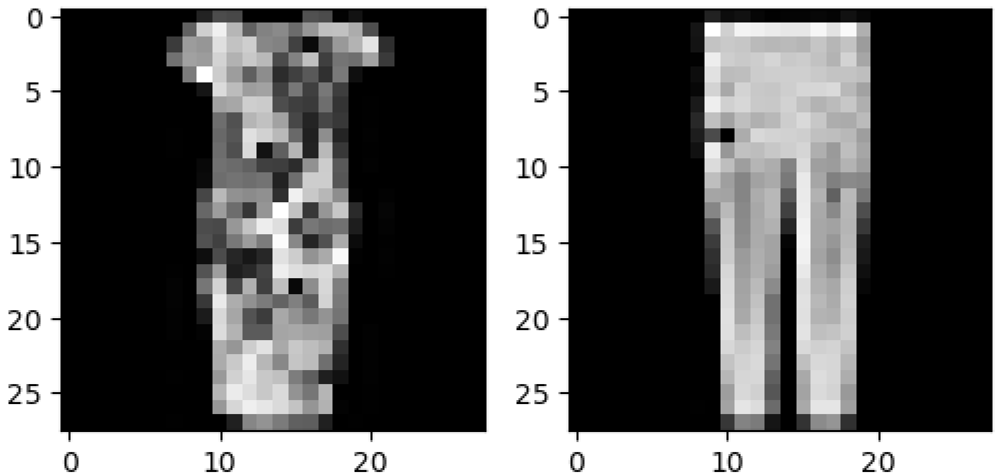


Reconstruction after interpolation in latent space

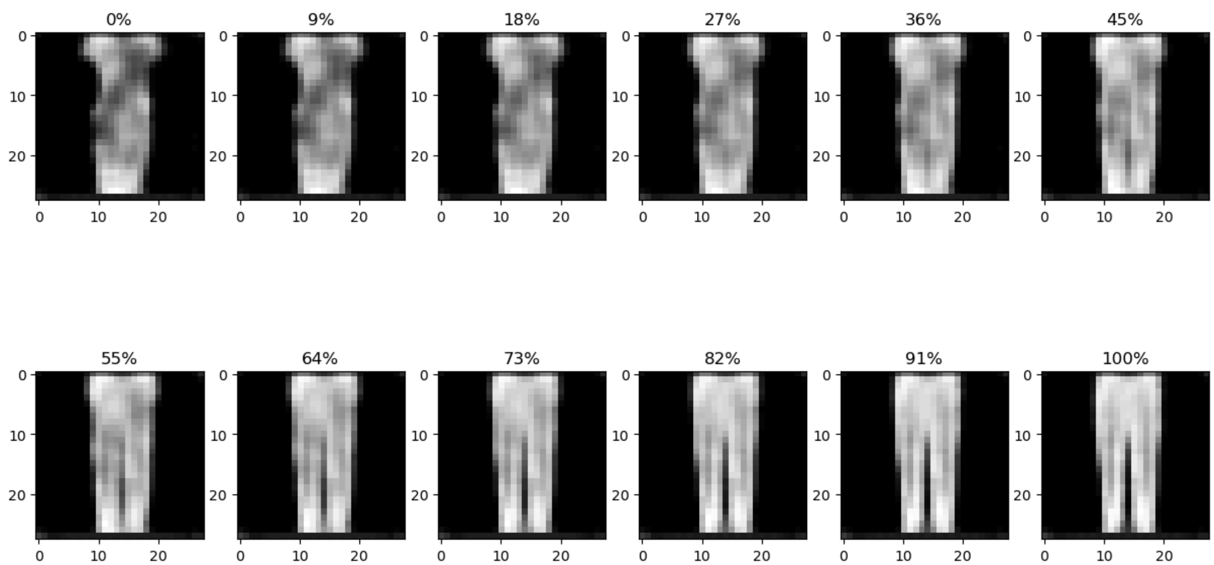


In [48]:
```
interpolate_between(model, test_dataset, 100, 200)
```

Original images



Reconstruction after interpolation in latent space



In [ ]: