Assignment 4 - Course Project

Sebastian Scholl 11827056
Clemens Berger 11821616
Philipp Temmel 00530740

1) Introduction

Our group chose topic 1, Correcting Images with Autoencoders. The task is to train an autoencoder that is capable of correcting and re-generating clean images from their distorted versions. For the data set we chose Kuzushiji-MNIST. Initially we stared out with FASHION-MNIST, had some learnings and then decided to challenge ourselves with the Kuzushiji-MNIST dataset. In order to create a perturbed data set we created our own functions, which modify the data with gaussian noise, rotate or flip the images, add black patches and change the brightness of the original images.

2) Methods

The network we build is an autoencoder, therefore we use mean squared error as loss function:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Autoencoder are built as an encoder-decoder architecture, the encoder receives an image, reduces the dimensions and compresses the input data. The decoder then reconstructs the original image from this compressed data. The autoencoder is fed with our perturbed data as input and their corresponding unmodified clean data samples as output. This way the model learns to reconstruct/correct small modifications.

We decided to implement different model approaches in separate methods. This allowed us to play with a new parameter or slightly different approach, try out some values and report the best working one for that respective model. Models with their ideas are reported below and can be found in the source code in their respective method. This allowed us to easily switch out models and allows to understand easily how we approached the problem.

The decoder part should reflect the encoder part in an autoencoder. We always thought about elements to be changed in the encoder and adjusted the respective counter-part in the decoder, i.e. we used Reshape instead of Flatten, UpSampling2D instead of MaxPooling2D, Conv2DTranpose as the deconvolution with regards to Conv2D, etc).

We trained all models with the same early stopping criteria and ensured that our perturbed data is created using a fixed random seed in order to make comparisons as deterministic as possible. Some slight variations were still in there due to the default glorot_uniform kernel initializer which we reduced later to he_uniform as suggested as the best overall go-to in recent publications. This also reduces the randomness of training runs since training partly took already pretty long and we wanted to gain reliable comparison results.

3) Results

Model 0:
The initial model is a not very deep standard encoder decoder architecture with two convolutions in the encoder and batch normalizations.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(1, activation='relu'))

# decoder
model.add(Dense(4, activation='relu'))
model.add(Reshape((2, 2, 1)))
model.add(Conv2D(4, (2, 2), activation='relu', padding='same'))
model.add(Conv2D(16, (2, 2), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((7, 7)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

Model 1:
Model 0 with more convolutional layers.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(1, activation='relu'))

# decoder
model.add(Dense(4, activation='relu'))
model.add(Reshape((2, 2, 1)))
model.add(Conv2D(4, (2, 2), activation='relu', padding='same'))
model.add(Conv2D(16, (2, 2), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((7, 7)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

Model 2:

Added more dense layers to model 1.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='relu'))

# decoder
model.add(Dense(4, activation='relu'))
model.add(Reshape((2, 2, 1)))
model.add(Conv2D(4, (2, 2), activation='relu', padding='same'))
model.add(Conv2D(16, (2, 2), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((7, 7)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

Model 3:

Added another convolutional layer with batch normalization to model 2.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='relu'))

# decoder
model.add(Dense(4, activation='relu'))
model.add(Reshape((2, 2, 1)))
model.add(Conv2D(4, (2, 2), activation='relu', padding='same'))
model.add(Conv2D(16, (2, 2), activation='relu', padding='same'))
model.add(Conv2D(32, (2, 2), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((7, 7)))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

Model 4:
Model 2 with Conv2DTranspose instead of Conv2D layers on the decoder.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='relu'))

# decoder
model.add(Dense(4, activation='relu'))
model.add(Reshape((2, 2, 1)))
model.add(Conv2DTranspose(4, (2, 2), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (2, 2), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((7, 7)))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(1, (3, 3), activation='relu', padding='same'))
```

Model 5 to 8.1:
With these models we tested out which number of nodes in the bottleneck layer performs the best. For this we fixed the general model structure and just changed the number of nodes in the middle layer. The structure looks as follows:

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(49, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(20, activation='relu'))

model.add(Dense(7, activation='relu'))

# decoder
model.add(Dense(20, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(49, activation='relu'))
model.add(Reshape((7, 7, 1)))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(1, (3, 3), activation='linear', padding='same'))
```

We tested a range of different numbers from 3 to 7 nodes. Model 7 was built with 7 nodes in the bottleneck layer. As one can see, this model performed the worst with a validation loss of 0.1561. Lowering the number of nodes increased the model's performance significantly. For example, the

models 8.1 and 5 have 5 and 4 nodes in the bottleneck layer and showed the best performance. However, if one further decreases the nodes like to 3 in model 6, the losses already increase again.

Since the model 5 with 4 nodes showed the best performance, we fixed this value for the subsequent models.

Model 10:

Here we tried to deepen the model structure by adding some convolutional layers. As one can see, this further increased the model's performance.

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(49, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(20, activation='relu'))

model.add(Dense(4, activation='relu'))

# decoder
model.add(Dense(20, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(49, activation='relu'))
model.add(Reshape((7, 7, 1)))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(1, (3, 3), activation='linear', padding='same'))
```

Model 11:

Since a deeper network seems to work better, we added two convolutional layers with 16 feature maps at the encoder and decoder stage. However, the training becomes very time consuming, since one epoch already takes quite some time and the convergence of the model's loss takes more epochs than before. However, the performance was increased again by deepening the model.

Model 12:

Here we tried to deepen the model at its dense layers. Therefore, we extended model 11 by adding two Dense layers with 10 nodes at the decoder and encoder stage. This led to a way better performance than we had achieved before with a validation loss of 0.0744. The model's structure looks as follows:

```
# encoder
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(28, 28, 1)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D((2, 2)))
model.add(BatchNormalization())
model.add(Flatten())
model.add(Dense(49, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(10, activation='relu'))

model.add(Dense(4, activation='relu'))

# decoder
model.add(Dense(10, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(30, activation='relu'))
model.add(Dense(40, activation='relu'))
model.add(Dense(49, activation='relu'))
model.add(Reshape((7, 7, 1)))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(16, (3, 3), activation='relu', padding='same'))
model.add(BatchNormalization())
model.add(UpSampling2D((2, 2)))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2DTranspose(1, (3, 3), activation='linear', padding='same'))
```

Model 13:

Since the last model already performed quite well we tried to add some regularizers in order to close the gap between the training and validation loss. In this model we just added two L2 regularizers with a factor of 0.001 at the first and last convolutional layer of model 12. The gap between training and validation loss got closer, but the value of the validation loss increased in comparison to the previous model.

Model 14:

For this model we added two Dropout layers with a dropout rate of 0.1 at the beginning and at the end of model 12. This led again to worse results than before without regularizers.

Model 15:

For this model we tried out a different approach than before reducing the dimensions early by directly applying max pooling layers after the first conv2d layer. Typical are either conv2d directly followed by relu and max pooling or stacking several conv2d layers on top of each other and then followed by relu and max pooling for these kinds of tasks. We also reduced the number of dense layers and added dropout layers with factor 0.5 which is very common. This brought down dimensions quickly since model runs took quite long already before.

We added he_uniform kernel initialization which has been recommended in recent publications as a go-to-default and usage of stochastic dropout layers after the pooling layers which showed best performance in similar tasks according to some publications. Performance dropped a little bit compared to recent very deep architectures we tried out but training was ridiculously fast in

comparison. Models 16 – 19 use a very similar layout with only slight modifications, hence we didn't copy the whole models here into the report.

```
# encoder
model.add(Conv2D(filters=32,  # number of convolutional kernels/channels; filters =
                 kernel_size=(3, 3),  # typical values in modern architectures: (1,
                 # 3) instead of once (5, 5) is a modern efficient approach)
                 strides=(1, 1),  # using the smallest convolutional stride since o
                 padding='same',  # using 'same' instead of default 'valid' padding
                 activation='relu',  # relu is considered a standard for conv layer
                 activity_regularizer=weight_regularizer,
                 kernel_initializer='he_uniform',  # good practice for convolution:
                 # 2)/num_inputs
                 input_shape=(28, 28, 1)))  # needs to be defined since it's the fi
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(Dropout(normal(0.2, 0.05)))  # convolutional dropout using normal distrib
model.add(Conv2D(filters=32,  # number of convolutional kernels/channels; filters =
                 kernel_size=(3, 3),  # typical values in modern architectures: (1,
                 # 3) instead of once (5, 5) is a modern efficient approach)
                 strides=(1, 1),  # using the smallest convolutional stride since o
                 padding='same',  # using 'same' instead of default 'valid' padding
                 activation='relu',  # relu is considered a standard for conv layer
                 activity_regularizer=weight_regularizer,
                 kernel_initializer='he_uniform'))  # good practice for convolution
# 2)/num_inputs
model.add(MaxPooling2D((2, 2), padding='same'))
model.add(Dropout(normal(0.2, 0.05)))  # convolutional dropout using normal distrib
model.add(Flatten())  # flatten convolution output in order to feed into affine lay
model.add(Dense(128, activation='relu', activity_regularizer=weight_regularizer))
# representation
model.add(Dropout(0.5))  # typical dropout factor for dense layers

# latent space
model.add(Dense(latent_space_dim, activation='relu'))  # encoder output
```

```
# decoder
model.add(Dense(49, activation='relu', activity_regularizer=weight_regularizer))
model.add(Dropout(0.5))
model.add(Reshape((7, 7, 1)))  # inverse operation of flatten in encoder
model.add(UpSampling2D((2, 2)))  # inverse operation of MaxPooling2D in encoder
model.add(Conv2DTranspose(filters=32,  # number of convolutional kernels/channel
                          kernel_size=(3, 3),  # typical values in modern archit
                          # 3) instead of once (5, 5) is a modern efficient appr
                          strides=(1, 1),  # using the smallest convolutional st
                          padding='same',  # using 'same' instead of default 'va
                          activation='relu',  # relu is considered a standard fo
                          activity_regularizer=weight_regularizer,
                          kernel_initializer='he_uniform'))  # good practice for
# 2)/num_inputs
model.add(Dropout(normal(0.2, 0.05)))  # convolutional dropout using normal dist
model.add(UpSampling2D((2, 2)))  # inverse operation of MaxPooling2D in encoder
model.add(Conv2DTranspose(filters=1,  # number of convolutional kernels/channels
                          kernel_size=(3, 3),  # typical values in modern archit
                          # 3) instead of once (5, 5) is a modern efficient appr
                          strides=(1, 1),  # using the smallest convolutional st
                          padding='same',  # using 'same' instead of default 'va
                          activation='relu',  # relu is considered a standard fo
                          activity_regularizer=weight_regularizer,
                          kernel_initializer='he_uniform'))  # good practice for
# 2)/num_inputs
model.add(Dropout(normal(0.2, 0.05)))  # convolutional dropout using normal dist
```

Model 16:

We attempted to fine-tune the lower dimension model approach from model 15 and maybe getting it to perform similar to the very deep network architecture from before. While a large batch size improved model training time, using too large of a batch size hinders model convergence. This model approach is really fast to train already, so we tried out different batch sizes and found that a batch size of 32 worked really well (compared to 512 from the deep networks from before). We also tried to double the previously low dimension of the latent space vector from 4 to 8. This resulted in improved performance.

Model 17:

Since previously we already tried out different amount of conv and dense layers and played around with neuron dimensions a lot, we decided to try out different scaling with the latent space vector which denotes the most important hyperparameter for an autoencoder. Setting it too low can lead to an information bottleneck while making it too large can result in bad generalization. Especially since we don't use any kind of additional information about the hidden states of the encoder (e.g. such as when using an attention mechanism) getting the latent space dimension right is of utter importance. Latent space dimensions were double here again to 16 now which resulted in a slight drop in performance.

Model 18:
Our image input dimensions are 28x28, hence flattened 784 pixels which could contain vital information. Based on our dataset we know that only part of the image contains useful information which needs to be encoded. Additionally, latent space can compress data additionally. We attempted increasing dimensions further to see how it performs. What we found is that going beyond (32, 64, 128) decreased performance. The decrease was not significant, but training times increased as well due to having more dimensions. Hence, we report the results for latent space dimension 32 here and kept it low to recently well working latent space dimensions (4 and 8).

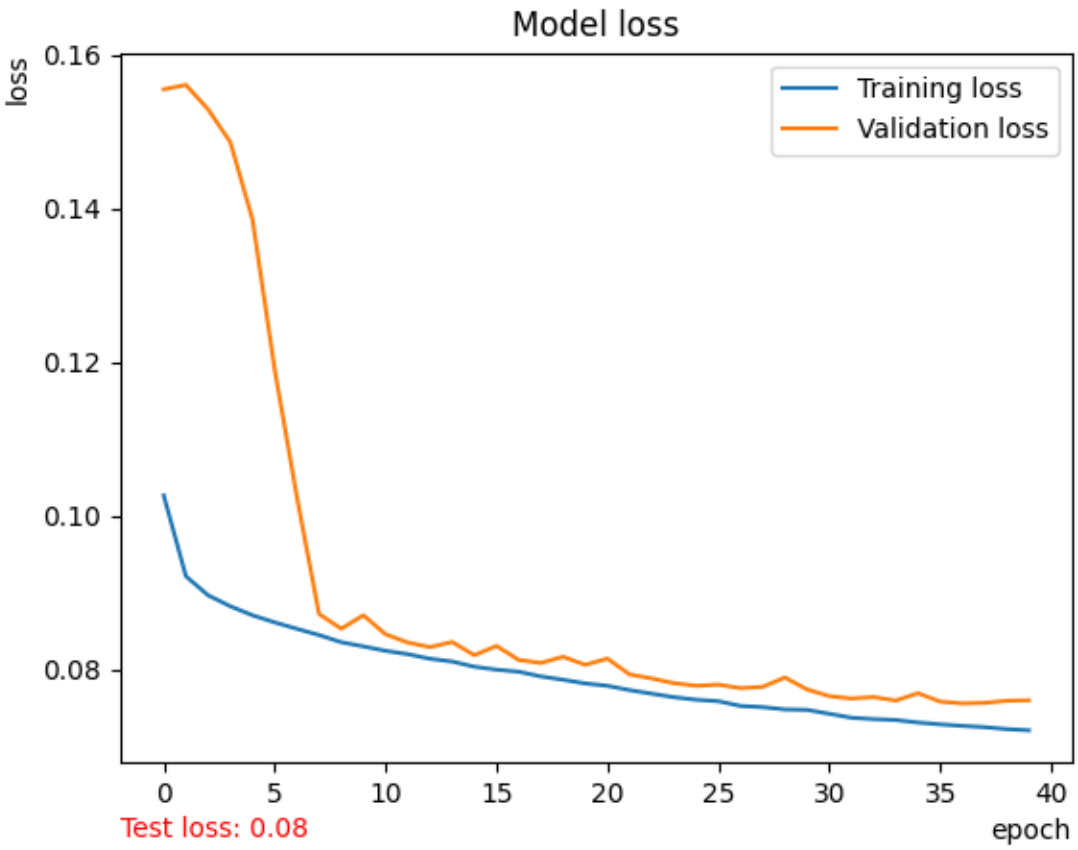
Model 19:
So far this new approach used a recommended working strategy of activity regularizer, stochastic dropout after pooling layers and fixed dropout layers for all dense layers. This approach was also based on findings in assignment 3 in which we dealt with the regular fashion mnist dataset and played around with many approaches. However, this dataset is a little bit more challenging. Hence, we completely replaced our regularization techniques with a very common strategy. We used only batch normalization after each pooling layer and fixed dropout layers for the dense layers.

Table of all models:

Model Nr.	Training loss	Validation loss
0	0.1	0.1
1	0.09	0.09
2	0.09	0.091
3	0.109	0.109
4	0.111	0.112
5	0.0856	0.0865
6	0.1158	0.1506
7	0.1113	0.1561
8	0.1088	0.1089
8.1	0.0880	0.0886
9	0.1091	0.1224
10	0.0820	0.0837
11	0.0818	0.0830
12	**0.0717**	**0.0756**
13	0.0771	0.0788
14	0.0730	0.0758
15	0.1574	0.1575
16	0.1413	0.1421
17	0.1631	0.1634
18	0.1601	0.1611
19	0.0901	0.1408

Model 12 performed best with a validation loss of **0.0756**.

Evolution of training losses:



The behavior we see here with the evolution of losses looked similar for all models with deeper architectures. They reached a decent performance soon but continued improving over long training periods (2-3 hours). The deeper models always fluctuated slightly with the validation loss during training but became better over time.

In comparison the simpler architectures (less capacity models) converged quickly and had a very constant validation loss over the training epochs without any visible fluctuations. This might be an effect of the stronger regularization applied to the larger capacity models. However, the deeper ones performed better.

4) Final model architecture

We chose model 12 as our final model architecture since it performed clearly the best. The exact model structure can be seen in the short description of the model above. The model's summary looks as follows:

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 32)	320
conv2d_1 (Conv2D)	(None, 28, 28, 32)	9248
conv2d_2 (Conv2D)	(None, 28, 28, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 14, 14, 32)	0
batch_normalization (BatchNormalization)	(None, 14, 14, 32)	128
conv2d_3 (Conv2D)	(None, 14, 14, 16)	4624
conv2d_4 (Conv2D)	(None, 14, 14, 16)	2320
conv2d_5 (Conv2D)	(None, 14, 14, 16)	2320
conv2d_6 (Conv2D)	(None, 14, 14, 16)	2320
max_pooling2d_1 (MaxPooling 2D)	(None, 7, 7, 16)	0
batch_normalization_1 (BatchNormalization)	(None, 7, 7, 16)	64
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 49)	38465
dense_1 (Dense)	(None, 40)	2000
dense_2 (Dense)	(None, 30)	1230
dense_3 (Dense)	(None, 20)	620
dense_4 (Dense)	(None, 10)	210
dense_5 (Dense)	(None, 8)	88
dense_6 (Dense)	(None, 10)	90
dense_7 (Dense)	(None, 20)	220
dense_8 (Dense)	(None, 30)	630
dense_9 (Dense)	(None, 40)	1240
dense_10 (Dense)	(None, 49)	2009
reshape (Reshape)	(None, 7, 7, 1)	0
batch_normalization_2 (BatchNormalization)	(None, 7, 7, 1)	4
up_sampling2d (UpSampling2D)	(None, 14, 14, 1)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 16)	160
conv2d_transpose_1 (Conv2DTranspose)	(None, 14, 14, 16)	2320
conv2d_transpose_2 (Conv2DTranspose)	(None, 14, 14, 16)	2320
conv2d_transpose_3 (Conv2DTranspose)	(None, 14, 14, 16)	2320
batch_normalization_3 (BatchNormalization)	(None, 14, 14, 16)	64
up_sampling2d_1 (UpSampling2D)	(None, 28, 28, 16)	0
conv2d_transpose_4 (Conv2DTranspose)	(None, 28, 28, 32)	4640
conv2d_transpose_5 (Conv2DTranspose)	(None, 28, 28, 32)	9248
conv2d_transpose_6 (Conv2DTranspose)	(None, 28, 28, 32)	9248
conv2d_transpose_7 (Conv2DTranspose)	(None, 28, 28, 1)	289

Total params: 108,007
Trainable params: 107,877
Non-trainable params: 130

This model is the deepest of all tested models. In the encoder part it consists of 3 + 4 convolutional layers, which are separated by a MaxPooling layer and a BatchNormalization layer. Afterwards the nodes get flatten and 5 Dense layers are following. This marks the end of the encoder part. The number of nodes of the bottleneck layer was set to 4. The decoder part is the exact counterpart of the encoder. Hence, the whole model is "symmetric" around the bottleneck layer. However, we used transposed convolutional layers instead of the regular ones and we replaced the MaxPooling layers by upsampling layers. We designed the whole structure such that the model only gets smaller or stays the same size with deeper layers until we reach the bottleneck. Then we just get larger again until the output. This should lead to a good compression of the input image. For all activation functions except the output layer we chose Relu activations, since they are a good working standard. For the output we decided to use a linear activation since the problem of removing disturbances is kind of a regression problem. For the training we used an Adam optimizer and a MSE loss function. Furthermore we use an EarlyStopping callback with a patience of 3 epochs.

5) Discussion

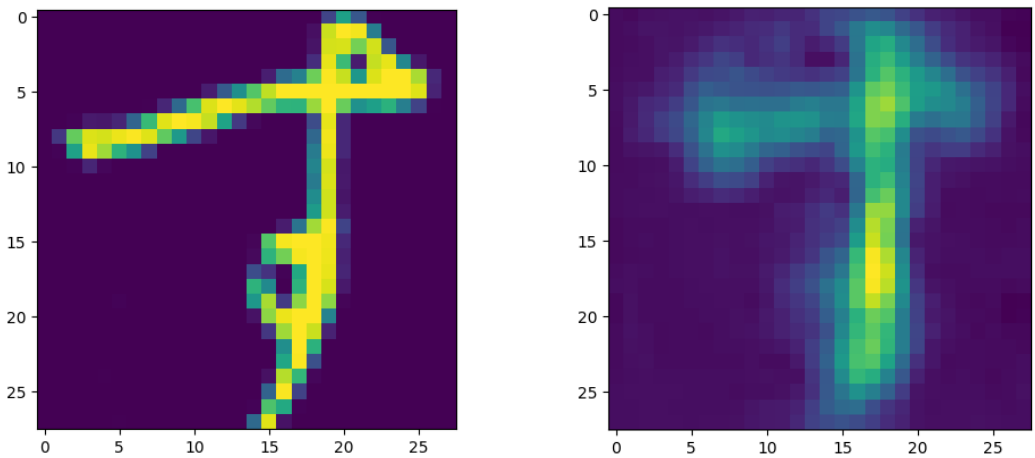
Initially we started out with the fashion mnist dataset. For this dataset we found that architecture don't need to be too deep in order to perform well. Using 2 layers of conv2d layers and two dense layers already perform really well.

One observation that we made is that once we switched to the Kuzushiji that much deeper network architectures were needed. The characters in this dataset are much more complex and this was expected. We had to stack several conv2d layers on top of each other before reducing dimensions via pooling layers. Furthermore, we had to add more dense layers. We didn't need extreme amounts of neurons in each layer, but the current trend of "deeper is better" worked here as well. We are confident that further improvements could be made by adding even more layers with corresponding stronger regularization but some models took us already 2-3 hours for a training run and we didn't have powerful GPUs to try out deeper architectures in a realistic scenario.

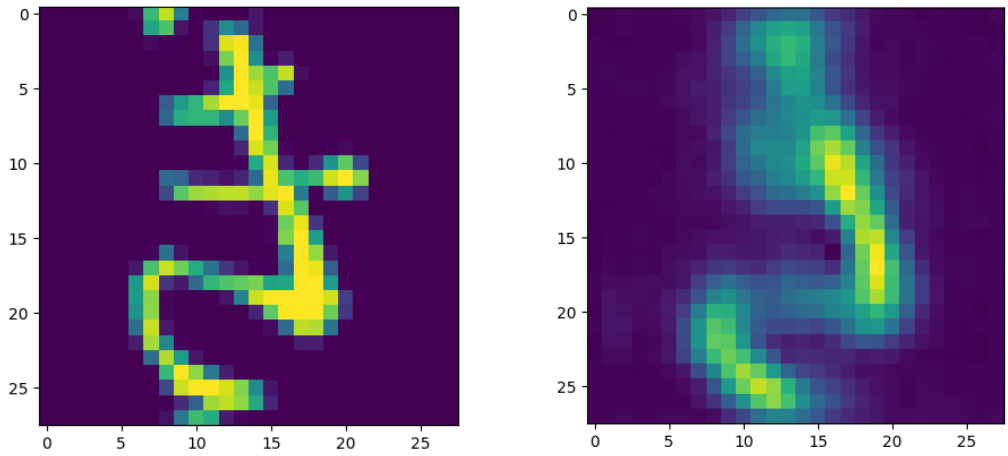
Defining the dimension of the latent space vector is also key to building an autoencoder. Being too low it may hinder the performance due to acting as an information bottleneck, if it is set too high it might not generalize too well after some point. We were surprised to find that very low latent space dimensions worked best for us suggesting that model compression worked really well and that higher dimensions were getting more in the way for good generalization results.

We found that while regularization was necessary when increasing the model capacity sufficiently, the different types of regularization didn't matter all that much. The key point seemed to be just adding the right amount of regularization preventing us from model overfitting but also not overdoing it (e.g. combining activity regularization with batch normalization already proved too strong in combination) which prevents the model from learning the differences. Comparison of same architectures just with very different regularization approaches (batch normalization, activity regularization, kernel normalization, dropout normal and stochastic dropout) performed surprisingly equal only differing slightly.

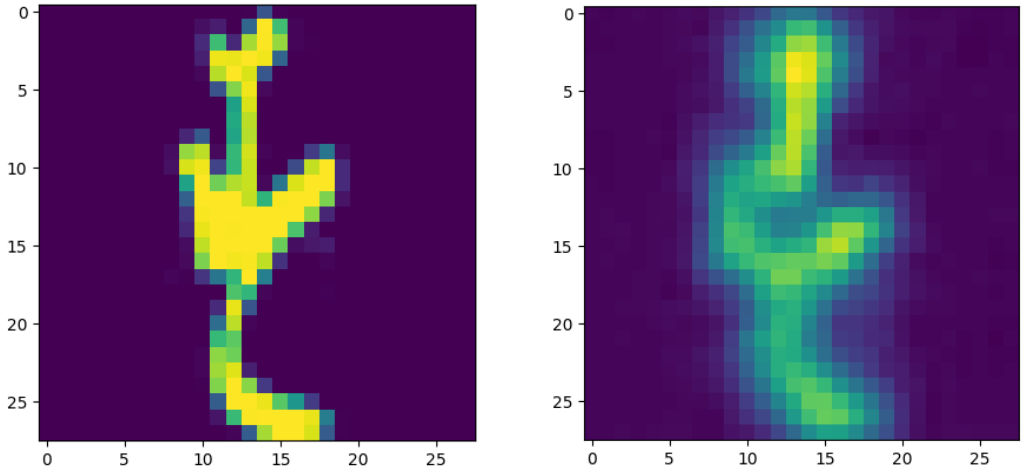
In total, we believe that making the model more complex seemed to provide the largest benefits but at some point we were restricted by hardware. We stopped increasing model capacity once one model run took us more than 2-3 hours to make the assignment doable.



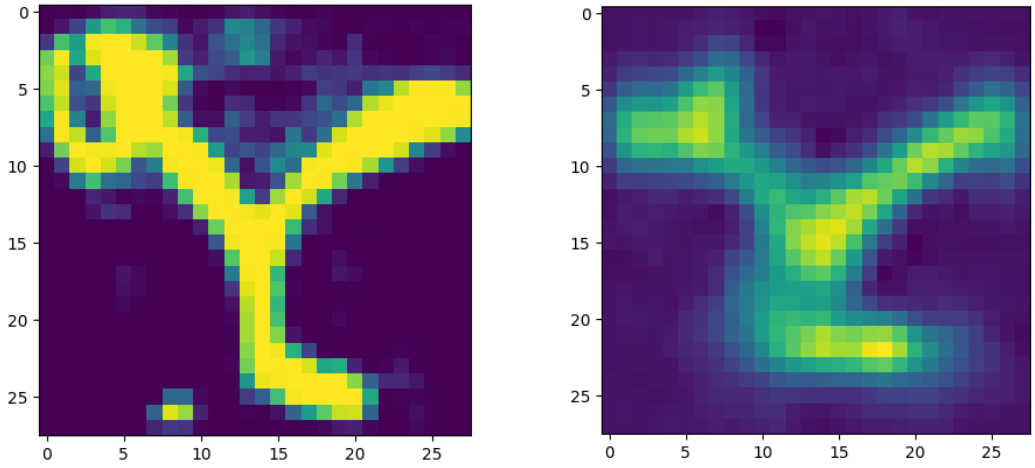
Result 1: Test sample with smallest loss (best case)



Result 2: Test sample with median loss



Result 3: Test sample with worst loss



Result 4: Random sample from validation set

We compared our generated/reconstructed samples from the autoencoder and compared them side by side with their corresponding clean image samples which were used as a basis for perturbation in order to be able to visually interpret how well our final model performs on the reconstruction task. We also included 3 of these reconstructed vs clean image tuples in the report which we chose by evaluating the model on the sample which performed best, worst and the median based on the test loss MSE metric when doing the final evaluation on the test set. This shows how well the final model visually in the best case, worst case and a regular case scenario.

We manually had a look at other indices as well and what we found is that all samples we looked at exclusively had the same overall structure even though we included flipped and black patches in the input samples. Since we created the perturbed data in a balanced dataset fashion (i.e. all pertubations had the same amount of samples so we had a balanced training set) via the modulo operator in the code, we could manually pick some samples with the right indices and compare them side by side. These findings suggest that the final model indeed managed to learn these more challenging aspects as well.

While the dataset itself only used greyscale values between 0 and 255 we plotted these images via matplotlib using their default pseudocolor rendering instead of setting it explicitly to greyscale and between ranges of 0 and 255. This allows for simpler visual inspection since we can discern slight color changes easier than slight shifts in greyscale in order to evaluate how well our model performs on the reconstruction task. The colors on all inspected samples were always the same and no notable shifts could be made out. This points to the model not performing any major incorrect reconstruction attempts.

At the same time the reconstructed images all look fuzzy in comparison (like applying a Gaussian blur in Photoshop). We assume that this is a result of training the model via loss penalizing for larger deviations in image values (MSE) so the model tried to stay as close to the original greyscale values as possible but there is no model metric in place enforcing clear separation (i.e. clearly defined image contrast edges). Higher input image resolution could help with that. Alternatively, we would potentially need a second loss term added for training which penalizes the model for no clear defined contrast edges.

Apart from this fuzziness the output seems really good and consistent when manually inspecting the output images. Only observing a low average test loss would not tell the whole story. While we could derive due to quadratic punishment of other grey values or larger shifts in pixel information if the overall shape is very different, it could easily be the case that the model fails to deal with certain perturbations and compensates with good results on others (hence we manually looked at our different modulo perturbations) or that there are certain discrepancies learned by the model which can be visually seen but don't necessarily lead to high losses (e.g. empty spot when using a black patch). This can be seen from other models in the industry when denoising or super-resolution model architectures amplify noise or create visual artifacts which can easily be noted by a human being while still having a low model loss overall.