Prof. Dr. Carsten Meyer / Lars Schmarje
Faculty of Engineering
Christian-Albrechts-Universität Kiel

Neural Networks and Deep Learning – Summer Term 2020

Exercise sheet 6

Submission due: Friday, July 03, 13:15 sharp

Exercise 1 (Transfer learning, CNN model library):

The following Jupyter notebook contains code to adapt a pre-trained model from the CNN model library (in this case MobileNetV2, pre-trained on ImageNet) to a different dataset (in this case the "cats_vs_dogs" dataset). Run the code, explain the individual steps and interpret the results. The code is based on https://www.tensorflow.org/tutorials/images/transfer_learning which may be consulted for reference.
Then, use a different model, e.g. 'ResNet101' instead of 'MobileNetV2', and a different data set, e.g. 'imagenette' instead of 'cats_vs_dogs' and report on your findings.

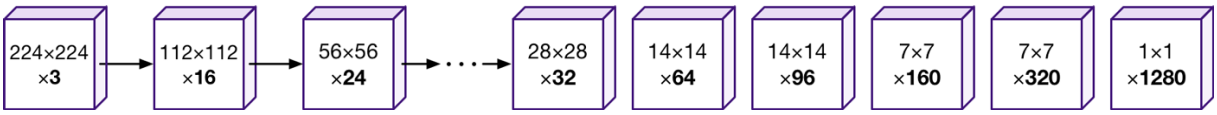
Solution:

First, the "cats_vs_dogs" data set is loaded. This data set has 23262 example images (as can be seen from the "metadata" description of the data set), organized in a single 'train' split (again found in the "metadata" description). Therefore, in order to obtain training, validation and test data, the first 80% of the data are used to define the training set, the next 10% for the validation set and the last 10% for the test set. There are 18610 images in the training subset.
Note that the pixel dimensions can vary between images; the first training image e.g. has a shape of 262 x 350 pixel, the second training image of 409 x 336 pixel etc. The mean pixel dimension over all training images is 365.05 x 410.36 pixel with a standard deviation of 96.32 x 107.78 pixel. The minimum and maximum pixel value in the training data is 0 and 255, respectively. The "cats_vs_dogs" data set defines a two-class classification problem; there are 9232 training images with non-zero label (i.e., label "1" corresponding to "dog"). Thus, there are 9378 images with label "0" corresponding to "cat".

After organizing the data into the three subsets, the data in all subsets are linearly normalized from the range [0, 255] to the range [−1, 1], and resized (using the Tensorflow "resize" method) to a fixed image size of 160 x 160. The training data are shuffled, and the training, validation and test data are organized in batches (of 32 images in the example). Thus, the shape of a training batch is 32 x 160 x 160 x 3.

Next, the MobileNetV2 is loaded. The second version of mobile net replaces the depthwise and pointwise convolution of the standard MobileNet by an expansion layer (to increase the number

of feature maps), a depthwise convolution layer (as before) and a projection layer (replaces the pointwise convolution and reduces the number of feature maps). In addition, residual connections are added (more information about both MobileNet versions can be found e.g. at https://machinethink.net/blog/mobilenet-v2/). Here is a simple diagram of MobileNetV2 (taken from the mentioned website):



The slightly different architecture of MobileNetV2 results in a slightly different output: If the input image has a dimension of 224 x 224 x 3 pixel (as is commonly used as input image size for the – differently sized – images of the ImageNet task), the output of the last convolutional layer (before the average pooling layer) has a size of 7 x 7 x 1024 (i.e., 1024 feature maps of size 7 x 7, see lecture slides). For the MobileNetV2 version the tensor sizes are given above. Note that both MobileNets use the ReLU6 variant (which has a maximal output value of 6): ReLU6(x) = min(max(0,x), 6). This is claimed to be more robust than regular ReLU.

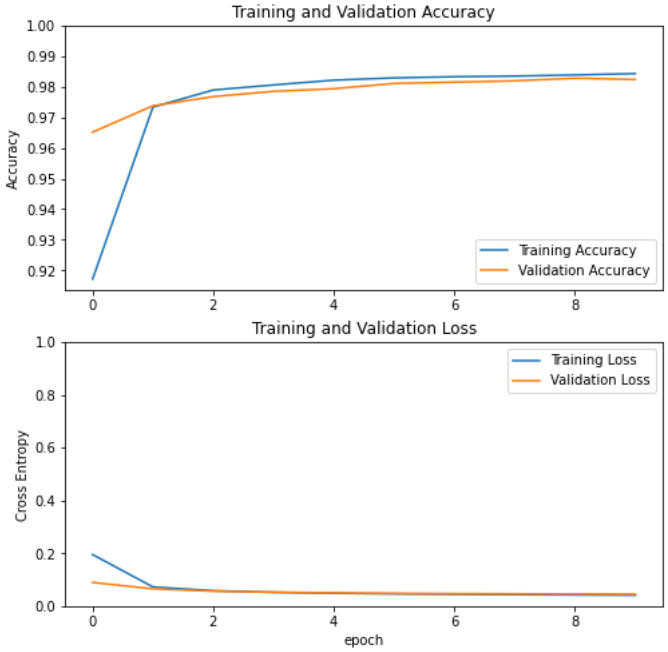
For transfer learning, we are going to remove the "classification head" of the MobileNet, i.e., use only the first part of the network with all layers up to the last convolutional layer. This is achieved by setting the `include_top=False` parameter. Furthermore, we are going to use the network weights which were pre-trained on the ImageNet task: `weights='imagenet'.` This defines the "base model" for transfer learning.

If we input an image of size 160 x 160 pixel to this network (in contrast to the commonly assumed size of 224 x 224 mentioned above), its output (before average pooling) will be downscaled by a factor of 32 (there are 5 convolutions with stride 2, see diagram above). Thus, the output size will be 5 x 5 pixel (160 / 32 = 5). Since there are 1280 feature maps in MobileNetV2 before average pooling (see the output of the model's `summary` method) and 32 images in a batch, one batch of images passed through the base model has an output size of 32 x 5 x 5 x 1280. The base model has 2.257.984, i.e., 2.26 million parameters, but no trainable parameters: In the first step of transfer learning, all weights are "frozen", i.e. shall not be trained on the new task: `base_model.trainable = False.`

The next step is to define a new classification head for the task at hand. In this example, the classification head consists of a global average pooling layer, followed by a fully connected layer with as many output neurons as there are classes in the classification problem at hand. Note that the "cats_vs_dogs" data set defines a two-class problem; here, we need only one output neuron. We choose to not apply a final sigmoid activation function (or a softmax activation function in case of a multi-class classification problem); instead, we directly use the "logits" output to predict the class and correspondingly set `from_logits=True.` The base model, the global average pooling layer and the final prediction layer compose the sequential model to be used for our task. For a two-class classification problem, we use the binary cross-entropy loss function; for a multi-class problem, we can use the sparse categorical cross-entropy loss when directly using integers as class labels (as is done here). One batch of input images now has a size of 32 x 1280 after global average pooling (since the 5 x 5 feature map is spatially averaged, independently for each input image and feature map) and 32 x 1 after the prediction layer (since there is only 1 output neuron for the two-class "cats_vs_dogs" task). This model can be evaluated on the test data (which, however, makes limited sense since the new classification head is not trained yet).

After defining the model for the task at hand, it can be trained on the corresponding training data (of the new task). More precisely, only the parameters of the layers of the new classification head are being trained (since the base model was "frozen", see above). The model summary shows that the 2.26 million parameters of the base model are non-trainable, and that there are 1.281 trainable parameters (1280 weights plus one bias of the final fully connected layer) which are trained on the "cats_vs_dogs" training data defined above).

Learning curves of training phase (using RMSProp optimizer with initial learning rate 0.0001):



"cats_vs_dogs", MobileNetV2 base model:

Training phase:
Final training loss (last epoch): 0.0411
Final training accuracy (last epoch): 0.9843
Final validation loss (last epoch): 0.0448
Final validation accuracy (last epoch): 0.9824
About 59s per epoch (100µs per step) training time

The trained model is then evaluated on the validation and test data; it already performs quite well on the new task (see the loss and accuracy curves and numbers on the validation data above and on the test data in the summary table below).

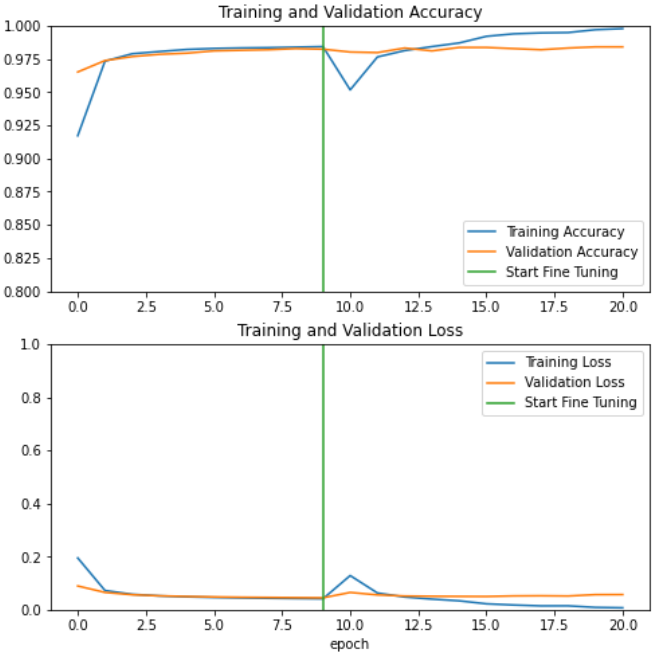
The final step consists in fine-tuning of the model parameters. To this end, the parameters of the base model are "unfrozen": `base_model.trainable = True.` However, since the first layers of the base model represent quite general features which are useful for the new task as well, the idea is not to fine-tune all layers of the model. Instead, the first `fine_tune_at_layer` number of layers (in this example, the first 100 out of 155 model layers) are still frozen and only the weights of the subsequent layers are being modified by gradients computed on the training data of the new task. The model summary reveals that out of the 2.259.265 total model parameters 1.863.873 (82.5%) parameters are trainable in the fine-tuning phase, whereas 395.392 parameters are frozen (non-trainable). 100 Layers are frozen and 58 layers are trainable (note the remarks regarding the number of layers given in the lecture).

Furthermore, it is important to reduce the learning rate for the fine-tuning (in this example, a ten times smaller learning rate than in the training phase is used in the fine-tuning phase). Otherwise, the computed gradients would be too large and "destroy" the pre-trained weights. The fine-tuning iterations start at the last iteration of the training phase, i.e., continue training.

Note that the use of batch normalization might require special attention, since batch normalization involves trainable parameters (when accumulating feature statistics). It might be that the feature statistics accumulated during pre-training of the network does not match the

feature statistics of the new task at hand at all, resulting in low performance of the final model. It can also be advisable to *not* "re-learn" the trainable batch normalization parameters on the new task (i.e., to use the parameters from pretraining) if the learned scale match the new task. In Keras, the latter can be achieved by operating batch normalization in the "inference" mode. More information about this issue can be found at https://keras.io/guides/transfer_learning/.

Learning curves of fine-tuning phase (using RMSProp with *reduced* learning rate 0.00001):



"cats_vs_dogs", MobileNetV2 base model:

Fine-tuning phase:
Final training loss (last epoch): 0.0072
Final training accuracy (last epoch): 0.9979
Final validation loss (last epoch): 0.0570
Final validation accuracy (last epoch): 0.9841
About 66s per epoch (115µs per step) training time

It can be seen that the training loss and accuracy improve during the fine-tuning phase after an initial performance degradation. This however only marginally translates to the validation data, i.e. the fine-tuning overfits to the new task. Therefore, it seems to be advisable to restrict the number of parameters which are trainable during fine-tuning and thus to freeze even more layers in the fine-tuning phase. Also note since more layers have to be trained, the time for one epoch in fine-tuning is larger than in training.

More information can be found e.g. at
https://www.tensorflow.org/tutorials/images/transfer_learning
https://keras.io/guides/transfer_learning/

Using ResNet101 on "cats_vs_dogs":

This is achieved by replacing

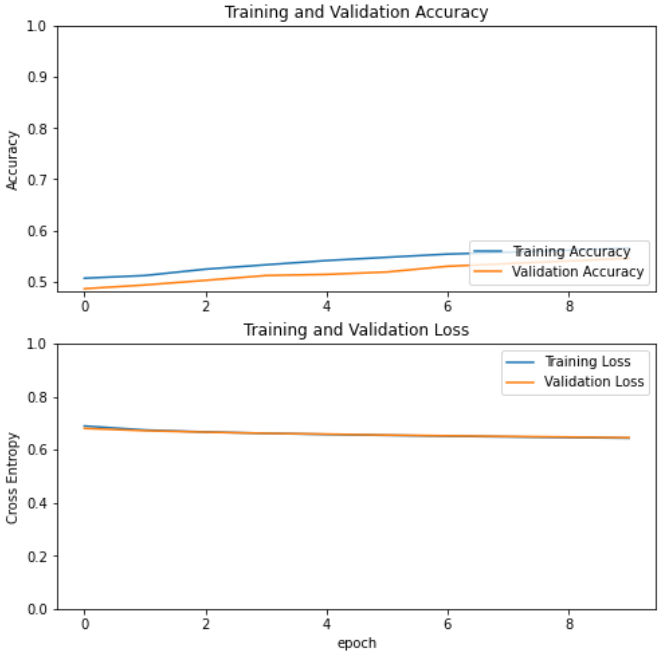
```
base_model = tf.keras.applications.MobileNetV2(input_shape=img_shape,
                                               include_top=False, weights='imagenet')
```

with

```
base_model = tf.keras.applications.ResNet101(input_shape=img_shape,
                                             include_top=False, weights='imagenet')
```

The last layer of the ResNet101 base model generates 2048 feature maps (instead of 1280 of the MobileNetV2 base model). Therefore, the classification head has 2049 trainable parameters. There are 42.658.176, i.e. 42.66 million parameters in the base model (instead of 2.26 million in case of MobileNetV2), plus the 2049 trainable parameters of the classification head. There are 345 layers in the model.

Learning curves of training phase (using RMSProp optimizer with initial learning rate 0.0001):



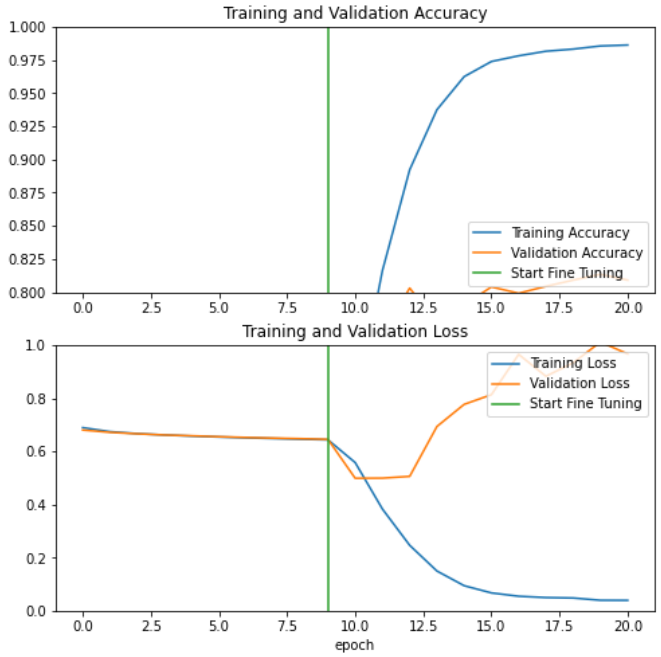
"cats_vs_dogs", ResNet101 base model:

Training phase:
Final training loss (last epoch): 0.6439
Final training accuracy (last epoch): 0.5654
Final validation loss (last epoch): 0.6456
Final validation accuracy (last epoch): 0.5456
About 109s per epoch (188µs per step) training time

It can be seen that the training phase only slightly improves the loss and accuracy on the training and validation data, arriving at unsatisfactory performance after 10 epochs.

In the fine-tuning phase, 296 layers are trainable (see the remarks regarding layer counting in Keras in the lecture); altogether, there are 38.473.217 (90.2%) trainable parameters and 4.187.008 non-trainable parameters.

Learning curves of fine-tuning phase (using RMSProp with *reduced* learning rate 0.00001):



"cats_vs_dogs", ResNet101 base model:

Fine-tuning phase:
Final training loss (last epoch): 0.0390
Final training accuracy (last epoch): 0.9862
Final validation loss (last epoch): 0.9677
Final validation accuracy (last epoch): 0.8091
About 216s per epoch (370µs per step) training time

The plots show that the training loss and accuracy are dramatically improved after fine-tuning. Interestingly, the validation loss is increased during fine-tuning, while the accuracy significantly improves (although not reaching the level of the MobileNetV2). Presumably the ResNet101 model is too big for the "cats_vs_dogs" two-class classification problem, so that the MobileNetV2 model is much better suited.

Using MobileNetV2 on "imagenette":

The "imagenette" data set is a subset of 10 easily classified classes from ImageNet. It can be loaded via

```
(raw_train, raw_validation, raw_test), metadata = tfds.load(
    'imagenette',
    split = ['train[:80%]', 'train[:80%]', 'validation'],
    with_info = True,
    as_supervised = True)
```

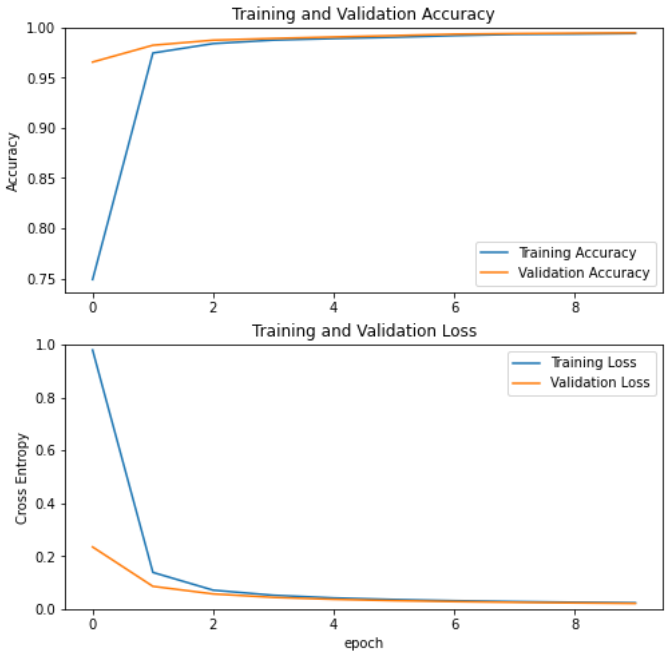
The number of classes has to be adapted accordingly:

```
num_classes = 10
```

Furthermore, since this is a multi-class classification problem, we are using the sparse categorical cross-entropy loss (instead of the binary cross-entropy loss). This has also to be changed when assessing the training history to generate training curves.

There are 10315 training images with largely varying shapes (shape of first training image: 101 x 125 x 3 pixel, shape of second training image: 375 x 500 x 3 pixel). The mean pixel dimension of the training images is 407.09 x 471.20 pixel with standard deviation of 205.94 x 231.99 pixel). Note that the pixel value range ([0, 255]) is the same as in "cats_vs_dogs" such that the image-preprocessing is still valid.
Since there are now 10 output neurons, the classification head has 1280*10 + 10 = 12.810 parameters. Otherwise, the number of parameters are unchanged compared to above.

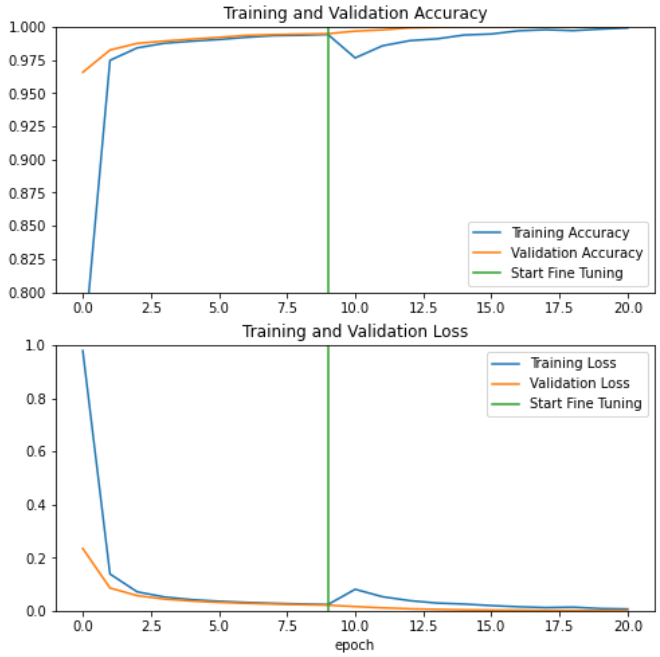
Learning curves of training phase (using RMSProp optimizer with initial learning rate 0.0001):



"imagenette", MobileNetV2 base model:

Training phase:
Final training loss (last epoch): 0.0232
Final training accuracy (last epoch): 0.9942
Final validation loss (last epoch): 0.0208
Final validation accuracy (last epoch): 0.9948
About 64s per epoch (200µs per step) training time

Learning curves of fine-tuning phase (using RMSProp with *reduced* learning rate 0.00001):



<u>"imagenette", MobileNetV2 base model:</u>

Fine-tuning phase:
Final training loss (last epoch): 0.0064
Final training accuracy (last epoch): 0.9991
Final validation loss (last epoch): 0.0005
Final validation accuracy (last epoch): 1.0000
About 67s per epoch (207µs per step) training time

There is not much to improve during fine-tuning since the task has already nearly been solved after training, so the improvements obtained by fine-tuning are rather small.

Summary of evaluation results on validation data:

Task	Base model	Before training		After training		After fine-tuning	
		loss	accuracy	loss	accuracy	loss	accuracy
cats_vs_dogs	MobileNetV2	0.6451	0.6281	0.0448	0.9824	0.0570	0.9841
	ResNet101	0.7226	0.5211	0.6456	0.5456	0.9677	0.8091
imagenette	MobileNetV2	2.7837	0.0976	0.0208	0.9948	0.0005	1.0000

Summary of evaluation results on test data:

Task	Base model	Before training		After training		After fine-tuning	
		loss	accuracy	loss	accuracy	loss	accuracy
cats_vs_dogs	MobileNetV2	0.6543	0.6156	0.0428	0.9837	0.0487	0.9832
		0.7317	0.5129	0.6376	0.5684	0.9765	0.8143
imagenette	MobileNetV2	2.7837	0.1140	0.0759	0.9800	0.0749	0.9820

Note that it is not guaranteed to obtain good results with any network on any data set. Many factors may influence performance (preprocessing, image size, task properties and difficulties, size and quality of training data, receptive field considerations, usage of batch normalization, choice of meta-parameters etc.). In case of failure, these factors have to be checked in detail.

Exercise 2 (Simple recurrent neural network for character sequences):

The python script in the Jupyter notebook, which is based on the script **`min-char-rnn.py`** written by Andrej Karpathy (see https://gist.github.com/karpathy/d4dee566867f8291f086), implements a simple recurrent neural network applied to predict the next character in a sequence of characters, based on the current character (see also the lecture slides).

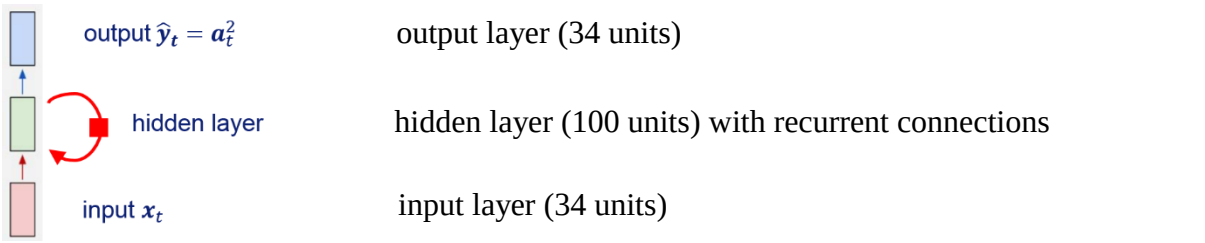
a) Depict a diagram of the recurrent neural network (including the number of neurons in each layer, the activation functions, the update equations and the dimensions of the matrices and vectors involved). Describe in detail how the network processes a sequence of characters in training (what is input, what is output, unfolding) and how the network generates a sequence of characters. Run the script (on the input file **`input_short.txt`**) and describe the output.

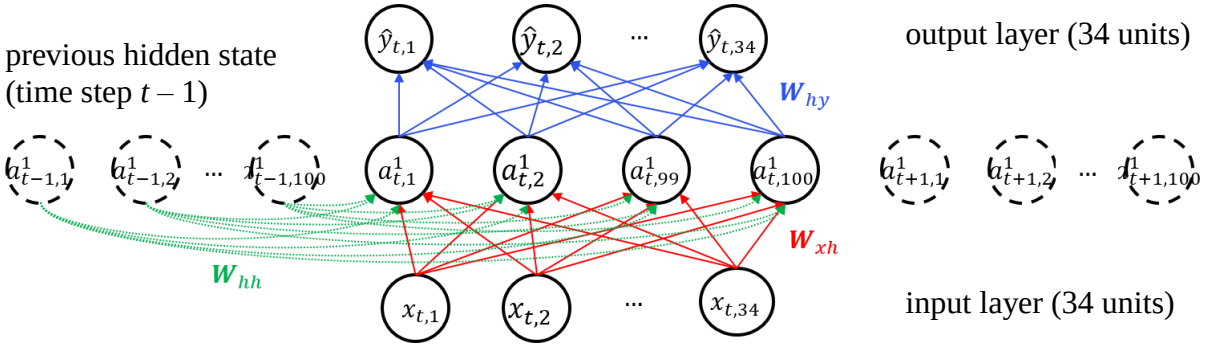
Solution:

The recurrent network has the following topology:

- Number of input units: As many units as there are different characters in the input file **`input.txt`**; in the example provided, there are 34 different characters (where upper case and lower case are distinguished, i.e. 'W' and 'w' are two different characters, and also punctuation symbols like '.', ',' and ' ' count as individual characters), so there are 34 input units.
- Number of hidden units: The network has **`hidden_size`** hidden units; in the example provided, there are per default 100 hidden units. There is only a single hidden layer in the network. The hidden layer uses a "tanh" activation function.
- Number of output units: There are as many output units as input units; in the example provided, there are 34 different output units. The output layer uses a softmax activation function.

Simple and extended diagram of the network:





Notation:

$x_{t,2}$: input at time *t* (first subscript), second component (second subscript)
$a^1_{t,100}$: activation in first hidden layer (superscript) at time *t* (1st subscript), component 100 (2nd subscript)
$\hat{y}_{t,34}$: output at time *t* (first subscript), component 34 (second subscript)

Remarks:

- The input-hidden connections W_{xh} connect the 34 input units to the 100 hidden units (100 × 34 matrix) and are shown in red
- The recurrent connections W_{hh} connect the 100 hidden units, i.e. their values e.g. at time *t* – 1 to their value at time *t* (100 × 100 matrix); they are shown dashed green; only the connections from time step *t* – 1 to *t* are shown for clarity, indicating that the hidden state at time *t* – 1 influences the hidden state at time *t*; similarly, the hidden state at time *t* influences the hidden state at time *t* + 1 (shown to the right of the diagram above, but without synaptic connections for clarity of presentation)
- The hidden-output connections W_{hy} connect the 100 hidden units to the 34 output units (34 × 100 matrix) and are shown in blue

The forward update equations are given as follows (see lecture slides and the code in **`exercise2.py`**, lines 78 – 80):

$$\boldsymbol{z}_t^1 = \mathbf{W}_{xh}\boldsymbol{x}_t + \mathbf{W}_{hh}\boldsymbol{a}_{t-1}^1 + \boldsymbol{b}^1 \quad (1)$$
$$\boldsymbol{a}_t^1 = \tanh(\boldsymbol{z}_t^1) \quad (2)$$
$$\boldsymbol{z}_t^2 = \mathbf{W}_{hy}\boldsymbol{a}_t^1 + \boldsymbol{b}^2 \quad (3)$$
$$\hat{\boldsymbol{y}}_t = \mathrm{softmax}(\boldsymbol{z}_t^2) \quad (4)$$

$\boldsymbol{a}_{t-1}^1, \boldsymbol{a}_t^1, \boldsymbol{b}^1, \boldsymbol{z}_t^1$: 100-dim.
$\boldsymbol{x}_t$: 34-dim.
$\mathbf{W}_{xh}$: 100 × 34, $\mathbf{W}_{hh}$: 100 × 100, $\mathbf{W}_{hy}$: 34 × 100
$\boldsymbol{b}^2, \boldsymbol{z}_t^2$: 34-dim.
$\hat{\boldsymbol{y}}$: 34-dim.

Training proceeds as follows:

- At the beginning of training or after each complete pass over the input data, the hidden layer activations are initialised to $\boldsymbol{a}_0^1 = 0$ ($t = 0$) and the algorithm starts (again) at the beginning of the input file ($p = 0$).
- A sequence of **`seq_length`** input characters is constructed (in the example per default 25), here the first 25 characters of the file **`input.txt`**. For each input character, the corresponding target is the letter immediately following the input character. Therefore, the targets are a sequence of **`seq_length`** characters, which are shifted by 1 character with respect to the inputs. Both inputs and targets are converted to a one-hot-encoding (here 34-dimensional vectors, see above), so the (current) inputs are a sequence of **`seq_length`** 34-dimensional vectors, and similarly the (current) targets are a sequence of **`seq_length`** 34-dimensional vectors, shifted by 1 character. The goal is to predict a single output character (the next character in a sequence) from a single input character at each time step.
- Starting with the initial value $\boldsymbol{a}_0^1 = 0$, a forward pass is performed sequentially for the **`seq_length`** input characters (one after the other, propagating the hidden layer activations), i.e., the network is unfolded for **`seq_length`** time steps. For each input character $\boldsymbol{x}_t$ (in one-hot-encoding) at time *t*, the network output $\hat{\boldsymbol{y}}_t$ is computed (as

probability distribution, 34-dimensional vector) and compared to the target (34-dimensional vector, one-hot-encoding). The log-likelihood loss is calculated over the sequence of examples.

- Based on the calculated log-likelihood loss, the backward equations are applied to the network which is unfolded for **`seq_length`** time steps and the network parameters are updated (truncated backpropagation through time) using "AdaGrad". Clipping is applied to the network parameters in order to prevent too small or too large values of the parameters.
- Then, the sequence is shifted by **`seq_length`** characters. For the forward equations, the last hidden layer activations are used (instead of setting the hidden layer activations to 0, which only occurs when iterations start at the beginning of the file).
- This is performed for a number of iterations (in the example, maximally 10000 iterations).

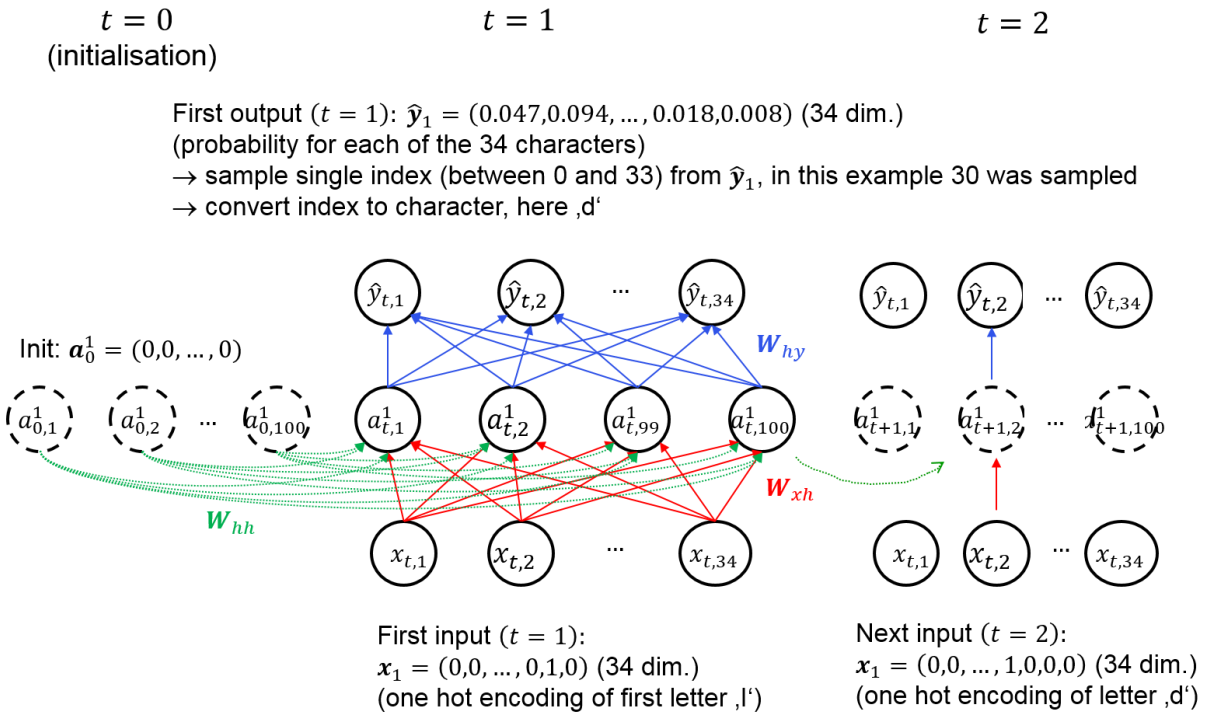
Since **`seq_length`** time steps are considered in a single backpropagation through time update step, the network has a "memory" of at most **`seq_length`** time steps. The next parameter update is then performed on the sequence shifted by **`seq_length`** time steps, where in the forward propagation the hidden layer activations are propagated, but in the backward equations, the individual sequences of **`seq_length`** time steps are treated independently from each other. To improve prediction performance, the **`seq_length`** should be made large; this, however, increases training complexity and slows down training.

Note that at each time, a *single* character is input to the network, *not* a sequence of characters. The latter would be the case if we would use a feedforward neural network, e.g. a multi-layer perceptron, to predict the sequence; in this case, we have to input a "history" of characters to the network, because otherwise the feedforward network had no chance of representing "long-range dependencies" in the character sequence beyond a single character. In case of a recurrent network, the history of characters is "remembered" within the hidden layer activations, which are passed via the recurrent connections from time step to time step in order to store the long-range dependencies. Together with the current input (the current character), the hidden layer activations must guide the network to predict the correct subsequent letter, depending on the history of characters "stored" in the hidden layer activations. Therefore, the hidden layer activations must be "rich" enough (i.e. there must be enough hidden neurons) to "store" the history of characters.

The network generates a sequence as follows (see also the diagram below):

- First, the hidden layer activations are initialised to $\boldsymbol{a}_0^1 = 0$ ($t = 0$).
- Then, at each time step, a single letter in form of a one-hot-encoding is input to the network. Here, there are 34 different characters (see above), so the input vector is 34-dimensional. The first input $\boldsymbol{x}_1$ ($t = 1$) corresponds to the first letter of either the current sequence, or of the file **`input.txt`**; in the latter case, the first letter is 'I'.
- Using the initial value $\boldsymbol{a}_0^1$ and the first input $\boldsymbol{x}_1$ as well as the current parameters for $\mathbf{W}_{xh}, \mathbf{W}_{hh}$ and $\boldsymbol{b}^1$, the postsynaptic potential $\boldsymbol{z}_1^1$ and the activation $\boldsymbol{a}_1^1$ at time $t = 1$ are calculated according to equations (1) and (2).
- With this values and the current parameters for $\mathbf{W}_{hy}$ and $\boldsymbol{b}^2$, the postsynaptic potential $\boldsymbol{z}_1^2$ and the output $\hat{\boldsymbol{y}}_1$ at time $t = 1$ are calculated according to equations (3) and (4). The output is a 34-dimensional vector containing probabilities for each of the 34 characters.

- From the output probabilities $\hat{\boldsymbol{y}}_1$, a single index is sampled and converted to the corresponding character, which is the first generated output, i.e. the character following the first character 'I'; in the example, the next character is 'd'.
- The predicted character (in form of a one-hot encoding) is then used as input at the next time step and the process is iterated until the end of the loop (in the example either 200 characters for the first letter of the current sequence and 100 characters for the first letter of the input file).



Note that the output $\hat{\boldsymbol{y}}_1$ has been (re-)calculated with the fully trained network.
The synaptic connections at time step t = 2 have only roughly been indicated; they correspond to the synaptic connections at time step t = 1.

Running the script **`exercise2.py`**, the following output was obtained (only the beginning and end of the output are shown; note that a random component is involved in the output):

```
data has 184 characters, 34 unique.
----
sample 200 letters from current position:
hukWVFldhb?bngVha
aGemfblijdzbIWkjVf           zoGwwD           t,bcinemVhdecDwm,,acGB
wbn.blHjBdbudzFwk, ?uHemzaWr.F dVt.?DkkrjumsdIFjmBcG
ltfankk
WnV f VWronogbIindl,kbkDno.s  rGBGz?DVBocD?F
BFFmFtjestWhawn
gFfodtBftbe
----
sample 100 letters from beginning of text:
I BWgnlglchFoH
fWtwmzFbozhF drFgdgfBeFahezWhoVnIejmsgasfVBDwm?Frk?GWob
```

```
fWuu?D?
fkteDrdcogsokdklt,oD?w
----

iter 0, loss: 88.159010
----
sample 200 letters from current position:
taenneiiz
 Dtjene           ,ansaeizeeaht,eazmener                          afaet
jtahcneneeGeneleeiascetrhnecercezeDenektes
awt,feasned,zD,ihs,ee,Gan iini
 rniee t tseekdiuD aieseetiseekaetlldeeaD,t teeeeeeeDueetdeihedten
seGreruaWf
----
sample 100 letters from beginning of text:
Ihr nbet,lt,
aeieateatseeteahac ea  eneettzt eee,lnDeen,rsatt
hnnnoa,aecaen eedutzfeeileica,ereneifie
----

iter 100, loss: 88.917367


...

sample 200 letters from current position:
hr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn genderuekeiderz noch jenem
----
sample 100 letters from beginning of text:
Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch
----

iter 9800, loss: 0.075354
----
sample 200 letters from current position:
ueh sich einst dem trueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gen Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mei
----
sample 100 letters from beginning of text:
Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch
----

iter 9900, loss: 0.073697
Training duration (s) :  26.78624725341797

Final test: Trying to reproduce training sequence; predicted text:
```

```
Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gesder, sch

[[   0.    1.    2. ...,  182.  183.  184.]
 [   1.    0.    1. ...,  181.  182.  183.]
 [   2.    1.    0. ...,  180.  181.  182.]
 ...,
 [ 183.  182.  181. ...,    6.    7.    8.]
 [ 184.  183.  182. ...,    7.    7.    8.]
 [ 185.  184.  183. ...,    8.    8.    8.]]

there are 8 errors
```

I.e., after 10000 iterations the network has learned the sequence quite well (with only 8 errors).

b) Run the script 10 times (you may modify the python script correspondingly) and report on your finding. Vary the configuration of the network and important parameters and assess the network performance based on 10 runs of each modified configuration for the input file **`input_short.txt`**. Report on your findings and conclusions.

Solution:

Running the script 10 times on **`input_short.txt`**, the following output was obtained (excerpt):

```
data has 184 characters, 34 unique.

Iteration number 1
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gestalten,

there are 7 errors

Iteration number 2
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch ich mein Her, such ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jen

there are 37 errors

Iteration number 3
Final test: Trying to reproduce training sequence; predicted text:
```

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gez noch wo

there are 9 errors

Iteration number 4
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich aedieschn gez noch jenem W, ni.
Versuch wohl, ich Her, dezenem Wahn gezeigt.
Versuch wohl, euch diesmal festzuhalten?
Fuehl i

there are 95 errors

Iteration number 5
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frues sich einst dem trueben Blick gezeiltnsHerz noch jenem Wahn gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein He

there are 60 errors

Iteration number 6
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Geaeben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gezeigt.
Versuch ich wohl, euch diesmal festzuh

there are 76 errors

Iteration number 7
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestesten,
Dioch jenen Her, schwankende Gestalten,
Die frueh sich einst dem trueben Blick gezeigt.
Veruch mhloch jeDich Gestalten,
Die frueh sich eVert

there are 97 errors

Iteration number 8
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gessin?
Fuehl ich mein Herz noch jenem Wahn geueben Blick gezeigt.
Versuch ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn geu

```
there are 41 errors

Iteration number 9
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich einst dem trswal ich wohl, euch diesmal festzuhalten?
Fuehl ich mein Her, schwankende Gestaht dem trsuch ich wohl, euch
diesm

there are 70 errors

Iteration number 10
Final test: Trying to reproduce training sequence; predicted text:

Ihr naht euch wieder, schwankende Gestalten,
Die frueh sich eich wohl, euch diesmal festzuhalten?
Fuehl ich mein Herz noch jenem Wahn gezeigt.
Versuch ich wohl, euch diesmal festzuhalte

there are 84 errors

Error summary:
[7.0, 37.0, 9.0, 95.0, 60.0, 76.0, 97.0, 41.0, 70.0, 84.0]
Mean error: 57.600000
Standard deviation: 32.945072

Training time summary:
[24.24244260787964,     23.89924192428589,     23.836841821670532,
23.852441787719727,     23.883641958236694,     23.774441719055176,
23.774441719055176,     23.94604206085205,     23.914842128753662,
25.833645582199097]
Mean error: 24.095802
Standard deviation: 0.624789
```

Note that the number of errors varies strongly between the different iterations. The training time is, however, quite comparable.

The results of testing different configurations are summarized in the following table:

hidden_size	seq_length	learning_rate	mean error	mean train time
100	10	0.1	63.50 ± 43.572	12.91 ± 1.310
100	25	0.1	57.60 ± 32.945	24.10 ± 0.625
100	50	0.1	31.90 ± 8.621	53.49 ± 2.674
100	100	0.1	63.80 ± 3.795	116.48 ± 4.858
50	50	0.1	53.10 ± 40.037	33.43 ± 4.272
200	50	0.1	115.50 ± 22.29	118.29 ± 3.74
100	50	0.2	77.50 ± 38.925	65.68 ± 2.790
100	50	0.05	37.50 ± 27.646	62.35 ± 4.93

The results are interpreted as follows:

- A shorter sequence length increases the mean error (while reducing mean training time), a larger sequence length improves results at the expense of a larger training time. If the sequence length is too large (here 100), results are getting worse again. This is potentially due to the short input file length, i.e. the sequence length should be much smaller than the length of the input file.
- Reducing the number of hidden neurons increases the mean error (while reducing mean training time), a larger number of hidden neurons improves results at the expense of a larger training time. Increasing the number of hidden neurons drastically increases the mean error, potentially due to the fact that the large number of parameters cannot be trained reliably anymore. Surprisingly, the training time increases only slightly.
- A larger learning rate deteriorates convergence, yielding a larger mean error at a larger training time. Reducing the learning rate slightly increased the mean error as well as the training time.
- Therefore, the optimal configuration (out of those tested) seem to be 100 hidden neurons, a sequence length of 50 and a learning rate of 0.1.

Exercise 3 (Long Short Term Memory LSTM, time series prediction):

a) Consider an LSTM network with a single hidden layer composed of two LSTM units, i.e. #hidden = 2, which receives three-dimensional inputs $\mathbf{x} = (x_1, x_2, x_3)^T$, i.e. the number of input features is #features = 3.

The synaptic weight matrices are given as follows:

Input gate:

$\mathbf{W}_{xi} = \begin{pmatrix} 2 & 0 & 3 \\ 4 & 1 & 0 \end{pmatrix}$ weights from input to input gate (#hidden × #features)

$\mathbf{W}_{hi} = \begin{pmatrix} -3 & 0 \\ -5 & 1 \end{pmatrix}$ weights from hidden units to input gate (#hidden × #hidden)

$\boldsymbol{b}_i = \begin{pmatrix} 0 \\ -2 \end{pmatrix}$ bias of input gate (vector of dimension #hidden)

Forget gate:

$\mathbf{W}_{xf} = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 1 & 2 \end{pmatrix}$ weights from input to forget gate (#hidden × #features)

$\mathbf{W}_{hf} = \begin{pmatrix} -2 & 0 \\ 2 & 1 \end{pmatrix}$ weights from hidden units to forget gate (#hidden × #hidden)

$\boldsymbol{b}_f = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$ bias of forget gate (vector of dimension #hidden)

Cell ("gate gate"):

$\mathbf{W}_{xg} = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$ weights from input to "gate" gate (#hidden × #features)

$\mathbf{W}_{hg} = \begin{pmatrix} 2 & 0 \\ -1 & 1 \end{pmatrix}$ weights from hidden units to "gate" gate (#hidden × #hidden)

$\boldsymbol{b}_g = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$ bias of "gate" gate (vector of dimension #hidden)

Output gate:

$\mathbf{W}_{xo} = \begin{pmatrix} -1 & 2 & -2 \\ 0 & 1 & 1 \end{pmatrix}$ weights from input to output gate (#hidden × #features)

$\mathbf{W}_{ho} = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix}$ weights from hidden units to output gate (#hidden × #hidden)

$\boldsymbol{b}_o = \begin{pmatrix} -1 \\ -1 \end{pmatrix}$ bias of output gate (vector of dimension #hidden)

Both the hidden layer activations and the cell activations shall be initialized with 0:

$$\boldsymbol{a}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \boldsymbol{c}_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} .$$

Further, there are four output units, i.e. #output = 4. The synaptic weight matrix between the hidden units and the outputs is given by

$$\mathbf{W}_{yh} = \begin{pmatrix} 2 & -1 \\ 0 & 3 \\ 1 & -1 \\ 4 & 0 \end{pmatrix}$$ weights from hidden unit activations to output (#output × #hidden)

$$\boldsymbol{b}_y = \begin{pmatrix} 2 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$ bias of outputs (vector of dimension #output)

For the output units, a ReLU activation function is used.

Calculate the outputs in a many-to-many scenario, i.e. an output is calculated for each input vector, at the time steps $t = 1$ and $t = 2$ for the input sequence

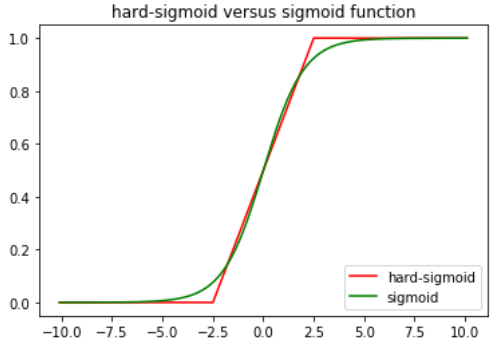
$$\boldsymbol{x}_1 = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \qquad \boldsymbol{x}_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

Optional: Draw a diagram of the LSTM network.

Note: Instead of the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ the so-called "hard-sigmoid" function σ_{hard} is being used at the input, forget and output gate, which is defined as follows:

$$\sigma_{hard}(z) = max\{0, min\{1, 0.2 * x + 0.5\}\}$$

$$\text{i.e. } \sigma_{hard}(z) = \begin{cases} 0 & for\ x < -2.5 \\ 0.2 * x + 0.5 & for -2.5 \leq x \leq 2.5 \\ 1 & for\ x > 2.5 \end{cases}$$



The python script in the Jupyter notebook demonstrates how this network can be implemented in Keras. You may use the script to check your solution. However, please also document intermediate steps in your calculations. Also be aware that the script uses a standard sigmoid activation function and *not* a hard-sigmoid; therefore, the script may produce slightly different results than those of your calculations (the error can be up to 0.1 for the second input; for every input, the deviation adds up and therefore increases over time).

Solution:

The update equations of the hidden LSTM units (with hard-sigmoid activation function at the input, forget and output gate) are given as follows:

$$\boldsymbol{i}_t = \sigma_{hard}(\mathbf{W}_{xi}\boldsymbol{x}_t + \mathbf{W}_{hi}\boldsymbol{a}_{t-1} + \boldsymbol{b}_i)$$
$$\boldsymbol{f}_t = \sigma_{hard}(\mathbf{W}_{xf}\boldsymbol{x}_t + \mathbf{W}_{hf}\boldsymbol{a}_{t-1} + \boldsymbol{b}_f)$$
$$\boldsymbol{o}_t = \sigma_{hard}(\mathbf{W}_{xo}\boldsymbol{x}_t + \mathbf{W}_{ho}\boldsymbol{a}_{t-1} + \boldsymbol{b}_o)$$
$$\boldsymbol{g}_t = \tanh(\mathbf{W}_{xg}\boldsymbol{x}_t + \mathbf{W}_{hg}\boldsymbol{a}_{t-1} + \boldsymbol{b}_g)$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t$$
$$\boldsymbol{a}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$$

Note that the hidden and the cell state have the same dimension (namely the number of hidden units, since each LSTM has a cell state in addition to the hidden state). By virtue of the update equations, all gates then also have the same dimension (there is one input gate, one forget gate, one "gate" gate and one output gate per hidden LSTM unit).

The equations to calculate the network output $\hat{\boldsymbol{y}}_t$ is given by

$\boldsymbol{z}_t = \mathbf{W}_{yh}\boldsymbol{a}_t + \boldsymbol{b}_y$ postsynaptic potential at output units from hidden unit activations
$\hat{\boldsymbol{y}}_t = \mathrm{ReLU}(\boldsymbol{z}_t)$ network output

Time step 1:

$$\boldsymbol{i}_1 = \sigma_{hard}(\mathbf{W}_{xi}\boldsymbol{x}_1 + \mathbf{W}_{hi}\boldsymbol{a}_0 + \boldsymbol{b}_i) = \sigma_{hard}\left(\begin{pmatrix}2 & 0 & 3\\ 4 & 1 & 0\end{pmatrix}\begin{pmatrix}1\\0\\-1\end{pmatrix} + \begin{pmatrix}-3 & 0\\ -5 & 1\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0\\-2\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}-1\\4\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0\\-2\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}-1\\2\end{pmatrix}\right) = \begin{pmatrix}-0.2+0.5\\0.2*2+0.5\end{pmatrix} = \begin{pmatrix}0.3\\0.9\end{pmatrix}$$

$$\boldsymbol{f}_1 = \sigma_{hard}(\mathbf{W}_{xf}\boldsymbol{x}_1 + \mathbf{W}_{hf}\boldsymbol{a}_0 + \boldsymbol{b}_f) = \sigma_{hard}\left(\begin{pmatrix}1 & 0 & -1\\ 2 & 1 & 2\end{pmatrix}\begin{pmatrix}1\\0\\-1\end{pmatrix} + \begin{pmatrix}-2 & 0\\ 2 & 1\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1\\-1\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}2\\0\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1\\-1\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}3\\-1\end{pmatrix}\right) = \begin{pmatrix}1\\0.3\end{pmatrix}$$

$$\boldsymbol{o}_1 = \sigma_{hard}(\mathbf{W}_{xo}\boldsymbol{x}_1 + \mathbf{W}_{ho}\boldsymbol{a}_0 + \boldsymbol{b}_o) = \sigma_{hard}\left(\begin{pmatrix}-1 & 2 & -2\\ 0 & 1 & 1\end{pmatrix}\begin{pmatrix}1\\0\\-1\end{pmatrix} + \begin{pmatrix}1 & 1\\0 & 2\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}-1\\-1\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}1\\-1\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}-1\\-1\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}0\\-2\end{pmatrix}\right) = \begin{pmatrix}0.5\\0.1\end{pmatrix}$$

$$\boldsymbol{g}_1 = \tanh(\mathbf{W}_{xg}\boldsymbol{x}_1 + \mathbf{W}_{hg}\boldsymbol{a}_0 + \boldsymbol{b}_g) = \tanh(\begin{pmatrix}1 & -1 & 0\\0 & 2 & 1\end{pmatrix}\begin{pmatrix}1\\0\\-1\end{pmatrix} + \begin{pmatrix}2 & 0\\-1 & 1\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}-1\\2\end{pmatrix}))$$
$$= \tanh(\begin{pmatrix}1\\-1\end{pmatrix} + \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}-1\\2\end{pmatrix}) = \tanh(\begin{pmatrix}0\\1\end{pmatrix}) = \begin{pmatrix}0\\\tanh(1)\end{pmatrix} = \begin{pmatrix}0\\0.7616\end{pmatrix}$$

$$\boldsymbol{c}_1 = \boldsymbol{f}_1 \odot \boldsymbol{c}_0 + \boldsymbol{i}_1 \odot \boldsymbol{g}_1 = \begin{pmatrix}1\\0.3\end{pmatrix} \odot \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0.3\\0.9\end{pmatrix} \odot \begin{pmatrix}0\\0.7616\end{pmatrix} = \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0\\0.6854\end{pmatrix} = \begin{pmatrix}0\\0.6854\end{pmatrix}$$

$$\boldsymbol{a}_1 = \boldsymbol{o}_1 \odot \tanh(\boldsymbol{c}_1) = \begin{pmatrix}0.5\\0.1\end{pmatrix} \odot \tanh\left(\begin{pmatrix}0\\0.6854\end{pmatrix}\right) = \begin{pmatrix}0\\0.1 * \tanh(0.6854)\end{pmatrix} = \begin{pmatrix}0\\0.0595\end{pmatrix}$$

The hidden layer activation at time $t = 1$ is calculated to be $\boldsymbol{a}_1 = \begin{pmatrix}0\\0.0595\end{pmatrix}$.

The output at time time $t = 1$ is calculated as follows:

$$\hat{\boldsymbol{y}}_1 = \text{ReLU}\left(\mathbf{W}_{yh}\boldsymbol{a}_1 + \boldsymbol{b}_y\right) = \text{ReLU}\left(\begin{pmatrix}2 & -1\\0 & 3\\1 & -1\\4 & 0\end{pmatrix}\begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}2\\0\\-1\\1\end{pmatrix}\right) =$$
$$\text{ReLU}\left(\begin{pmatrix}-0.0595\\0.1785\\-0.0595\\0\end{pmatrix} + \begin{pmatrix}2\\0\\-1\\1\end{pmatrix}\right) = \text{ReLU}\left(\begin{pmatrix}1.9405\\0.1785\\-1.0595\\1\end{pmatrix}\right) = \begin{pmatrix}1.9405\\0.1785\\0\\1\end{pmatrix}$$

Therefore, the network output at time $t = 1$ is $\hat{\boldsymbol{y}}_1 = \begin{pmatrix}1.9405\\0.1785\\0\\1\end{pmatrix}$.

Time step 2:

$$\boldsymbol{i}_2 = \sigma_{hard}(\mathbf{W}_{xi}\boldsymbol{x}_2 + \mathbf{W}_{hi}\boldsymbol{a}_1 + \boldsymbol{b}_i) = \sigma_{hard}\left(\begin{pmatrix}2 & 0 & 3\\4 & 1 & 0\end{pmatrix}\begin{pmatrix}0\\1\\1\end{pmatrix} + \begin{pmatrix}-3 & 0\\-5 & 1\end{pmatrix}\begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}0\\-2\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}3\\1\end{pmatrix} + \begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}0\\-2\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}3\\-0.9405\end{pmatrix}\right) = \begin{pmatrix}1\\0.3119\end{pmatrix}$$

$$\boldsymbol{f}_2 = \sigma_{hard}\left(\mathbf{W}_{xf}\boldsymbol{x}_2 + \mathbf{W}_{hf}\boldsymbol{a}_1 + \boldsymbol{b}_f\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}1 & 0 & -1\\ 2 & 1 & 2\end{pmatrix}\begin{pmatrix}0\\1\\1\end{pmatrix} + \begin{pmatrix}-2 & 0\\ 2 & 1\end{pmatrix}\begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}1\\-1\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}-1\\3\end{pmatrix} + \begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}1\\-1\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}0\\2.0595\end{pmatrix}\right) = \begin{pmatrix}0.5\\0.9119\end{pmatrix}$$

$$\boldsymbol{o}_2 = \sigma_{hard}(\mathbf{W}_{xo}\boldsymbol{x}_2 + \mathbf{W}_{ho}\boldsymbol{a}_1 + \boldsymbol{b}_o)$$
$$= \sigma_{hard}\left(\begin{pmatrix}-1 & 2 & -2\\ 0 & 1 & 1\end{pmatrix}\begin{pmatrix}0\\1\\1\end{pmatrix} + \begin{pmatrix}1 & 1\\ 0 & 2\end{pmatrix}\begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}-1\\-1\end{pmatrix}\right)$$
$$= \sigma_{hard}\left(\begin{pmatrix}0\\2\end{pmatrix} + \begin{pmatrix}0.0595\\0.1190\end{pmatrix} + \begin{pmatrix}-1\\-1\end{pmatrix}\right) = \sigma_{hard}\left(\begin{pmatrix}-0.9405\\1.1190\end{pmatrix}\right) = \begin{pmatrix}0.3119\\0.7238\end{pmatrix}$$

$$\boldsymbol{g}_2 = \tanh(\mathbf{W}_{xg}\boldsymbol{x}_2 + \mathbf{W}_{hg}\boldsymbol{a}_1 + \boldsymbol{b}_g) = \tanh(\begin{pmatrix}1 & -1 & 0\\ 0 & 2 & 1\end{pmatrix}\begin{pmatrix}0\\1\\1\end{pmatrix} + \begin{pmatrix}2 & 0\\ -1 & 1\end{pmatrix}\begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}-1\\2\end{pmatrix}))$$
$$= \tanh(\begin{pmatrix}-1\\3\end{pmatrix} + \begin{pmatrix}0\\0.0595\end{pmatrix} + \begin{pmatrix}-1\\2\end{pmatrix}) = \begin{pmatrix}\tanh(-2)\\\tanh(5.0595)\end{pmatrix} = \begin{pmatrix}-0.9640\\0.9999\end{pmatrix}$$

$$\boldsymbol{c}_2 = \boldsymbol{f}_2 \odot \boldsymbol{c}_1 + \boldsymbol{i}_2 \odot \boldsymbol{g}_2 = \begin{pmatrix}0.5\\0.9119\end{pmatrix} \odot \begin{pmatrix}0\\0.6854\end{pmatrix} + \begin{pmatrix}1\\0.3119\end{pmatrix} \odot \begin{pmatrix}-0.9640\\0.9999\end{pmatrix} = \begin{pmatrix}-0.9640\\0.9369\end{pmatrix}$$

$$\boldsymbol{a}_2 = \boldsymbol{o}_2 \odot \tanh(\boldsymbol{c}_2) = \begin{pmatrix}0.3119\\0.7238\end{pmatrix} \odot \tanh\left(\begin{pmatrix}-0.9640\\0.9369\end{pmatrix}\right) = \begin{pmatrix}-0.2327\\0.5311\end{pmatrix}$$

The hidden layer activation at time $t = 2$ is calculated to be $\boldsymbol{a}_2 = \begin{pmatrix}-0.2327\\0.5311\end{pmatrix}$.

The output at time time $t = 1$ is calculated as follows:

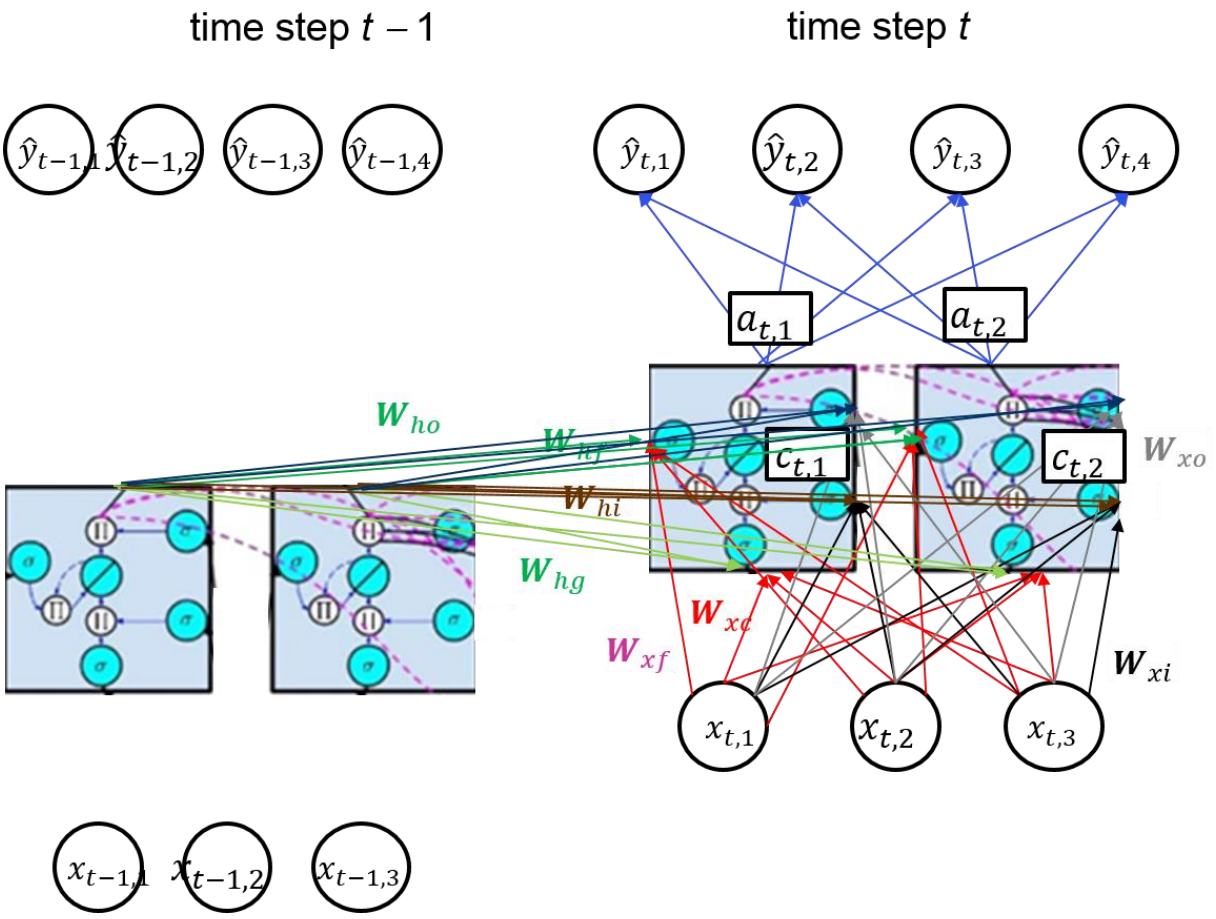
$$\hat{\boldsymbol{y}}_2 = \mathrm{ReLU}\left(\mathbf{W}_{yh}\boldsymbol{a}_2 + \boldsymbol{b}_y\right) = \mathrm{ReLU}\left(\begin{pmatrix}2 & -1\\ 0 & 3\\ 1 & -1\\ 4 & 0\end{pmatrix}\begin{pmatrix}-0.2327\\0.5311\end{pmatrix} + \begin{pmatrix}2\\0\\-1\\1\end{pmatrix}\right) =$$
$$\mathrm{ReLU}\left(\begin{pmatrix}-0.9965\\1.5933\\-0.7638\\-0.9308\end{pmatrix} + \begin{pmatrix}2\\0\\-1\\1\end{pmatrix}\right) = \mathrm{ReLU}\left(\begin{pmatrix}1.0035\\1.5933\\-1.7638\\0.0692\end{pmatrix}\right) = \begin{pmatrix}1.0035\\1.5933\\0\\0.0692\end{pmatrix}$$

Therefore, the network output at time $t = 2$ is $\hat{\boldsymbol{y}}_2 = \begin{pmatrix}1.0035\\1.5933\\0\\0.0692\end{pmatrix}$.

To summarize, the following results are obtained:

time	LSTM cell state $\boldsymbol{c}_t$	Hidden LSTM activation $\boldsymbol{a}_t$	Network output $\hat{\boldsymbol{y}}_t$
$t = 1$	$\begin{pmatrix} 0 \\ 0.6854 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0.0595 \end{pmatrix}$	$\begin{pmatrix} 1.9405 \\ 0.1785 \\ 0 \\ 1 \end{pmatrix}$
$t = 2$	$\begin{pmatrix} -0.9640 \\ 0.9369 \end{pmatrix}$	$\begin{pmatrix} -0.2327 \\ 0.5311 \end{pmatrix}$	$\begin{pmatrix} 1.0035 \\ 1.5933 \\ 0 \\ 0.0692 \end{pmatrix}$

A diagram of the LSTM network is shown on the next page. Please note that the synaptic connections have only been shown for time t, together with the recurrent connections from time $t - 1$ to t.



b) The corresponding script in the Jupyter notebook (copied from https://machinelearningmastery.com/stateful-stateless-lstm-time-series-forecasting-python/) implements a simple LSTM network to predict shampoo sales over a period of three years. Describe the LSTM network, the way how the network processes the time sequence to predict future shampoo sales (how many inputs of which dimension, how many outputs of what dimension?), and how training is performed (how many samples, what is the length of the sequence considered in truncated backpropagation through time?). Vary important parameters (repeating over several training runs) and comment on your findings.

Note that to modify other parameters, certain conditions must hold, e.g. the batch size must be a factor of the number of training samples; but then, other architectural changes of the LSTM network have to be applied. This is beyond this small introductory exercise...

Solution:

The LSTM consists of a single input unit, i.e. input at time *t* is a single number, the pre-processed shampoo sales (see below) at time *t*, a single hidden layer consisting of 4 LSTM units and a single output unit predicting a single number, for which the pre-processing steps are then inverted to give the predicted shampoo sales at time *t* + 1.
Denote the original shampoo sales at time *t* as *s*(*t*).

Preprocessing:

First, instead of the raw values, differences (in this case between two consecutive time steps, i.e., interval = 1) are considered (function **`difference`** in **`exercise3b.py`**):
$d(t) = s(t) - s(t-1)$
Then, supervised data are created by assigning to each difference value $d(t)$ the subsequent difference value $d(t)$ (function **`timeseries_to_supervised`** with lag = 1). The last 12 values (i.e., the sales of the last year of the data) are used as test data, the remaining values (i.e., the first two years) as training data.
The difference values of the training data (both input and output) are scaled to the interval $[-1, 1]$; therefore, the smallest training difference value is -1, the largest difference value is +1. Note that the scaling function is estimated on the training data and then applied to the test data, i.e., the smallest test difference value can be smaller than -1 and the largest test difference value can be larger than +1 (as is the case for this data set). Since the LSTM output activation $\boldsymbol{a}_t$ is defined by a tanh with values in $[-1, 1]$ (and the dense output weights are trained on data which are limited to that range), it cannot be expected that input values outside $[-1, 1]$ can be predicted correctly; a potential solution could be to use a more clever scaling of the data.

LSTM training:

As stated above, the LSTM network consists of a single input unit (which receives the scaled difference value at time *t*), a single hidden layer consisting of 4 LSTM units and a single output unit which predicts the scaled difference value at time *t* + 1. The activation functions of the LSTM units are the tanh and hard-sigmoid functions (see above). The activations of the 4 hidden LSTM units are fully connected ("dense") to the output unit, which has a linear

activation function (see the **`get_config()`** method in Keras). This LSTM network is trained on the scaled training data (see the **`fit_LSTM`** function in **`exercise3b.py`**).

Keras expects the training data in the format of a 3-dimensional tensor with shape (batch_size, #timesteps, #features). Here, # features is 1, and the batch size is also set to 1. Interestingly, the number of time steps is also set to 1, meaning that the network is unfolded to a single time step only. However, the "**`stateful`**" parameter is set to "true", such that the activation at the end of a batch (i.e. after processing each element from the input time sequence) is used to initialize the next input. This makes only sense if the sequence of input samples is ordered (i.e., not shuffled). Therefore, the entire training sequence is used in forward propagation. Backward propagation, however, seems to use a single time step only (which may not make too much sense).

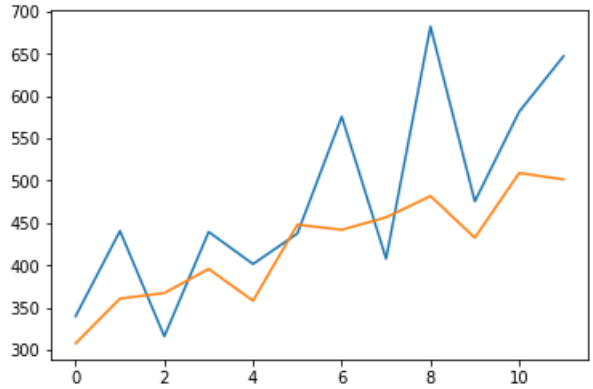
Predictions:

After training, the LSTM network can be used to predict the sale of the next month based on the sale of the current month. In particular, the scaled difference sale is input to the model, and a scaled difference sale is predicted (please note the remark regarding preprocessing above). The predicted scaled difference sale is then scaled using the inverse scaling transformation estimated in training. Afterwards, the absolute sale is computed from this difference sale, which is the output of the network. Note that the prediction of the LSTM network for the subsequent month is not the current prediction (for the current month), but the actual true prediction for the current month. A different evaluation scenario (where a prediction is used to calculate the next prediction) could be constructed.

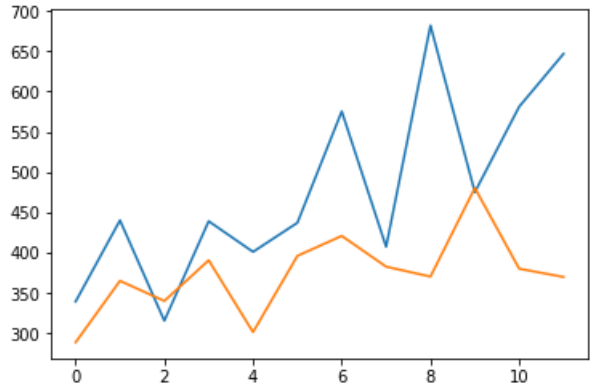
Due to different random initializations of network weights, different training runs result in different outcomes. To investigate the influence of parameters, we therefore average over four training runs.

Here are the outcomes for four training runs with standard parameters:

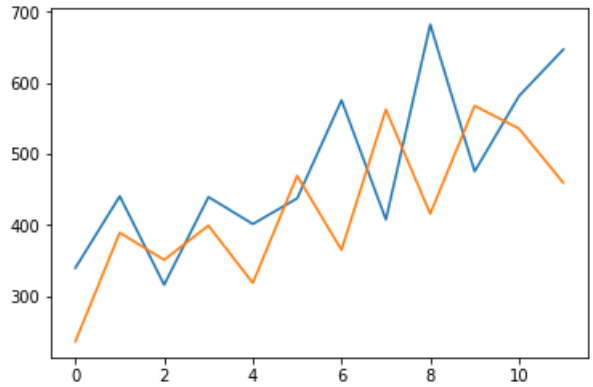
```
Month=1, Predicted=307.617829, Expected=339.700000
Month=2, Predicted=360.529975, Expected=440.400000
Month=3, Predicted=367.070183, Expected=315.900000
Month=4, Predicted=395.553830, Expected=439.300000
Month=5, Predicted=357.947693, Expected=401.300000
Month=6, Predicted=447.805232, Expected=437.400000
Month=7, Predicted=441.785465, Expected=575.500000
Month=8, Predicted=456.486775, Expected=407.600000
Month=9, Predicted=481.610897, Expected=682.000000
Month=10, Predicted=432.425494, Expected=475.300000
Month=11, Predicted=508.856209, Expected=581.300000
Month=12, Predicted=501.295139, Expected=646.900000
Test RMSE: 92.485
```



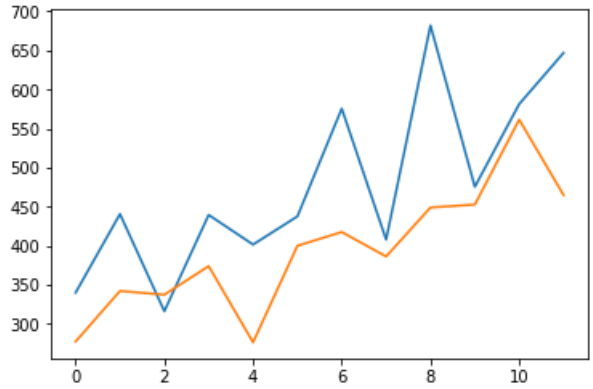

```
Month=1, Predicted=289.103660, Expected=339.700000
Month=2, Predicted=365.371617, Expected=440.400000
Month=3, Predicted=340.406122, Expected=315.900000
Month=4, Predicted=390.896310, Expected=439.300000
Month=5, Predicted=301.760642, Expected=401.300000
Month=6, Predicted=396.340573, Expected=437.400000
Month=7, Predicted=421.090896, Expected=575.500000
Month=8, Predicted=382.941611, Expected=407.600000
Month=9, Predicted=370.604889, Expected=682.000000
Month=10, Predicted=480.514345, Expected=475.300000
Month=11, Predicted=380.399994, Expected=581.300000
Month=12, Predicted=370.142176, Expected=646.900000
Test RMSE: 147.515
```




```
Month=1, Predicted=236.283877, Expected=339.700000
Month=2, Predicted=389.110425, Expected=440.400000
Month=3, Predicted=350.902507, Expected=315.900000
Month=4, Predicted=399.031481, Expected=439.300000
Month=5, Predicted=318.349944, Expected=401.300000
Month=6, Predicted=469.121649, Expected=437.400000
Month=7, Predicted=364.832112, Expected=575.500000
Month=8, Predicted=562.343578, Expected=407.600000
Month=9, Predicted=415.667641, Expected=682.000000
Month=10, Predicted=567.542148, Expected=475.300000
Month=11, Predicted=535.546724, Expected=581.300000
Month=12, Predicted=459.327775, Expected=646.900000
Test RMSE: 132.005
```

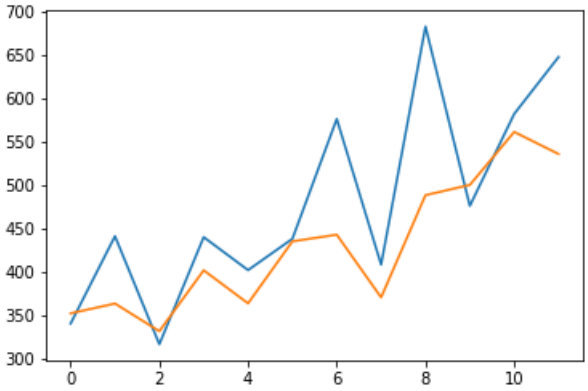



```
Month=1, Predicted=277.048288, Expected=339.700000
Month=2, Predicted=341.801691, Expected=440.400000
Month=3, Predicted=337.055216, Expected=315.900000
Month=4, Predicted=373.649205, Expected=439.300000
Month=5, Predicted=276.008283, Expected=401.300000
Month=6, Predicted=399.720811, Expected=437.400000
Month=7, Predicted=417.356965, Expected=575.500000
Month=8, Predicted=386.014490, Expected=407.600000
Month=9, Predicted=448.771867, Expected=682.000000
Month=10, Predicted=452.534953, Expected=475.300000
Month=11, Predicted=561.388796, Expected=581.300000
Month=12, Predicted=464.643055, Expected=646.900000
Test RMSE: 111.626
```

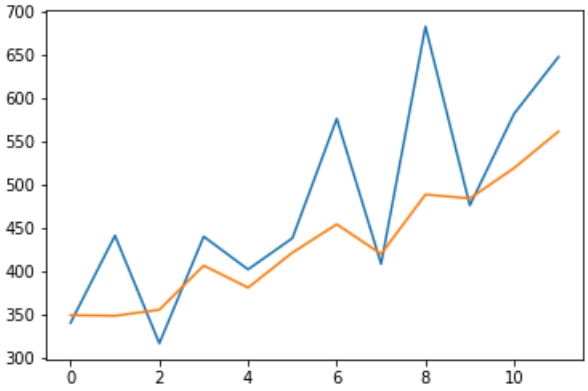


Here are another 4 runs, used to create the following table:

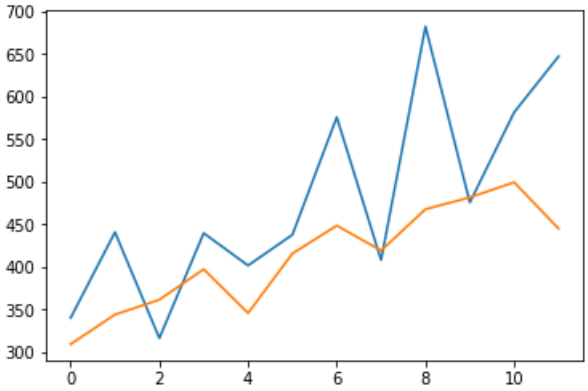
```
Month=1, Predicted=351.431687, Expected=339.700000
Month=2, Predicted=362.759781, Expected=440.400000
Month=3, Predicted=330.887229, Expected=315.900000
Month=4, Predicted=401.083292, Expected=439.300000
Month=5, Predicted=362.951946, Expected=401.300000
Month=6, Predicted=434.429900, Expected=437.400000
Month=7, Predicted=442.113748, Expected=575.500000
Month=8, Predicted=369.824232, Expected=407.600000
Month=9, Predicted=487.678792, Expected=682.000000
Month=10, Predicted=499.459529, Expected=475.300000
Month=11, Predicted=560.556339, Expected=581.300000
Month=12, Predicted=535.086074, Expected=646.900000
Test RMSE: 81.561
```



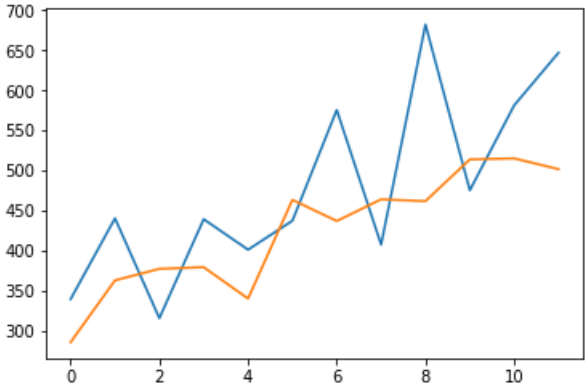

```
Month=1, Predicted=348.491192, Expected=339.700000
Month=2, Predicted=347.754788, Expected=440.400000
Month=3, Predicted=354.691124, Expected=315.900000
Month=4, Predicted=405.780926, Expected=439.300000
Month=5, Predicted=380.301420, Expected=401.300000
Month=6, Predicted=420.735112, Expected=437.400000
Month=7, Predicted=453.497750, Expected=575.500000
Month=8, Predicted=419.172899, Expected=407.600000
Month=9, Predicted=487.863057, Expected=682.000000
Month=10, Predicted=483.316202, Expected=475.300000
Month=11, Predicted=518.541183, Expected=581.300000
Month=12, Predicted=560.717394, Expected=646.900000
Test RMSE: 79.658
```



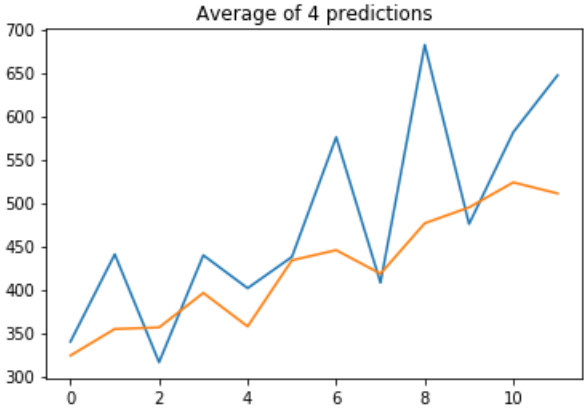

```
Month=1, Predicted=308.805498, Expected=339.700000
Month=2, Predicted=343.609009, Expected=440.400000
Month=3, Predicted=361.021203, Expected=315.900000
Month=4, Predicted=396.901144, Expected=439.300000
Month=5, Predicted=345.326456, Expected=401.300000
Month=6, Predicted=415.351005, Expected=437.400000
Month=7, Predicted=448.169191, Expected=575.500000
Month=8, Predicted=418.816764, Expected=407.600000
Month=9, Predicted=467.431479, Expected=682.000000
Month=10, Predicted=481.306621, Expected=475.300000
Month=11, Predicted=499.042685, Expected=581.300000
Month=12, Predicted=444.637444, Expected=646.900000
Test RMSE: 103.226
```




```
Month=1, Predicted=285.934202, Expected=339.700000
Month=2, Predicted=363.068711, Expected=440.400000
Month=3, Predicted=377.418197, Expected=315.900000
Month=4, Predicted=379.487674, Expected=439.300000
Month=5, Predicted=340.364450, Expected=401.300000
Month=6, Predicted=463.448448, Expected=437.400000
Month=7, Predicted=437.032611, Expected=575.500000
Month=8, Predicted=464.087331, Expected=407.600000
Month=9, Predicted=461.820576, Expected=682.000000
Month=10, Predicted=513.834426, Expected=475.300000
Month=11, Predicted=515.101704, Expected=581.300000
Month=12, Predicted=501.684119, Expected=646.900000
Test RMSE: 99.340
```



Average results:



Average RMSE: 90.946

With regard to a few parameter modifications, we obtain the following results:

Configuration	Average RMSE (over 4 runs)	Remark
Number of LSTM units = 4	90.946	Baseline
Number of LSTM units = 8	133.407	... getting worse
Number of LSTM units = 2	108.952	No improvement

Note that to modify other parameters, certain conditions must hold, e.g. the batch size must be a factor of the number of training samples; but then, other architectural changes of the LSTM network have to be applied. This is beyond this small introductory exercise...

Exercise 4 (Autoencoder):

a) (Standard autoencoder) The jupyter notebook provides code to train a standard autoencoder for encoding representations of the MNIST handwritten digits data. Run the code using different sizes of the encoded representations and different optimizers and discuss your results. In addition, add a sparsity constraint (L1 regularization) to the activities of the hidden neurons:

```
encoded = Dense( encoding_dim, activation = 'relu',
                 activity_regularizer=regularizers.l1(1e-5))(input)
```

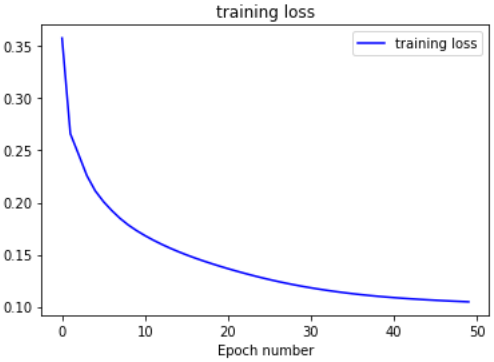
Again run the code and discuss your results.

Solution:

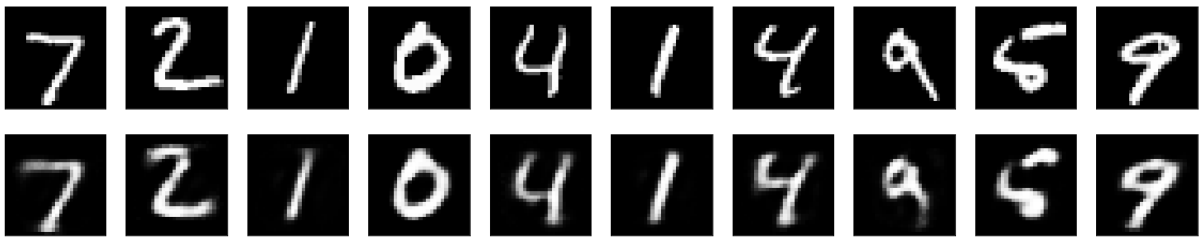
i) Varying the optimizer:

Using 32 hidden neurons (50.992 trainable parameters), i.e., compression factor 784 / 32 = 24.5

<u>Adadelta ('adadelta'):</u>

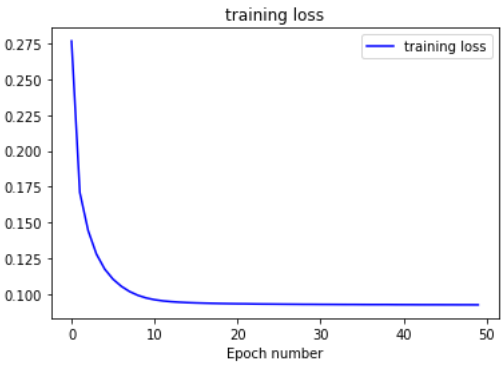


Final loss (last epoch): 0.1048
Final validation (i.e., test) loss (last epoch): 0.1029
About 1s per epoch (19-20µs per step) training time

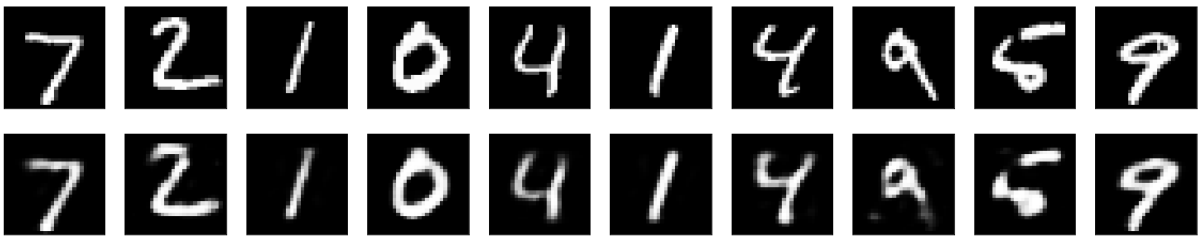


A similar behaviour is observed for Adagrad ('adagrad').

<u>Adam ('adam'):</u>

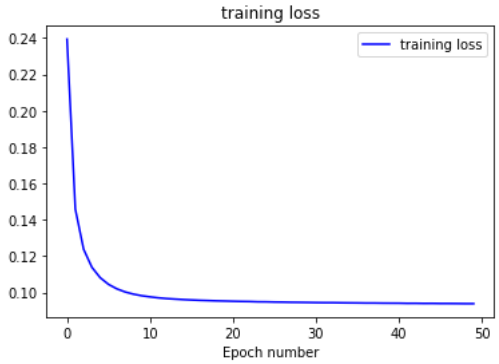


Final loss (last epoch): 0.0926
Final validation (i.e., test) loss (last epoch): 0.0914
About 1s per epoch (18-19µs per step) training time

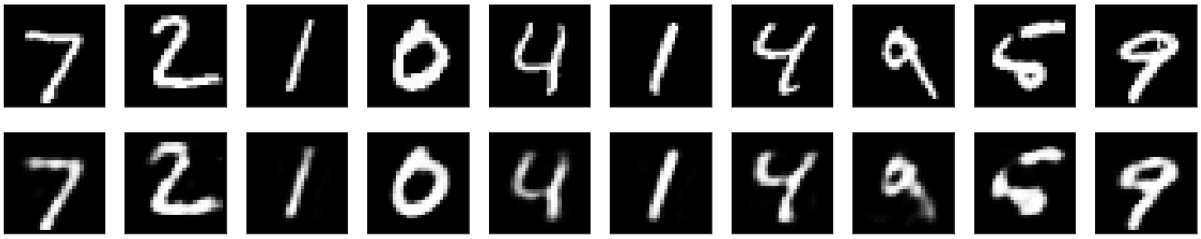


Adam leads to much faster convergence; the reconstructed images are very similar (there is a minimal difference in the second last image, regarding the upper horizontal line of the digit "5").

RMSprop ('rmsprop'):



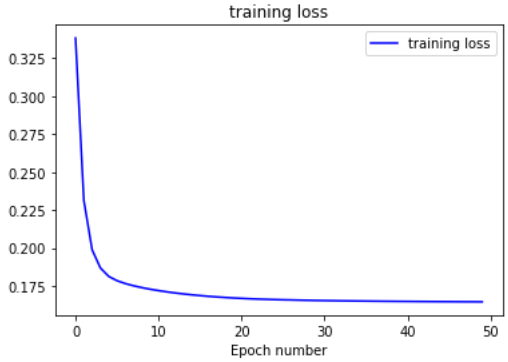
Final loss (last epoch): 0.0938
Final validation (i.e., test) loss (last epoch): 0.0926
About 1s per epoch (17-18µs per step) training time



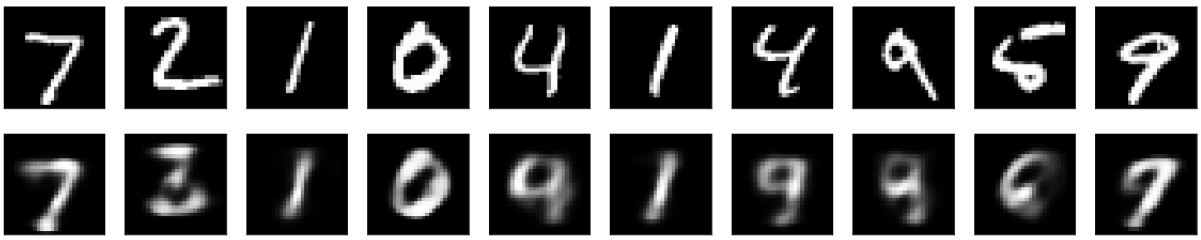
Only minimal difference to Adam are observed.

ii) Varying the number of hidden neurons, using the Adam ('adam') optimizer:

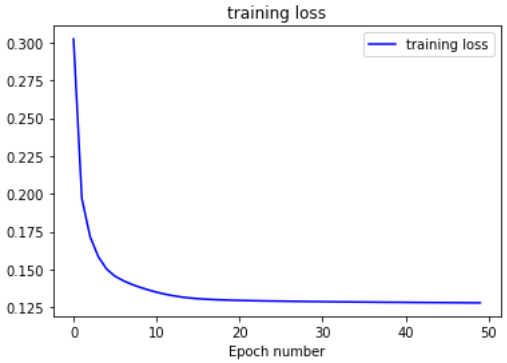
8 hidden units (13.336 trainable parameters), i.e., compression factor 784 / 8 = 98:



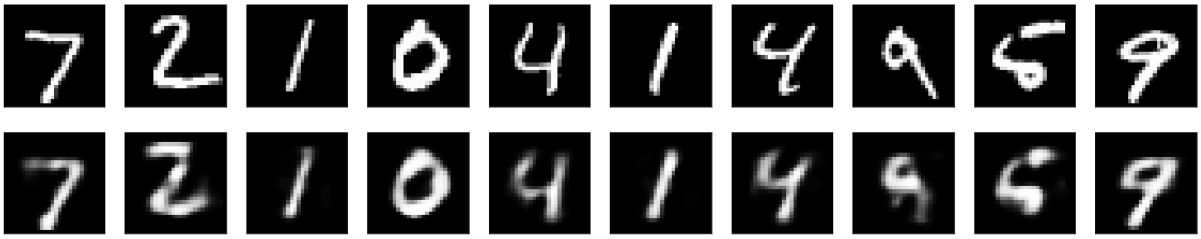
Final loss (last epoch): 0.1646
Final validation (i.e., test) loss (last epoch): 0.1630
About 1s per epoch (18-19µs per step) training time



16 hidden units (25.888 trainable parameters), i.e., compression factor 784 / 16 = 49:

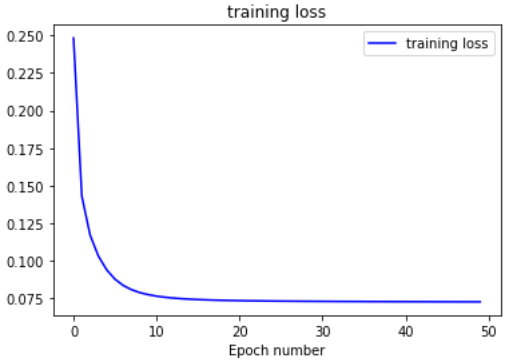


Final loss (last epoch): 0.1280
Final validation (i.e., test) loss (last epoch): 0.1265
About 1s per epoch (18µs per step) training time

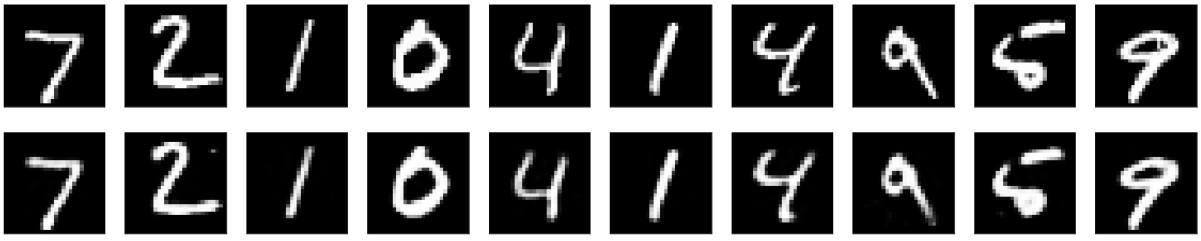


32 hidden units (50.992 trainable parameters), i.e., compression factor 784 / 32 = 24.5 see above

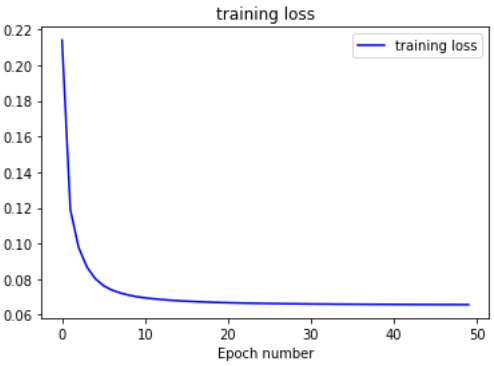
64 hidden units (101.200 trainable parameters), i.e., compression factor 784 / 64 = 12.25:



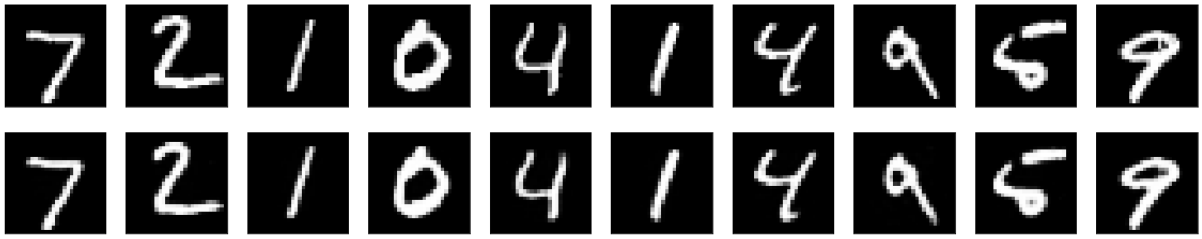
Final loss (last epoch): 0.0727
Final validation (i.e., test) loss (last epoch): 0.0723
About 1s per epoch (18-19µs per step) training time



128 hidden units (201.616 trainable parameters), i.e., compression factor 784 / 128 = 6.125:



Final loss (last epoch): 0.0656
Final validation (i.e., test) loss (last epoch): 0.0655
About 1s per epoch (19µs per step) training time



Summary:

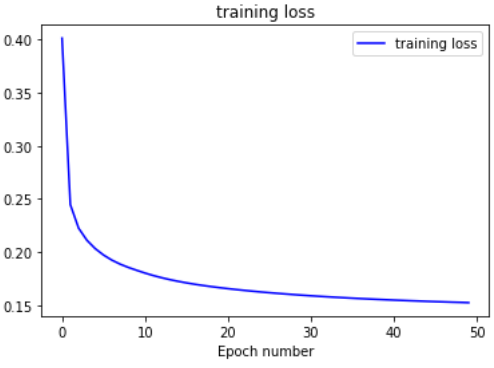
As expected, it can be seen that the image quality increases with increasing size of the encoded representation (i..e., number of hidden neurons). This corresponds to a decrease in the (final) training and validation loss. 8 hidden neurons are not enough to encode the input digits. On the other hand, using 64 hidden neurons still leads to visible improvements in the quality of the reconstructed digits compared to using 32 hidden neurons, while a further increase to 128 hidden neurons provides only small additional improvements.

Adding a sparsity constraint (L1 regularization) to the activities of the hidden neurons:

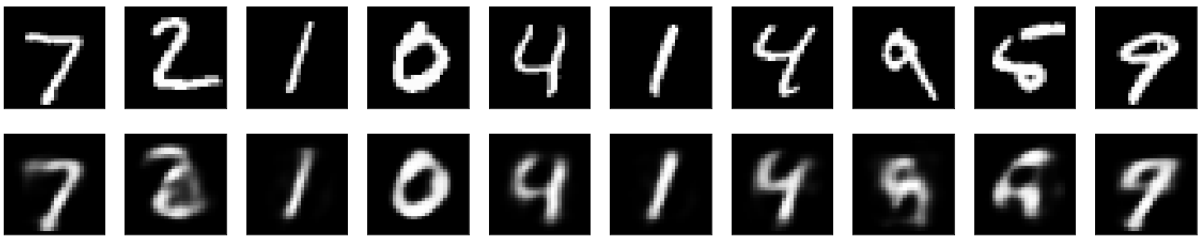
This is realized by the following code (which needs `from keras import regularizers`):

```
encoded = Dense( encoding_dim, activation = 'relu',
                 activity_regularizer=regularizers.l1(1e-5))(input)
```

Using 64 hidden neurons and the Adam optimizer with 50 epochs the following results have been obtained for a regularization factor of 1e-5:

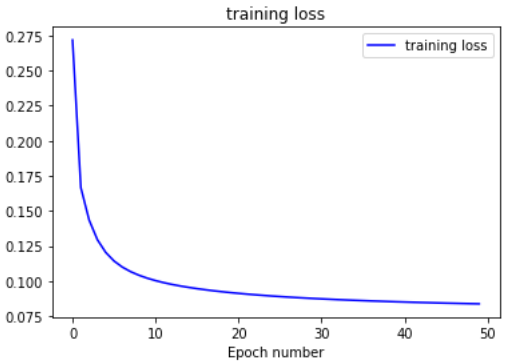


Final loss (last epoch): 0.1525
Final validation (i.e., test) loss (last epoch): 0.1510
About 1s per epoch (13µs per step) training time

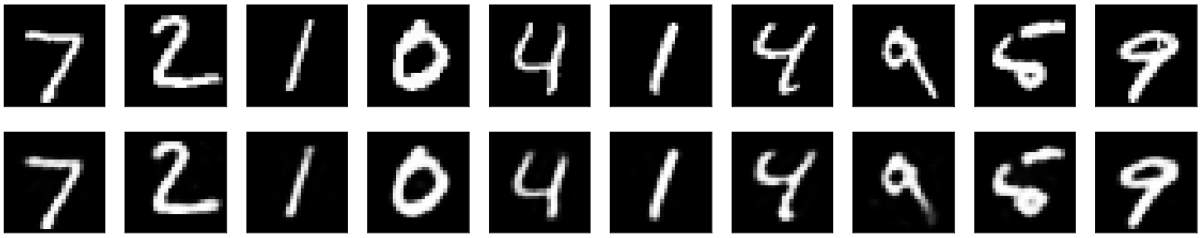


The image quality is much worse than without activity regularization. The loss is increased from about 0.07 to about 0.15 which can be attributed to the additional regularization term. On the other hand, the mean value of the encoded images (computed over the 10.000 test images) is 0.082 compared to 6.737 without regularization; thus, the model with activity regularization leads to encoded representations which are 80 times sparser.

Using a regularization factor of 1e-6 leads to the following results:



Final loss (last epoch): 0.0838
Final validation (i.e., test) loss (last epoch): 0.0833
About 1s per epoch (13-14 µs per step) training time



Here, the image quality is comparable to that without regularization. The loss is slightly increased (from 0.07 to 0.08), which is attributed to the regularization term. The mean value of the encoded representations of the 10.000 test images is about 0.533, which is a 12 times sparser representation than without regularization with nearly similar quality of the reconstructed images.

This also demonstrates that the value of the regularization factor (i.e., its order of magnitude) has to be carefully chosen.

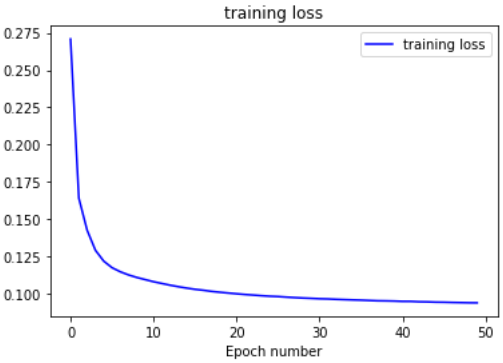
b) (Convolutional autoencoder) The next jupyter notebook provides code to train a convolutional autoencoder for encoding representations of the MNIST handwritten digits data. Run the code and interpret your results including a comparison to the autoencoder from part a).

Solution:

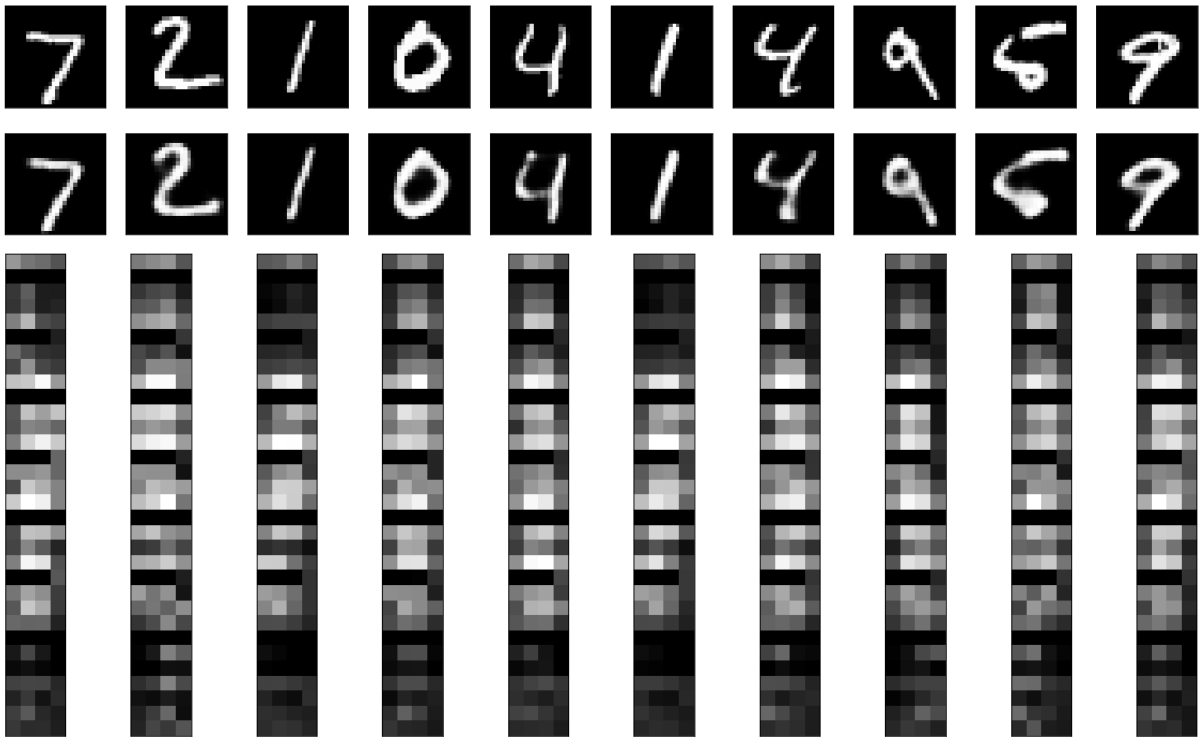
The convolutional autoencoder has the following form (output of "summary" method):

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_26 (InputLayer)        (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 4, 4, 8)           0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 4, 4, 8)           584
_________________________________________________________________
up_sampling2d_7 (UpSampling2 (None, 8, 8, 8)           0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 8, 8, 8)           584
_________________________________________________________________
up_sampling2d_8 (UpSampling2 (None, 16, 16, 8)         0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 14, 14, 16)        1168
_________________________________________________________________
up_sampling2d_9 (UpSampling2 (None, 28, 28, 16)        0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 28, 28, 1)         145
=================================================================
Total params: 4,385
Trainable params: 4,385
Non-trainable params: 0
```

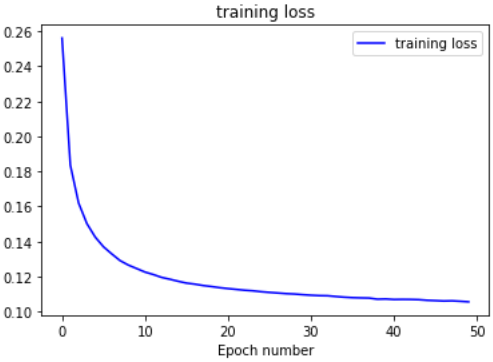
Thus, there are only 4.385 parameters, which is an order of magnitude less than the number of trainable parameters of the "plain" autoencoder from part a) with 32 hidden units. On the other hand, the convolutional autoencoder has 7 hidden layers (with trainable parameters, i.e., without pooling layers) instead of a single hidden layer. The training time is about 4 times as large as before. The training output – using the Adam ('adam') optimizer – is as follows:



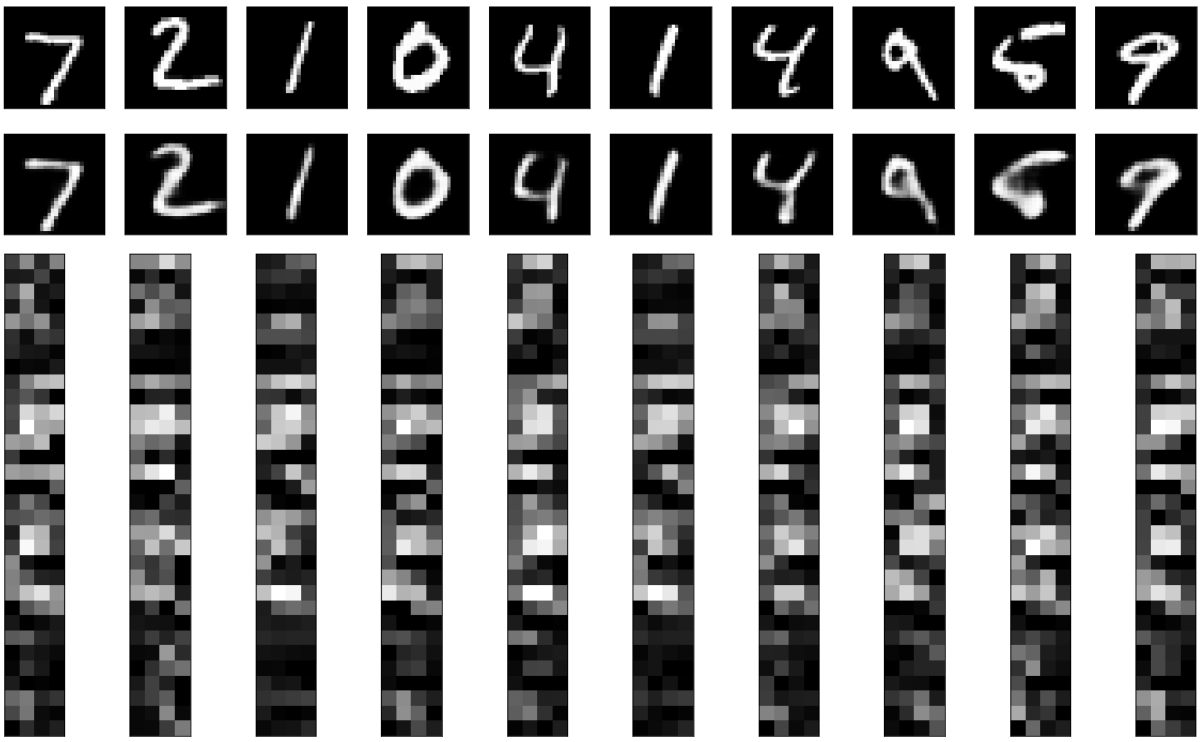
Final loss (last epoch): 0.0940
Final validation (i.e., test) loss (last epoch): 0.0932
About 4s per epoch (64 - 65µs per step) training time



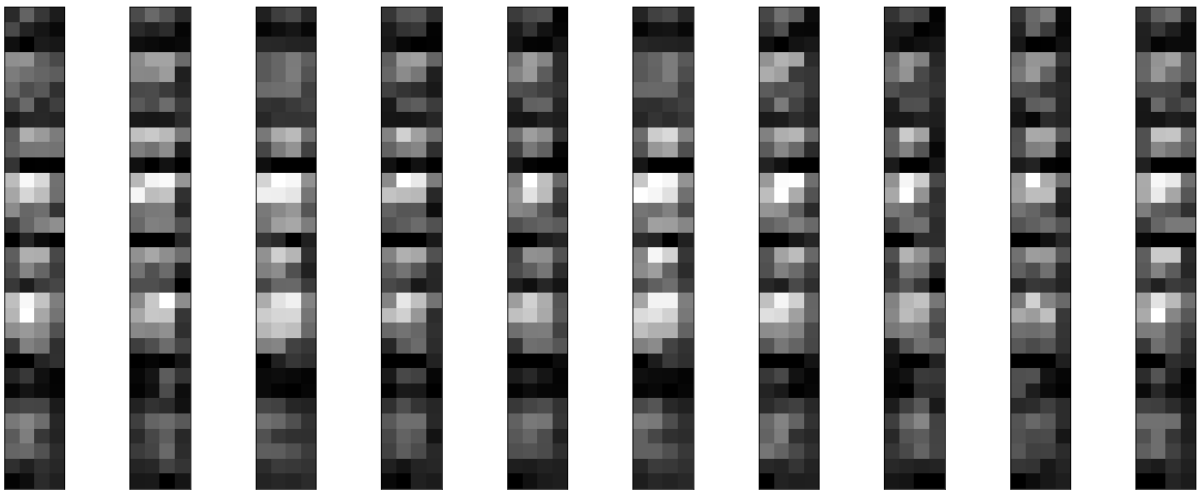
For comparison, the output for the (‘adadelta’) optimizer is as follows:



Final loss (last epoch): 0.1056
Final validation (i.e., test) loss (last epoch): 0.1051
About 4s per epoch (66µs per step) training time



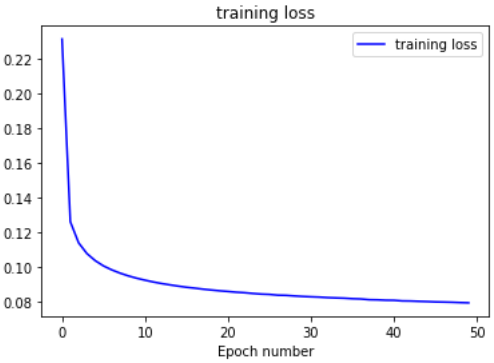
Adam seems to provide slightly more accurate reconstructions, which is also reflected by the slightly lower loss. There are visible differences in the learned representations of the digits, which may be due to using a different optimizer or using different random weight initializations. Repeating training using the Adam optimizer, the following representations have been obtained:



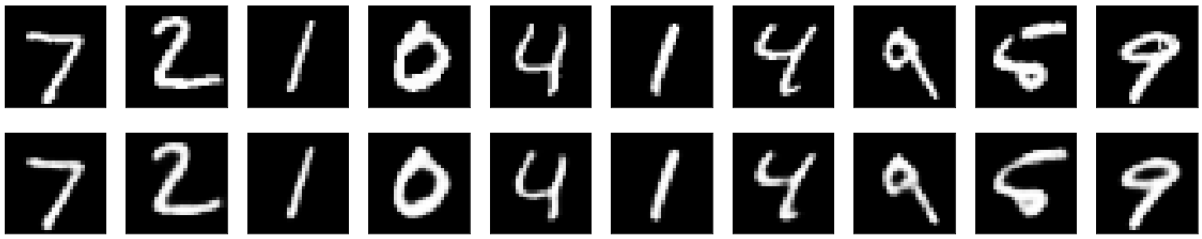
This supports the notion that the difference in the obtained representations can be attributed to different initializations of the optimizer, leading to different found local optima.

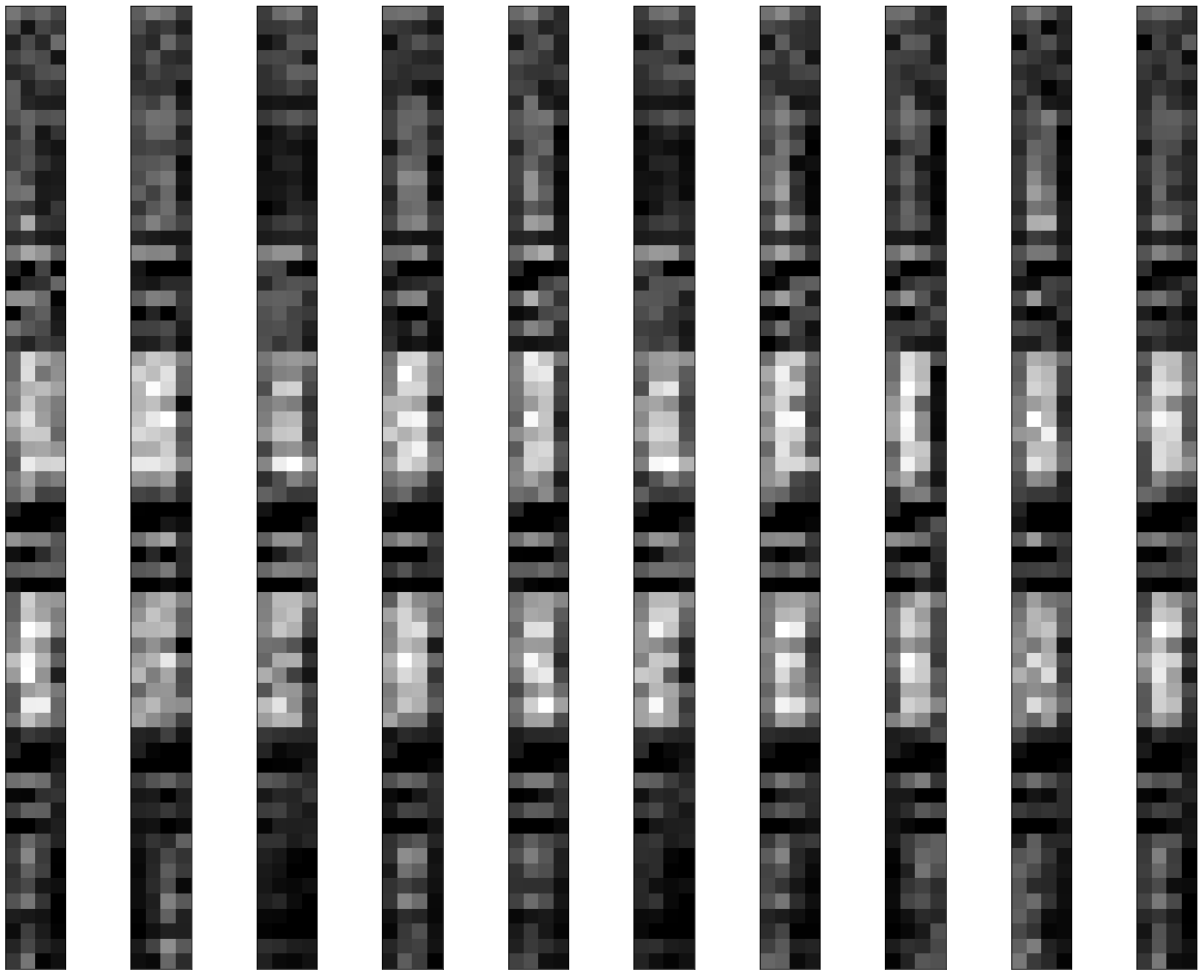
Doubling the number of feature maps leads to a model with 16.833 trainable parameters. Using the Adam optimizer, the training output is as follows:

```
num_feature_maps_high = 32
num_feature_maps_low = 16
```



Final loss (last epoch): 0.0793
Final validation (i.e., test) loss (last epoch): 0.0787
About 5s per epoch (88µs per step) training time





The quality of the reconstructed images is slightly improved – corresponding to a slightly lower loss – at the expense of a larger number of trainable parameters and a larger training time.

c) (Denoising autoencoder) Apply the convolutional autoencoder from part b) to noisy input digits. To this end, synthetic Gaussian noise is added pixel-wise to the input images (where the resulting noisy pixel values arethen clipped to values in the interval [0,1]). This is achieved by the following code, which must be executed after the "process data" step:

```
###----------------------
# add noise to input data
###----------------------

noise_factor = 0.5

training_input_noisy = training_input + noise_factor * np.random.norm
al(loc=0.0, scale=1.0, size = training_input.shape)
test_input_noisy = test_input + noise_factor * np.random.normal(loc=0
.0, scale=1.0, size = test_input.shape)

training_input_noisy = np.clip(training_input_noisy, 0., 1.)
test_input_noisy = np.clip(test_input_noisy, 0., 1.)
```

In this case, training has to be performed using the noisy images as input and the original images as targets. In addition, the noisy images are used to calculate internal representations (and of course, the noisy input images have to be plotted):

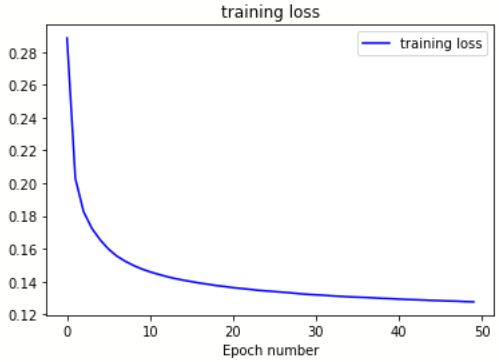
```
history = autoencoder.fit(training_input_noisy, training_input,
                          epochs=num_epochs, batch_size=batch_size,
                          shuffle = True,
                    validation_data=(test_input_noisy, test_input))

decoded_input = autoencoder.predict(test_input_noisy)

encoded_input = encoder.predict(test_input_noisy)
```

Modify the code accordingly, perform some experiments and report on your results.

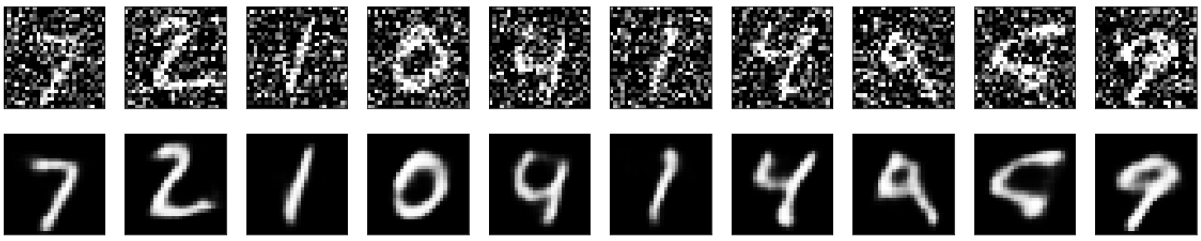
Solution:

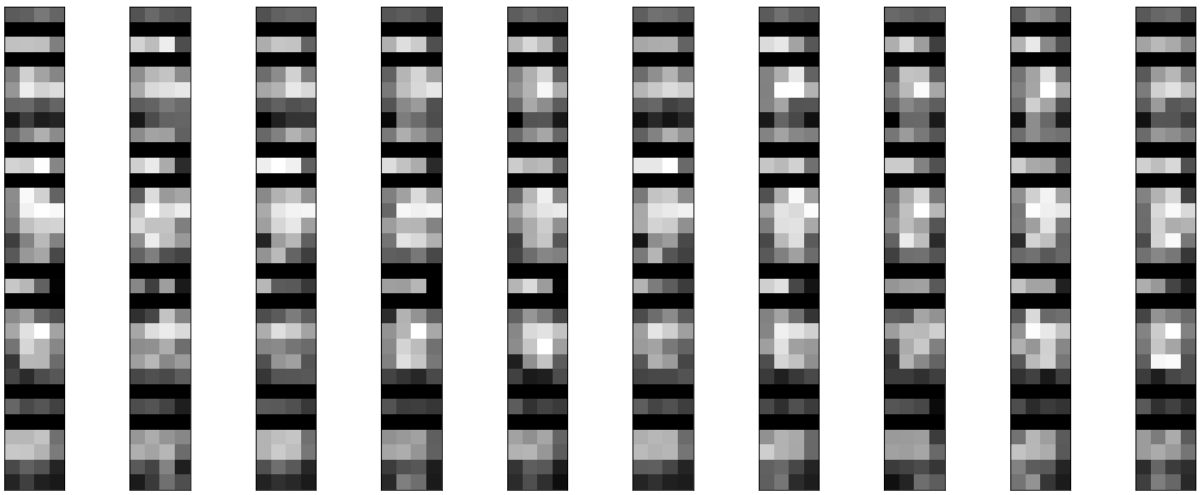
Using 16 / 8 feature maps, respectively, the Adam optimizer run for 50 epochs and a noise factor of 0.5 leads to the following results:

```
num_feature_maps_high = 16
num_feature_maps_low = 8
```



Final loss (last epoch): 0.1276
Final validation (i.e., test) loss (last epoch): 0.1266
About 2s per epoch (31µs per step) training time



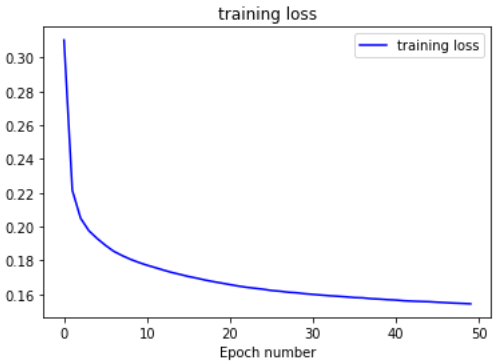


Thus, the images can be denoised pretty well (although the loss is increased from about 0.09 to 0.12). There seems to be no fundamental difference in the internal reconstructions.

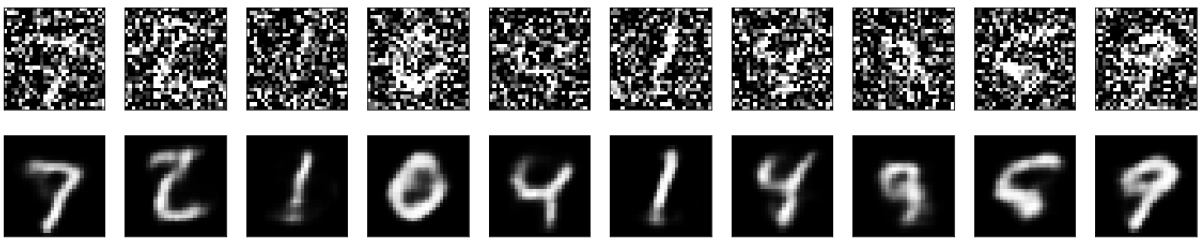
Note, however, that the (synthetic) noise is random and uncorrelated, which may not be the case in real applications. Thus, denoising of real images is a much harder task.

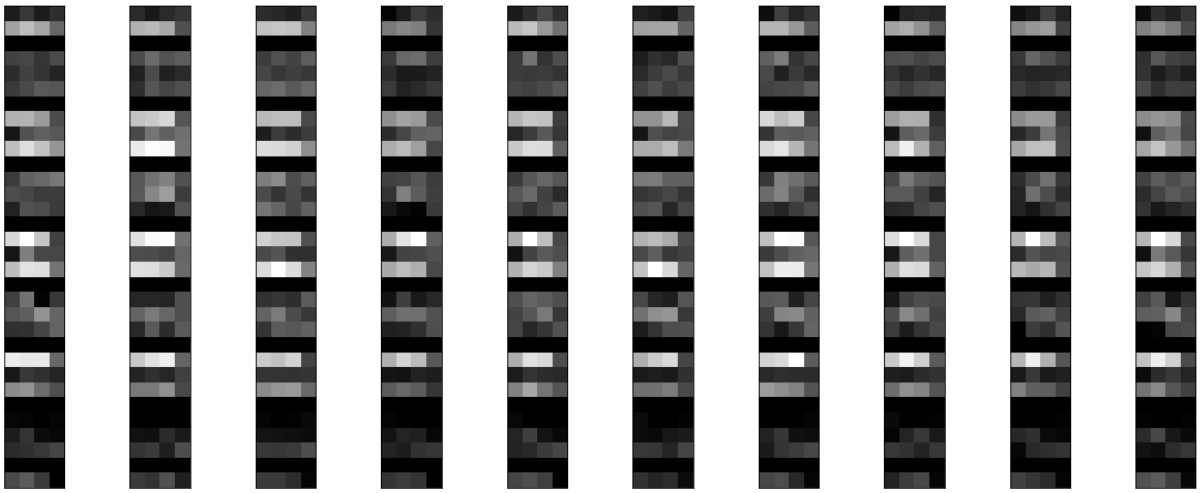
Increasing the noise factor to 0.75 yields the following results:

```
noise_factor = 0.75

num_feature_maps_high = 16
num_feature_maps_low = 8
```



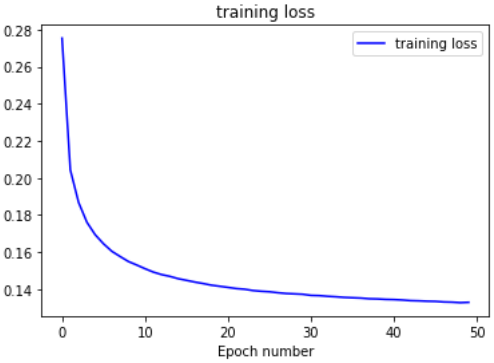
Final loss (last epoch): 0.1543
Final validation (i.e., test) loss (last epoch): 0.1531
About 2s per epoch (31µs per step) training time



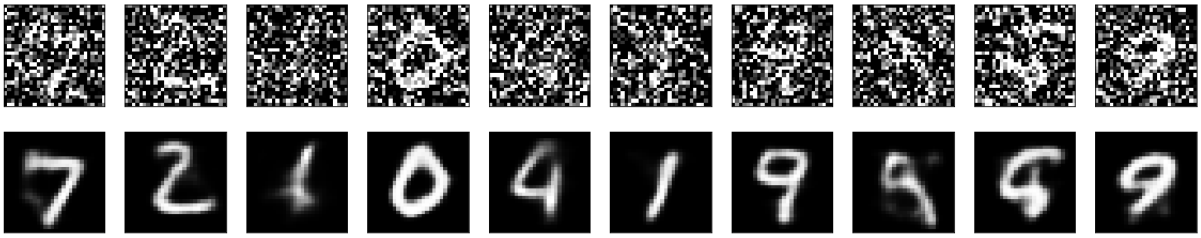


Here it can be seen that the quality of the reconstructed images is poorer, which is also reflected in an increased loss. This can be countered by an increased number of feature maps:

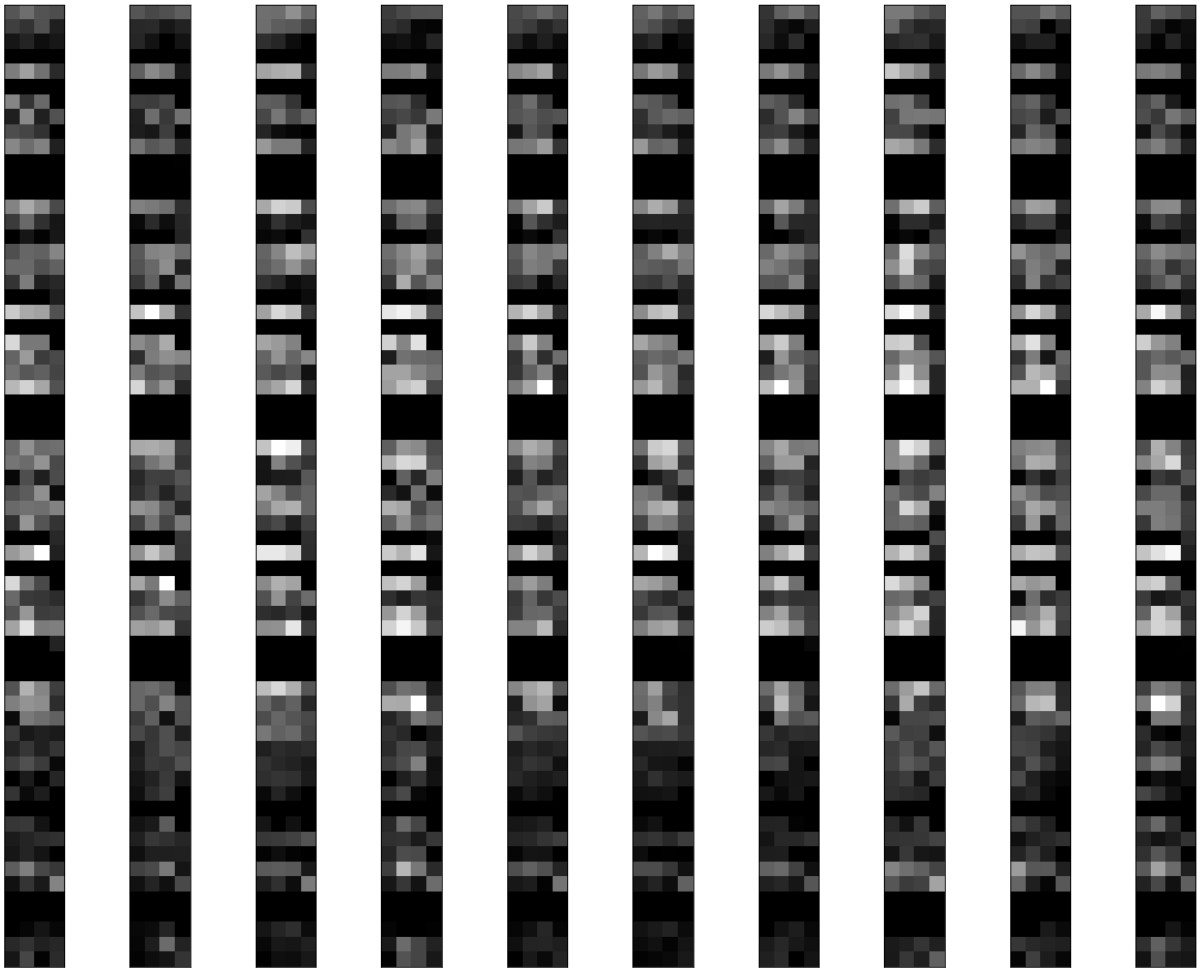
```
noise_factor = 0.75

num_feature_maps_high = 32
num_feature_maps_low = 16
```

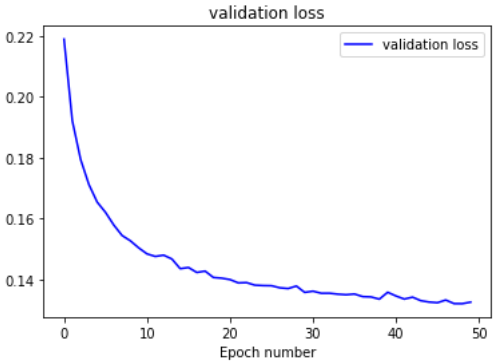


Final loss (last epoch): 0.1330
Final validation (i.e., test) loss (last epoch): 0.1326
About 2s per epoch (37µs per step) training time



Here, the quality of the reconstructed images seems to be (sometimes) improved compared to the lower number of feature maps before, corresponding to a lower loss; note, however, that the seventh reconstructed digit is wrong.





For comparison, the validation, i.e. test loss is plotted. It basically follows the training loss, without signs of overfitting, however shows slightly more fluctuations than the training loss.

Additional analysis:

To motivate the following experiments with a variational autoencoder, some experiments have been performed to a) add Gaussian noise to the calculated encodings, and b) to generate completely random encodings, from which decoded images are being reconstructed.

The first step is to define a separate decoder model by connecting the last layers of the autoencoder:

```
# create a separate decoder model (to later decode representations): th
is model maps a representation to a lossy reconstruction
representation_shape = (4,4,8)
representation_input = Input(shape=representation_shape)
decoder = Model(representation_input, autoencoder.layers[-
1](autoencoder.layers[-2](autoencoder.layers[-3](autoencoder.layers[-
4](autoencoder.layers[-5](autoencoder.layers[-6](autoencoder.layers[-
7](representation_input)))))))
```

Then, the value range of the reconstructed images is being plotted. A scaling factor for the Gaussian noise to be added to the calculated test image encodings is chosen based on the calculated value range (the given noise scaling factor of 0.5 fits an observed value range of [0,10]; for other value ranges, adjust the scaling factor accordingly):

```
###-----------------------------------------------
# construct representations and try to decode images
###-----------------------------------------------

print("minimal value of encoded images: %f" % np.min(encoded_input))
print("maximal value of encoded images: %f" % np.max(encoded_input))

num_repetitions = 3
noise_factor_encoding = 0.5

print("noise factor in encoding: %f" % noise_factor_encoding)
print("adding Gaussian noise to calculated encodings:")
# add noise to calculated encoded representations of test images
for i in range(num_repetitions):
  encoded_input_noisy = encoded_input + noise_factor_encoding * np.rand
om.normal(loc=0.0, scale=1.0, size = encoded_input.shape)
  decoded = decoder.predict(encoded_input_noisy)

  plt.figure(figsize=(20, num_feature_maps_low))
  for i in range(num_examples):
    # display representation
    ax = plt.subplot(1, num_examples, i + 1)
    plt.imshow(encoded_input_noisy[i].reshape(4, 4*num_feature_maps_low
).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
  plt.show()

  plt.figure(figsize=(20, 4))
  for i in range(num_examples):
    # display reconstruction
    ax = plt.subplot(1, num_examples, i + 1)
    plt.imshow(decoded[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
  plt.show()
```

Finally, completely random encodings are being generated, again using the adjusted noise scaling factor:

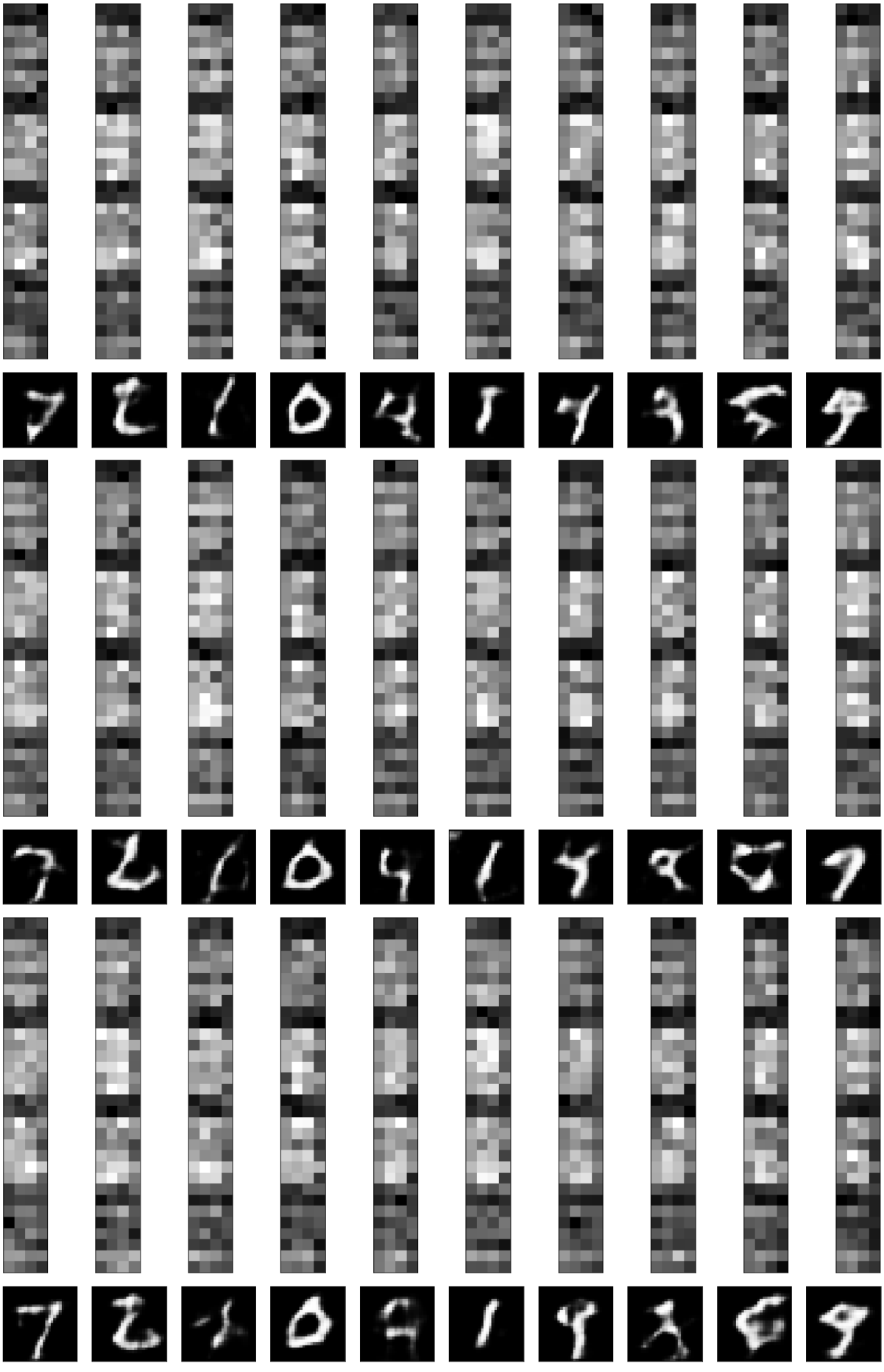
```
# now generate completely random encoded representations
print("now generate completely random encodings:")
for i in range(num_repetitions):
  encoded_input_noisy = noise_factor_test * np.random.normal(loc=0.0, s
cale=1.0, size = encoded_input.shape)
  decoded = decoder.predict(encoded_input_noisy)

  plt.figure(figsize=(20, num_feature_maps_low))
  for i in range(num_examples):
    # display representation
    ax = plt.subplot(1, num_examples, i + 1)
    plt.imshow(encoded_input_noisy[i].reshape(4, 4*num_feature_maps_low
).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
  plt.show()

  plt.figure(figsize=(20, 4))
  for i in range(num_examples):
    # display reconstruction
    ax = plt.subplot(1, num_examples, i + 1)
    plt.imshow(decoded[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
  plt.show()
```

The ideas is to check whether small deviations from the calculated encodings still lead to meaningful reconstructed images.

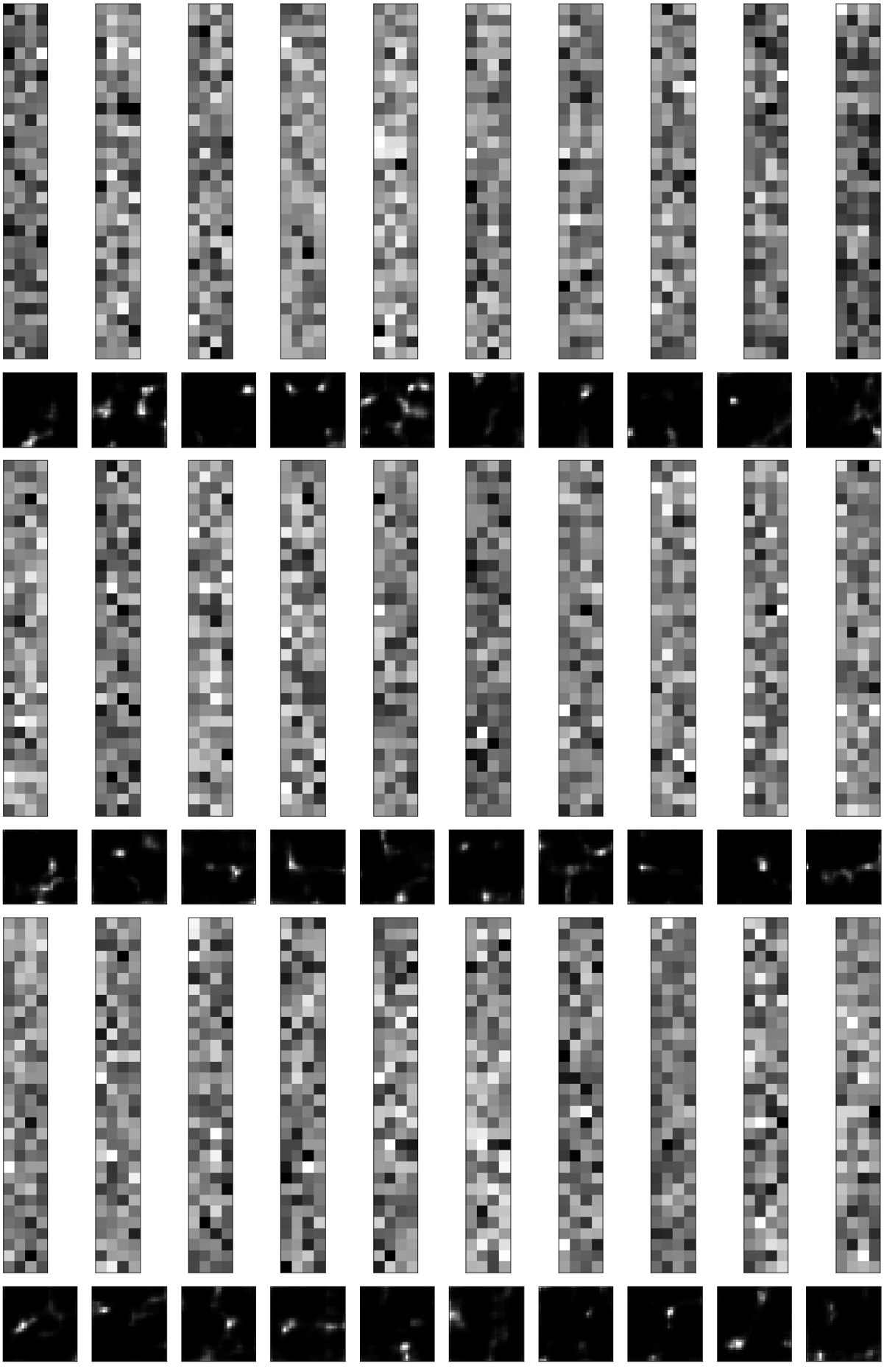
The following plots show that adding Gaussian noise, multiplied with a factor of 0.5 (for an observed value range of the calculated test image encodings of [0, 10]), renders the reconstructions mostly meaningless:

```
minimal value of encoded images: 0.000000
maximal value of encoded images: 9.782259
noise factor in encoding: 0.500000
```

adding Gaussian noise to calculated encodings:



now generate completely random encodings:



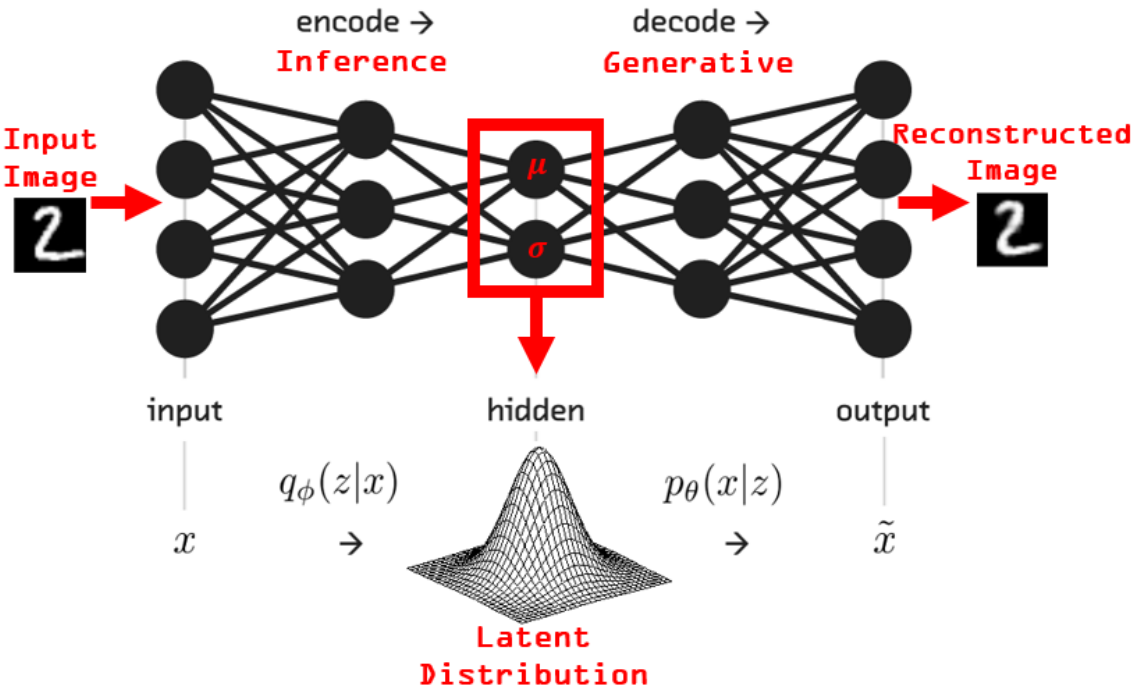
d) (Variational autoencoder) The subsequent Jupyter notebook contains code to train a (convolutional) variational autoencoder on the MNIST data. Run the code and interpret the figures, thereby explaining briefly how a variational autoencoder works.

(example based on https://www.kaggle.com/rvislaywade/visualizing-mnist-using-a-variational-autoencoder)

Solution:

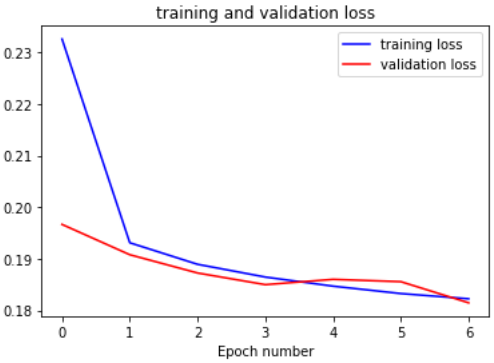
The convolutional variational autoencoder consist of an encoder and a decoder part. The encoder part has 4 convolutional layers (one of which has stride 2 and thus downsamples the input images) and 3 fully connected layers. Its output consists of two real numbers, i.e. the representation of a digit in the two-dimensional latent space.

The decoder takes a two-dimensional latent representation as input (i.e., any two-dimensional vector in the range of the latent space), and transforms it to a 28 x 28 image (via a fully connected layer which is reshaped to a two-dimensional 14 x 14 matrix, followed by a transposed convolution to for upsampling to a 28 x 28 matrix, and a standard convolution). This is shown in the following plot (taken from https://www.kaggle.com/rvislaywade/visualizing-mnist-using-a-variational-autoencoder):



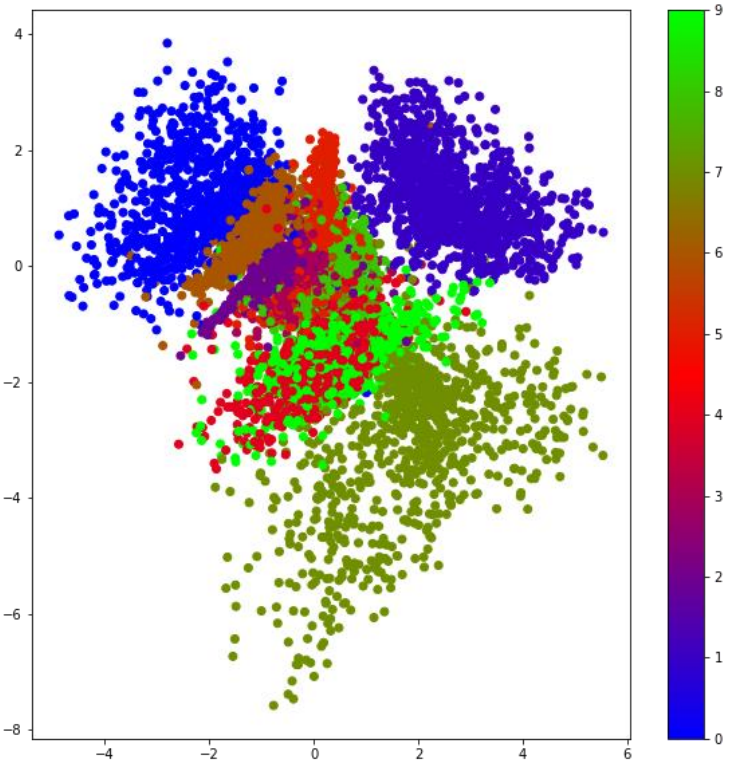
In training, the training images are first transformed to the latent space by the encoder, leading to a latent representation of each training image. Then, each latent representation is slightly "perturbed" to get a similar latent point: Based on the encoder output ("z_mu") and the additional output "z_log_sigma", a new latent representation is constructed from a normally distributed random variable "epsilon" by adding epsilon times z_log_sigma to the latent representation. From this "perturbed" latent representation, a digit is reconstructed. The loss is computed as the binary cross-entropy loss between the original digit and the reconstructed digit (to enforce the possibility to reconstruct the original digit from its latent representation; this loss may be changed to the mean squared error loss) plus the Kullback-Leibler distance between the latent (Gaussian) distribution parameterized by z_mu and z_log_sigma and the prior.

The following training and validation (i.e., test) loss were obtained in training using the 'rmsprop' optimizer:



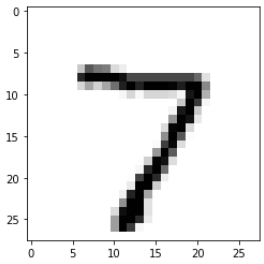
Final loss (last epoch): 0.1822
Final validation (i.e., test) loss (last epoch): 0.1815
About 20s per epoch (340µs per step) training time

Since the latent space is two-dimensional, it can be easily plotted. The figure below shows the latent representations of the test images (each dot corresponds to one test image), color-coded by their true target labels (this is the only reason that we need the true labels!). It can be seen that the digits form quite distinct clusters (with some overlaps though). Close clusters are digits that are structurally similar (i.e. digits that share information in the latent space).

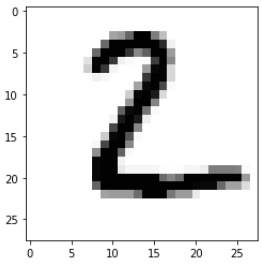


Because the variational autoencoder is a generative model, we can also use it to generate new digits. As an example, we compute the latent representations of the first two test images (shown as the end points of the solid line in the subsequent plot of the latent representations), and linearly interpolate between these latent representations (shown as solid line). The interpolated latent representations are then reconstructed. The following plot shows the results:

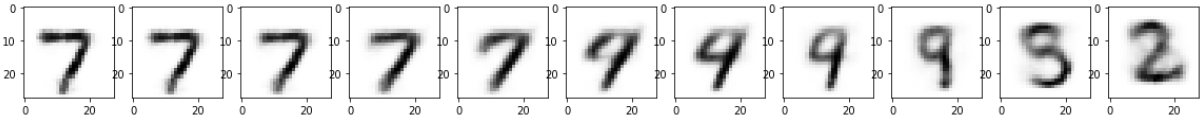
First test image:



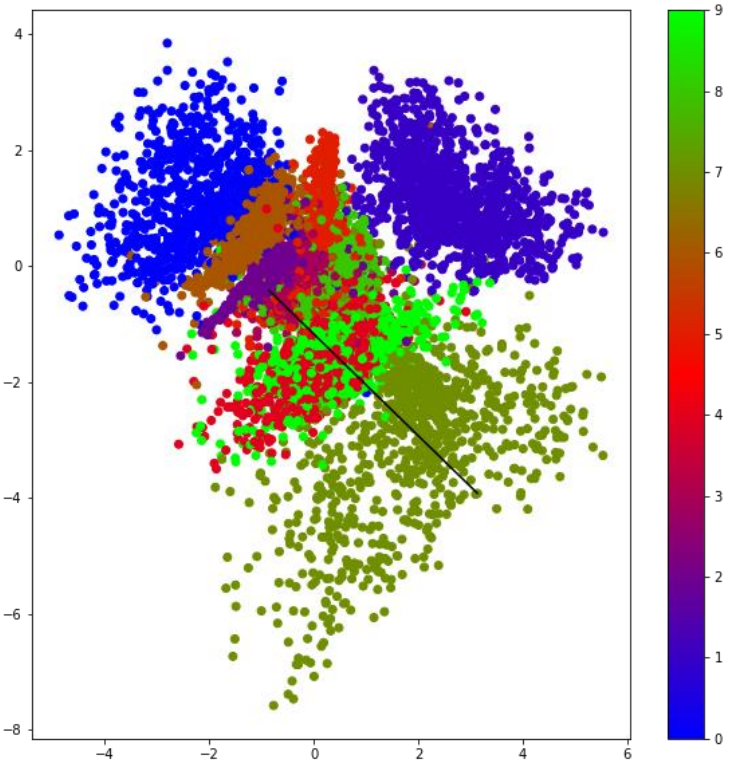
Second test image:



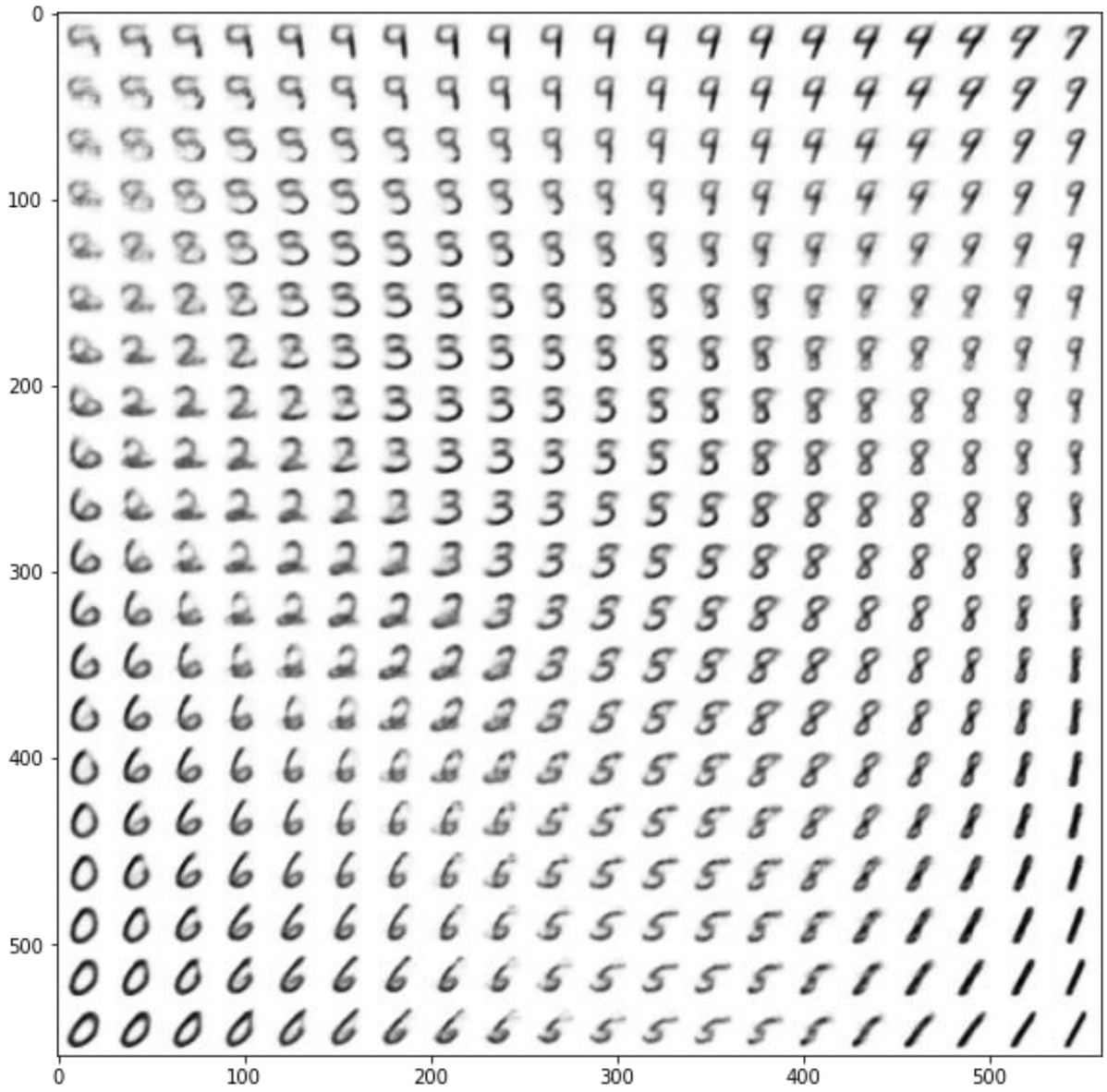
Interpolated images:



Starting from the digit “7” (right end of the solid line in the following figure), a number of interpolated latent representations belong to the cluster corresponding to the digit “7”. Then, the cluster “9” is crossed and some area of the digit “3”, before finally arriving in the cluster corresponding to the digit “2” (which is located at the border of the corresponding cluster). This demonstrates that one may interpolate between representations in latent space and still obtain meaningful reconstructions. This is in contrast to the standard autoencoder considered before (see part c), where a random internal representation usually did not lead to a valid reconstructed digit, unless it was very close to a learned representation from the training data.



This idea can further be extended by scanning the latent space, sampling latent points at regular intervals, and reconstructing the corresponding images, shown in the following figure. Only the digits “4” and “7” seem to be underrepresented (compare the scatter plots above).



Finally, we demonstrate how a random latent representation can be used to reconstruct a 28 x 28 image which may correspond to a valid digit. As an example, we chose the random latent representation (–1.1, 0.8). The reconstructed image shows a “6”: