Assignment 7: Neural Networks using Keras and Tensorflow

Neha Devi Shakya (CY Tech Exchange Student): 15 Hours

Sarvesh Meenowa (CY Tech Exchange Student): 15 Hours

Question 1: Preprocessing [1 Point]

In the notebook, the data is downloaded from an external server imported into the notebook environment using the mnist.load_data() function call.
Explain the data pre-processing high-lighted in the notebook.

First, a dataset of 60,000 greyscale images of the ten digits is downloaded along with a test set of 10,000 images.

```
(x_train, lbl_train), (x_test, lbl_test) = mnist.load_data()
```

x_train and x_test contain a numerical representation of the number of pictures. One element is one pixel, and the value represents its grayscale level as a byte image - a value between 0 and 255 where 255 represents white, 0 represents black, and in-between values represent shades of grey.

```
x_train = x_train.astype("float32")
x_test = x_test.astype("float32")
```

These first two lines of code convert all training and testing data from integers to floating-point numbers (float-32 type) as python's division assignment operator /= does not output float when the inputs are integers.

```
x_train /= 255
x_test /= 255
```

After that, these two lines of code divide everything by the max-grayscale level (255) to get a value between 0 and 1 (normalize the values). Normalizing the data is very important, as it makes the input features have the same order of magnitude, making the training easier. (source: Module 7: Machine learning and neural networks, neural_network8_intro.ipynb)

```
y_train = keras.utils.np_utils.to_categorical(lbl_train, num_classes)
y_test = keras.utils.np_utils.to_categorical(lbl_test, num_classes)
```

Then these last two lines of code create y-train and y-test variables, representing the correct value the image is displaying. The variables lbl_train and lbl_test are vectors with the corresponding correct integers. lbl_train = [5, 0, 4, ..., 5, 6, 8], meaning that the first number is a 5, the second is a 0, the third is 4, and so on. The method to_categorical then converts these integers to binary representations stored in y-train and y-test, i.e. converts a 1-D array of 60000 to an array of shape (60000, 10). The labels (correct integers) are converted into binary class matrices, meaning that the correct number is indicated by a 1 in the proper index instead of a number. For example the first y_train element: y_k=[0,0,0,0,0,5,0,0,0,0] represents the value 5 as index 5 is equal to 1.

Question 2: Network model, training, and changing hyper-parameters [4 Points]

A) How many layers does the network in the notebook have? How many neurons does each layer have? What activation functions and why are these appropriate for this application? What is the total number of parameters for the network? Why does the input and output layers have the dimensions they have?

```
## Define model ##
model = Sequential()

model.add(Flatten())
model.add(Dense(64, activation = "relu"))
model.add(Dense(64, activation = "relu"))
model.add(Dense(num_classes, activation="softmax"))
```

The layers created in these ines of code resulted in

- 1 input layer
- 2 hidden layers
- 1 output layer

In total: `4` layers.

Where

- Input layer (Flatten) has `28 * 28 = 1 * 784 =` **`784`** neurons (*converts input from a 28x28 array into a 1-D array of 1x784.*)
- Hidden layer 1 (Dense) has `1 * 64 =` **`64`** neurons
- Hidden layer 2 (Dense) has `1 * 64 =` **`64`** neurons
- Output layer (Dense) has `1 * 10 =` **`10`** neurons

In total: `784 + 64 + 64 + 10 =` **`922`**

The application uses the ReLU activation function in the hidden layers and Softmax activation function in the output layer.

$$ReLU: f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} = max(0,\ x)$$

The **Rectified Linear Unit (ReLU)** function is a non-linear activation function that does not activate all neurons simultaneously, unlike other activation functions. If the output of the linear transformation is less than 0, the **ReLU function** deactivates the neurons, meaning that negative input values are converted to 0, and the neuron is not activated. When compared to the **Sigmoid** and **Tanh** functions, which use an exponent that is computationally slow when done frequently, the **ReLU function** is considerably more computationally efficient because only a small number of neurons are engaged.

Due to its linear, non-saturating quality, **ReLU** also speeds up the convergence of gradient descent towards the global minimum of the loss function. Because the gradient of **ReLU** is also very easy to compute compared to **Sigmoid**, this acceleration in convergence is true for both feedforward and backpropagation. Another advantage of **ReLU** is sparsity which occurs when x ≤ 0. Dense representations appear to be less useful than sparse representations. The more such units in a layer, the more sparse the representation becomes. On the other hand, **Sigmoid functions** are prone to producing non-zero values, resulting in dense representations.

$$Softmax: f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{num\_classes} e^{x_j}}$$

In contrast to other activation functions that yield a single output for a single input, **Softmax** generates multiple outputs from a single input array. Instead of a binary class solution, **Softmax** can be used to develop neural network models that can categorise more than two classes. **Softmax**, which calculates relative probabilities, is used over **Sigmoid**, whose probability values are independent, meaning the probability that the data point belongs to 1 class does not consider the other classes' probability.

The output layer's output dimension should be configured according to the model's predicted output, which for the handwritten digit problem can be one neuron per digit, i.e. a vector of 10 nodes. This layer's **Softmax** activation function provides a vector representing the probability distributions (0-1) of a list of possible outcomes (digits 0-9), allowing the neural network to output the digit with the highest probability value. **Softmax** is the only activation function recommended for the *categorical cross-entropy loss function* since it adjusts the output to fit the loss function's required properties. (It normalises input values into values between 0 and 1 and sums up to 1, which can be interpreted as the probability of the input value being in a specific class.)

The parameters in the network is the sum of the weights and biases.

The total number of weights (the number of connections between the neurons) are: ($\Sigma\ neurons\ in\ currrent\ layer\ *\ neurons\ in\ previous\ layer$)

`784 * 64 + 64 * 64 + 64 * 10 =` **`54912`**

The total number of biases (the number of nodes) are: ($\Sigma\ neurons\ in\ currrent\ layer$)

`64 + 64 + 10 =` **`138`**

In total, the network has `54912 + 138 =` **`55050`** parameters.

We can verify this by using the `summary()` method that is available for keras-models.

```
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
    Layer (type)                 Output Shape              Param #
=================================================================
    flatten (Flatten)            (None, 784)               0

    dense (Dense)                (None, 64)                50240

    dense_1 (Dense)              (None, 64)                4160

    dense_2 (Dense)              (None, 10)                650

=================================================================
Total params: 55,050
Trainable params: 55,050
Non-trainable params: 0
_________________________________________________________________
None
```

B) What loss-function is used to train the network? What is the functional form (mathematical expression) of the loss function? and how should we interpret it? Why is it appropriate for the problem at hand?

```
model.compile(
        loss=keras.losses.categorical_crossentropy,
        optimizer=tf.keras.optimizers.SGD(learning_rate = 0.1),
        metrics=["accuracy"]
)
```

From this line of code we can see that the **catergorical cross-entropy loss function** is used to train the network, which is an optimization function used in the case of training a classification model. This function classifies the data by predicting the probabilities of which class the data belongs to. The functional form (mathematical expression) of the loss function is:

$$\mathrm{Loss} = -\sum_{i=1}^{output \atop size} y_i \cdot \mathrm{log}; {\hat{y_i}}$$

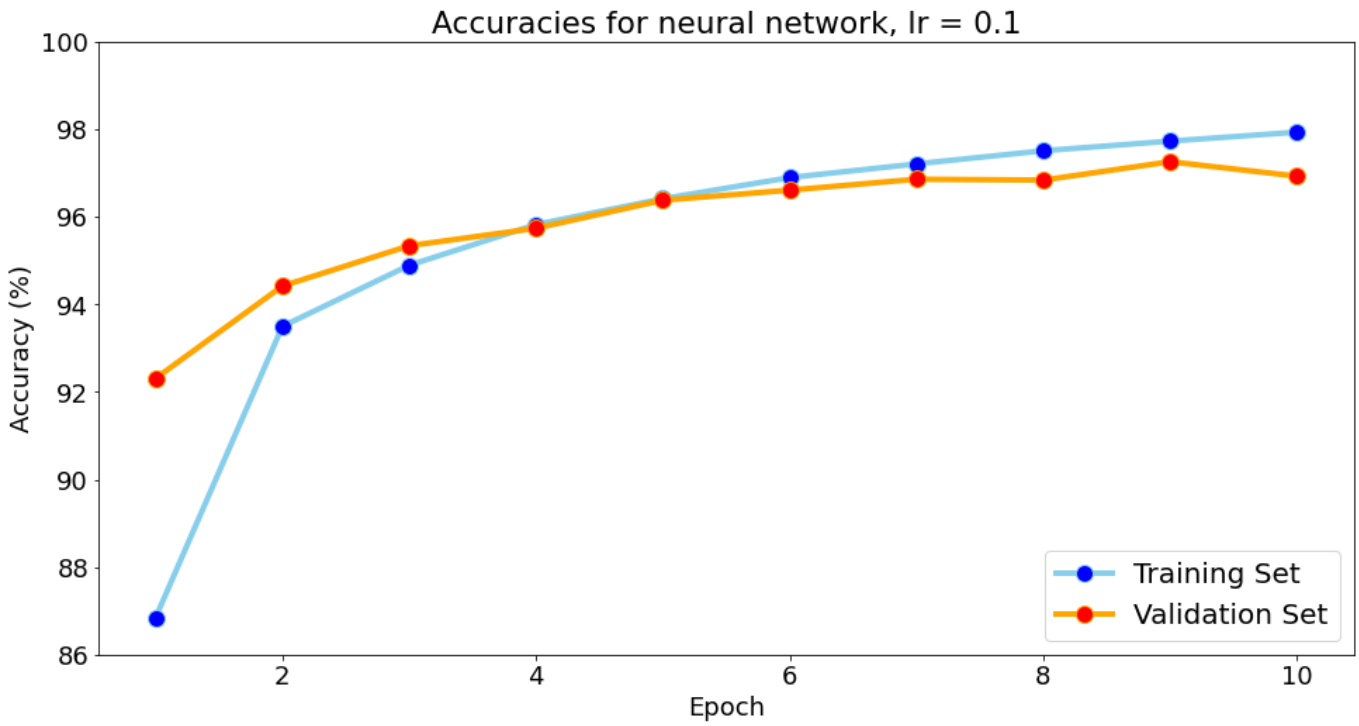
where $\hat{y}_i$ is the ith scalar value in the model output (predicted result), y_i is the corresponding target value (correct result), and the output size is the number of scalar values in the model output. Here $\hat{y}_i$ will only be 1 for one index in the $\hat{y}_i$ vector and 0 for every other number. Thus the loss will only be affected by the prediction corresponding to the actual number it represents,

as all other predictions will be multiplied by 0, meaning only the loss of the probability of the actual correct integer is considered and minimised.

This loss is a good measure of how distinguishable two discrete probability distributions are from each other. In this context, y_i is the probability that event i occurs, and the sum of all y_i is 1, meaning that precisely one event may occur. The minus sign ensures that the loss gets smaller when the distributions get closer.

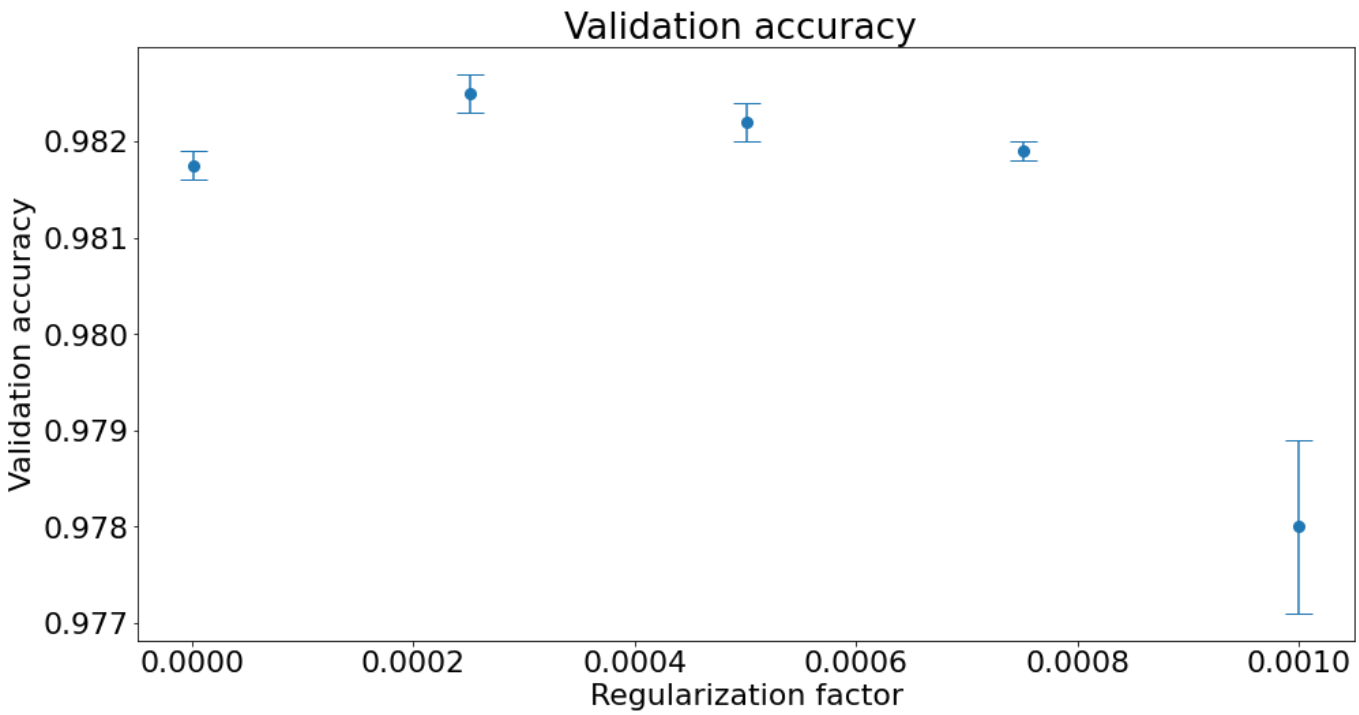
The **categorical cross-entropy loss function** is appropriate because this is a multi-class classification model and the model utilizes **softmax** as an activation function with multiple output labels. All other possibilities are erroneous, and only one option (represented by a 1 in the vector) is correct (represented by a 0). The many numbers that the graphic represents are the options. It also works nicely with **softmax** since it ensures the input for the log function is a positive value. In other words, when there is only one correct result (class) and several erroneous results, it is clear that each number can only belong to one class.

C) Train the network for 10 epochs and plot the training and validation accuracy for each epoch



D) Update model to implement a three-layer neural network where the hidden-layers has 500 and 300 hidden units respectively. Train for 40 epochs. What is the best validation accuracy you can achieve? – Geoff Hinton (a co-pioneer of Deep learning) claimed this network could reach a

validation accuracy of 0.9847 (http://yann.lecun.com/exdb/mnist/) using weight decay (L2 regularization of weights (kernels): https://keras.io/api/layers/regularizers/). Implement weight decay on hidden units and train and select 5 regularization factors from 0.000001 to 0.001. Train 3 replicates networks for each regularization factor. Plot the final validation accuracy with standard deviation (computed from the replicates) as a function of the regularization factor. How close do you get to Hintons result? – If you do not get the same results, what factors may influence this? (hint: What information is not given by Hinton on the MNIST database that may influence Model training)




```
Our maximum validation accuracy:  0.9828
Hilton validation accuracy:  0.9847
The difference between our validation accuracy from Hilton:  0.0019
```

When assessing the highest validation accuracy of the three replicates performed for each of the five regularisation parameters examined, the highest validation accuracy attained by applying the three-layer neural networks differs from Hilton's by roughly 0.19%. Because only three replicates were conducted for each regularisation factor, each result significantly impacts the mean, and more repeats could lead to greater accuracy.

The number of epochs ran and the learning rate, which Hilton does not provide, impact accuracy. In the figure for section C, the accuracy improves dramatically in the early epochs and then steadily converges. It is possible that increasing the number of epochs even further

(currently at 40) will not affect accuracy. The learning rate determines how quickly the model modifies (its weights), i.e. the step size used to minimise the loss function. The convergence of Hilton's validation accuracy might differ from ours if he utilised a different mix of step size/learning rate and epochs.

The regularisation factor has an impact on validation accuracy. When the accuracy of 0.9847 was received, Hilton did not specify which factor was employed. Other settings of the regularisation factor could have resulted in a higher validation accuracy because only five replicates were done. We utilise the "softmax," an activation function that penalises squared weights (l2 weight decay) in the output layer. However, we have only used five regularisation factors and trained three times for each, and we have not used a validation set to choose these weights. Instead, we just used averages inside the range specified in the question. As a result, we believe that if we choose a more acceptable regularisation factor, we should be able to achieve a validation accuracy equal to Hinton's.

Question 3: Convolutional layers [2 Points]

A) Design a model that makes use of at least one convolutional layer – how performant a model can you get? -- According to the MNIST database it should be possible reach to 99% accuracy on the validation data. If you choose to use any layers apart from convolutional layers and layers that you used in previous questions, you must describe what they do. If you do not reach 99% accuracy, report your best performance and explain your attempts and thought process

The table below gives a rundown of the CNNs we tested and their corresponding validation accuracy

Name	Classifier	Maximum Validation Accuracy [%]
CNN1	conv2d_32, maxpooling2d, flatten, dense_128, dense_softmax	98.20%
CNN2	conv2d_32, conv2d_64, maxpooling2d, flatten, dense_128, dense_softmax	98.63%
CNN3	conv2d_32, maxpooling2d, conv2d_64, maxpooling2d, flatten, dense_softmax	98.03%
CNN4	conv2d_32, maxpooling2d, dropout_0.5, flatten, dense_128, dense_softmax	98.58%
CNN5	conv2d_32, conv2d_64, maxpooling2d, dropout_0.5, flatten, dense_128, dense_softmax	98.93%
CNN6	conv2d_32, conv2d_64, maxpooling2d, dropout_0.5, flatten, dense_128, dropout_0.5, dense_softmax	99.06%

The correct optimal approach may differ from the one we discovered, which might be determined by performing numerous replicates for the same CNN and averaging the results. Due to time constraints, we chose to use the best we could with only one replication.

We attempted different network construction approaches to achieve a high enough validation accuracy. All models used two dense layers, 1-2 convolutional layers and a max-pooling layer, but they also used 1-2 dropout layers in some cases.

- For the dense layer (non-output layer), we changed the number of neurons to 128.
- For the convolution layers, we tested both 32 and 64 filters of size 3x3.
- We supplied 2x2 as the kernel size for the max-pooling layer. We set the stride size equal to the kernel size. This layer looks in a 2x2 kernel and produces the maximum value for that range to downsample the data.
- The dropout layer deactivates specific neurons at random. The input parameter is the frequency of deactivated neurons, which we set as 0.5. This layer helps in avoiding overfitting by deactivating neurons.

The optimal approach we discovered can be further fine-tuned to get better results. Below we provide a summary of our process and results:

1. For the first approach, we added a convolution layer with 32 filters and a max-pooling layer, which resulted in a maximum validation accuracy of 98.20%.
2. We added a second convolution layer with 64 filters for the second approach, which resulted in an increased maximum validation accuracy of 98.63%.
3. We wanted to test a method described in a publication by Yann LeCun et al . on the convolutional network for image recognition of hand-written digits [http://www.iro.umontreal.ca/lisa/pointeurs/handbook-convo.pdf]. The only difference is that instead of a convolutional layer, we used a dense layer as the output layer. We used two convolutional layers, followed by a max-pool layer for each convolutional layer, i.e. added a second max-pooling layer before the second convolution layer. This change resulted in a reduced maximum validation accuracy of 98.03%. So for the other approaches, we stuck to using the max-pooling layer after the two convolution layers.
4. We swapped the second convolution layer in approach 2 with a dropout layer for the fourth approach, which resulted in a reduced maximum validation accuracy of 98.58%.
5. For the fifth approach, we added a dropout layer after the max-pooling layer for the CNN used in approach 2, which resulted in an increased maximum validation accuracy of 98.93% (close to the proposed 99%).
6. We added a second dropout layer just before the output layer for our best performing model, which resulted in a maximum validation accuracy of 93.06%.

All the different CNN models had similar validation accuracy, but we would still say that "CNN6" performed the best since it achieved the highest validation accuracy. We trained and saved this best performing model to later use in question 4.

The following is the code for the best performaning model:

```
## Define model ##
model = Sequential()

# add convolutional layer with 32 convolution filters used each of size 3x3
model.add(Conv2D(
        32,
        kernel_size = (3, 3),
        activation = 'relu',
        input_shape = input_shape))

# add convolutional layer  with 64 convolution filters used each of size 3x3
model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu'))

# add max pooling layer to downsize input
model.add(MaxPooling2D(pool_size = (2, 2)))

# add dropout layer to help prevent overfitting
model.add(Dropout(0.5))

# add flatten layer to decrease the dimensions
model.add(Flatten())

# add dense layer to get fully connected and get all the relevant data
model.add(Dense(128, activation = 'relu'))

# add dropout layer
model.add(Dropout(0.5))

# add output layer with softmax to get probabilities
model.add(Dense(num_classes, activation = 'softmax'))

# add the loss-function and compile
model.compile(
        loss = keras.losses.categorical_crossentropy,
        optimizer = tf.keras.optimizers.SGD(learning_rate = 0.1),
        metrics = ["accuracy"]
)

epochs = 10
fit_info = model.fit(
        x_train, y_train,
        batch_size = batch_size,
        epochs = epochs,
        verbose = 1,
        validation_data = (x_test, y_test))

score = model.evaluate(x_test, y_test, verbose = 0)

print(f"Test loss: {score[0]}, Test accuracy {score[1]}")
```

```
# save the model to use in question 4C.
model.save("models/best_model_3", compile = True)
```

B) Discuss the differences and potential benefits of using convolutional layers over fully connected ones for the particular application

Convolutional layers are good at recognising patterns, which is one of the main reasons for using them. The layer could detect edges or circles at a low level, but it could recognise faces or whole numbers at a higher level.

By sliding filters over the input image, convolutional layers in a convolutional neural network extract features from the input. Unlike fully linked layers, Convolutional layers use parameter sharing, which means that all neurons in a feature map share weights. Patterns can be detected by sharing weights, and the model will be more efficient because the number of parameters to choose from will be reduced.

In completely linked layers, all input neurons are connected to all output neurons. Because additional parameters (weights) are placed in these layers, the model may be more prone to overfitting, resulting in the model memorising the test data input into the system rather than discovering patterns.

Question 4

A) The notebook implements a simple denoising deep autoencoder model. Explain what the model does: use the data-preparation and model definition code to explain how the goal of the model is achieved. Explain the role of the loss function? Draw a diagram of the model and include it in your report. Train the model with the settings given

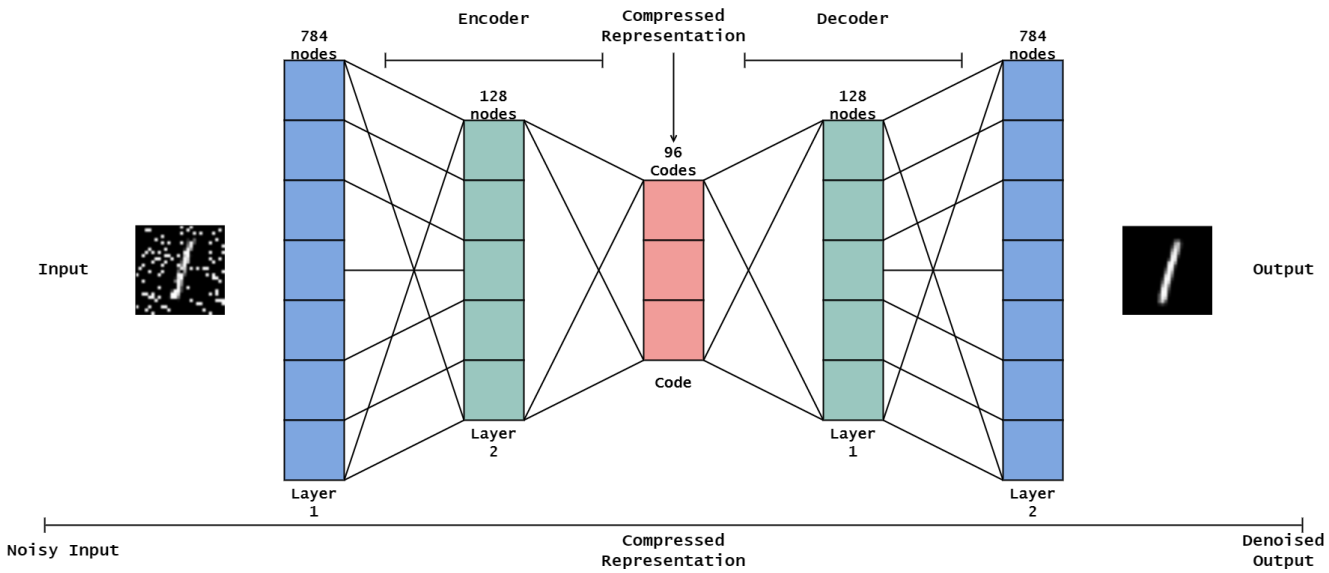
The model will be trained by adding noise to the photos and compressing (dimensional reduction) and decompressing the images to remove the noise and map images with some noise as the correct integer result.

The `salt_and_pepper ()` function modifies the x-train and x-test data sets by adding noise. This function sets specific pixels to black or white at random (binomial distribution). The images are encoded using a factor of `784/128=6.125`. The data is subsequently compressed to the dimension of latent dim by another layer. The encoded images are then transmitted to another layer, the decoder, where the compressed images are resized sequentially to 784 once more -

the images have been reconstructed (denoised). After compression and decompression, the autoencoder translates the actual input images to their reconstruction. Only the encoding and decoding parts of the model are constructed.

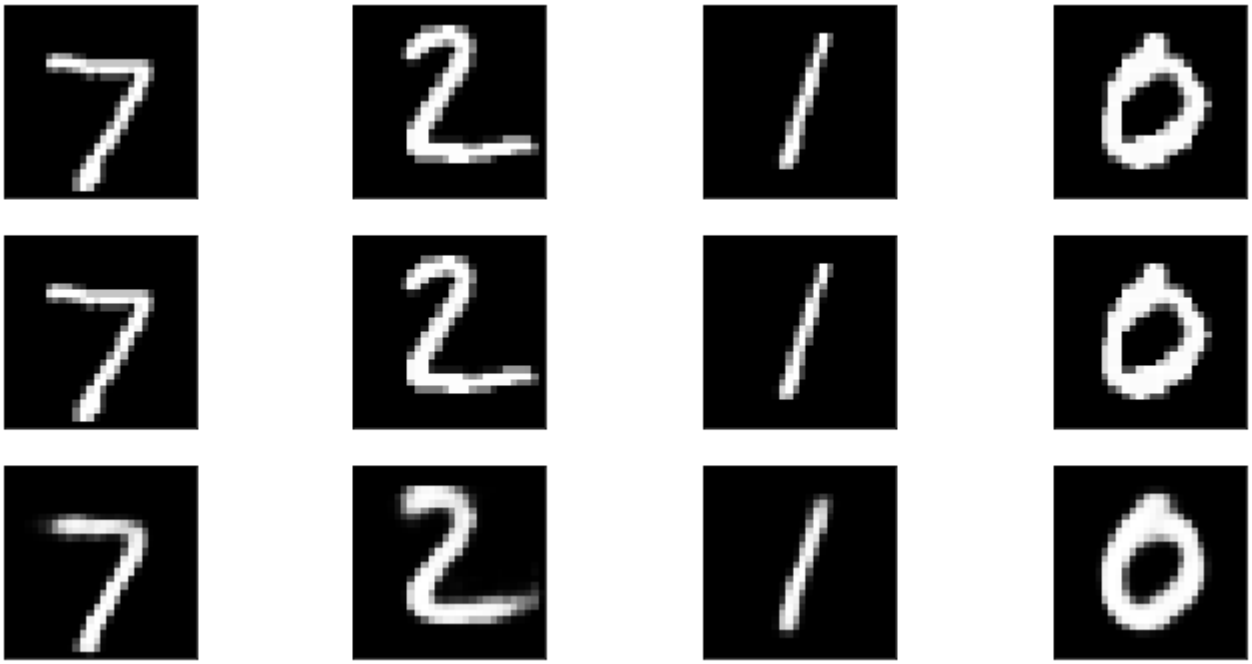
During dimension reduction, the features of the images were stored as vectors in the latent space by compressing and decompressing them. The features are then mapped during picture reconstruction, and the decoder removes the noise. The autoencoder smoothes the digits and removes all noise from the images.

The model minimises the reconstruction loss, the distance between the original images and their corresponding reconstruction after autoencoding, using the loss function binary cross-entropy (since we are working with binary inputs). The loss function forces the autoencoder model to match decoded images as closely as possible to original images.

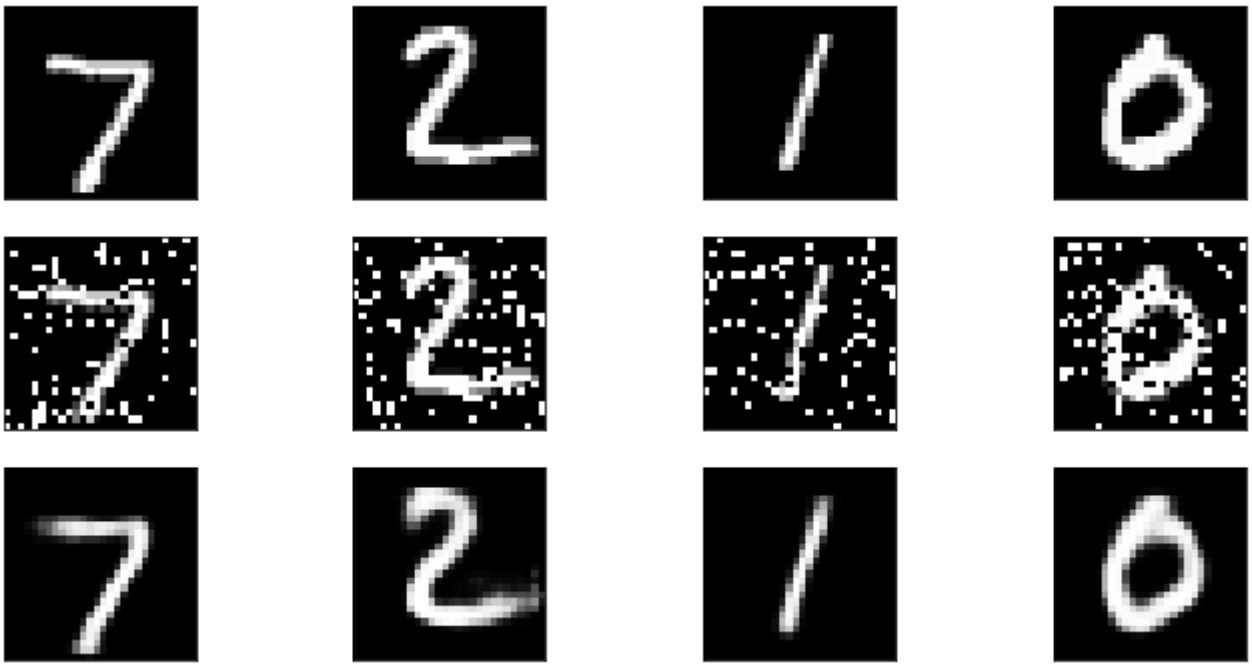


B) Add increasing levels of noise to the test-set using the salt_and_pepper()-function (0 to 1). Use matplotlib to visualize a few examples (3-4) in the original, “seasoned” (noisy), and denoised versions (Hint: for visualization use imshow(), use the trained autoencoder to denoise the noisy digits). At what noise level does it become difficult to identify the digits for you? At what noise level does the denoising stop working?

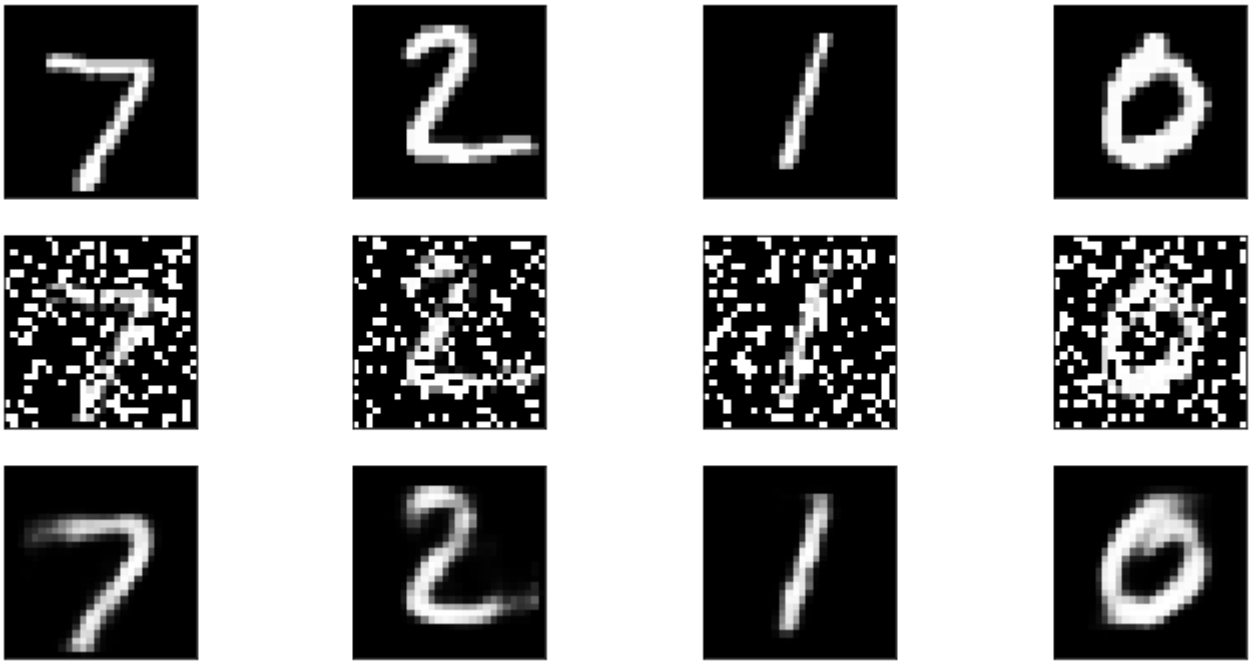
Noise: 0



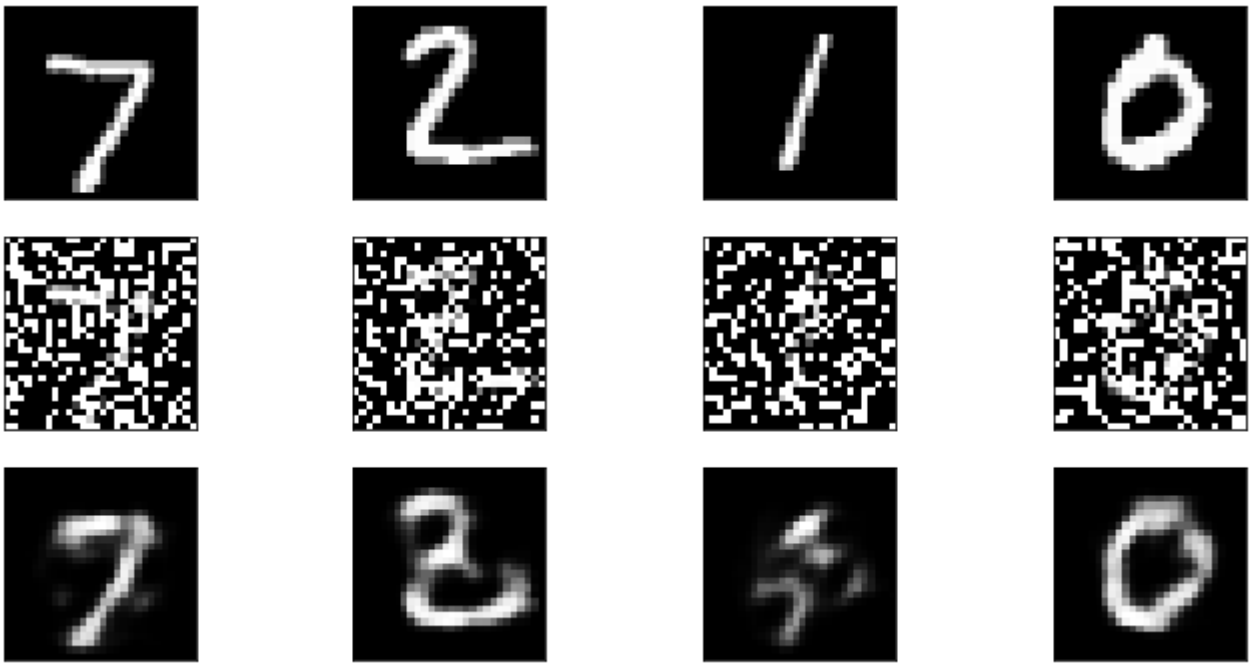
Noise: 0.2



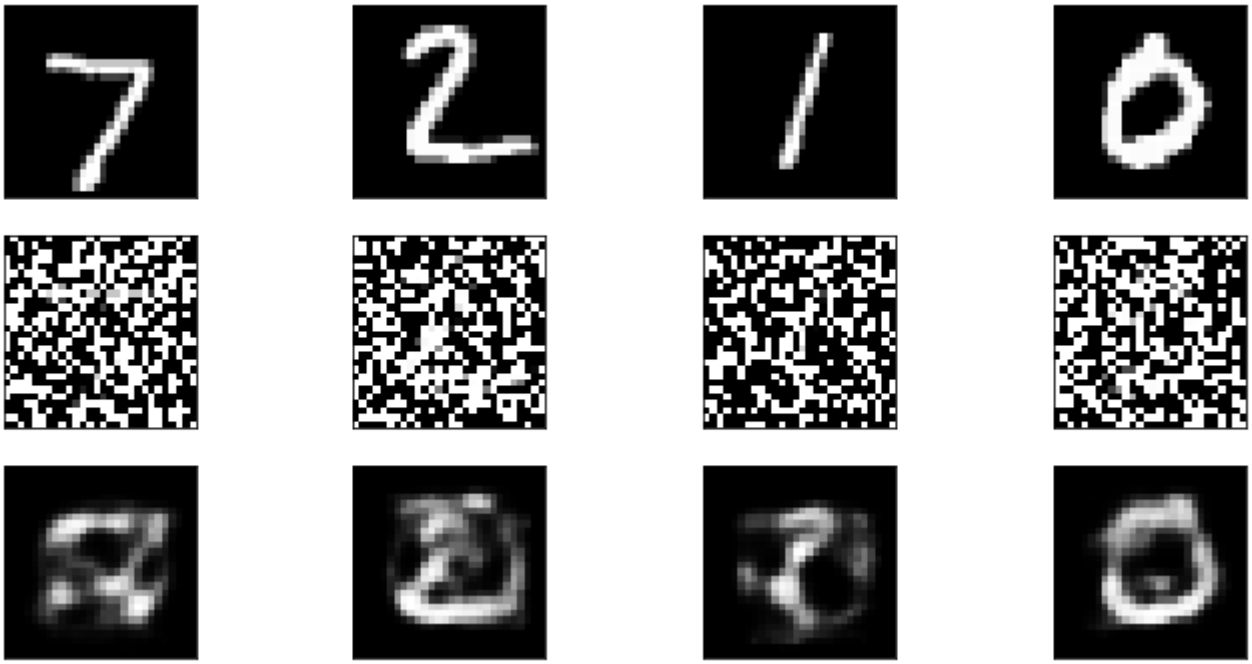
Noise: 0.4



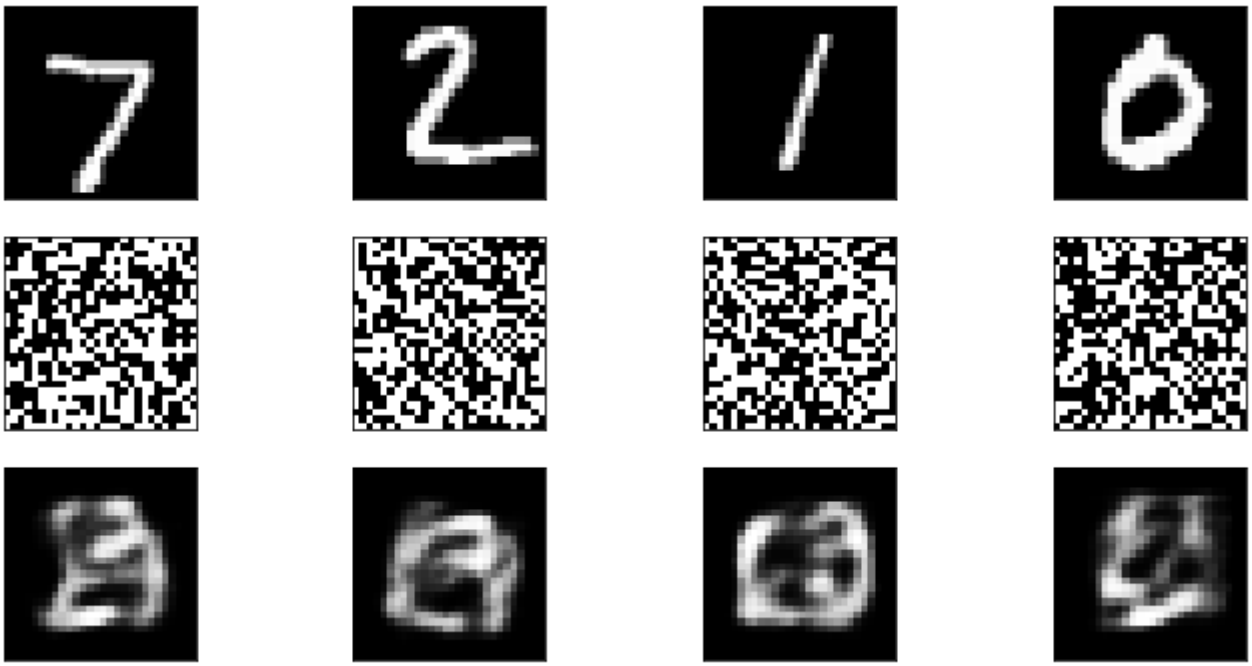
Noise: 0.6



Noise: 0.8

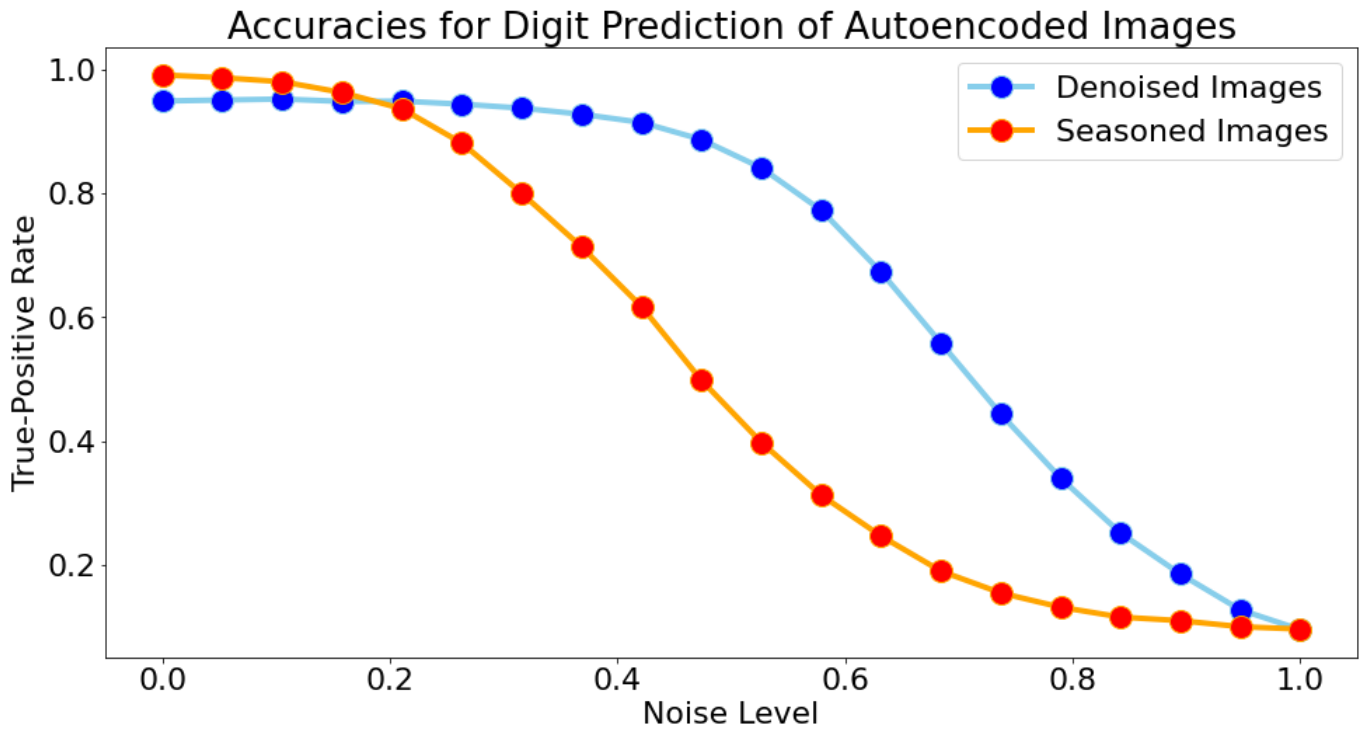


Noise: 1



At noise level 0.6, it becomes hard for us to identify what digit an image represents, but the autoencoder is still somewhat helpful at this stage. However, at the noise levels of 0.8 and 1, the decoded images do not show the actual number and can be said to the levels where the autoencoder stops working.

C) Test whether denoising improves the classification with the best performing model you obtained in questions 2 or 3. Plot the true-positive rate as a function of noise-level for the seasoned and denoised datasets – assume that the correct classification is the most likely class-label. Discuss your results



When we plot the true-positive rate as a function of the noise level, we get something that looks like an inverted and shifted sigmoid curve. The true-positive rate is relatively high until a noise level of 0.4 is reached when applying the autoencoder before evaluating the model from question 3. The function exhibits a significant fall at a noise level of 0.6 due to the high degree of noise stated in question 4B; even the human eye has difficulty distinguishing which digit the image represents.

Assume the autoencoder is not utilised, and the model directly assesses the images with noise. The true-positive rate is much lower, especially in the range of 0.2 to 0.8, indicating that autoencoders can eliminate noise and improve image evaluation.

The true-positive rate for both noised and denoised photos is close to zero for a noise level of 1 because very little of the original image is maintained, but all pixels are set randomly.

The seasoned images have a greater true-positive rate than the denoised images at a noise level of 0, indicating that the autoencoder is a trained model that can make mistakes. Denoising photos with no noise can result in less clear digits than when the image is used directly.

D) Explain how you can use the decoder part of the denoising auto-encoder to generate synthetic “hand-written” digits? – Describe the procedure and show examples in your report

When compressing an image, the compressed data points are stored as a vector defining the image's features, lowering the image's dimensions. The autoencoder's encoder transforms the image into a latent space representation. Because the features are compressed in this latent space, comparable data is clustered and can be modified to make the image look different from the original.

A new image can be formed by interpolating this latent space, and a perfect digit can be reconstructed as a not-so-perfect digit that seems handwritten. The encoder compresses the image, then resets the vectors to a value between the two images used for interpolation (the function "interpolate" below), and finally decodes the image - see code below. The decoder is now utilised to recreate the image from the latent space with some adjusted characteristics. It is also feasible to employ images representing different numbers, resulting in more variance in the images and images where the human eye cannot distinguish between two different numbers. On the other hand, the model trained above will always classify the image as a digit.

Another method for producing handwritten numbers is to add noise to the image and then run it through the autoencoder, as described in the subquestions above. Even if the source image were a flawless digit, the autoencoder would output slightly different digits by adding some random noise. As a result, the noise + autoencoder will replicate human behaviour because no two digits will be identical. See the photos above for an example of how we compressed and uncompressed the digits using various noise levels (4B).