Assignment 6 - Neural Networks
Group 62
Fredrik Lilliecreutz & Erling Hjermstad

Time spent on the module:
Fredrik Lilliecreutz: 15 hours
Erling Hjermstad: 15 hours

1) Preprocessing

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, the pixel values are normalized so that they are in the range 0-1, not 0-255. 0-255 is the normal numerical representation of pixel values, but in this case it is beneficial to make it a percentage, which is represented by 0-1 representing the percentage of grayscale in each pixel.

y-train and y-test are then initialized by the to_categorical function which converts the labels to one-hot encoded vectors. This format is easily comparable to the output of a multi-class neural net.

2) Network model, training, and changing hyper-parameters.

A) How many layers does the network in the notebook have?
The network has 4 layers, including the input flattening layer. So the layers are one input layer, two hidden layers and an output layer.
How many neurons does each layer have?

1) 28*28 = 784
2) 64
3) 64
4) 10

What activation functions and why are these appropriate for this application?
The two hidden layers use the Relu Function and the output layer uses the Softmax Function.
Softmax gives a probability as an output, i.e a value between 0 and 1. It is used to normalize the inputs to the model, which can range from large numbers, small numbers negative etc, to probabilities that sum up to 1. Large numbers will still be reflected in the probability but the output from the Softmax function will always be a value between 0 and 1. With asymptotes to both 0 and 1. This is compatible with the categorical cross entropy loss-formula as it does not accept 0 or negative values (due to a logarithm).

The relu function is the most common activation formula as it has a constant gradient which makes learning faster. The fact that it maps all negative values to 0 gives the neural network a characteristic known as sparsity. A sparse network has a lot of 0 weights/values, which makes backpropagation, and thus training, faster.

What is the total number of parameters for the network?
55050 parameters; 54912 weights and 138 biases.
Why do the input and output layers have the dimensions they have?
The input layer has the dimension because the images are represented by 28*28 = 784 pixel-values. The output gets its dimension because we have 10 labels, i.e 10 possible items that each input can correspond to. We want to break down all the input data to the type of items we are looking for and output the possibilities of each item.

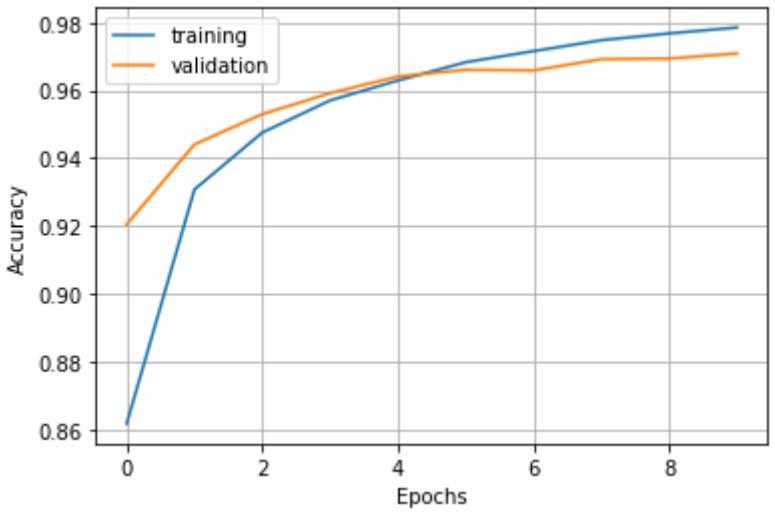
B) What loss-function is used to train the network?
Categorical crossentropy
What is the functional form (mathematical expression) of the loss function? and how should we interpret it?

$Loss = -\sum_i y_i log\, \hat{y}_i$ where $\hat{y}_i$ is the output at node i, and y is the one-hot encoded actual label. As both vectors sum to one they can be viewed as probabilities. The greater the loss, the further we are away from the actual one-hot encoded label. However this does not necessarily mean that we are classifying wrong.

In information theory entropy encoding is a lossless compression algorithm based on Shannon's coding theorem. Here each symbol has code length $-\ log\ P_i$.where P_i is the probability of symbol i appearing. $-\sum_i P_i log\ P_i$ is then expected message length.
Crossentropy can be viewed as an encoding where the probabilities are sampled. This gives an expected message length of $-\sum_i P_i log\ \hat{P}_i$ where $\hat{P}$ is the sampled probability. The shorter expected message length, the better the compression is, and the closer we are to the true probabilities. i.e. it can be used to measure the similarity of two datastreams.

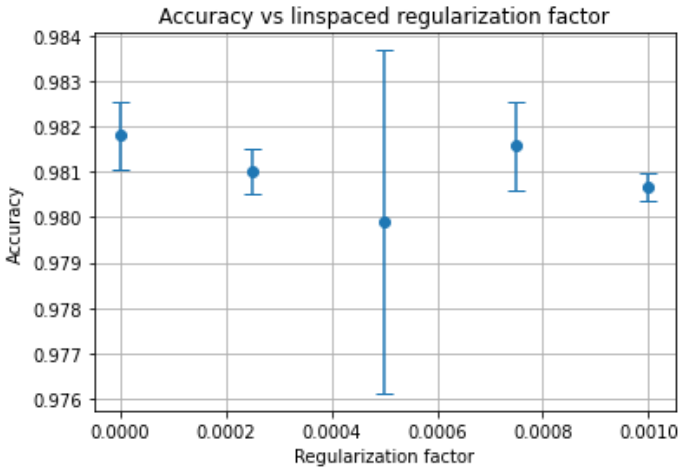
Why is it appropriate for the problem at hand?
Categorical crossentropy is commonly used for multiclass classification. That is, classification where each input corresponds to exactly one of the available classes. This is the case with handwriting. A digit is a single number between 0-9, never multiple digits.

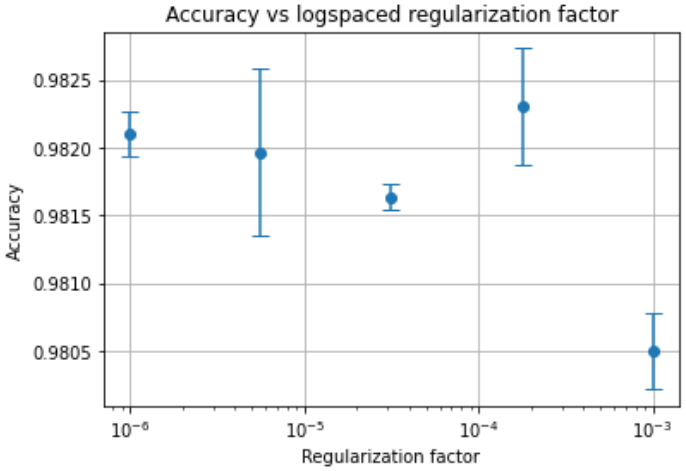
C) Train the network for 10 epochs and plot the training and validation accuracy for each epoch.



D) Update model to implement a three-layer neural network where the hidden-layers have 500 and 300 hidden units respectively. Train for 40 epochs. What is the best validation accuracy you can achieve?
We achieve a test accuracy of 98,12% on the first run of the network.

–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)
Training the 15 different networks takes around 30 minutes. Ran it first with linearly equally spaced regularizations values, then with logarithmically equally spaced regularization values. The results are saved in two csv files.



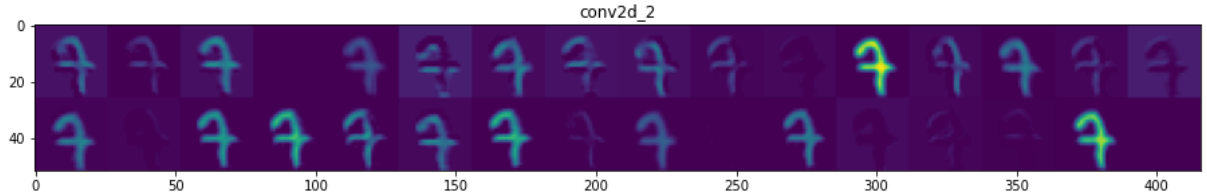


Our best result was 98,31 % with a regularization factor of 0.0005, however the standard deviation was also the greatest for this value. This result is just below the value Geoff Hinton claimed the network could achieve. We do not know all the parameters Hinton used, especially the learning rate could be interesting to tweak. However, it is apparent that our accuracy does plateau under training, suggesting that the model is either converging or overfitting, making tweaks in epochs and learning rate unnecessary (although we might be able to converge faster, and speed up our training). Hinton used a 3-layer neural network with 500 + 300 hidden units, softmax activation, cross-entropy loss-function and weight decay just like us. What optimizing function Hinton used is not given. He can have used true gradient descent instead of the stochastic version we use. This is computationally more demanding, but each improvement is guaranteed to be in the correct direction.

3) Convolutional layers.

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.
Best performance: 98.8%
Model:
2 convolutional layers, 32 and 64 filters
1 max pooling (2x2 groups)
1 dropout (rate 0.2)
1 flatten
2 dense (64 and 10 units)

The max pooling layer divides the image into 2x2 groups and throws out the 3 smallest pixels. At the output of this layer, the shape is half in each dimension of the size at the input. This layer extracts the important features of our image. We also used a dropout layer to counter overfitting before flattening and running through two dense layers. We tried a lot of different configurations, both with and without the max-pooling and dropout layers, and different hyper-parameters. We found that in most cases the model performed about equally good, with around 98 % accuracy.



To understand what hyper-parameters to use we made a plot which showed our image on the output of our convolution-, dropout- and maxpooling-layers. Here the first convolution-layer with 32 filters is shown.It is clear that some of the filters highlight features we as humans also use to identify digits. Such as loops and and lines. Still the correct

amount of filters, size of groups and dropout rate was still hard to reason about, so we mostly went for trial and error.

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

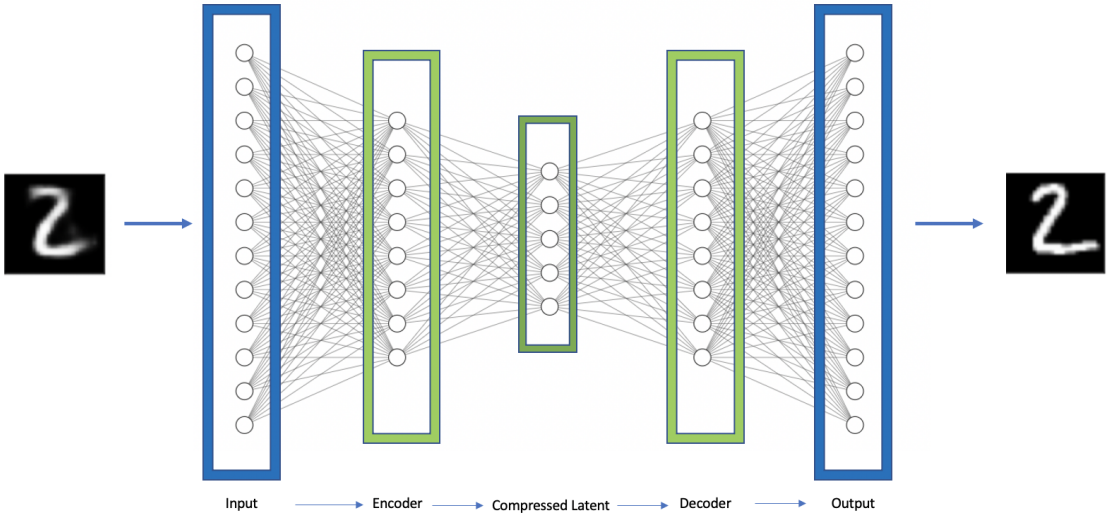
In fully connected ones, all of the input parameters directly affect the output layers. Although the nodes are connected, the weights do not necessarily have to affect the output layers. There are some nodes that just adds weight to specific nodes in the output layer. When using a conventional layer, all the input parameters are connected to the output layers, which adds additional flexibility. Inspired by an article from TowardsDataScience (https://towardsdatascience.com/convolutional-layers-vs-fully-connected-layers-364f05ab460b), the number of weights in each layer is smaller which is beneficial when using high-dimensional input such as image data. This strengthens our thesis that using a convolutional layer is particularly good for our application of neural networks, indicated by the instruction of the task, achieving about 99% accuracy instead of 98% in the previous task.

4) Auto-Encoders for denoising.

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.

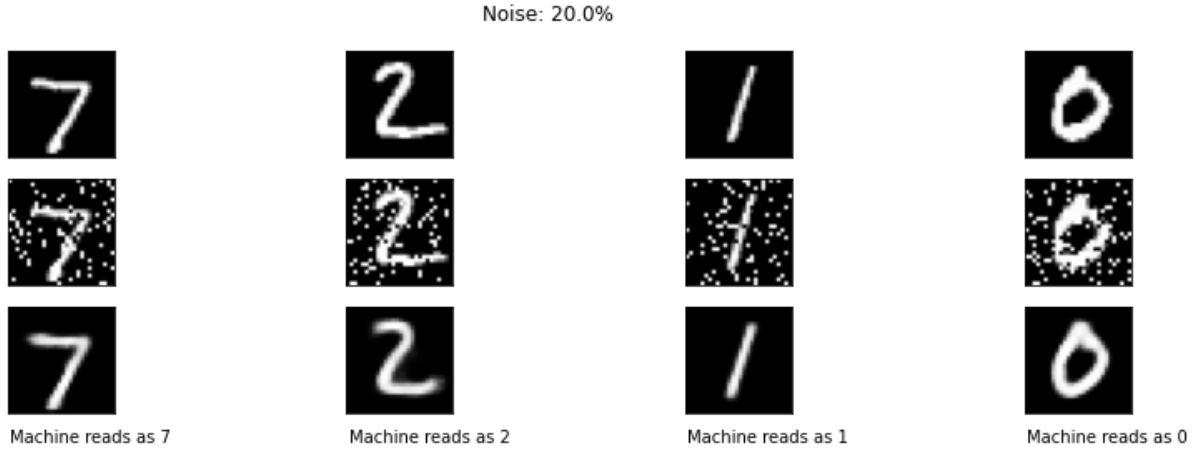
The model is compressing an image(encoding) and later decodes it. As described in the code, the size of the layer goes from 784 to 128, making it approximately 6 times smaller for the first layer. In the third layer, the size is the latent_dim which is 96 in the code. We also used a far smaller latent_dim, with equally good results. The goal is to perform an identity function where the input layer is equal to the output layer. However, the hidden layers are small in relation to the input layer. A solution to this is to add random numbers as noise in the model to try to improve the bottleneck described in our lecture notes, also seen in the code(salt & pepper function). This is to add some variations to the image of our original image in order to train the model to recognize patterns with some noise. The model later decodes it again and goes from 128 to 784 making the model's input equal to the output. Finally, it is being denoised and a reconstruction of the image has been made.

Python visualization tools did unfortunately not work properly for us. There were some other tools that could be used as well, but they required too much computing power. We are therefore drawing a visionary diagram for the neural network down below. Note, that it has been a down-scale due to computer power. The graph is describing how a picture, built upon pixels, is inputted into the input layer. The layers are then shrunk (encoded) to compressed latent. It is later decoded to its original size and we get an output from the model, the same number but clearer as the minor details, like disturbances, are lost in the compression.



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?



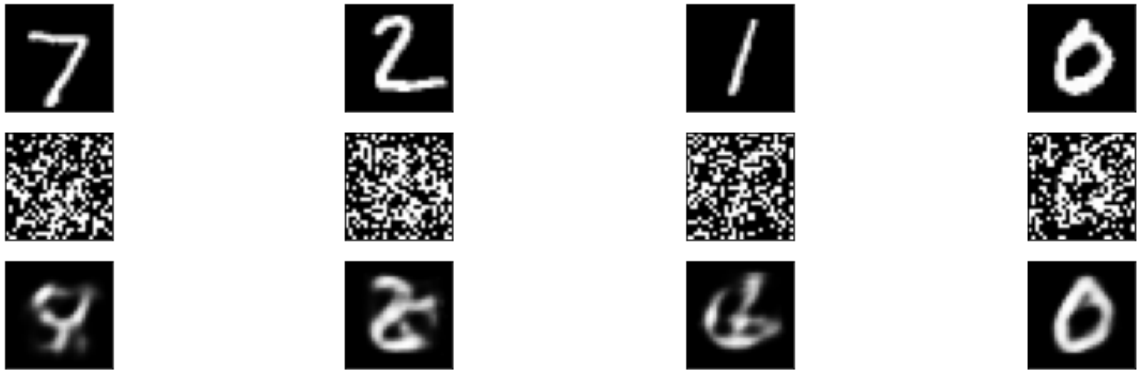
Noise: 32.0%

Machine reads as 7 | Machine reads as 2 | Machine reads as 1 | Machine reads as 0

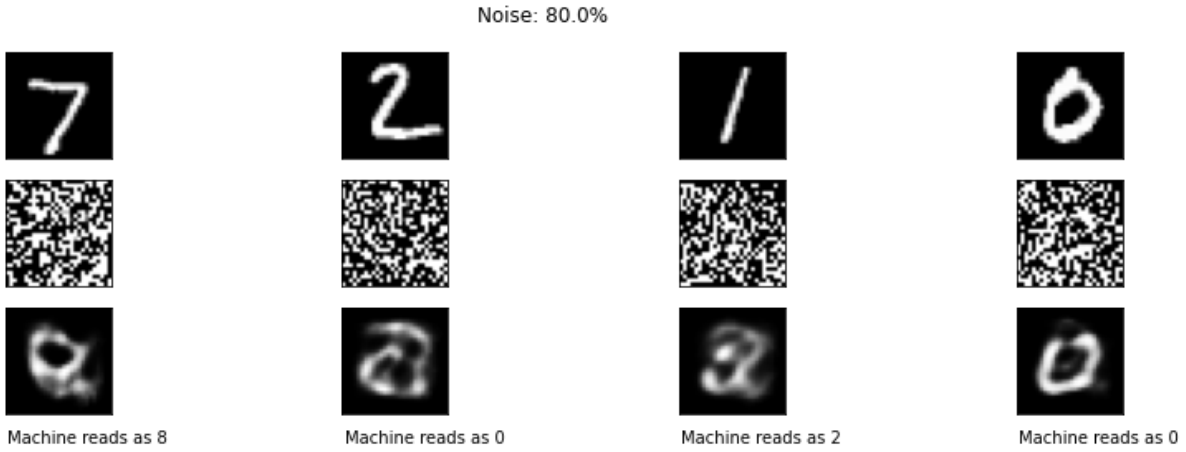
Noise: 56.0%

Machine reads as 7 | Machine reads as 2 | Machine reads as 1 | Machine reads as 0

Noise: 68.0%



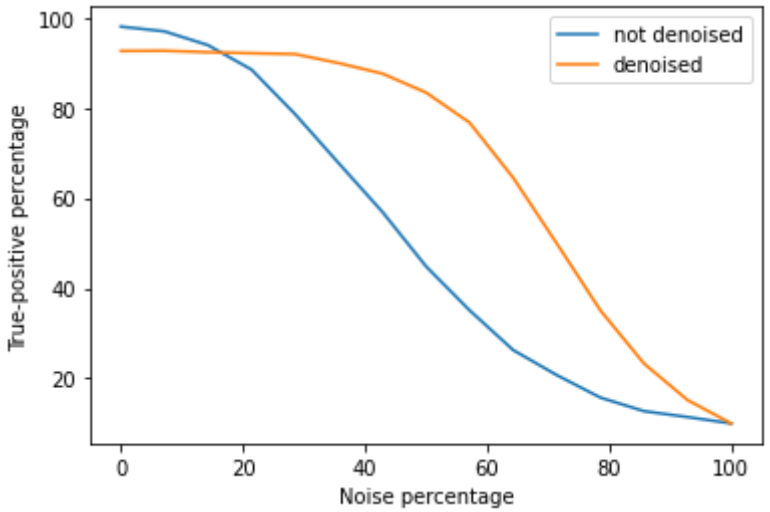
Machine reads as 4 | Machine reads as 8 | Machine reads as 6 | Machine reads as 0



We plotted with many different noise values. Some are shown above. The noise level of 0.5-0.6 is when it becomes very difficult to identify the digits. The original salt and pepper function was noise=0.4 which looked like the most noise you could have before the picture got too blurry. The denoising definitely stops working at noise = 0.6-0.7 as well so it seems like a noise level of 0.5 is the threshold for making the denoising work regularly.

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.

Since we got the best result from question 3, we decided to test it on that model.

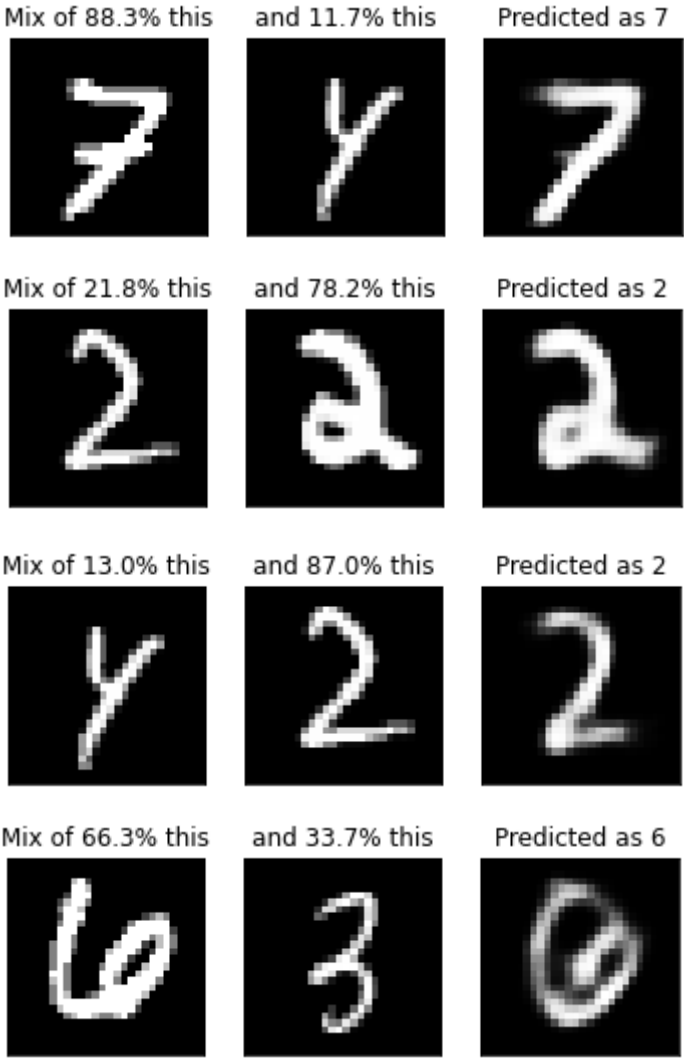


As can be expected the model which incorporates denoising before prediction performs best when there is noise in the image. However, the model which does no denoising performs better when there is no to little noise added to the image. This makes sense as the model is trained on images of this kind. The fact that it performs a bit worse on the denoising model when noise is not introduced indicates that the denoiser removes some vital information in some images. It can be seen that the denoiser can be a vastly better choice when noise is expected in the input, e.g. poor camera quality.

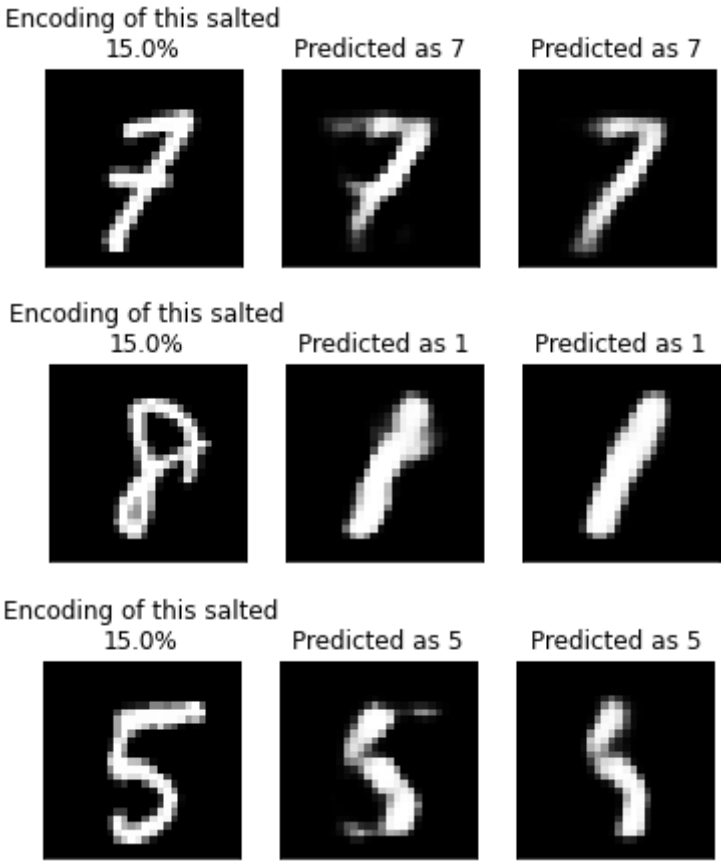
D) Explain how you can use the decoder part of the denoising auto-encoder to generate synthetic "hand-written" digits? – Describe the procedure and show examples in your report.

We tried 3 different methods for creating fake hand-written digits. All of them utilized the encoded version of known images. In method 1 we took 2 digits and mixed their encoding. In method 2 we salted the encoding of a single digit. In method 3 we created a very small latent space so that we hoped all possible encodings would map to something close to a digit, and created random encodings.

Method 1:
Encoded each image. Then did a random weighted sum of the encodings: (1-p)*e1+p*e2, where p was drawn from a uniform distribution from 0 to 1. Then decoded the mixed encoding. This method worked best in images of the same digits.

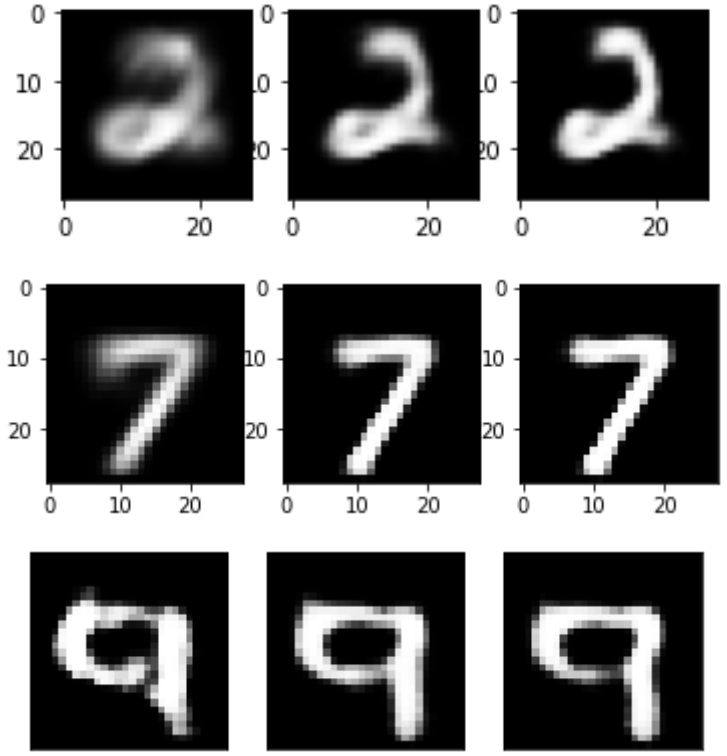


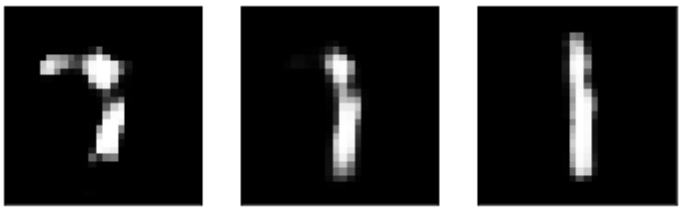
Method 2:
Salted the encoding of an image. Used a different salter than given, since we did not want to restrict the salt 0 and 1, but every number between the minimum and maximum of the input. Ran the output of the decoder through the entire autoencoder again to sharpen the image.



Method 2 was not reliable enough to be used. Maybe the latent space was too big?

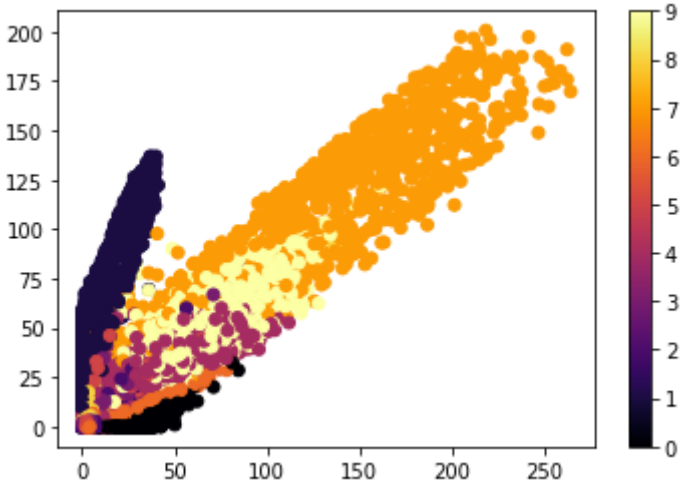
Method 3:
Trained a new autoencoder, with a tiny 2D latent-space.





Here the first image is generated from a random array in a 2D latent space. The second image is the first image run through our original autoencoder, which sharpens the image. The last image is the one before after another run through the autoencoder.

The autoencoder to 2D space is pretty useless for actually encoding digits. This is because it does not create distinct groups for the digits. This is shown in the plot below. Here the 2d space is plotted. The color signifies what digit it is. As can be seen there is significant overlap between digits and the space they occupy.



A fourth possible method could be to train the network in reverse. 10 input nodes and 28*28 output nodes. To introduce some kind of randomness in the outcome a random input could be added as well.