Lab 6 Neural Networks

Malte Carlstedt & Johan Östling

1)

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

From this line we divide the dataset into a training set and a test set. The Mnist dataset consists of 60 000 28x28 gray scale training images along with 10 000 28x28 gray scale test images.

x_train and x_test are numpy arrays with shape (60000, 28, 28) and (10000,28,28) respectively. Each element in these numpy arrays represents the shade of gray on a scale of a pixel, ranging from 0 to 255. 255 represents white and 0 represents black. lbl_train and lbl_test are also numpy arrays but with the shape of (60000) and (10000) respectively. The elements of these numpy arrays are values between 0-9 and represent the integer each image shows.

Next section of code is the preprocessing of the data:

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

x_train /= 255
x_test /= 255

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

The astype() function converts the type of an array. In our case we convert all values, representing the shade of gray, in x_test and x_train into type float32. This was necessary since the next two lines divide all values in the x sets with 255, and to avoid rounding mistakes in python at least either the denominator or numerator should be of type float. After the division, the values of the x sets now represent the ‘percentage of white’.

keras.utils.to_categorical() converts the values of the lbl sets, which are integers from 0 to 9, into binary representation with length of num_classes. For example the integer 5 will become [0,0,0,0,0,1,0,0,0,0], where 1 is at index 5.

2)

```
## 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'))


model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr = 0.1),
        metrics=['accuracy'],)

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('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

A)

The model consists of 4 layers. We can see that we add four layers by using the add() function. 3 of the 4 layers are Dense layers, which is a Keras layer where each neuron receives inputs from every neuron in the layer before it. The first layer is a Flatten layer and its purpose is to flatten the input. In our case it takes in matrices with 28 rows and 28 columns and outputs change the shape into a one dimensional list with the length of 784.

Number of neurons of each layer is:

1. 784
2. 64
3. 64
4. 10

Activation functions of each layer:

1. In the first layer all neurons are activated
2. ‘relu’
3. ‘relu’
4. ‘softmax’

ReLu is defined as R(z) = max(0,z). This means that R(z) is 0 when z is less than 0 and z when the input is above or equal to z. This activation is appropriate for this application since we only have positive values ranging from 0 to 1. Also the ReLu function has shown to converge quicker than other activation functions such as sigmoid function. Intuitively this seems reasonable since it outputs 0 for inputs below 0 instead of sigmoid which outputs other values from a more 'complicated' formula. In our case Sigmoid would be unnecessary since we do not have any inputs below 0.

SoftMax is used to normalize the output of a network. All of the neurons in our network get a probability distribution that all together sum to 1. For example the list [1,2,3,4] would get weight distributed between 0 to 1 where the sum of the list is 1. Where in this case since 4 is the largest value would have the largest value between 0 to 1. However if we convert values that are already below zero and which sum is not equal to 1 we do not get an accurate distribution of which element is the largest. Softmax is appropriate for this application since we want to be able to give a probability classification. It is important that our values all sum up to 1. By doing this we can give a value to the probability of a certain value belonging to category A or category B and so on.

Why the input layer and output layer have the dimensions that they have:

The first layer has 784 layers. The reason for this is that all neurons in the first layer represent the shade of gray for every pixel in an image. Since the image has 28x28 pixels the input layer has 784 neurons.

The output layer consists of 10 neurons. Each neuron in the output layer represents each label on the image. Since the images represent numbers from 0-9 the output layer should have 10 neurons.

B)

The loss function used is 'Categorical cross entropy'. The mathematical formula is:

$$loss = -\sum_{i=1}^{output\ size} y_i \cdot log\, \bar{y}_i$$

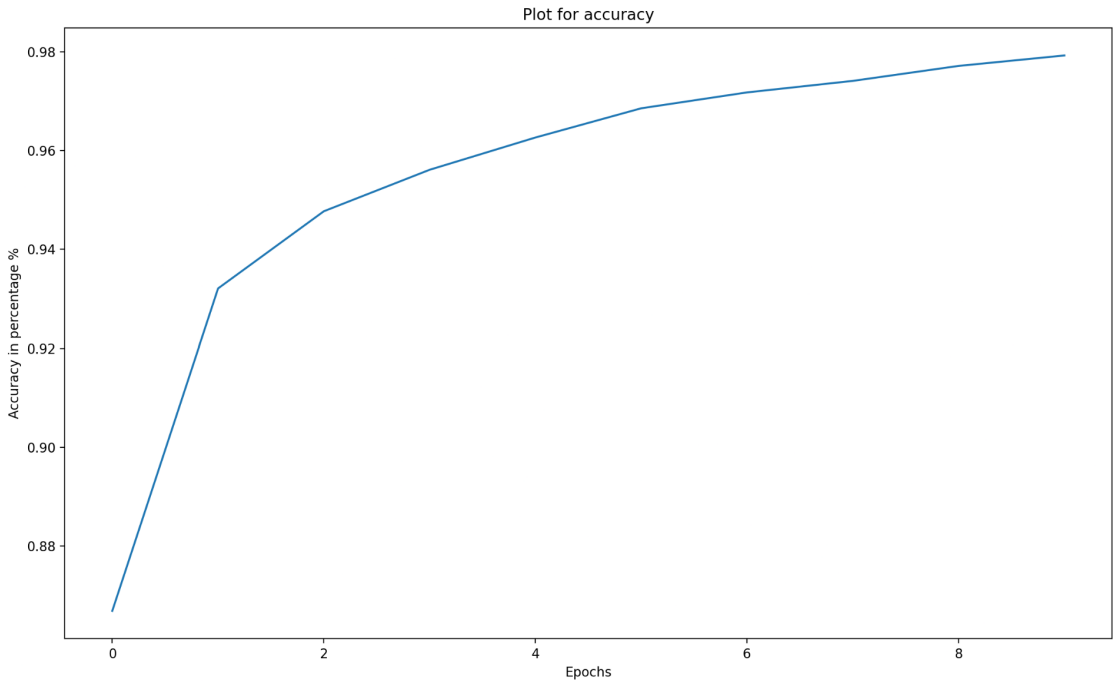
y_i is the target value and $\bar{y}_i$ is the value in the model output. y_i can either be 1 or 0 since we represent the integers binary where the index of where the 1 is in the list corresponds to the correct integer. $\bar{y}_i$ is a probability since we used the softmax function. Hence the formula takes the sum of each probability that the correct integer has been predicted. The loss can tell how disinghusable two probability distributions are.

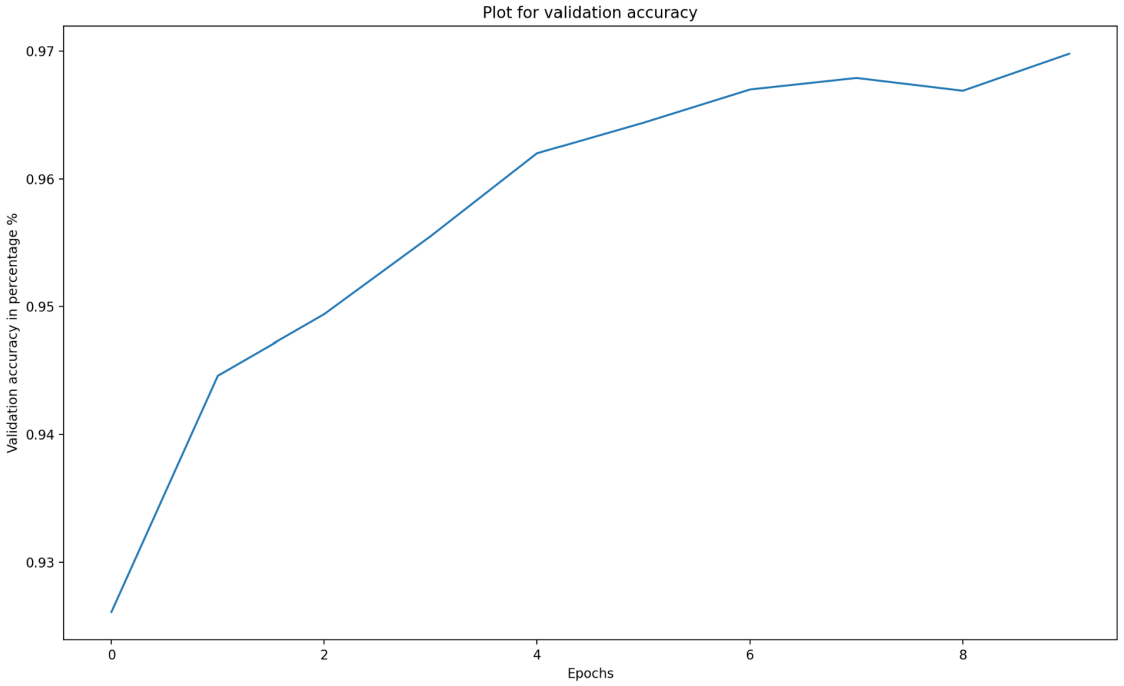
The categorical cross entropy is a loss function used for when we have two or more output labels. As this assignment consists of (10). This loss function uses the data to classify it and then predict whether this data belongs to a certain class or another. In this case using categorical cross entropy in conjunction with the softmax activation, the loss function learns

to give the correct answer a high probability output and everything else a low output. Since in our case there is only a right answer (the correct digit) and a lot of incorrect answers the categorical cross entropy is well suited for this assignment. Note that we need to use the softmax activation for the categorical cross entropy to work properly.

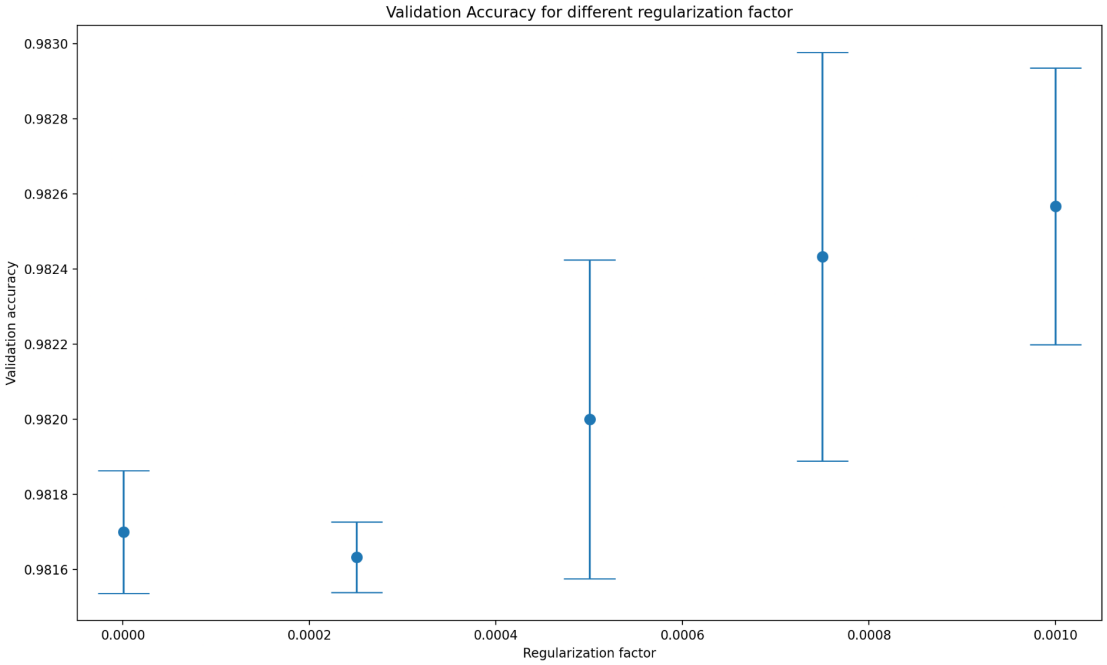
C)

```
# For printing accuracy
x = np.arange(epochs)
y = np.array(fit_info.history["accuracy"])

plt.title("Plot for accuracy")
plt.ylabel("Accuracy in percentage %")
plt.xlabel("Epochs")
plt.plot(x,y)
plt.show()


# For plotting validation accuracy.
plt.title("Plot for validation accuracy")
x = np.arange(epochs)
y = np.array(fit_info.history["val_accuracy"])

plt.ylabel("Validation accuracy in percentage %")
plt.xlabel("Epochs")
plt.plot(x,y)
plt.show()
```





D) Plot of of the final validation accuracy with standard deviation



Our highest validation score was: 0.9829999804496764. This is a difference of: 0.0017000195503236 from Hinton's score of 0.9847. The way Hinton might have managed to get a higher score is probably due to the fact that Hinton has not reported what parameters were being used. His model might have gone through more epochs which might have increased the accuracy to a small extent. Nor is it reported how many replications he used. Hilton did not state which values he used for the regularizers. Other parameters that might have influenced his higher score is which learning rate he used. There are multiple parameters that could differ from ours and therefore we did not expect to get the exact accuracy that Hinton did.

3)

A)

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(5,
5),activation='relu',input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, kernel_size=(5, 5),activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(num_classes,activation='softmax'))
```

With this model we managed to reach a validation accuracy score of 0.9915000200271606. To reach this result we used the convolutional layer Conv2d and on top of this we used MaxPooling2D twice. Every layer except MaxPooling2D are layers we have used before or are convolutional layers.

MaxPooling2d is used in between convolutional layers to basically reduce the computing load. It does this by reducing the amount of pixels by searching for the most valued pixels. In our MNIST dataset we have blurry numbers. In our case the most important pixels are the ones containing angles, shapes and specific attributes for a certain number. A pixel with just gray is not very giving for our convolutional layers. By searching in the image in a 2 x 2 manner.

```
model.add(MaxPooling2D(pool_size=(2,2)))
```

MaxPooling2D searches for the most valued pixel in each 2x2 square.

23	47
32	46

In this example, MaxPooling2D would only return the pixel which is valued at 47.

This means that the output from MaxPooling2D has reduced by a factor of 2 and this will decrease the computational load for the next convolutional layer (since there are fewer pixels). This will also give the next layer pixels that are more important for the process.

B)

A convolutional layer is not densely connected as fully connected ones are. Meaning that not all neurons receive inputs from each neuron in the previous layer. The neurons in the convolutional layer are connected to ‘local’ neurons in the previous layer. This means that convolutional layers are cheaper in memory and compute power. It is often used for image analysis and is appropriate for this application since the layer does not need to look for patterns in the whole picture. For example, in an image, a certain pattern like an ‘top edge’ does not appear over the entire image, this feature is placed locally in the image. Hence it is better to look for patterns in an image by inspecting parts of the image, and not the entire image.

4)

A)

Explain what the model does: use the data-preparation and model definition code to explain how the goal of the model is achieved.

```
#data preparation
flattened_x_train = x_train.reshape(-1,784)
flattened_x_train_seasoned = salt_and_pepper(flattened_x_train,
noise_level=0.4)

flattened_x_test = x_test.reshape(-1,784)
flattened_x_test_seasoneed = salt_and_pepper(flattened_x_test,
noise_level=0.4)
```

This is the data preparation. It uses the function salt_and_pepper() which randomly sets pixels in the images to values of 1 or 0, meaning white or black. First we flatten the images by doing reshape() so the 28x28 pixels becomes an array with length of 28*28=784. Then salt_and_pepper() is used on the flatten list to set random elements to 0 and 1. We do this preparation for both x_train and x_test. The reason why is because we want to make sure that after encoding and decoding these salt-and-peppered images they are restored to not being salt-and-peppered. If that is the case, then the model works.

```
latent_dim = 96

input_image = keras.Input(shape=(784,))
encoded = Dense(128, activation='relu')(input_image)
encoded = Dense(latent_dim, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded)

autoencoder = keras.Model(input_image, decoded)
encoder_only = keras.Model(input_image, encoded)

encoded_input = keras.Input(shape=(latent_dim,))
decoder_layer = Sequential(autoencoder.layers[-2:])
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
```

Here we create three models. Here we add layers that take care of the compression and decompression of the image. There are 2 encoded layers and the last one is a dense layer with the size of the decompressed image, in this case 96. Hence we have a compression factor of $\frac{784}{96} \simeq 8.2$. There are also 2 decoded layers and the last one is a dense layer with the same size we had on the images before they were encoded, which is 784.

The autoencoder model maps an input to its reconstruction. The encoder_only model maps an input to its encoded representation. And lastly, the decoder model takes an encoded image and produces the decoded image by using the layers of autoencoder.

Then the autoencoder model is trained by the following code:

```
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

fit_info_AE = autoencoder.fit(flattened_x_train_seasoned, flattened_x_train,
                epochs=32,
                batch_size=64,
                shuffle=True,
                validation_data=(flattened_x_test_seasoneed,
flattened_x_test))
```

Then it is visualized by the following code which is taken from https://blog.keras.io/building-autoencoders-in-keras.html. But instead of the encoder taking in the unseasoned images we changed it to the seasoned images.

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder_only.predict(flattened_x_test_seasoneed)
decoded_imgs = decoder.predict(encoded_imgs)

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(flattened_x_test_seasoneed[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

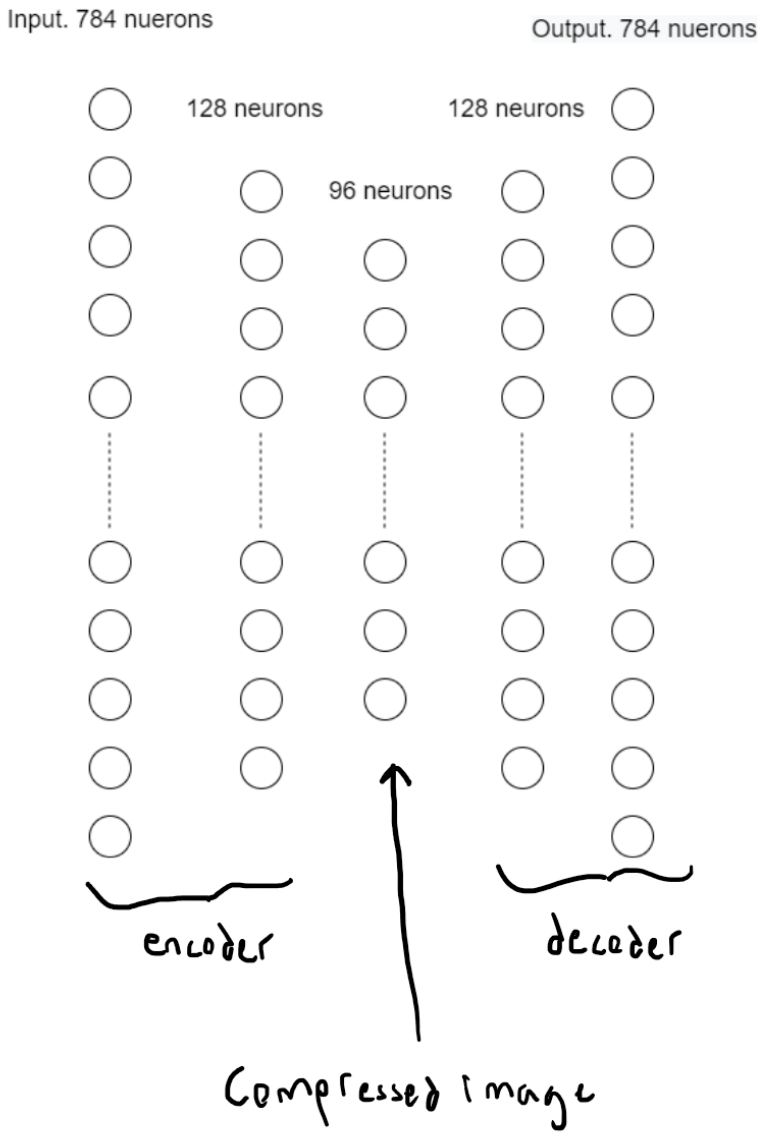
This gave the following plot:

Where the first row is seasoned images before they are encoded and decoded and the second row is after. As can be seen, the ‘noise’ in the images disappeared and therefore we can say that the model’s goal was achieved.

Explain the role of the loss function?

The loss function can be seen as a distance function of information loss from the original images to their reconstruction. The neural network optimizes the parameters for the autoencoder to minimize the loss. Hence the purpose of the loss function is to force the autoencoder model to make the output image as close to the input image as possible.

Draw a diagram of the model and include it in your report. Train the model with the settings given.

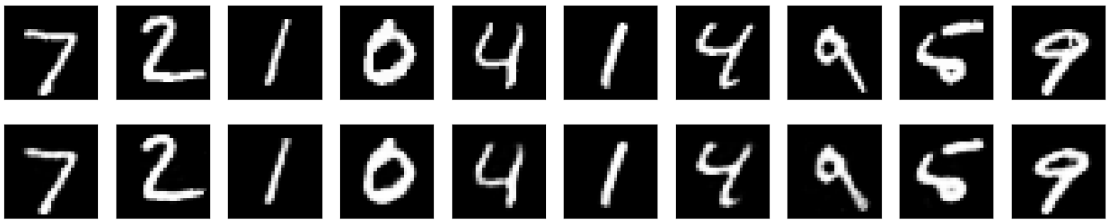


This is the model. Imagine that every neuron in each layer is connected with every neuron in the previous layer. I could not find a good program to draw a neural network diagram.

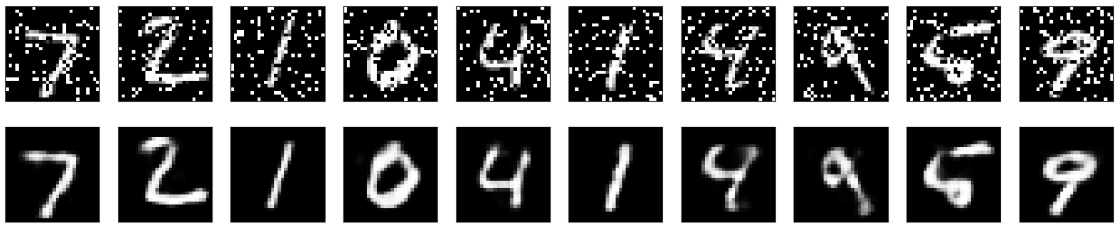
B)

We will plot the seasons images with their corresponding unseasoned image after using the autoencoder for five different value for noise level, noise level = [0, 0.2, 0.4, 0.6, 0.8,1]

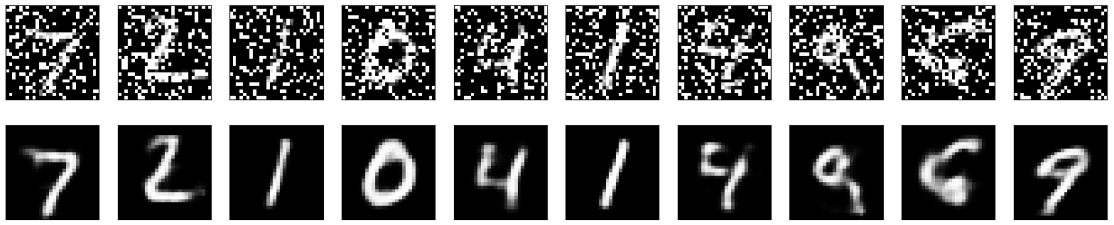
noise level = 0



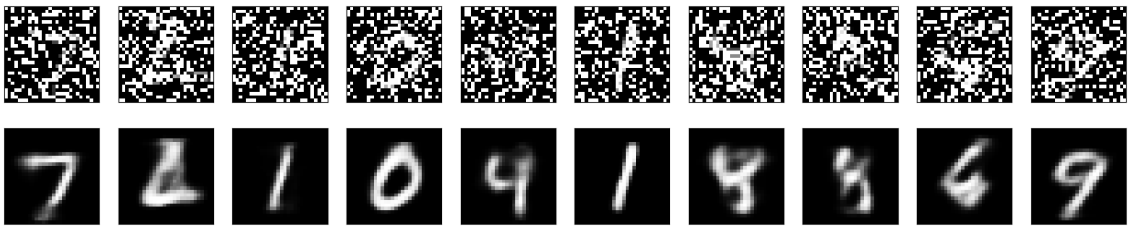
noise level = 0.2



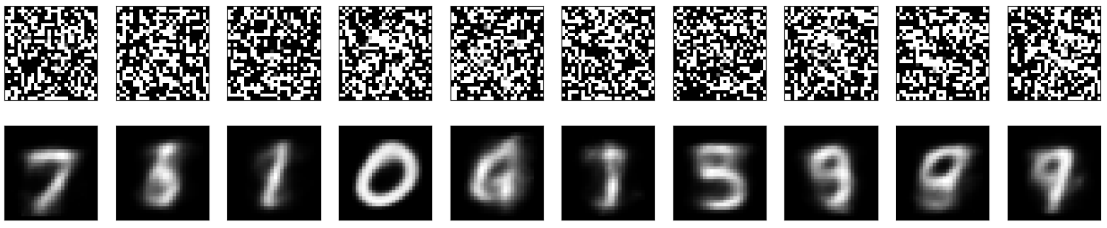
noise level = 0.4



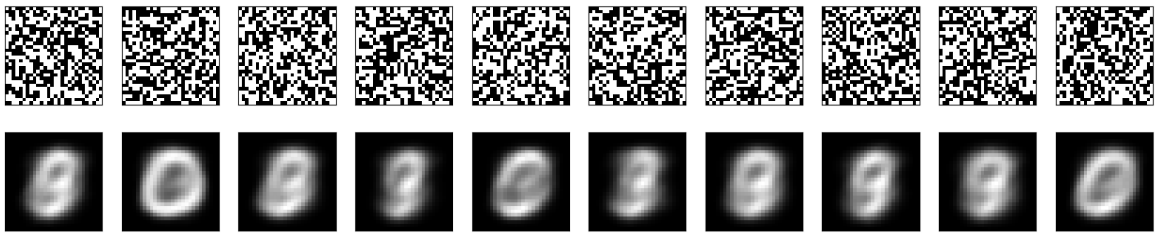
noise level = 0.6



noise level = 0.8

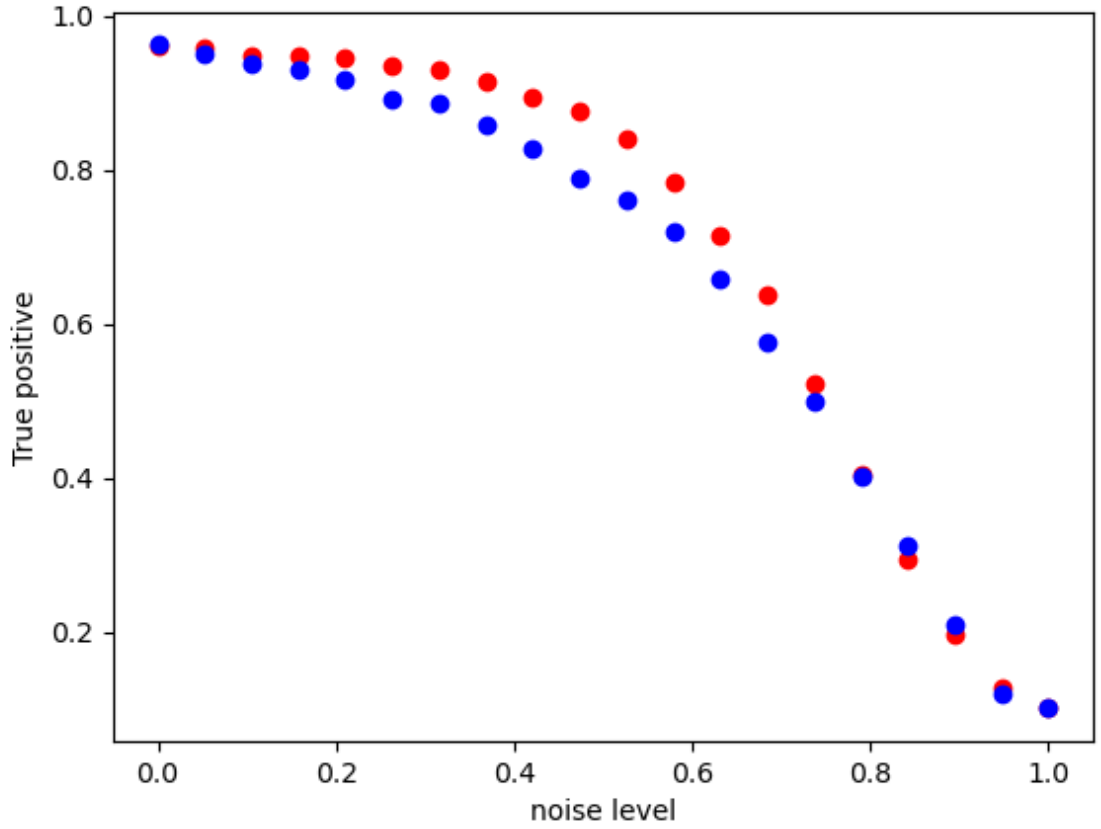


noise level = 1



For us the integers in the images get difficult to read at noise level 0.4. The denoising stops working at noise level = 0.6, where we can see that it denoise some images incorrectly.

C)



Red points represents **denoised**
Blue points represent **seasoned**

We plotted this for the model we did on question 2. This model:

```
## 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'))
```

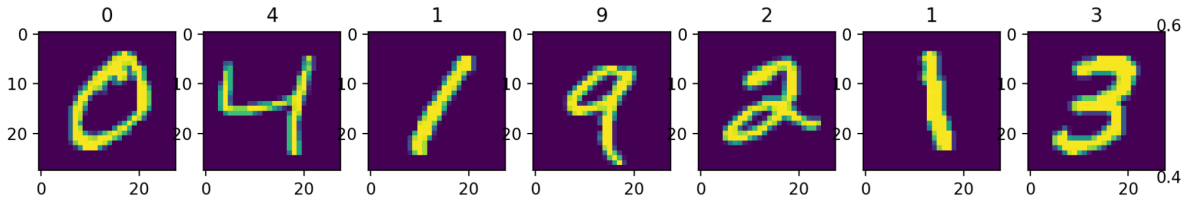
```
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.SGD(lr = 0.1),
        metrics=['accuracy'],)

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('Test loss: {}, Test accuracy {}'.format(score[0], score[1]))
```

The result obtained seems reasonable. In Q4 B we said that the integers represented in each image were more difficult to see when we increased noise level compared with the denoised images which were easier to read. Therefore we thought that the true positive rate was going to be generally higher for the denoised images. But when the noise level gets closer to one, even the denoised images are very hard to read and therefore the true positive rate for the seasoned and denoised images should be approximately the same for a high percentage of noise level. When the noise level is low or close to 0 it is approximately as simple to read both the seasoned and denoised images. But when the noise level is in the middle of 0 and 1 the denoised images are easier to read than the seasoned images and therefore the true positive rate should be higher for the denoised images. This was confirmed by our plot where we can see that for low noise level the true positive rate for seasoned and denoised images are pretty alike, and same thing for high noise level. We also see from the plot that the true positive rate for noise level in the middle of 0 and 1 is higher for the denoised images than the seasons images. In conclusion our plot seems very reasonable.

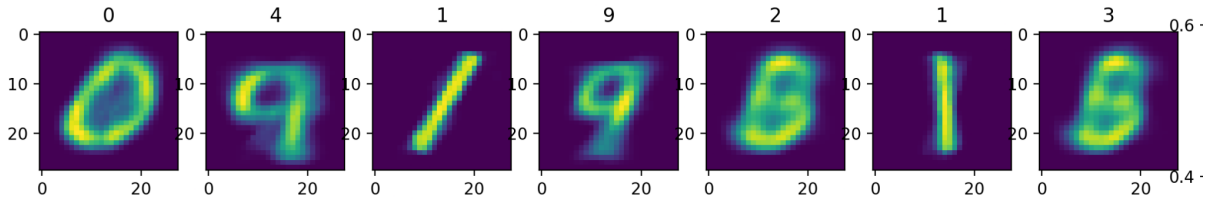
D)

To be able to create handwritten digits with the decoder we can use the vector/latent space that gets created from the input. Our encoder is essentially a bad cloning machine. When we input an image the encoder passes the image into the auto-encoder which creates a vector/latent space of the original image. Then our **decoder** tries to recreate the original input image as an output image. This might feel pointless but the important takeaway from this process is what the encoder has learnt in the process. Important to note is that the output from the encoder is not the same image as the output, it is a recreation of the original input. Since the recreated image is a *version* of the original image it could be regarded as a generated handwritten digit.

Original Image



Recreated Image



The recreated image was created by using:

```
decoded_values_pred = decoder.predict(encoded_pred)
```

Since the encoded_pred already held the data from earlier we can reconstruct this by predicting the values with our decoder. Our decoder is having some issues recreating some of these images. This might have been improved if I ran more epochs (only ran 10). The digits that are hard to tell what it's supposed to be are also digits that are form-wise close to another digit. Which is why our decoder has a hard time recreating it.