DIT406: Assignment 7

Calvin Smith, Emir Zivcic

Chalmers University of Technology/University of Gothenburg

December 21, 2021

1 Preprocessing

Depending on if you are using Tensorflow or Theano as backend, the data needs to be reshaped in a specific way. Both require that we add a dimension that represents the depth of images, which in our case is 1 since the data consists of grayscale images. The difference is that Tensorflow takes the depth as the last argument of the shape and Theano takes it as the second.

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

if K.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)
```

Second step is to convert our data to type **float32** and then normalize the values to be between $[0, 1]$.

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

x_train /= 255
x_test /= 255
```

Then we convert the class label vectors to arrays with 6000 rows and 10 columns. Each row represents the class label of an observation by having every column equal to zero except for the class that the observations belongs to which is set to 1.

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

2 Network model, training, and changing hyperparameters

A)

The network has 4 layers with 784, 64, 64 and 10 neurons respectively. The activation functions used are ReLU and Softmax. These are standard activation functions that are very commonly used today and good when applying on multiclass data. The softmax function is very suitable when the ouptut can be seen as "probabilities" of an image belonging to a certain class, since the function normalizes values from the previous layer to be between 0 and 1. The total number of parameters in the network is 55050. The input layer has the dimension (N, 784) because each image is 28X28 (= 784) pixels and the output layer is (N, 10) since we are outputting a predictions for each of the ten classes.

B)

The loss function used is called Categorical Cross-Entropy. The output of our Neural Network is a vector of "probabilites" from the softmax function. This vector represents the belief our network has that an image belongs to one of our ten classes. For example:

$$\hat{y} = [0.05, 0.05, 0.05, 0.05, 0.1, 0.1, 0.05, 0.05, 0.05, 0.45] \tag{1}$$

and an example of a target vector or "true" vector could be:

$$y = [0, 0, 0, 0, 0, 0, 0, 0, 0, 1] \tag{2}$$

The Cross-Entropy loss function is then defined as:

$$Loss = -\sum_{i=1}^{10} y_i \log(\hat{y}_i) \tag{3}$$

As we can see from the equation, the only class that contributes to the sum is the class that is non-zero in y, everything else will be zero. The purpose of the loss function is to measure the distance of the predictions from the truth. And the goal is to minimize it.

C)

Figure 1 is a plot of training and validation accuracy for each of the 10 epochs.

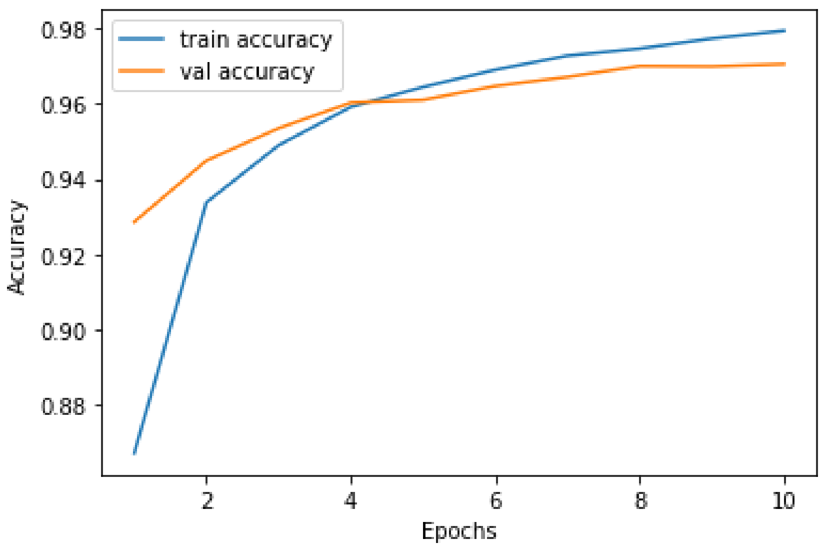


Figure 1

D)

Results do not vary greatly from that of Hinton's. We can see from the figure below that one regularization factor had a span from about 0.983 to about 0.986 indicating a sizeable standard deviation and results could be different from run to run.

Factors that may affect the result includes epochs and learning rate. Increasing epochs is unlikely to help since the accuracy converges quickly in the first few epochs.

The learning rate could be lowered to make the model more hesitant to change and thus potentially giving greater results.

No information is given from Hinton on what regularization factor was used to achieve his result. Perhaps greater results would be achieved from running even higher regularization factors. This makes sense since a fully-connected network tends to overfit to the training data.

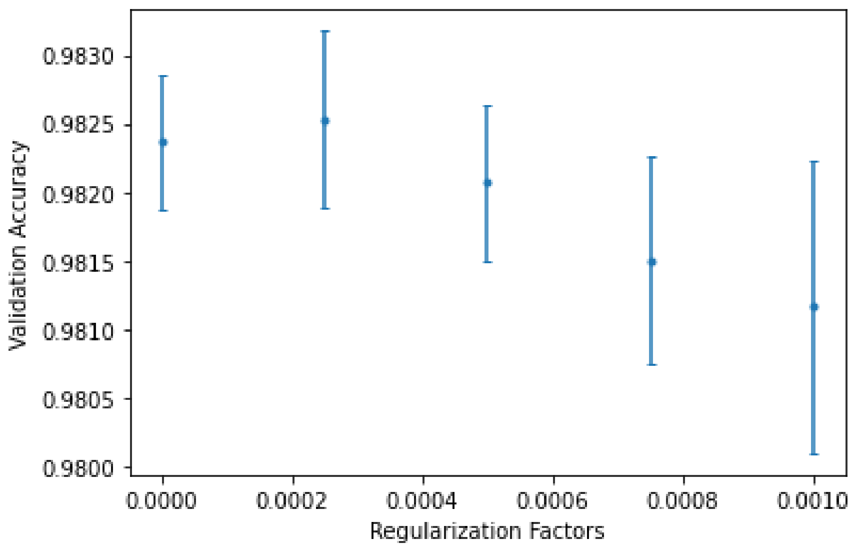


Figure 2: Error plot over regularization factors

3 Convolutional layers

A)

Three different iterations were trained with three models in each iteration with various differences between them. The layers were as following in the first iteration.

1. Convolutional
2. MaxPooling2D
3. Dropout
4. Convolutional
5. Flatten
6. Dense

The first iteration tested different values for the filters in the convolutional layers. The models had 32, 64 and 128 set respectively but this made no noticeable difference thus 32 was chosen for the next iteration to save time.

The second iteration added a dense layer with 32, 64 and 128 neurons in each model trained. This was placed after the flatten operation. This did have a significant increase in validation accuracy and could be due to the model being able to extract a higher amount of features before the output-layer. 32 was chosen for the third iteration since there were again no noticeable differences between the values used.

The third iteration used various values for the dropout. A good value for **dropout** is between 0.5 and 0.8 so the three different models were trained with 0.5, 0,65 and 0.8. This did not have a significant effect on the models so 0.5 was chosen arbitrarily for the final model.

The reason **MaxPooling2D** was utilized is too save computation-time by down-scaling the data.

Regularization methods like L2-regularization reduce overfitting by modifying the cost function, in our case we simply added a dropout-layer. That layer will modify the network itself by randomly dropping neurons from the network during training in each iteration.

The final model achieves a accuracy-score of 98.89 which we find is close enough to **99%**. This could however be improved by completing more iterations of the final mode. However, this was not done due to it being computationally heavy and consuming a lot of time.

B)

The name ”convolutional” comes from the fact that the local connection pattern can be thought of as a set of filters being ”convolved” over the image.

Convolutional layers are good for feature extraction in image analysis since they automatically detect patterns without human intervention. Which is perfect in the case of evaluating hand-drawn numbers. The convolutional layer would for example identify that a 6 always has a loop, in the first filter, and then additional features in the next filters that are characteristics of the number 6.

A fully connected layer connects each neuron of the previous layer to each neuron of the next layer. A convolutional layer differs in that is only connected with a few local neurons.

The potential benefit is that a convolutional layer tends to be cheaper since it has fewer connections and weights. This can be seen in how much faster the convolutional model is in question 3a than in the fully connected version in question 2d.

4 Auto-Encoders for denoising

A)

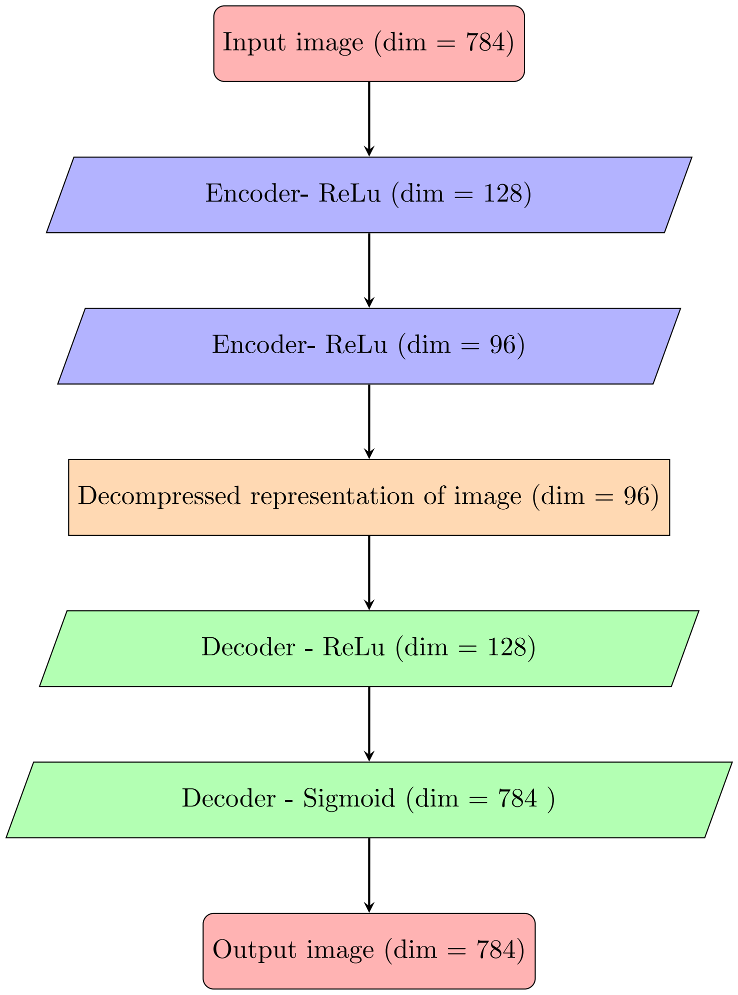
We begin by reshaping the data so that each row represents an image, that is we get a matrix with size (n,784). Then we apply the function **salt_and_pepper** to create new training and testing data with random noise.

```
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)
```

The variable **latent_dim = 96** represents the dimension to which our encoder will compress the input images to. The model uses two layers for encoding and two layers for decoding, with the number of nodes representing the dimension that the images are compressed and decompressed respectively. The purpose of the loss function is to minimize the reconstruction error of our noisy images using the normal images as reference. In other words, from the noisy images, we want to reconstruct the images that as closely as possible resembles the normal images.

```
latent_dim = 96

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

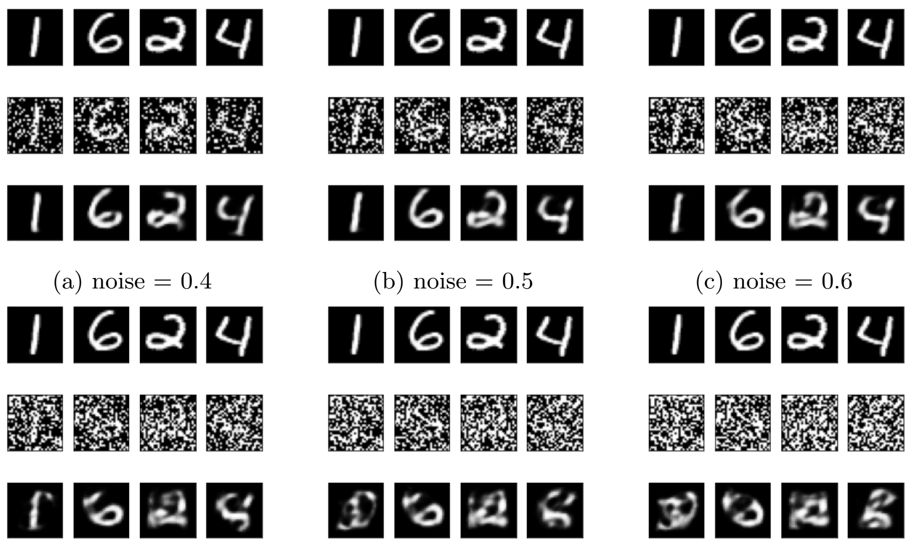
```
decoded = Dense(128, activation='relu')(encoded)
decoded = Dense(784, activation='sigmoid')(decoded)

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))

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))
```


Input image (dim = 784)
Encoder- ReLu (dim = 128)
Encoder- ReLu (dim = 96)
Decompressed representation of image (dim = 96)
Decoder - ReLu (dim = 128)
Decoder - Sigmoid (dim = 784)
Output image (dim = 784)

The diagram above represents what our model looks like. We feed the noisy image to the network, with two encoding layers we obtain an encoded representation of the image. The encoded image is then decompressed (reconstructed) through two decoding layers and the output is then compared to the original images using the loss function.

B)

Figure 3 shows the results of denoising with different levels of noise. The top row in the figures show the orignal images, the middle rows shows the figures after applying random noise and the bottom row is the result after denoising. Already at 0.5 it becomes pretty hard to identify the noisy images. However, the denoising performs quite well up until 0.7 when the reconstructed images start to become hard to indentify. After 0.7 it becomes virtually impossible to



(a) noise = 0.4 (b) noise = 0.5 (c) noise = 0.6

(d) noise = 0.7 (e) noise = 0.8 (f) noise = 0.9

Figure 3: Denoising results for different noise levels.

identify the denoised images.

C)

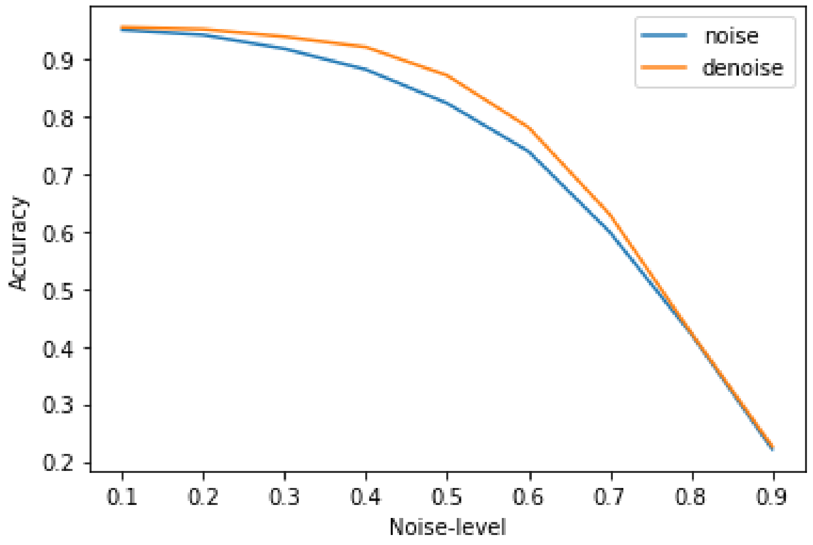


Figure 4: Classification vs noise-level.

Figure 4 shows how the accuracy of our classification changes with the level of noise added to the data. The blue line is the accuracy of a model trained and tested on noisy data. The orange line is the accuracy of a model trained and tested on the same noisy data, but after applying the autoencoder for denoising. The figure illustrates that denoising data before a classification task can improve the accuracy of your classifications.

D)

The process of generating synthetic ”hand-written” digits using the autoencoder is essentially to make the encoder part of the network output vectors of estimated means and standard deviations from a standard normal distribution, which we then sample from to create a latent (compressed) vector which is passed on to the decoder. So instead of decompressing each image to a point in our ”latent space”, as we previosuly did, we are now mapping each image to a normal distribution, which we can sample from. This can then be used to create new digits, by randomly sampling points from a normal distribution and the feeding it to our decoder.

Unfortunately, we didn’t manage to figure out how to do this in Python and the sampled images came out looking nothing like digits.