Assignment 7 - Neural Networks with Keras and TensorFlow

Nathan HAUDOT (14 hours), Hugo MATH (15 hours)

December 21, 2021

1 Preprocessing

The code for preprocessing the data showed in the notebook is as follow :

```
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)
```

There is a lot of benefits to preprocess the data by shifting and scaling it. For example, if we scaled and shift the data so that for a given dataset $\vec{X}$ we get its mean to 0 and its variance to 1, we obtained a more clear decision boundary for our classification.

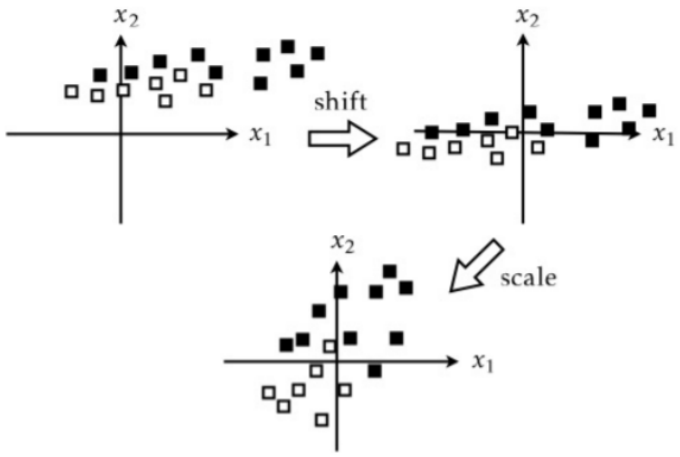

Figure 1: Shifting and scaling a two-dimensional dataset to achieve 0 mean and variance of 1

There would be also another problem if keeping the same scale at the beginning of our network. In fact, one of the challenge of NN is the vanishing gradient problem. If we keep large mean value in our dataset, we might have large gradient in our energy function, causing the gradient to explode and the convergence of our training being stuck. Similar effects with the variance.
The last lines simply convert string classes into numbers that the NN model can classify.

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

2.a Network description

In the notebook example there is 4 layers. One flatten layer as input layer, two fully connected hidden layer (Dense) and one fully connected output layer for the classification (Dense).
The flatten layer has an input size of (28, 28, 1) and an output size of (784, 1) since its utility is to flatten a matrix (0 neurons there) represented by our dataset (28, 28, 1) we get $-> 28*28 = 784$. The two hidden layer have 64 neurons each, with the rectified linear activation function (typical activation function here for an hidden layer). The final layer is working as the output layer, each neuron is for each class (0, 1, 2, 3 .. 10) so to get the probability of a class at the output, we then use the softmax function as the activation function. This function is very suitable for classification problems as describe above, we get the probability of each class.
To get the number of parameters in our model we simply do : $784*64+64+64*64+64+10*64+10 = 55050$ The Dense layer is a fully connected layer meaning that every neurons are connected to each neurons of the previous layer. Plus, we add the thresholds.

2.b What loss-function is used to train the network?

The loss function is as follow in keras :

```
keras.losses.categorical_crossentropy
```

This loss function is useful when dealing with multiple classes. The cross-entropy loss function is as follow :

$$Loss = -\sum_{i=1}^{ouputSize} y_i.log(\widehat{y_i})$$

where y_i is the target from the dataset, outputSize is the number of classes and $\widehat{y_i}$ is the computed output with our updated weights which is the prediction for each x_j in the dataset x. The minus sign ensures that the loss function get smaller each step, indeed :
$log(x) <= 0 \forall x \in [0, 1]$
The log function is here to serve non-linear purpose, thus, to give more error on low probability of prediction.

2.c Train the network for 10 epochs and plot the training and validation accuracy for each epoch.

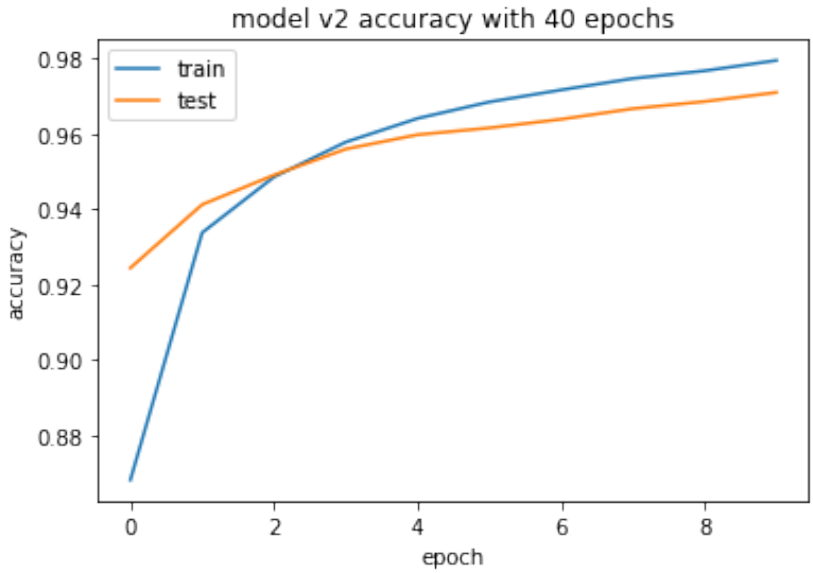

Figure 2: Evolution of the accuracy with 10 epochs

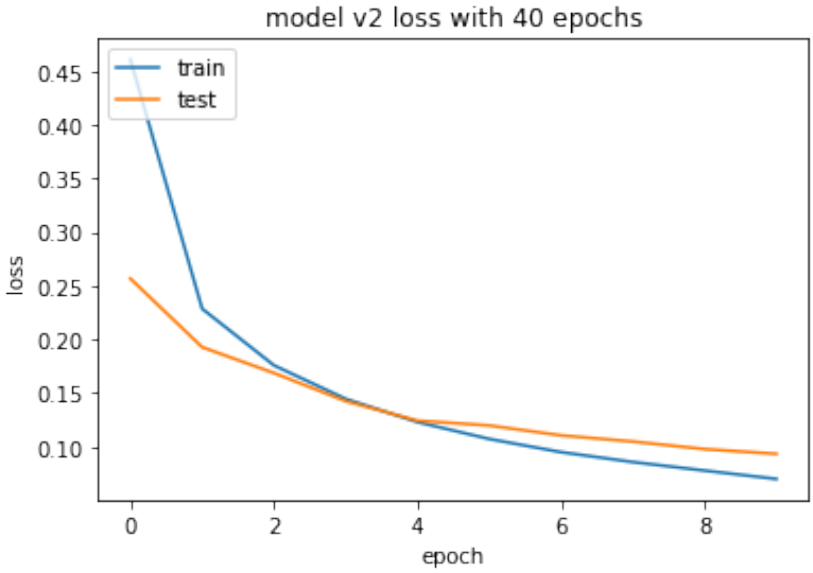

Figure 3: Evolution of the loss function with 10 epochs

We obtained a test loss of 0.0933 and a test accuracy of 0.971. With just two hidden layer with 64 neurons each, we achieve 97% of test accuracy.

2.d Update model to implement a new three-layer model

With the new model we get a test loss: 0.067 and a test accuracy of 0.982

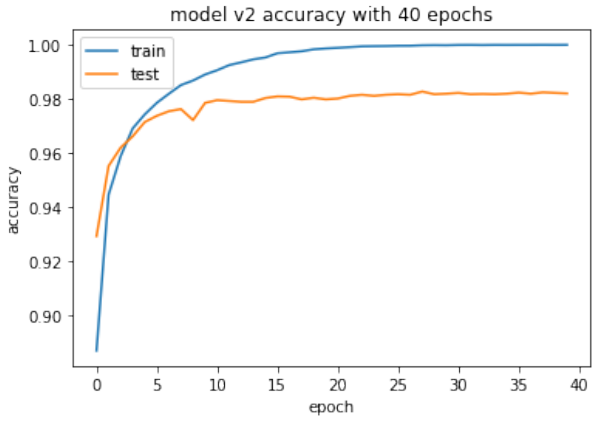

Figure 4: Evolution of the loss function for the second model with 40 epochs

For the second part we trained three Hintons model with decays of 0.000001, 0.00001 and 0.001. We have got theses results :

Decays	Final Test Accuracy	Mean Test Accuracy	Std Test Accuracy
0.000001	0.9815	0.9774	0.0092
0.00001	0.9816	0.9761	0.0093
0.0001	0.9815	0.9762	0.0096

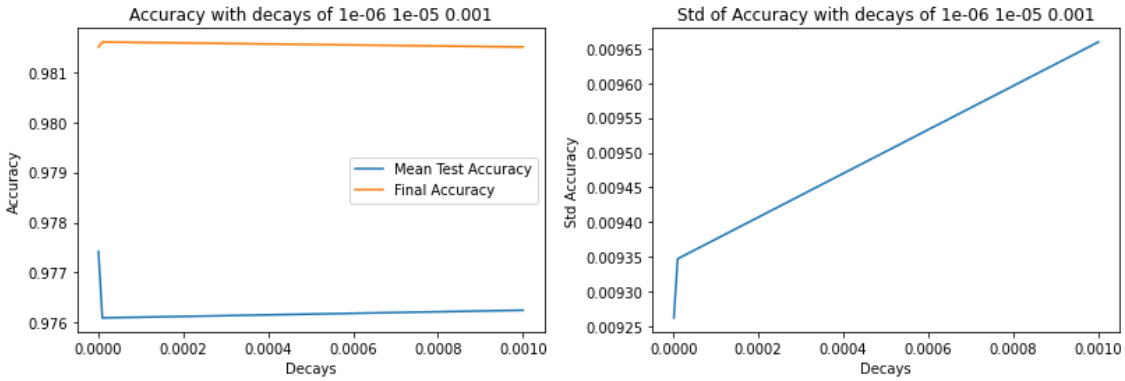

Figure 5: Evolution of the accuracy and its standard for 3 regularizer parameter

As we can notice, we got very close results to Hinton but still not achieving 0.9847 (we have 0.9816 final validation accuracy). One reason would be that we are using SGD : stochastic gradient descent and we might not have the same result each time due to the stochastic component of this algorithm. However, the difference of result is too noticeable for that, so we can think that we are picking the wrong decays and haven't tried so much regularizer factors. There might be also our hyper-parameters and especially the learning rate which influences a lot the convergence of our model. We have put an LR of 0.1 but we could have done an adaptive learning rate to more efficiently converge and don't get stuck in a local minima.

3 Convolutional layers

3.a Design a model that makes use of at least one convolutional layer

We created a simple ConvNet of three hidden convolutional layer respectively with 32 64 and 100 filters with Maxpooling layer between each, and one Flatten layer and finally two fully-connected layer with 100 and 10 neurons at the end for the classification (softmax activation). We could achieve with the Adam optimizer (not stochastic gradient descent this time) a Test loss of 0.0348 and a Test accuracy of 0.992.
For the convolutional layer we have put a padding of 0 meaning that there is no "starting gap" between the filter and the image matrix. We let the strides to 1 meaning a step of 1 on every filter (we don't loose information but the training speed can be a bit slow). And for the feature we gently increased it until we achieve good accuracy.
Our method was to put multiple convolutional layer, not too much because the dataset was not high-dimensional and some MaxPooling layer to speed-up the training and get the most interesting part of the features. (MaxPooling takes the max of a kernel in the different resulting filters of the previous convolutional layer).

Starting with features size of 16 32 64 and then we find that 32 64 100 was enough. After testing different features size and neurons size, we came to this model :

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 26, 26, 32)        320

 max_pooling2d_6 (MaxPooling  (None, 25, 25, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 23, 23, 64)        18496

 max_pooling2d_7 (MaxPooling  (None, 22, 22, 64)       0
 2D)

 conv2d_8 (Conv2D)           (None, 20, 20, 100)       57700

 max_pooling2d_8 (MaxPooling  (None, 19, 19, 100)      0
 2D)

 flatten_2 (Flatten)         (None, 36100)             0

 dense_3 (Dense)             (None, 200)               7220200

 dense_4 (Dense)             (None, 10)                2010

=================================================================
Total params: 7,298,726
Trainable params: 7,298,726
Non-trainable params: 0
_________________________________________________________________
```

Figure 6: Model Summary

3.b Discuss the differences and potential benefits of using convolutional layers over fully connected ones for the particular application?

Convolutional layers works well on images classification and more generally in image processing application. Since it's using convolutional filters on matrix represented by an image, it can detect abstracts features like objects or small geometric forms. One issue with convolutional layer is when dealing with high dimensional data (full-hd image for example) it can be tricky and slow to trained model like that. So we put MapPooling layer to speed-up training without loosing too much information.

One other issue is that convolutional neural networks tend to detect very small and abstract features, this has trouble detecting an whole geometric form on images. While a children just have to see one time a cat to classify it at a cat next time, for a convolutional network it can takes thousand of good quality images to achieve same accuracy. Another related problem is when dealing with adversarial attack, in fact by just putting a noise in the image that human cannot discern, this can dramatically changed the accuracy of the model.

4 Auto-encoders for denoising

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.

This auto encoder model takes our x_train images as input. First, we transform the images into a 1x784 vector to obtain a "flattened" dataset. Then, we apply to our dataset the salt_and_pepper() function defined in the notebook: this function adds random noise to our images, and we do the same thing with our x_test dataset (which is "x_test_seasoneed" in the notebook). The first noise level applied is 0.4.

The deep network autoencoder is composed of five layers. We can segment the model into three parts:

4.a.1 The encoder

It is composed of two fully connected layers: the input layer of 784 (28*28=784) neurons, as well as the first hidden layer of the 128 neurons network (ReLu activation function).

4.a.2 The latent space representation

Also called bottleneck layer, the layer in the middle of the network, the one with the most compressed representation of the model. It is composed of a fully connected layer on both sides, and 96 neurons (ReLu activation function).

4.a.3 The decoder

It is composed of a fully connected layer of 128 neurons (ReLu activation function), as well as another fully connected layer of 784 neurons but this time with a sigmoid activation function: it is relevant because we need an output between 0 and 1, which will correspond to a "pixel intensity" of a grey scale image, so the neural network will not need to learn that the output must be between 0 and 1.

The binary cross-entropy loss (a specific case of cross-entropy where the target is binary) computes the cross-entropy loss between the train labels and the labels predicted with the model. It is used during a binary classification, and thus corresponds to our use case.

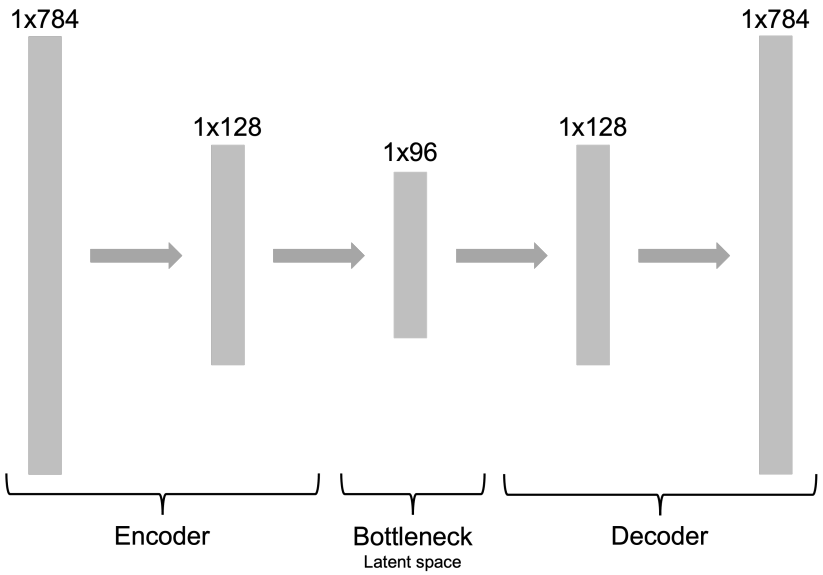

Figure 7: Autoencoder diagram

4.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 from 4A to denoise the noisy digits). At what noise level does it become difficult to identify the seasoned digits for you? At what noise level does the denoising stop working?

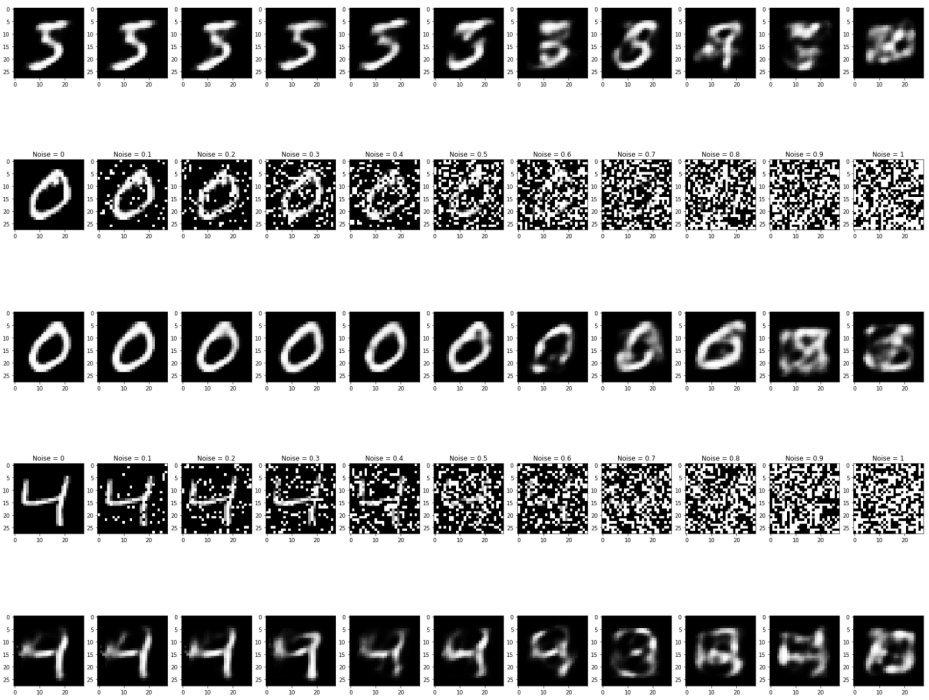

Figure 8: Noisy images vs. autoencoder result

We use the salt_and_pepper() function at different noise levels (from 0 to 1, increment of 0.1) on our dataset: we took the first three images as example. We then use our autoencoder trained in the previous question to denoise our noisy numbers. In our noisy images, it is difficult to recognize the numbers above a noise of 0.5. After processing through the autoencoder, the denoising stops working around 0.7.

4.c Test whether denoising improves the classification with the best performing model you obtained in questions 2 or 3. Plot the accuracy as a function of noise-level for the seasoned and denoised datasets. Discuss your results.

Using the seasoned & denoised data with our convolutional neural network, it generally does better with the denoised inputs than with the seasoned inputs :

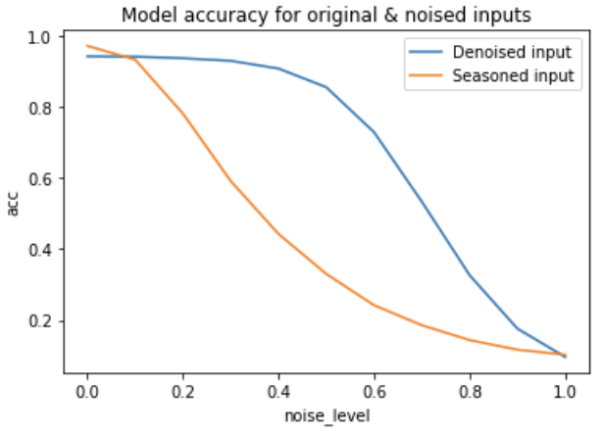

Figure 9: Seasoned vs. denoised datasets accuracy

With a noise level of less than 0.1, we can see that the convolutional neural network does better with noisy inputs. We can understand that the addition of light noise forces the network to train better on its predictions. When the noise level is greater than 0.1, the neural network does much better with the denoised inputs, especially from 0.4-0.5. This confirms our observations in the previous question about the ability of the autoencoder to de-noise the image at a maximum of a noise level of 0.4-0.5.

4.d Explain how you can use the decoder part of the denoising autoencoder to generate synthetic "hand-written" digits? – Describe the procedure, implement it and show examples in your report.

```
def generate_digit(model, digit, decoded_img, threshold=0.99):
    np.random.shuffle(decoded_img)
    for img in decoded_img:

        imgreshape = img.reshape(28, 28, 1)

        predictions = model.predict(imgreshape)
        maxIndex = np.argmax(predictions)
        maxAcc = predictions[0, maxIndex]

        if (maxIndex == digit) and (maxAcc >= threshold):
            return img

plt.imshow(generate_digit(model, 4, decoded),  cmap='gray')
plt.show()
```

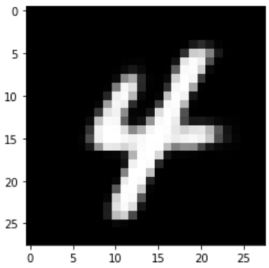

Figure 10: Generated number with the decoder part of an autoencoder

The decoder part of the autoencoder can be used to generate new images.

In the last cell of the notebook, we train our encoder as well as our decoder with our original dataset. Then, we created the function "generate_digit" taking as parameters our convolutional neural network "model" previously trained on the MNIST, also the "digit" that we want to generate, and the decoder part of our autoencoder "decoded_img".

For each decoded image, we try to classify it to get the right digit generated. If the accuracy of the predicted label is greater than the threshold, and the requested digit corresponds to the classifying result of the neural network, then we return the image. Here, our classifier is used to "catch" our generated image.