Neural Networks: Implementing a Neural Network with Keras Deep Learning in Computer Vision Lab 3

Awet Haileslassie Gebrehiwot, Santiago Herrero Melo

I. INTRODUCTION

In this third lab session of *Deep Learning in Computer Vision* we will see how to build a neural network with the Keras framework in Python. We will use the MNIST dataset, one of the most used datasets in machine learning research. This dataset consists of 70000 grayscale images of handwritten digits (of size 28x28). The project will be developed in python language, using Keras in the tensorflow environment.
A neural network consists of the following components

- *An input layer, x*
- *An arbitrary amount of hidden layers*
- *An output layer, y*
- *A choice of activation function for each hidden layer, and optimaizer.*

A. Single Neuron:

As a first exercise, we will consider a single neuron with 28x28=784 inputs and a single sigmoid unit generating the output. Our neuron will simply learn to distinguish between the digit 0 and the other digits (in the MNIST dataset).

B. A Neural Network with one Hidden Layer:

In this exercise, we will add an intermediate hidden layer which consists of 64 units. This means that the 784 inputs will be connected to each of the 64 units of the hidden layer, which will also be connected to the output layer. The purpose of this hidden layer is to create system that allows us a faster and better learning for the neural network. It also provide the discrimination necessary to be able to separate the training data.

C. Multiclass Neural Network:

In this section, we will recognize each of the ten different handwritten digits (from 0 to 9) individually. This means that now, the output will have ten units, being each of them connected to each of the units of the hidden layer.

II. IMPLEMENTATION

Taking the artificial neural network approach, the computer is fed training examples of known handwritten digits(MNIST dataset), that have been previously labeled as to which number they correspond to, and via the algorithm the computer then learns to recognize each character. The Neural network via single Neuron (Perceptron) algorithm allows the computer to learn by incorporating new data. In this occasion, differently from the previous lab, we just have to load certain specific functions from the Keras framework that will perform all the work.

A. Single Neuron:

In the first place, we have to import the *Sequential* Keras model in order to create our model. This model is a sequential stack of layers which performs the learning making minimal assumptions on the sequence structure.
We add a dense layer in which we define the *number of outputs* (in this case 1), the *activation function* (which will be the sigmoid function) and the *size of the input shape*. In the case of the single neuron, this value will be equal to the number of neurons in the input (784 in this case).
Once the neural network is created, we have to configure the learning process via the *compile* function. We have to set parameters such as *loss function*, *optimizer* and *metrics*. In the case of our single neural network, we will use the binary cross entropy for the loss function, sgd as optimizer and measure the accuracy.
The following step is to train the model, for which we will use the *fit* function. In this case, we will have to set which are the training data and labels, the number of epochs (100 in this case), the batch size (that we will modify) and the validation data that we will use, in our case, the 20% of the training data.
We will finally evaluate the accuracy of our model with the *evaluate* function, using the test data and labels.

B. A Neural Network with one Hidden Layer:

The only difference between this exercise and the one from the previous section is the addition of the hidden layer. For doing so, we have to call the *add* twice for adding the hidden layer. In the first call, we will define as input the number of neurons in the input layer (784), as an output the number of neurons in the hidden layer (64 originally, but we will study the behaviour with different number of neurons). Then, in the second call, the input will be the number of neurons in the hidden layer, and the output number of the neural network (1).

We have now set the batch size to 100, used the adam function as optimizer and studied the use of different activation functions.

C. Multiclass Neural Network:

In this exercise, the output layer changes from the previous one, modifying its number of output nodes from one to ten,

as shown in Fig. (1).

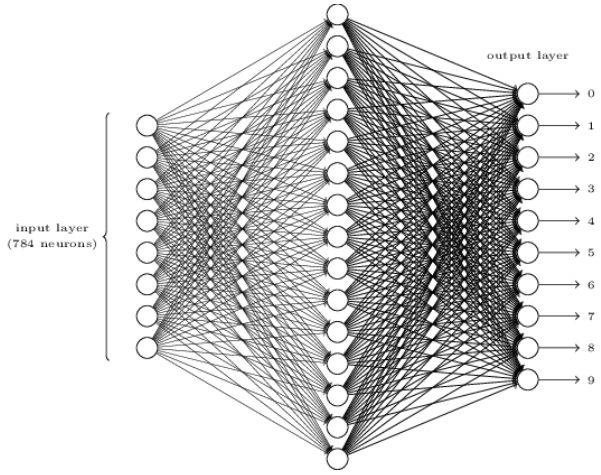


Fig. 1: Multi-class Neural Network with 10 outputs

In terms of code, it is the same as the previous exercise, only making sure that the value of the number of classes (neurons in the output layer) has been modified. In this case we will use the *ReLu* function as activation function for the input layer, and the *softmax* function as activation function for the hidden layer, and the employed loss function will be the categorical cross entropy.
We will explore the behaviour of the neural network testing different optimizers, setting the number of neurons of the hidden layer in 64, and the batch size as 100.

D. Running the code

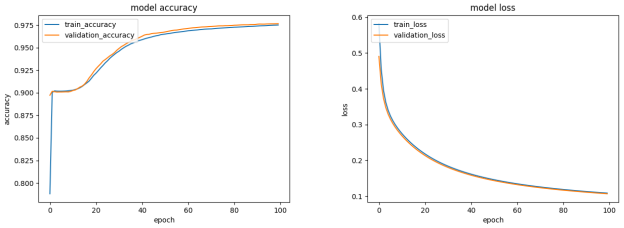
For running the code we first have to activate the Tensorflow environment. Once that is done, we access the folder where the skeleton of the code is saved, and type the following line in the command window:
python lab3_1_skeleton.py for the single neuron script,
python lab3_2_skeleton.py for the single class script with hidden layer,
python lab3_3_skeleton.py for the multiple class with hidden layer script,
In some cases, we have encountered a small issue when changing the computer where we work. The returned parameters of the *fit* function are sometimes called *acc, acc_value* and sometimes *accuracy, accuracy_value*. As this function does not allow indexing, this may need to be modified manually by the user if desiring to obtain the graphs.

III. RESULTS AND ANALYSIS

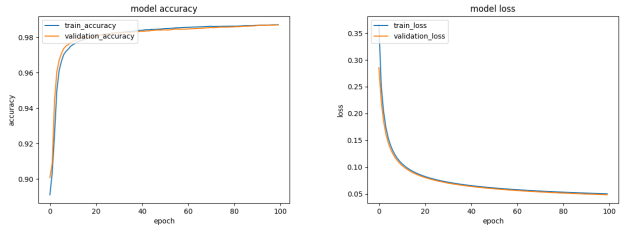
A. Single Neuron:

The accuracy of the network has been tested for different batch sizes, which are shown in Fig. (2).

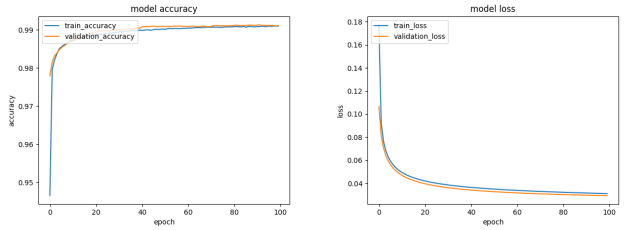
As we reduce the size of the batch, the accuracy increases and the loss function decreases. For the test set, we obtain



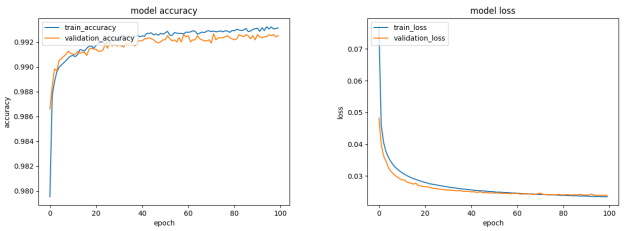
(a) Accuracy for a batch of size 10,000 (b) Loss density function for a batch size of 10,000



(c) Accuracy for a batch of size 1,000 (d) Loss density function for a batch size of 1000



(e) Accuracy for a batch of size 100 (f) Loss density function for a batch size of 100



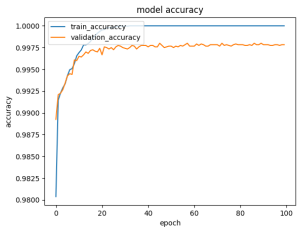
(g) Accuracy for a batch of size 10 (h) Loss density function for a batch size of 10

Fig. 2: Accuracy performance and loss functions for single neural network

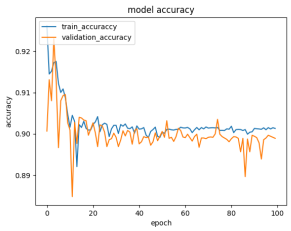
an accuracy of *0.9767* when using a 10000 batch, and an accuracy of *0.9922* when using a batch of size 10. In the same way, the loss accuracy is smaller in the second case, passing from *0.098576* to *0.02365*. Also, it is important to point out that the smaller the batch size, the more time the computation takes to be performed.
As the difference of accuracy is negligent when using a batch size of 100 instead of 10 (accuracy of *0.9921* compared to *0.9922*), but the computational time is much smaller with batch size 100, we decided to use a batch size of 100 from now on.

B. A Neural Network with one Hidden Layer:

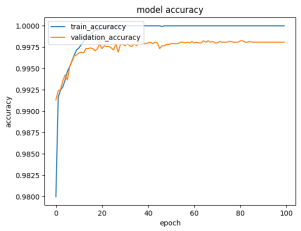
The accuracy of the network has been tested for different combinations of number of neurons in the hidden layer and activation functions, which are shown in Fig. (3).



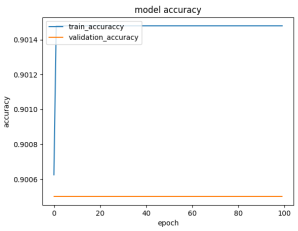
(a) Accuracy for 64 neurons in the hidden layer with sigmoid activation function



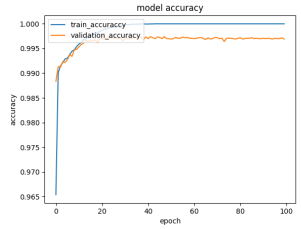
(b) Accuracy for 64 neurons in the hidden layer with ReLu activation function



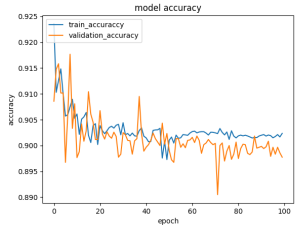
(c) Accuracy for 128 neurons in the hidden layer with sigmoid activation function



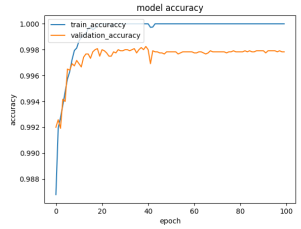
(d) Accuracy for 128 neurons in the hidden layer with ReLu activation function



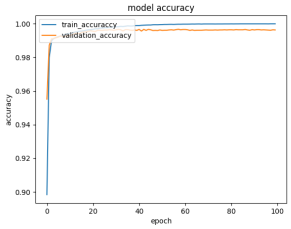
(e) Accuracy for 32 neurons in the hidden layer with sigmoid activation function



(f) Accuracy for 32 neurons in the hidden layer with ReLu activation function



(g) Accuracy for 512 neurons in the hidden layer with sigmoid activation function

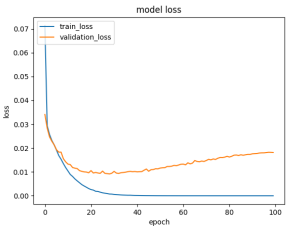


(h) Accuracy for 8 neurons in the hidden layer with sigmoid activation function

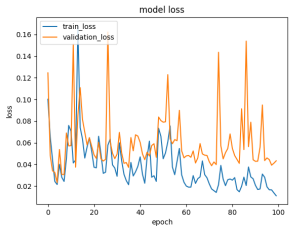
Fig. 3: Accuracy performance for a single hidden layer

The network has clearly a problem with the *ReLu* function, as the answers both for the accuracy and the loss function are highly oscillating.

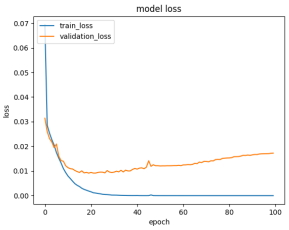
As it can be seen in the images of Fig. (3), the accuracy increments when using a hidden layer with a larger number of neurons. Although in all cases the accuracy is very good,



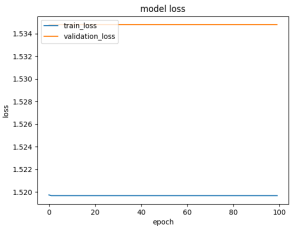
(a) Loss density function for 64 neurons in the hidden layer with sigmoid activation function



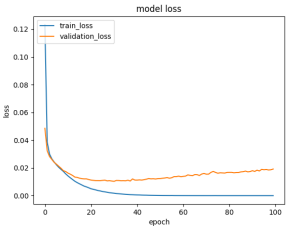
(b) Loss density function for 64 neurons in the hidden layer with ReLu activation function



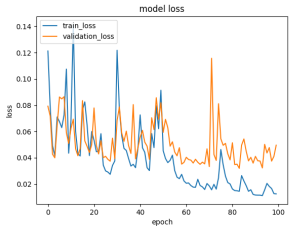
(c) Loss density function for 128 neurons in the hidden layer with sigmoid activation function



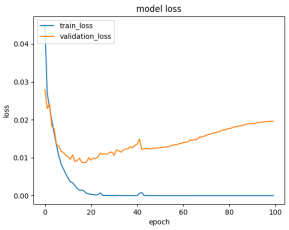
(d) Loss density function for 128 neurons in the hidden layer with ReLu activation function



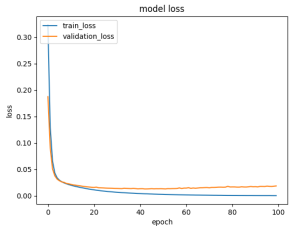
(e) Loss density function for 32 neurons in the hidden layer with sigmoid activation function



(f) Loss density function for 32 neurons in the hidden layer with ReLu activation function



(g) Loss density function for 512 neurons in the hidden layer with sigmoid activation function



(h) Loss density function for 8 neurons in the hidden layer with sigmoid activation function

Fig. 4: Loss functions for a single hidden layer

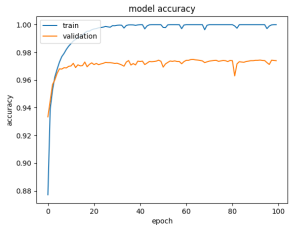
we can see a small improvement from the hidden layer with less neurons shown in image (3h) which reaches an accuracy of *0.9964* for the test set, to the hidden layer with more neurons, in image (3g), which reaches an accuracy of *0.998* for the test set. For the original hidden layer, with 64 neurons, shown in image (3a), the accuracy achieved for the test set is *0.9976*.

All the precedent values where calculated with the *sigmoid* activation function. For the *ReLu* activation function, the results are worse, obtaining an accuracy for the test set of around *0.90*. This may be due to the ReLu function being

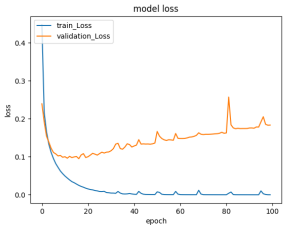
considered more for multiclass problems. Therefore, we can conclude that for one single hidden layer, the best option is to use a *sigmoid* activation function, with as many neurons in the hidden layer as possible.

C. Multi-class Neural Network:

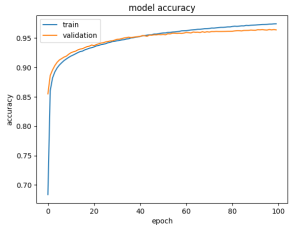
The accuracy of the network has been tested for different optimizers, which are shown in Fig. (5).



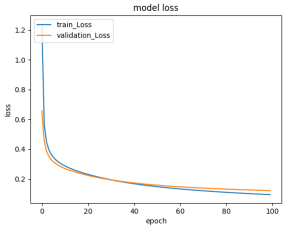
(a) Accuracy performance function using Adam optimizer



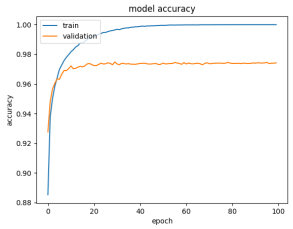
(b) Loss density function using Adam optimizer



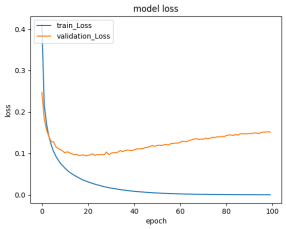
(c) Accuracy performance function using SGD optimizer



(d) Loss density function using SGD optimizer



(e) Accuracy performance function using adaDelta optimizer

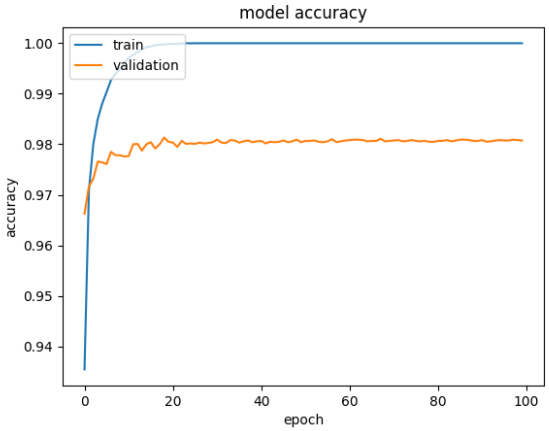


(f) Loss density function using adaDelta optimizer

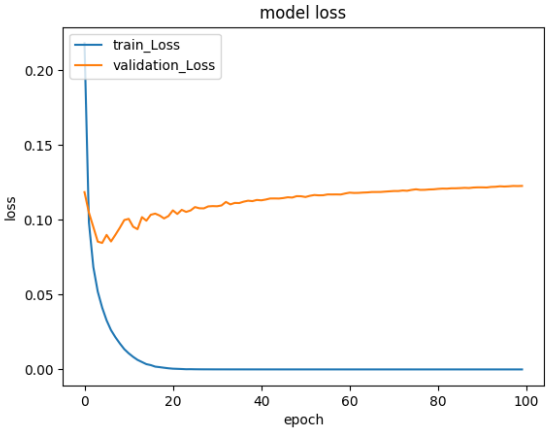
Fig. 5: Accuracy performance and loss functions for multi-class neural network

We tried as different optimizers *Adam*, *SGD* and *adaDelta*, obtaining the highest accuracy for the network with this last optimizer (adaDelta). In this case, the accuracy reached a value of *0.9767*, slightly higher than with *Adam* optimizer (*0.976* of accuracy).
Taking into account this exercise and the previous ones, we have selected our best network to optimize accuracy as a network with a small batch size (10), a high number of neurons in the hidden layer (512), the *ReLu* and *softmax* activation functions and the *adaDelta* optimizer.

For this last architecture, in the test set we achieve a value of *0.9823* in accuracy and *0.0998* in the loss function. On the other hand, we reach loss values on the scale of 10^{-7}



(a) Accuracy performance



(b) Loss density

Fig. 6: Best neural architecture for multiclass neural network

and accuracy of 1, which means perfect matching during the training.