Vadim Stasiev - 1900872

Design of Machine Learning Solution for Biometric Recognition Task

Assignment 1

Contents

[1. Introduction 2](#_Toc66062606)

[2. Designing a solution 2](#_Toc66062607)

[Parsing the data 2](#_Toc66062608)

[Splitting the data 3](#_Toc66062609)

[Creating the model 3](#_Toc66062610)

[Training the model 4](#_Toc66062611)

[Overfitting Data – Why should it be avoided? 6](#_Toc66062612)

[Evaluating the model 6](#_Toc66062613)

[3. Experiments 9](#_Toc66062614)

[4. Conclusion 10](#_Toc66062615)

[5. References 11](#_Toc66062616)

[6. Appendix 13](#_Toc66062617)

# 1. Introduction

This report covers the approach taken in the completion of a biometric recognition task which was assigned and outlined in the assignment one brief (Schetinin, 2021). The brief provides a dataset of 1500 photos that in all contain 30 different people where each person has multiple photos that were purposefully taken under different lighting conditions.

However, the dataset already came normalized as all of the photos were black and white, containing only the area of interest, which was in fact the same across all photos at a width of 68 and height of 77 pixels. These facts made the difficulty of this task much easier as it meant that a simple machine learning solution would likely suffice to achieve a solution that would accurately recognize one of the 30 individuals on new similarly normalized photos of them.

In other words, the developed solution being covered in this report assumes that any future data passed into it will come in a similar form to that of the dataset given, this assumes they should come in black and white, correct rotation, same resolution and ratio, same centred position, and that any new photo belongs to one of the 30 individuals from the original dataset.

Had the data not already been normalized so well, it would have likely been necessary to use a convolutional neural network implementation which could have allowed for face recognition under different rotations, positions and resolutions, however given the limited dataset the outcome of such might have fallen short of its original goal of achieving the highest accuracy possible and naturally, all the processing overhead that comes from a more complex machine learning solution. (Saha, 2018)

# 2. Designing a solution

As previously mentioned, the assignment brief (Schetinin, 2021) provided a dataset which contained 1500 photos that belonged to a total of 30 different people, the photos came with the following naming style: “yaleB02\_P00A+015E+20.jpg”. In the name, the number following “*yaleB*” represents the number of the person the photo belongs to, however, these numbers are not sequential, meaning that the 30 people do not start at 0 and end at 29, which would have been ideal for using this data for labels, instead it ranges randomly from 2 to 39, this had to be accounted for when parsing the data.

## Parsing the data

When reading and parsing the dataset, first, it was necessary to iterate through all the files and append them as grayscale arrays to the “images”, which is later converted into a Numpy Array which is what is compatible with the machine learning libraries being used here (NumPy, 2021). The “labels” are then read from the filenames and lastly a simple algorithm is used to convert the labels into a sequential number of individuals, each number representing each individual, from 0 to 29.

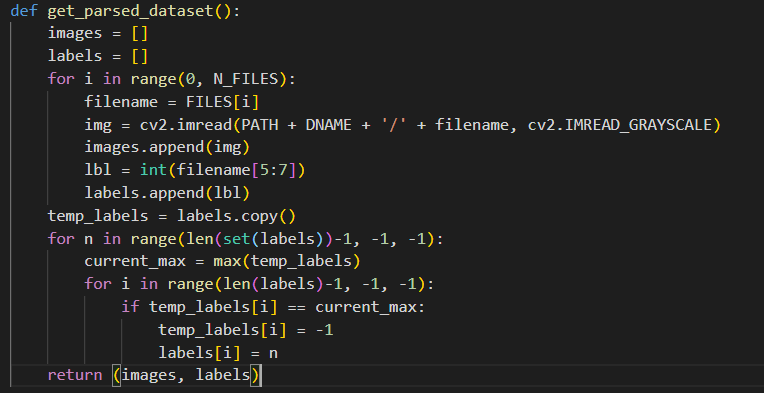


Figure 1 – Parsing the dataset

The photos obtained from the dataset are then divided by 255 in order to restrict the range of the values between 0 and 1 and therefore converting all the integer values into floating point values as this is computationally more convenient for GPU processing as GPUs are optimized for floating point math. (Nvidia, 2021).

## Splitting the data

In order to ensure that the model can classify unseen data the dataset given must be split into training data, for which the majority of the dataset will be allocated, while the remaining is set aside to be used as testing data. To do that, there is a simple function in the machine learning module “*sklearn*” named “*train\_test\_split*” (scikit-learn, 2021).

In this case it was used a test-size of 30%.



Figure 2 – Splitting the dataset into 2 sections, one for training and a smaller one for testing.

## Creating the model

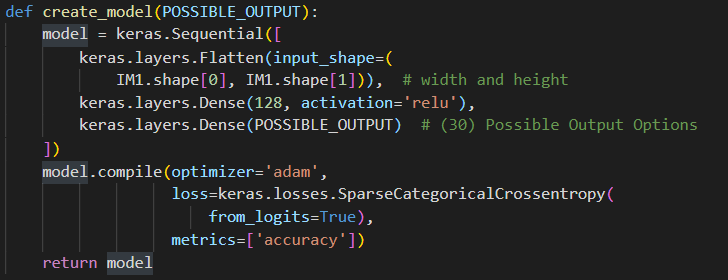


Figure 3 - Creating the model

The model is shown in the function of the picture above, which first creates a “*Flatten*” layer that takes the input image and places it all sequentially in a single dimension array, effectively getting an array of *width \* height* number of elements (Tensorflow, 2021), this process is demonstrated in the picture bellow.

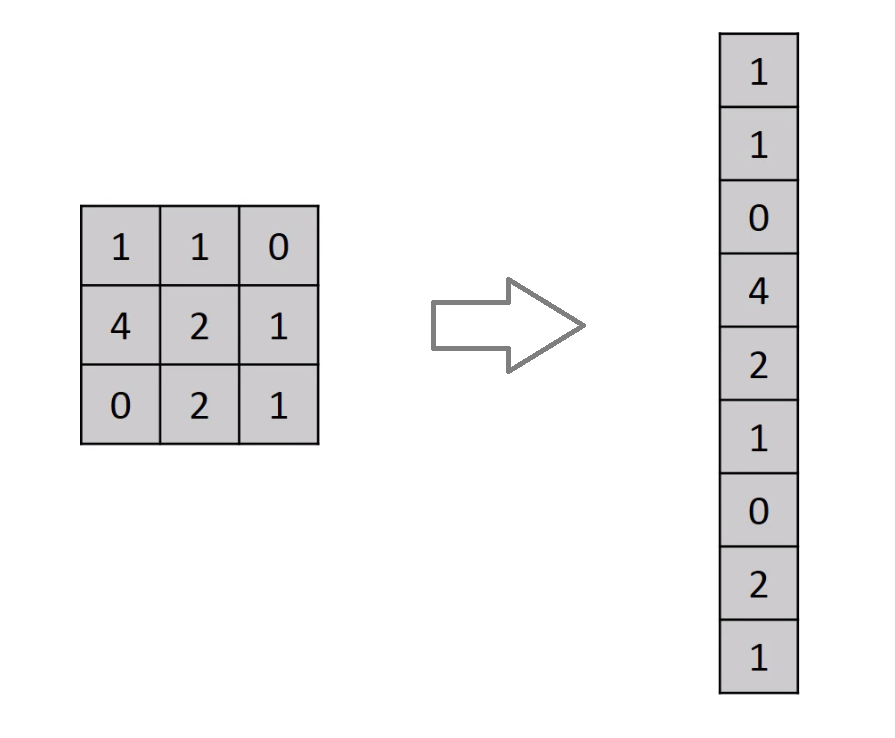


Figure 4 - Flatten Layer Example (Dabakoglu, 2018)

A dense layer, as the name suggests, is when a layer is fully connected to the neurons in the next layer (Rampurawala, 2019). In this case this is used with an activation function called “*RELU”*, which is nowadays commonly used for its performance in detriment of Sigmoid (Versloot, 2019), to simplify, its purpose is to essentially determine whether or not a neuron is active.

That said, the second layer used was a dense layer with an initial arbitrary number of 128 neurons, which seemed to work perfectly fine without any need for immediate tweaking as it was evaluating the test-data with accuracies ranging between 85 and 95.

The last layer used was also a dense layer with a non-arbitrary number of 30 neurons as these represent the possible outputs as there were a total of 30 different individuals.

The model is then compiled using “*Adam*” as the go-to general-purpose TensorFlow-recommended optimizer (TensorFlow, 2017) and set to use “*accuracy*” for its metrics as this will calculate how often the predictions equal the given labels, in this case, the individuals. (Keras, 2021)

## Training the model

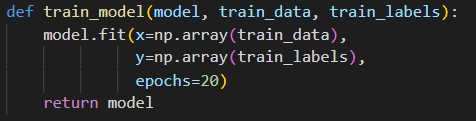


Figure 5 - Train model function



Figure 6 - Passing into the “train\_model(),” the training data and labels as well as retrieving the model for further usage

After getting a compiled model, the next step was to train it by fitting the model with the “*train\_data*” and the matching “*train\_labels*”, which, just like the original labels array, is a list with all of the matching individual’s numbers from the “*train\_data*”, so they are exactly in the same order as the photos, this will let the model know which photo belongs to which person. Both “*train\_data*” and “*train*\_*labels*” had to be converted to a numpy array, it is not entirely clear why this is, as the original data was a numpy array already but it could have possibly been due to the “*train\_test\_split*”, though this was not verified.

### Overfitting Data – Why should it be avoided?

In short, overfitting happens to a model when it has been trained too many time using a given dataset and as such, the model starts losing its ability to generalize the task that it was targeting to accomplish, which means that, past a certain point of training over the same data, while the model may seem to improve in accuracy, it may in reality start to “*overfit*” the training data and perform worse on new and unseen data. This is important to take into account when developing a machine learning solution as it is most efficient to strike a balance between making the most usage of the available data without overfitting the model to the point where it cannot serve its intended purpose over new data.

The “epochs” is the setting that defines how many times the model trains over the same data, and for the model at hand, it was initially set to be trained for 20 epochs, which again, is an arbitrary number which was picked in accordance to the TensorFlow examples and recommendation of not setting it too high (TensorFlow, 2017), this particular value seemed to work well, however further optimization was to take place while optimizing the model, which in this report can be located under the “Experiments” section.

## Evaluating the model

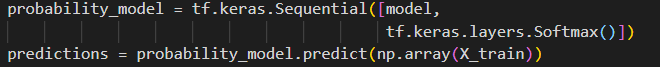


Figure - Creating a probability model from the existing model

To evaluate the model, another model was created from it, but with an added “*Softmax*” layer, this is used for determining the prediction probability of each possible output, this is demonstrated in the upcoming graphs bellow.

The graphs bellow were generated using the python “matplotlib” library, using the following script:

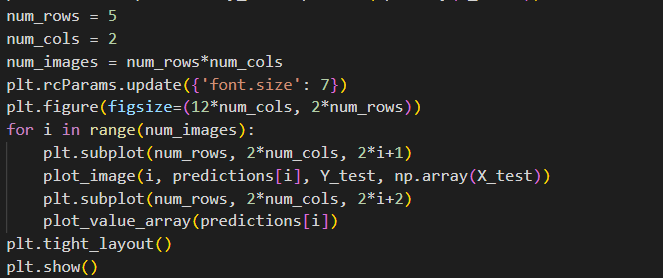


Figure - Script that generates the "matplotlib" graphs bellow

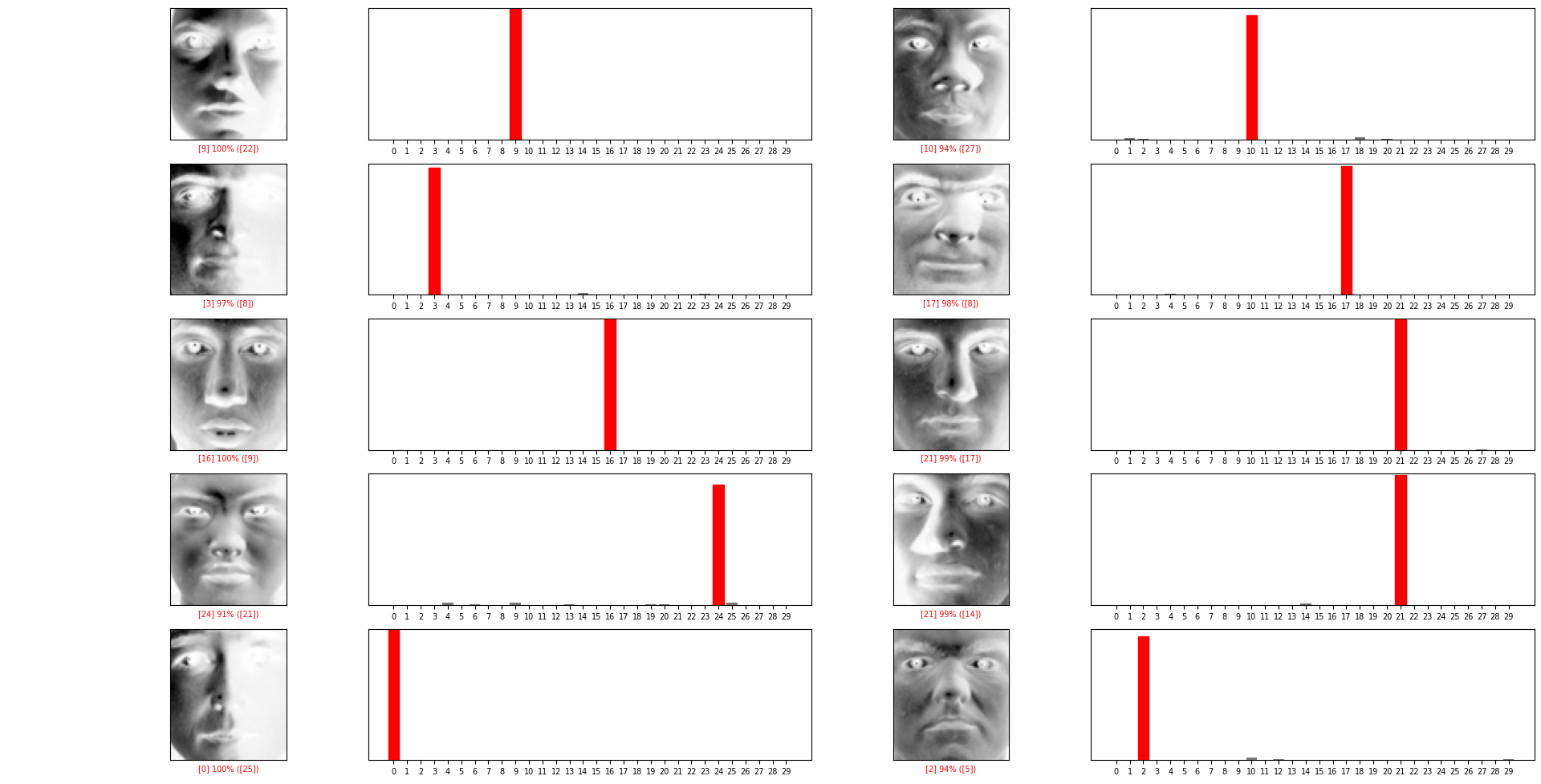


Figure - A few samples from the test-data with their evaluation probability on the right-hand side

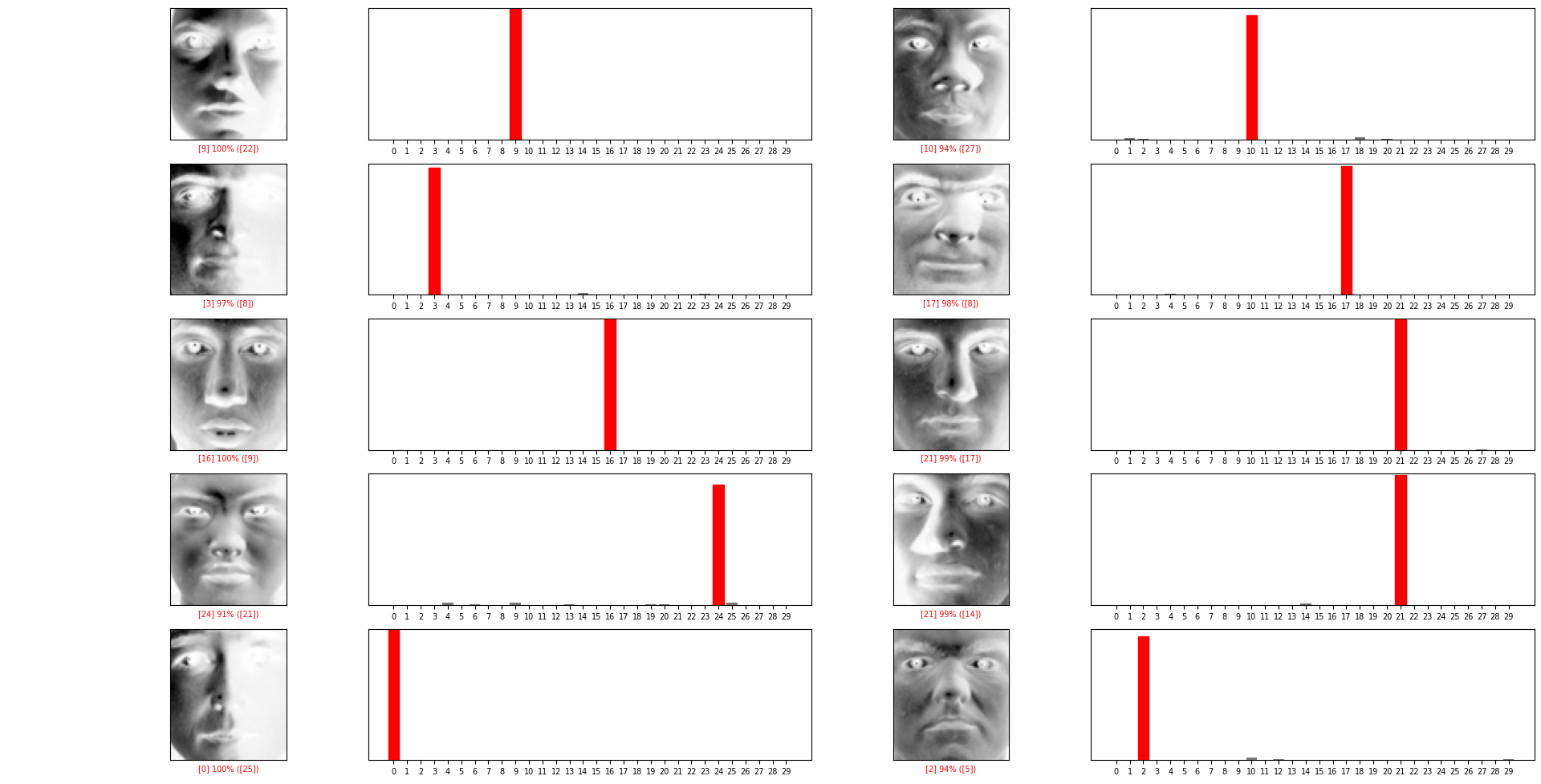


Figure - (...continuation) a few more samples of the evaluation of the test-data

# 3. Experiments

Up until this point it was covered the rough architecture of the neural network, with little to no regard to the performance impact of the hyper-parameters.

In this section, it is described the experimentation of different possible hyper-parameters which were all tested individually in order to graph and visualize their impact on the accuracy of the model.

To that end, it was created a script that tested all the possible combinations of the hyper-parameters to be adjusted, which in this case, were only two, the number of neurons and the epochs of training, the script is shown below.

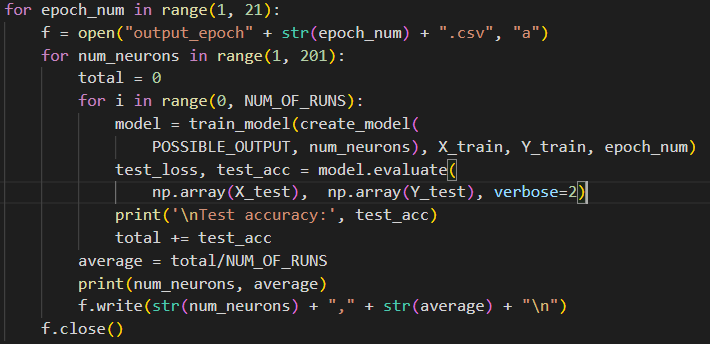


Figure - Hyper-parameter testing script

It iterates through the 20 different epochs so that it can be observed how the number of epochs affects the other hyper-parameter being tested here, it was stopped at 20 due to the high computational requirements as for each epoch it would also be tested how the number of neurons affected the network. That being the next step of the loop, as it iterated through the given testing range of possible neurons it was also decided to test each neuron number setting for 5 times and then average out the result as it typically seemed that the accuracy commonly had a wide range of accuracy results. The averaged accuracy result would then be written in the csv file as a new line. As each epoch iteration saved a new csv file table as all the computations were finished it resulted into the following files being created.

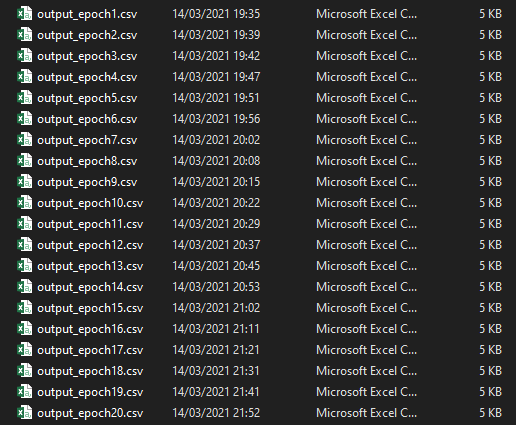


Figure - Script output files

Finally, for better visualization of the results, these files were processed into a single file so that this data could be saved in a single place and graphed for easy analysis. The graph repetition of said data is displayed bellow.

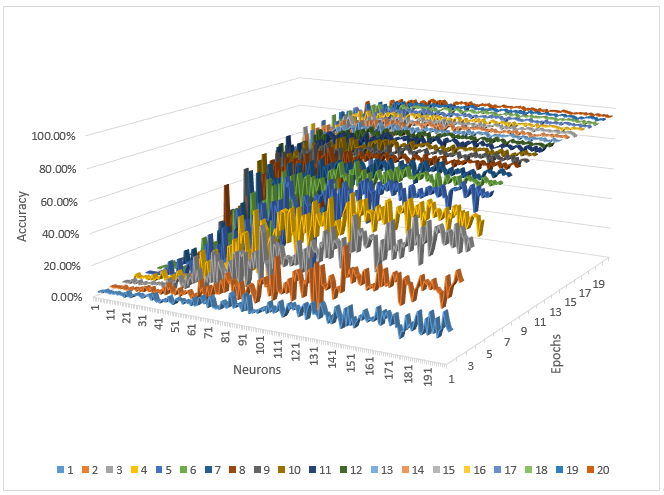


Figure – 3D Representation of the impact of a different number of epochs and different number of neurons on the accuracy of the model

From the graphic representation above it can be inferred that:

* the impact of the number of neurons on the accuracy of the model grows with the number of epochs;
* the epochs cause the greatest improvements up until epoch 8, despite this, it is clear that there is an improvement in accuracy all the way up until epoch 20;
* the model seems to become more consistent with more epochs, as it can be observed that during the first epochs, the increase of neurons causes a highly inconsistent improvement in the accuracy;
* after approximately 8 epochs, the number of neurons beyond approximately 120, do not bring major improvements to the accuracy, to better illustrate this, the best performing tested epoch was sliced out of the graph for better visualization, displayed below.

Figure - The impact of the number of neurons on the accuracy of the model

The conclusion of the tuning of the aforementioned two hyper-parameter settings comes down to the performance requirements of the model, the number of epochs will only affect the performance of training the model whereas the neurons will affect the performance of both training and using the model. This may mean that if the model is targeted to run on a slower device the number of neurons have to be reduced, but if the target system offered no performance compromises then the value could be easily set to a very high value even if this offers little improvement.

# 4. Conclusion

As mentioned in the introduction, while the accuracy of the solution is quite high, it comes with some serious limitations, limitations that essentially render this solution with hard applicability to any real life applications as, to start with, it assumes that new data will come normalized as provided by the dataset. This means, of course, that above all else the photos must be perfectly cropped around the face area and that these are in a perfect vertical rotation as in the dataset.

In theory the solution should not be vulnerable to the vertical flipping of the image, given that most faces are symmetrical, however if that is not the case than the solution will not work on flipped photos.

A more robust solution using this same dataset could also be developed using Convolutional Neural Networks that could at the very least take into account possible rotation angles, however for more significant and stout improvements it would likely be required another dataset altogether.

In the experiments section it was aimed to tune two parameters and as such these were both tested in all of their possible combinations, however this implies a very high computational requirement in order to determine the ideal values and this is given only 2 parameters, the introduction of a third one would cause these requirements to multiply by however many options the new parameter introduces. This makes it typically impossible to find ideal parameters for any given model and intrinsically there are other more efficient methods for obtaining good-enough parameters, such as random selections of different parameters and their comparison with each other which is roughly one of the algorithms that are already implemented in the many community available data science tools, such as “sklearn”.

# 5. References

Dabakoglu, C., 2018. *What is Convolutional Neural Network (CNN) ? — with Keras.* [Online]   
Available at: https://medium.com/@cdabakoglu/what-is-convolutional-neural-network-cnn-with-keras-cab447ad204c  
[Accessed 08 03 2021].

Keras, 2021. *Accuracy metrics.* [Online]   
Available at: https://keras.io/api/metrics/accuracy\_metrics/  
[Accessed 08 03 2021].

NumPy, 2021. *Array creation.* [Online]   
Available at: https://numpy.org/doc/stable/user/basics.creation.html  
[Accessed 08 03 2021].

Nvidia, 2021. *Arithmetic Instructions.* [Online]   
Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions  
[Accessed 7 03 2021].

Rampurawala, M., 2019. *Classification with TensorFlow and Dense Neural Networks.* [Online]   
Available at: https://heartbeat.fritz.ai/classification-with-tensorflow-and-dense-neural-networks-8299327a818a  
[Accessed 08 03 2021].

Saha, S., 2018. *A Comprehensive Guide to Convolutional Neural Networks.* [Online]   
Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53  
[Accessed 07 03 2021].

Schetinin, V., 2021. *Assignment 1.* [Online]   
Available at: https://docs.google.com/document/d/1KsEsZVNdJZ1rr3YbAwKSfPpQkgUNRzk7CCNKr4Momak  
[Accessed 07 March 2021].

scikit-learn, 2021. *sklearn.model\_selection.train\_test\_split.* [Online]   
Available at: https://scikit-learn.org/stable/modules/generated/sklearn.model\_selection.train\_test\_split.html  
[Accessed 08 03 2021].

TensorFlow, 2017. *Basic classification: Classify images of clothing.* [Online]   
Available at: https://www.tensorflow.org/tutorials/keras/classification  
[Accessed 08 03 2021].

Tensorflow, 2021. *tf.keras.layers.Flatten.* [Online]   
Available at: https://www.tensorflow.org/api\_docs/python/tf/keras/layers/Flatten  
[Accessed 08 03 2021].

Versloot, C., 2019. *ReLU, Sigmoid and Tanh: today’s most used activation functions.* [Online]   
Available at: https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/  
[Accessed 08 03 2021].

# 6. Appendix

import sys

from os import listdir

from matplotlib import image

from matplotlib import pyplot

import numpy as np

from numpy import asarray

from sklearn.model\_selection import train\_test\_split

import tensorflow as tf

from tensorflow import keras

import cv2

import matplotlib.pyplot as plt

tf.get\_logger().setLevel('INFO')

PATH = "."

DNAME = "/dataset"

FILES = listdir(PATH + DNAME)

N\_FILES = len(FILES)

IM1 = image.imread(PATH + DNAME + '/' + FILES[0])

N\_PIXELS = IM1.shape[0]\*IM1.shape[1]

def get\_parsed\_dataset():

    images = []

    labels = []

    for i in range(0, N\_FILES):

        filename = FILES[i]

        img = cv2.imread(PATH + DNAME + '/' + filename, cv2.IMREAD\_GRAYSCALE)

        images.append(img)

        lbl = int(filename[5:7])

        labels.append(lbl)

    temp\_labels = labels.copy()

    for n in range(len(set(labels))-1, -1, -1):

        current\_max = max(temp\_labels)

        for i in range(len(labels)-1, -1, -1):

            if temp\_labels[i] == current\_max:

                temp\_labels[i] = -1

                labels[i] = n

    return (images, labels)

def create\_model(POSSIBLE\_OUTPUT):

    model = keras.Sequential([

        keras.layers.Flatten(input\_shape=(

            IM1.shape[0], IM1.shape[1])),  # width and height

        keras.layers.Dense(110, activation='relu'),

        keras.layers.Dense(POSSIBLE\_OUTPUT)  # (30) Possible Output Options

    ])

    model.compile(optimizer='adam',

                  loss=keras.losses.SparseCategoricalCrossentropy(

                      from\_logits=True),

                  metrics=['accuracy'])

    return model

def train\_model(model, train\_data, train\_labels):

    model.fit(x=np.array(train\_data),

              y=np.array(train\_labels),

              epochs=20)

    return model

def plot\_image(i, predictions\_array, true\_label, img):

    true\_label, img = true\_label[i], img[i]

    plt.grid(False)

    plt.xticks([])

    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted\_label = np.argmax(predictions\_array)

    if predicted\_label == true\_label:

        color = 'blue'

    else:

        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format([predicted\_label],

                                         100\*np.max(predictions\_array),

                                         [true\_label]),

               color=color)

def plot\_value\_array(predictions\_array):

    plt.grid(False)

    x\_range = range(len(predictions\_array))

    plt.xticks(x\_range)

    plt.yticks([])

    thisplot = plt.bar(x\_range,

                       predictions\_array, color="#777777")

    plt.ylim([0, 1])

    predicted\_label = np.argmax(predictions\_array)

    thisplot[predicted\_label].set\_color('red')

if \_\_name\_\_ == "\_\_main\_\_":

    dataset\_images, dataset\_labels = get\_parsed\_dataset()

    dataset\_images = np.array(dataset\_images)

    dataset\_images = dataset\_images / 255.0

    # X represents the dataset\_images and Y represents the dataset\_labels

    X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(

        dataset\_images, dataset\_labels, test\_size=0.3, random\_state=42)

    POSSIBLE\_OUTPUT = len(set(dataset\_labels))  # 30

    model = train\_model(create\_model(POSSIBLE\_OUTPUT), X\_train, Y\_train)

    test\_loss, test\_acc = model.evaluate(

        np.array(X\_test),  np.array(Y\_test), verbose=2)

    print('\nTest accuracy:', test\_acc)

    probability\_model = tf.keras.Sequential([model,

                                             tf.keras.layers.Softmax()])

    predictions = probability\_model.predict(np.array(X\_train))

    num\_rows = 5

    num\_cols = 2

    num\_images = num\_rows\*num\_cols

    plt.rcParams.update({'font.size': 7})

    plt.figure(figsize=(12\*num\_cols, 2\*num\_rows))

    for i in range(num\_images):

        plt.subplot(num\_rows, 2\*num\_cols, 2\*i+1)

        plot\_image(i, predictions[i], Y\_test, np.array(X\_test))

        plt.subplot(num\_rows, 2\*num\_cols, 2\*i+2)

        plot\_value\_array(predictions[i])

    plt.tight\_layout()

    plt.show()