NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

MSc: DATA SCIENCE AND INFORMATION TECHNOLOGIES

Machine Learning
2ND Assignment

Xristos Morfopoulos (7115152100011)

Dimitris Plakas (DS2190015)

Supervisor: Stavros Perantonis

Winter Semester

2021 – 2022

Table of Contents

Exercise 1.

Part A.

The main mathematical computations that take part in a Multi-Layered Perceptron are explained briefly below.

We start with the initialization of the weights with small values, uniformly distributed between -a and a, where a is a positive real number in order to avoid the vanishing gradients issue.

The next step of the process is the Feed-Forward calculations, which are calculations of every neuron starting from the 1st hidden layer to the output layer. As we have calculated the final output from the output layer, we can compute the Cost Function. In this way, using backward equations and with the help of the chain rule we can calculate the gradients recursively from the last layer to the previous layer and finally update the weights.

The feedforward, backward equations as well as the weight updates is an iterative process, and it terminates when the cost drops below a given threshold, or the norm of the cost function gradient drops below a given threshold.

The general form of the backpropagation equations for Multi-layered Perceptrons and certain activation functions such as ReLU, Hyperbolic Tangent and Sigmoid are described below:

i) Using ReLU

Forward Calculations with ReLu Activation function:

$$y_{i\mu}^{(r)} = f\left(\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right) = max\left(\sum_k w_{ij}^{(r)} y_{i\mu}^{(r-1)}, 0\right)$$

Cost Function Calculations:

$$E = \frac{1}{2}\sum_{k\mu} \left(e_{k\mu}\right)^2$$
$$e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$$

Backward Calculations for the last layer R and any layer r:

$$\delta_{i\mu}^{(R)} = \frac{1}{2}\left(1 + y_{i\mu}^{(R)}\right)\left(1 - y_{i\mu}^{(R)}\right)\left(y_{i\mu}^{(R)} - T_{i\mu}\right)$$
$$\delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} w_{ki}^{(r+1)} \left(\frac{1}{2}\left(1 + y_{i\mu}^{(r)}\right)\left(1 - y_{i\mu}^{(r)}\right)\right)$$

Weight Updates:

$w_{ij}^{(r)}(new) = w_{ij}^{(r)}(old) + \delta w_{ij}^{(r)}, \delta w_{ij}^{(r)} = -\epsilon \sum_k \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}$

Range of Gradient:

$0 < \nabla E < 1/2$

ii) Using Hyperbolic Tangent

Forward Calculations with Hyperbolic Tangent Activation function:

$y_{i\mu}^{(r)} = f\left(\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right) = \frac{1-\exp\left(-\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right)}{1+\exp\left(-\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right)}$

Cost Function Calculations:

$E = \frac{1}{2}\sum_{k\mu} \left(e_{k\mu}\right)^2$
$e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$

Backward Calculations for the last layer R and any layer r:

$\delta_{i\mu}^{(R)} = \frac{1}{2}\left(1 + y_{i\mu}^{(R)}\right)\left(1 - y_{i\mu}^{(R)}\right)\left(y_{i\mu}^{(R)} - T_{i\mu}\right)$
$\delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} w_{ki}^{(r+1)} \left(\frac{1}{2}\left(1 + y_{i\mu}^{(r)}\right)\left(1 - y_{i\mu}^{(r)}\right)\right)$

Weight Updates:

$w_{ij}^{(r)}(new) = w_{ij}^{(r)}(old) + \delta w_{ij}^{(r)}, \delta w_{ij}^{(r)} = -\epsilon \sum_k \delta_{i\mu}^{(r)} y_{j\mu}^{(r-1)}$

Range of Gradient:

$0 < \nabla E < 1/2$

iii) Using Sigmoid

Forward Calculations with Sigmoid Activation function:

$$y_{i\mu}^{(r)} = f\left(\sum_k w_{ij}^{(r)} y_{j\mu}^{(r-1)}\right) = \frac{1}{1+\exp\left(-\sum_k w_{ij}^{(r)} y_{i\mu}^{(r-1)}\right)}$$

Cost Function Calculation:

$$E = \frac{1}{2}\sum_{k\mu} \left(e_{k\mu}\right)^2$$
$$e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}$$

Backward Calculations for the last layer R and any layer r:

$$\delta_{i\mu}^{(R)} = y_{i\mu}^{(R)}\left(1 - y_{i\mu}^{(R)}\right)\left(y_{i\mu}^{(R)} - T_{i\mu}\right)$$
$$\delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} w_{ki}^{(r+1)} \left(y_{i\mu}^{(r)}\left(1 - y_{i\mu}^{(r)}\right)\right)$$

Weight Updates:

$$w_{ij}^{(\mathbf{r})}(\text{ new }) = w_{ij}^{(r)}(old) + \delta w_{ij}^{(r)}, \delta w_{ij}^{(r)} = -\epsilon \sum_k \delta_{i\mu}^{(r)} y_{j\mu}^{(\mathbf{r}-1)}$$

Range of Gradient:

$$0 < \nabla E < 1/4$$

Part B.

In part B, our task was to compile various MLP models with certain activation functions. We downloaded the MNIST dataset, and we normalized the array dividing with 255.Each hidden layer of the model should have 32 units and as an output layer a SoftMax function. Starting with a certain activation function such as Sigmoid, we should construct 3 different models with 5,20,40 layers respectively. We implemented the same process with the hyperbolic tangent and ReLu activation functions. In this way, we constructed 9 different models in total. We should also mention that the loss function is categorical cross entropy, and the optimizer is SGD with learning rate 0.01. The training of each model is set to 10 epochs and 64 batch sizes.

In our implementation, each model is stored locally in a Python dictionary, with respect to their activation function, and with its hidden layers as keys. For example, the dictionary **dict_relu_models** have 3 different keys (5,10,20) with respect to their “hidden” layers and Relu activation functions.

The **performance** of the 9 models is validated on the test dataset of the MNIST and it is described below:

The performance of the “**ReLu**” model with **5** hidden layers is at **96.2% Accuracy** in the Test Dataset, while the “**ReLu**” model with **20** hidden layers is at **91.8% Accuracy**. Finally, the “**ReLu**” model with **40** hidden layers is at **19.6% Accuracy.**

The performance of the “**Hyperbolic Tangent**” model with **5** hidden layers is **at 96.1% Accuracy** in the Test Dataset, while the “**Hyperbolic Tangent** model with **20** hidden layers is at **96.2% Accuracy**. Finally, the “**Hyperbolic Tangent**” model with **40** hidden layers is at **95.4% Accuracy**.

The performance of the “**Sigmoid**” model with **5** hidden layers is at **11.3% Accuracy** in the Test Dataset, while the “**Sigmoid**” model with **20** hidden layers is at **11.3% Accuracy**. Finally, the “**Sigmoid**” model with **40** hidden layers is at **11.3% Accuracy.**

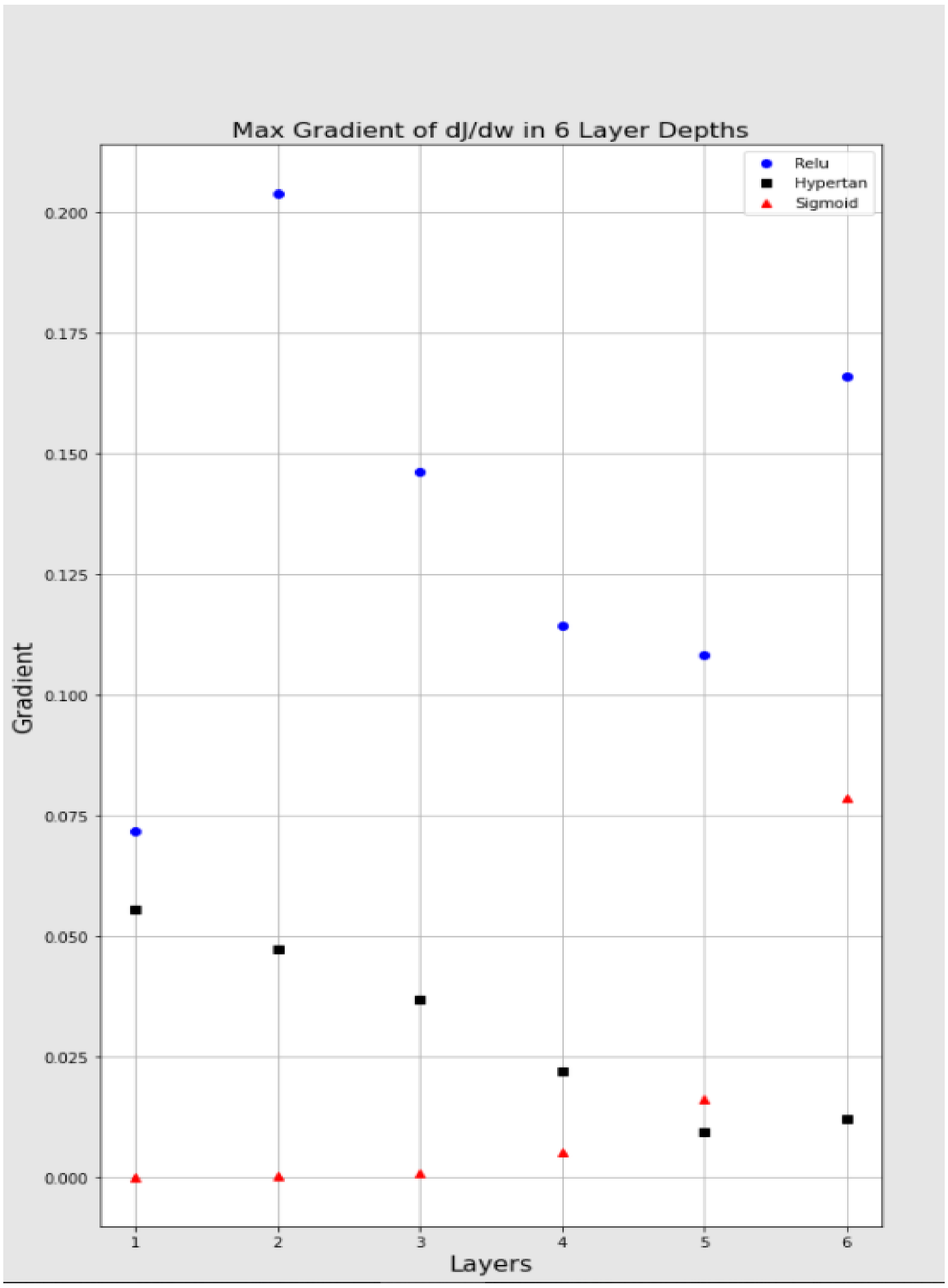
From the above findings, we can easily conduct that the “**Sigmoid**” model has very low performance in each case. We believe that the main issue for Sigmoid is the vanishing gradient (because of the calculation of its derivative) and that’s why the sigmoid activation function is easily neglected now days

We should mention that the “**Hyperbolic Tangent**” model (20 layers) has achieved the best performance at **96.2% Accuracy** along with of the “**ReLu**” model with **5** hidden layers. **However, we can easily validate that the all the “Hyperbolic Tangent” models are more robust in each case keeping high performance, whereas the “ReLu” models have lower performance as the hidden layers increasing**.

Part C.

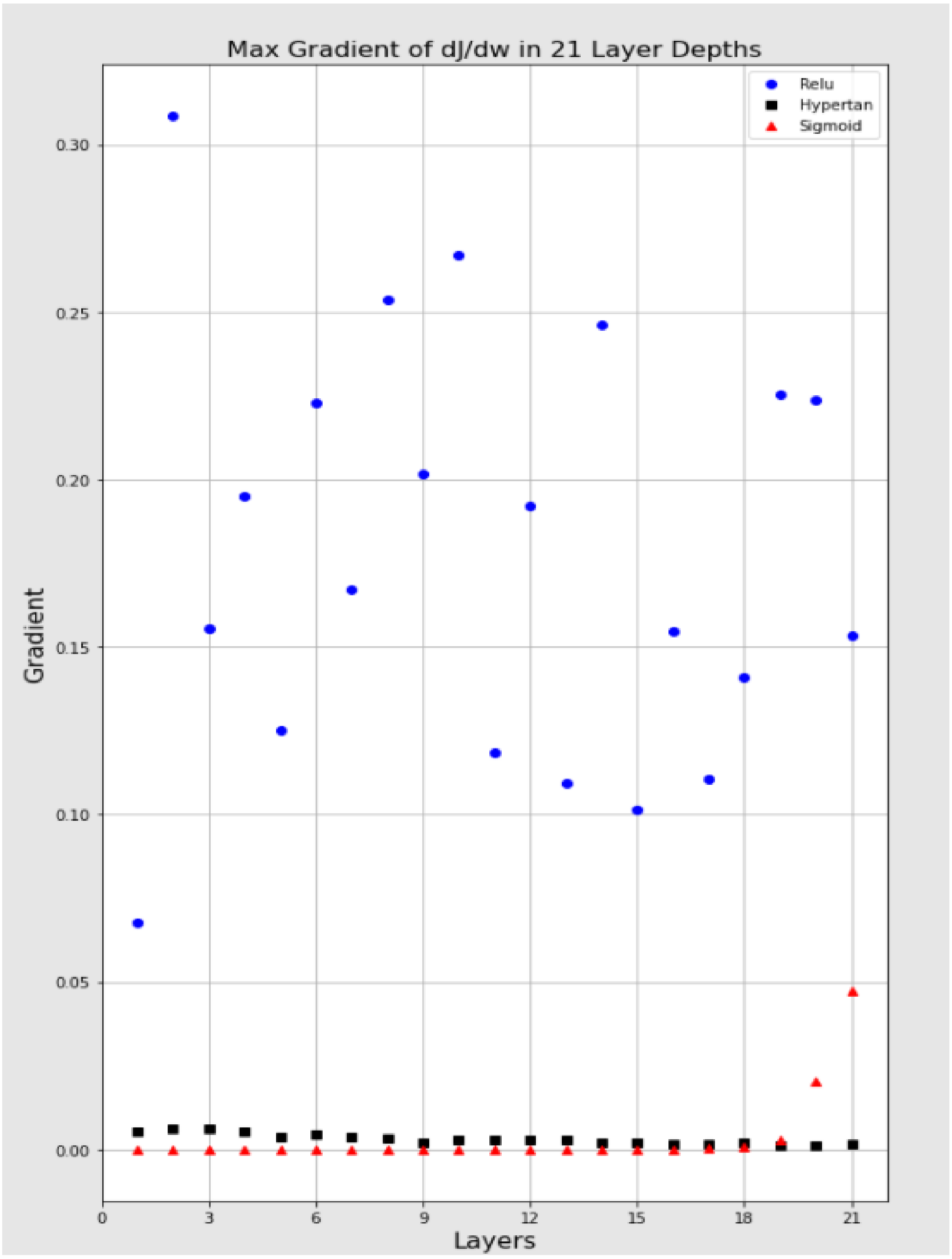
In part C, our goal was to obtain all the gradients for each layer from the 9 models. In the training process, we trained the models with 3 epochs & 64 batch size and **with the usage of tf.GradientTape(),** we can easily track the gradients from each layer.

Fig. 1   Plot “layer depth vs max gradient”, of models with 5 Hidden Layers



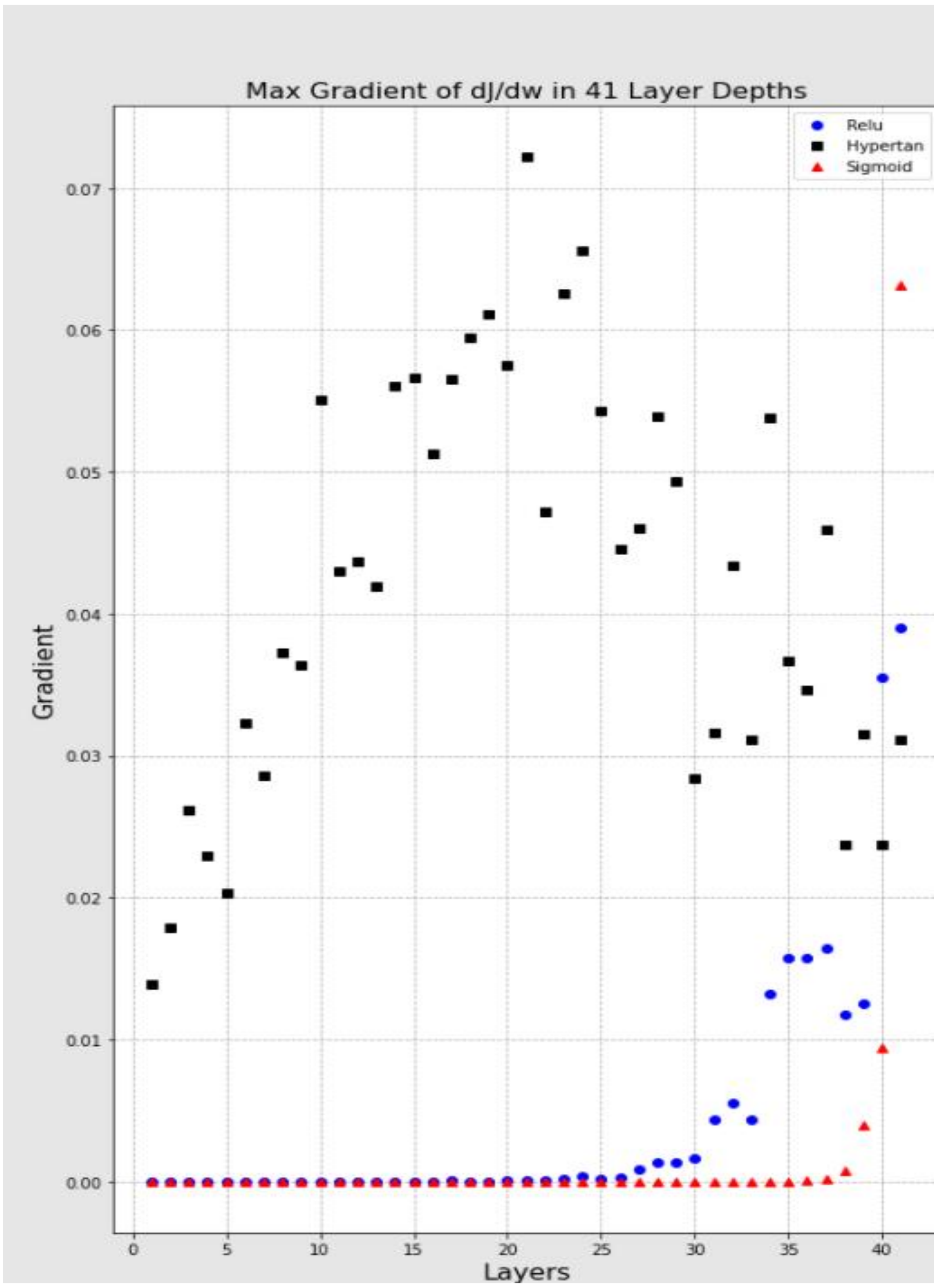
From the graph above, we can easily visualize the gradients values per Layer Depth. In this diagram, the 6 Layer Depths denotes the “5” hidden layer plus the output layer. We can easily validate that the gradients of the sigmoid “5” hidden layer model and the hypertan “5” hidden layer model are the smallest, whereas the ReLu gradients are the biggest. We should also mention that the gradients of the Hypertan model are decreasing per epoch

Fig. 2 Plot "layer depth vs max gradient", of models with 20 Hidden Layers



From this graph, we can easily conduct that the gradients of sigmoid and hypertan (20 layers) model follow the same pattern (smallest value) and their values ranges from 0 to 0.05. However, the Relu gradients values range above 0.05, reaching the maximum value equal to 0.3 at the second hidden layer.

Fig. 3   Plot "layer depth vs max gradient", of models with 40 Hidden Layers



In this graph, we can easily conclude that the Hypertan 40 hidden model has the biggest values in contrast to the previous hypertan models. In addition, the Relu gradients have also smaller values in the 40-layer model with respect to the previous models. Therefore, the "behaviour" of Hypertan model is preferable along the 40-layer models. Finally, the sigmoid 40 Layer model has very small gradients, 'keeping' the same pattern with the previous sigmoid models.

Part D.

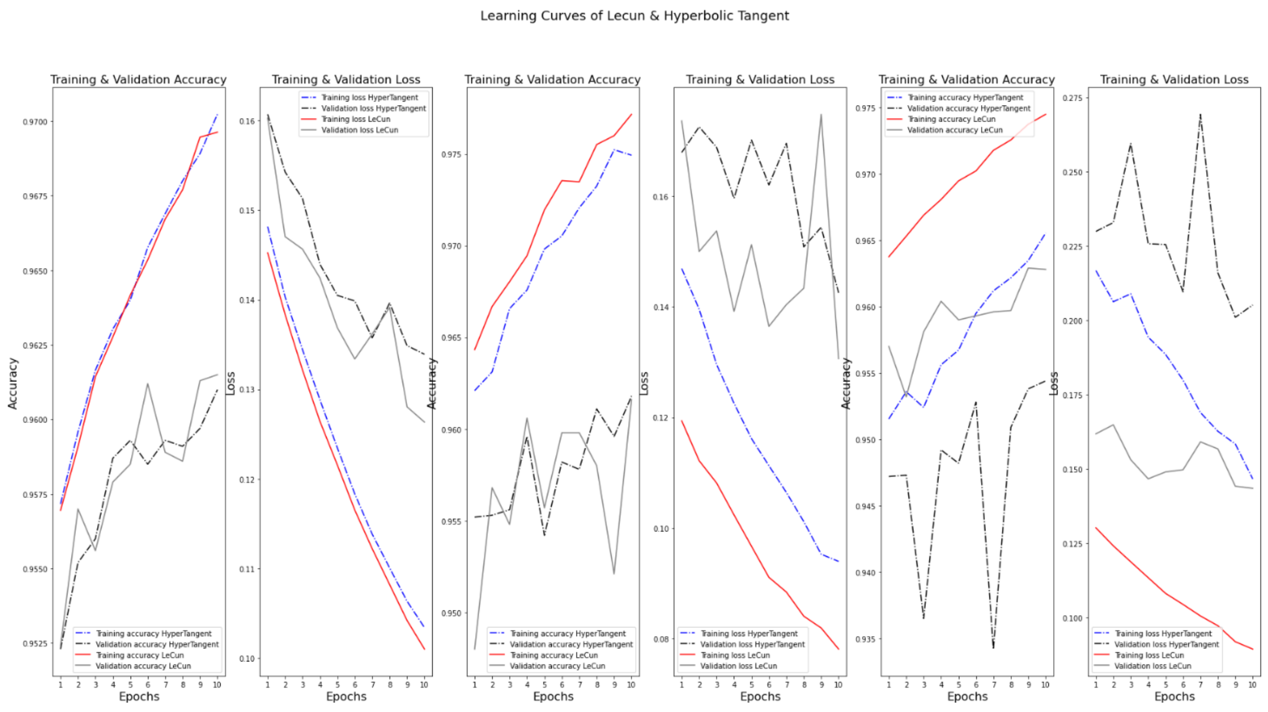
In this part, we had to implement LeCun as an activation function. We defined a custom function that implements the LeCun function and finally integrate it in the Dense Layers. In this way, we constructed 3 LeCun Models with 5,20,40 Hidden Layers.

The performance of the LeCun models is remarkable, reaching the maximum Test Accuracy of 96.3%.

In more detail, the performance of the "**LeCun**" model with **5** hidden layers is at **96.1% Accuracy** in the Test Dataset, while the "**LeCun**" model with **20** hidden layers is at **96.2% Accuracy**. Finally, the "**LeCun**" model with **40** hidden layers is at **96.3% Accuracy.**

A powerful tool to compare the LeCun models with the Hyperbolic Tangent models is the Learning Curves with respect to Training & Validation Accuracy and Training & Validation Loss.

Fig. 4 Learning Curves of Lecun & Hyperbolic Tangent Models



At a quick glance from the above graph, we can easily conclude that the LeCun Models are better from the Hyperbolic Tangent models in terms of the performance. With respect to 5 hidden layers models, the metrics are more or less the same**, however for the 20,40 hidden layers LeCun Models have not only bigger Training & Validation Accuracy, but also less Training & Validation Loss**. In particular, the 40 hidden LeCun Model has 97 % training Accuracy whereas the Hyperbolic Tangent has 96% Accuracy. In addition, the Model has also 96 % validation Accuracy whereas the Hyperbolic Tangent has 95% validation Accuracy.

The backpropagation equations and the gradient range for the LeCun activation function are defined below:

Cost Function Calculations and its Partial Derivative:

$$E = \frac{1}{2}\sum_{k\mu}\left(e_{k\mu}\right)^2, e_{k\mu} = T_{k\mu} - y_{k\mu}^{(R)}, \quad \delta\omega = -\varepsilon\nabla E$$

$$\delta\omega_{ij}^{(r)} = -\varepsilon\frac{\theta E}{\theta\omega_{ij}^{(r)}}$$

$$\frac{\theta E}{\theta_{\omega_i}} = \sum_t \delta_{i\mu}^{(r)} y_{i\mu}^{(r-1)}$$

Backward Equations & Weight Updates:

$$\delta_{i\mu}^{(r)} = \sum_k \delta_{k\mu}^{(r+1)} \cdot \omega^{(r+1)} \cdot F'\left(v_{i\mu}^{(r)}\right)$$

$$\delta_{it}^{(R)} = F'\left(v_{i\mu}^{(R)}\right)\left(y_{i\mu}^{(R)} - T_{i\mu}\right)$$

$$\omega_{ij}^{(r)}(new) = \omega_{ii}^{(r)} - \varepsilon\frac{\theta F}{\theta_{w:j}^{(r)}}$$

LeCun Activation Function & Derivative:

$$F\left(v_{i\mu}{}^{(r)}\right) = \mathrm{Le\,Cun}\left(v_{i\mu}^{(r)}\right) = 1.7159\tanh\left(\frac{2}{3}v_{i\mu}^{(r)}\right) + 0.01v_{i\mu}^{(r)}$$

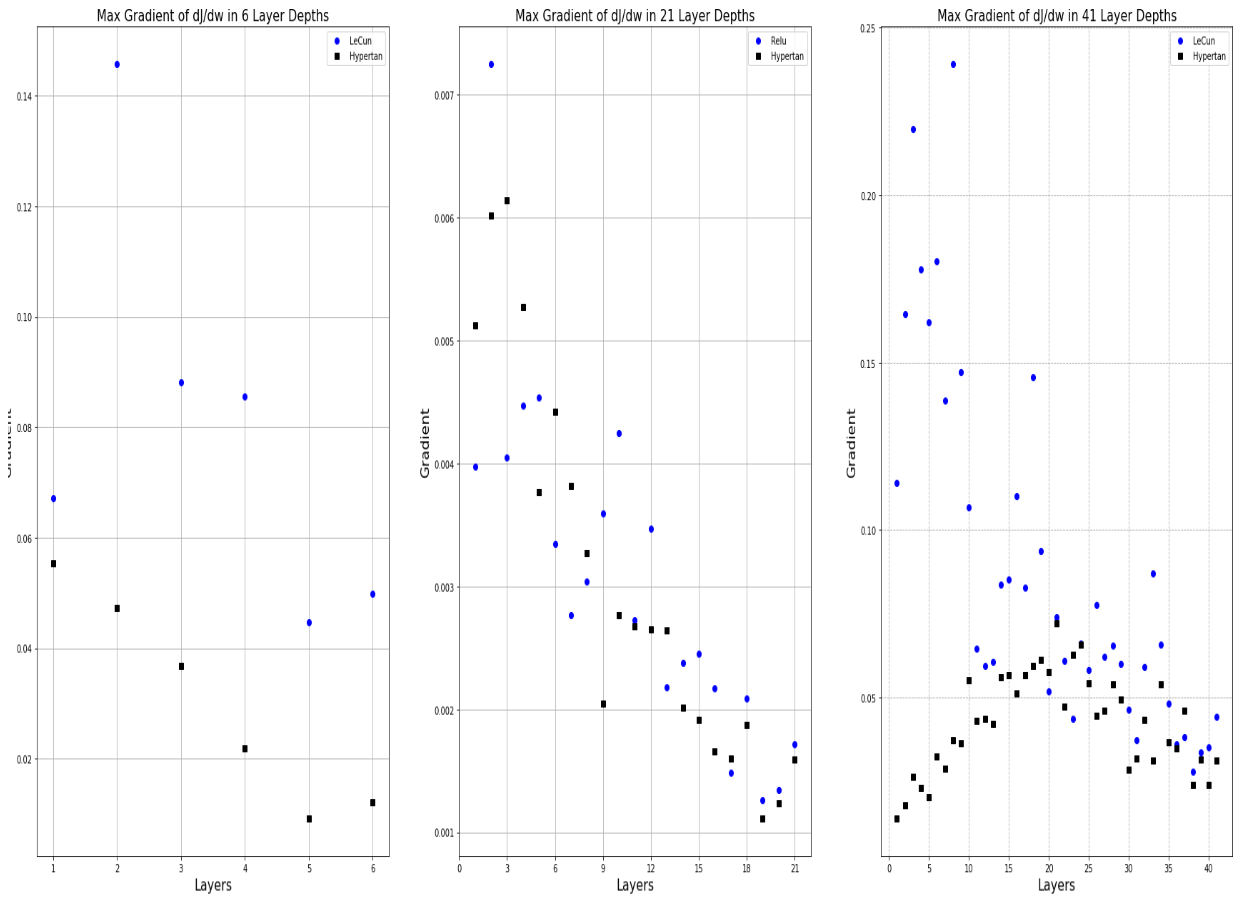
$$F'\left(v_{i\mu}^{(r)}\right) = 1.14393\left(1 - \tanh^2\left(\frac{2}{3}v_{ip}^{(r)}\right) + 0.01\right.$$

$$\tanh^2 ax \in (0,1) \quad \forall a \in \mathbb{R}$$

Range of Gradient:

$$0.01 \leqslant F'\left(v_{i\mu}^{r}\right) \leqslant 1.15393$$

Fig. 5 Plot "layer depth vs max gradient", of LeCun and Hyperbolic Tangent



In the graph above we can visualize the gradients of the LeCun Models and the Hyperbolic Tangent models in each layer. In general, the gradients of the Lecun Models are bigger than the gradients of the Hyperbolic Tangent models, especially in the models with 5 Hidden Layers and 40 Hidden Layers.

Exercise 2.

In this exercise our goal is to construct and train a convolutional neural network for the MNIST dataset. The network consists of 2 consecutive convolution layers of 32 and 64 3x3 filters respectively along with ReLu activation function. After the convolution layers we apply MaxPooling size 2x2 and a Dropout with 25% probability. Then we flatten the neurons, and we add a Dense Layer with 128 units and a ReLu activation function. Finally, we added a Dropout with 25% probability and for the output layer a Softmax. The loss function is catergorical crossentropy and the optimizer is Adam with learning rate 0.01. We trained the model for 4 epochs and with a batch size of 128 images. **The performance of the model is very good on the test dataset with a 97% Accuracy.**

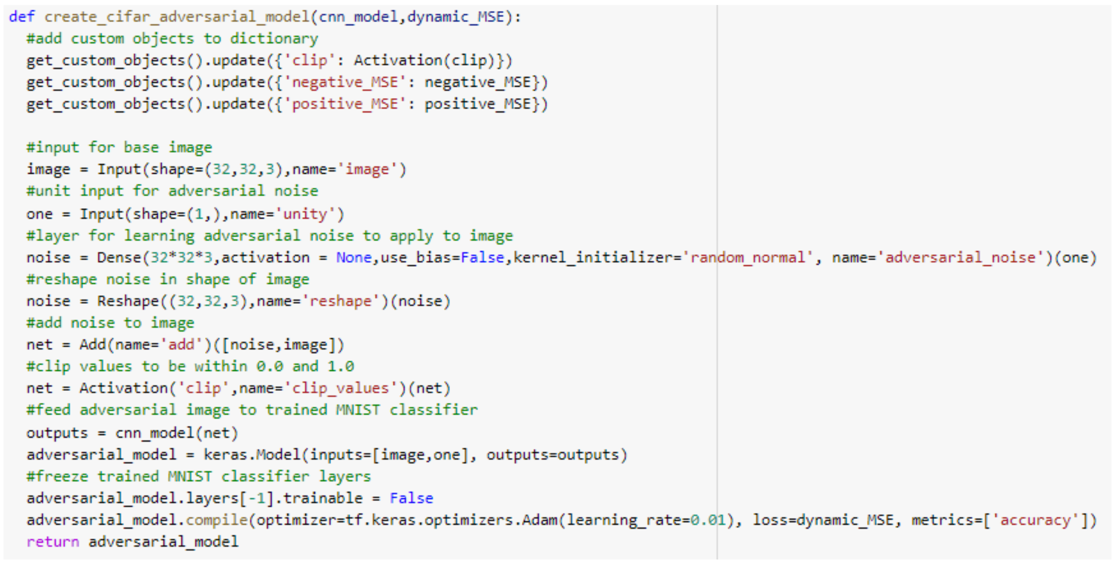
In the same way, we trained a Convolutional Neural Network for the **CIFAR dataset**. We should mention that the CIFAR dataset is quite different from the MNIST dataset while each image has 32 x 32 dimensions and 3 channels since they are RGB images and not Grayscale. The architecture for the CIFAR CNN model consists of 2 consecutive convolution layers of 64 3x3 filters with ReLu activation function and a MaxPooling size 2x2. We added a Dropout layer with 40% probability and then we apply again 2 consecutive convolution layers of 128 3x3 filters. In order to avoid overfitting, we apply again a Dropout Layer with 40% probability and then we flatten the neurons. Finally, we added 2 consecutive Dense layers of 1024 units with ReLu activation function and for the output layer a Softmax. The loss function is catergorical crossentropy and the optimizer is Adam. **The network has been trained for 50 epochs and the performance on the test dataset is acceptable with a 75 % Accuracy.**

Both models have been trained and saved in Google Colab for later use. The import /load of the models can be achieved after we have mounted our Google Drive.

The second goal of this exercise is to construct adversarial examples. An adversarial example can be built with a certain procedure, focusing and training on the noise of a certain image, so that the image is not classified correctly anymore from the model.

The architecture of our adversarial model consists of an input layer of 28x28x1 for the input images and then apply a Dense Layer of 784 (28*28) units for the noise. Afterwards we reshape the noise in a 28x28x1 size and finally added to the image. Finally, all the later layers come from the pretrained MNIST classifier and are excluded from training. We have used Adam optimizer with 0.01 learning rate and the loss function is MSE. We should mention that the loss function is defined dynamically through the code, and it is custom built, when the target is equal to the “desired” target which we want to train our adversarial example the loss function is set to Negative MSE and when the target is not equal to the “desired” target the loss function is set to Positive MSE.

Fig. 6 Python Function of Adversarial Model Architecture

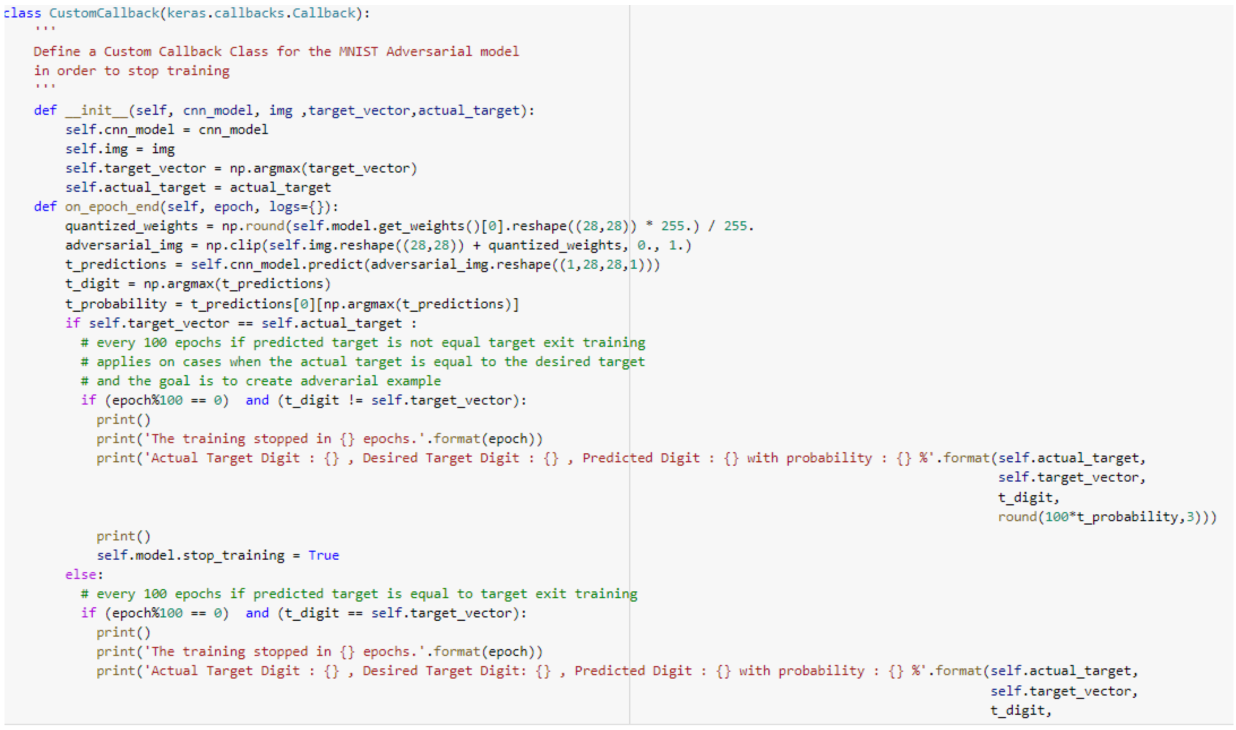


```python
def create_cifar_adversarial_model(cnn_model,dynamic_MSE):
  #add custom objects to dictionary
  get_custom_objects().update({'clip': Activation(clip)})
  get_custom_objects().update({'negative_MSE': negative_MSE})
  get_custom_objects().update({'positive_MSE': positive_MSE})

  #input for base image
  image = Input(shape=(32,32,3),name='image')
  #unit input for adversarial noise
  one = Input(shape=(1,),name='unity')
  #layer for learning adversarial noise to apply to image
  noise = Dense(32*32*3,activation = None,use_bias=False,kernel_initializer='random_normal', name='adversarial_noise')(one)
  #reshape noise in shape of image
  noise = Reshape((32,32,3),name='reshape')(noise)
  #add noise to image
  net = Add(name='add')([noise,image])
  #clip values to be within 0.0 and 1.0
  net = Activation('clip',name='clip_values')(net)
  #feed adversarial image to trained MNIST classifier
  outputs = cnn_model(net)
  adversarial_model = keras.Model(inputs=[image,one], outputs=outputs)
  #freeze trained MNIST classifier layers
  adversarial_model.layers[-1].trainable = False
  adversarial_model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01), loss=dynamic_MSE, metrics=['accuracy'])
  return adversarial_model
```

The training of the adversarial model is set to **3000 epochs**, however to avoid delay we have defined **a CallBack Class for the Network**. The Callback Class checks per 100 epochs if the pretrained MNIST classifier can correctly classify the adversarial example. In this way if an adversarial example has been properly built then **it stops training**.

Fig. 7 Python Class of CustomCallback



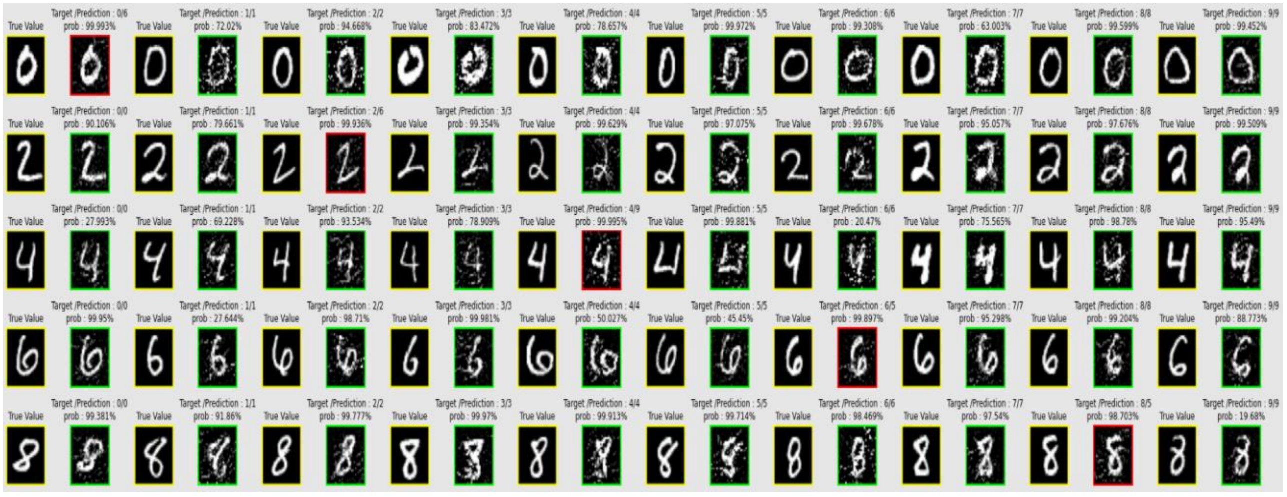
```python
class CustomCallback(keras.callbacks.Callback):
    '''
    Define a Custom Callback Class for the MNIST Adversarial model
    in order to stop training
    '''
    def __init__(self, cnn_model, img ,target_vector,actual_target):
        self.cnn_model = cnn_model
        self.img = img
        self.target_vector = np.argmax(target_vector)
        self.actual_target = actual_target
    def on_epoch_end(self, epoch, logs={}):
        quantized_weights = np.round(self.model.get_weights()[0].reshape((28,28)) * 255.) / 255.
        adversarial_img = np.clip(self.img.reshape((28,28)) + quantized_weights, 0., 1.)
        t_predictions = self.cnn_model.predict(adversarial_img.reshape((1,28,28,1)))
        t_digit = np.argmax(t_predictions)
        t_probability = t_predictions[0][np.argmax(t_predictions)]
        if self.target_vector == self.actual_target :
          # every 100 epochs if predicted target is not equal target exit training
          # applies on cases when the actual target is equal to the desired target
          # and the goal is to create adverarial example
          if (epoch%100 == 0)  and (t_digit != self.target_vector):
            print()
            print('The training stopped in {} epochs.'.format(epoch))
            print('Actual Target Digit : {} , Desired Target Digit : {} , Predicted Digit : {} with probability : {} %'.format(self.actual_target,
                                                                                                                               self.target_vector,
                                                                                                                               t_digit,
                                                                                                                               round(100*t_probability,3)))


            print()
            self.model.stop_training = True
        else:
          # every 100 epochs if predicted target is equal to target exit training
          if (epoch%100 == 0)  and (t_digit == self.target_vector):
            print()
            print('The training stopped in {} epochs.'.format(epoch))
            print('Actual Target Digit : {} , Desired Target Digit: {} , Predicted Digit : {} with probability : {} %'.format(self.actual_target,
                                                                                                                              self.target_vector,
                                                                                                                              t_digit,
```

As the exercise suggests, we took a sample (10 observations) of our five favourite digits (0,2,4,6,8) and we start training our adversarial model focusing to build adversarial examples for the 10 respective classes (from 0 to 9).

We have built an Image Grid in order to display all the adversarial examples along with the Original Images (True Values).

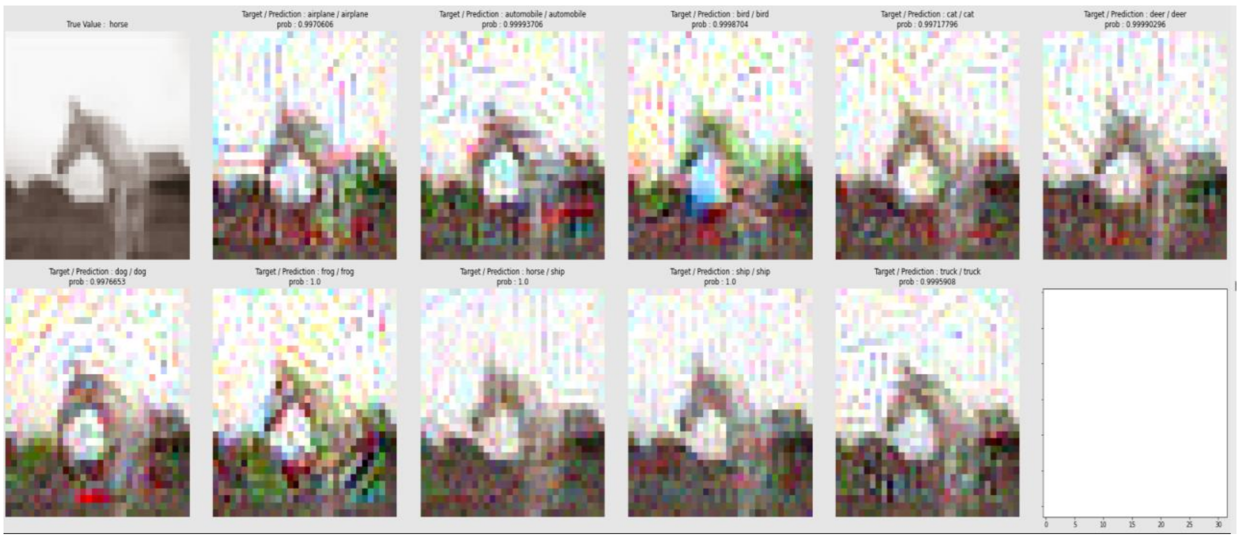
Fig. 8 Image Grid of Original & Adversarial Images along with predictions & probabilites


True Value
Target /Prediction : 0/6 prob : 99.993%
Target /Prediction : 1/1 prob : 72.02%
Target /Prediction : 2/2 prob : 94.668%
Target /Prediction : 3/3 prob : 83.472%
Target /Prediction : 4/4 prob : 78.657%
Target /Prediction : 5/5 prob : 99.972%
Target /Prediction : 6/6 prob : 99.308%
Target /Prediction : 7/7 prob : 63.003%
Target /Prediction : 8/8 prob : 99.599%
Target /Prediction : 9/9 prob : 99.452%
Target /Prediction : 0/0 prob : 90.106%
Target /Prediction : 1/1 prob : 79.661%
Target /Prediction : 2/6 prob : 99.936%
Target /Prediction : 3/3 prob : 99.354%
Target /Prediction : 4/4 prob : 99.629%
Target /Prediction : 5/5 prob : 97.075%
Target /Prediction : 6/6 prob : 99.678%
Target /Prediction : 7/7 prob : 95.057%
Target /Prediction : 8/8 prob : 97.676%
Target /Prediction : 9/9 prob : 99.509%
Target /Prediction : 0/0 prob : 27.993%
Target /Prediction : 1/1 prob : 69.228%
Target /Prediction : 2/2 prob : 93.534%
Target /Prediction : 3/3 prob : 78.909%
Target /Prediction : 4/9 prob : 99.995%
Target /Prediction : 5/5 prob : 99.881%
Target /Prediction : 6/6 prob : 20.47%
Target /Prediction : 7/7 prob : 75.565%
Target /Prediction : 8/8 prob : 98.78%
Target /Prediction : 9/9 prob : 95.49%
Target /Prediction : 0/0 prob : 99.95%
Target /Prediction : 1/1 prob : 27.644%
Target /Prediction : 2/2 prob : 98.71%
Target /Prediction : 3/3 prob : 99.981%
Target /Prediction : 4/4 prob : 50.027%
Target /Prediction : 5/5 prob : 45.45%
Target /Prediction : 6/5 prob : 99.897%
Target /Prediction : 7/7 prob : 95.298%
Target /Prediction : 8/8 prob : 99.204%
Target /Prediction : 9/9 prob : 88.773%
Target /Prediction : 0/0 prob : 99.381%
Target /Prediction : 1/1 prob : 91.86%
Target /Prediction : 2/2 prob : 99.777%
Target /Prediction : 3/3 prob : 99.97%
Target /Prediction : 4/4 prob : 99.913%
Target /Prediction : 5/5 prob : 99.714%
Target /Prediction : 6/6 prob : 98.469%
Target /Prediction : 7/7 prob : 97.54%
Target /Prediction : 8/5 prob : 98.703%
Target /Prediction : 9/9 prob : 19.68%

From the image above, we should mention that all the original images have a '**yellow'** border line, whereas all the adversarial examples have a '**green'** border line. The special cases where the adversarial examples have been trained with the 'desired' target equal to the 'actual' target have a '**red'** border line. At a quick glance at the image in the first row, we can easily see the true value of the image 0 and the special adversarial example 0 which is classified with 99% certainty as 6. The rest images in the first row are adversarial examples which have been trained with desired target not equal to their actual target (classes from 1 to 9) and we can easily validate the 4th image in the first row that is predicted from the MNIST pretrained classifier as 1 with 72% probability.

The same procedure for adversarial examples applies for the CIFAR dataset. We took randomly an image of a horse from the test dataset (the 87th index of the test) and we have created adversarial examples for each of the 10 classes.

Fig. 9   CIFAR Original & Adversarial Images along with predictions & probabilites


True Value : horse
Target / Prediction : airplane / airplane
prob : 0.9970606
Target / Prediction : automobile / automobile
prob : 0.99993706
Target / Prediction : bird / bird
prob : 0.9998704
Target / Prediction : cat / cat
prob : 0.99717796
Target / Prediction : deer / deer
prob : 0.99990296
Target / Prediction : dog / dog
prob : 0.9976653
Target / Prediction : frog / frog
prob : 1.0
Target / Prediction : horse / ship
prob : 1.0
Target / Prediction : ship / ship
prob : 1.0
Target / Prediction : truck / truck
prob : 0.9995908

We can easily visualize from the image above all the adversarial examples along with the original image. In more detail, in the first row in the second image the noise on a RGB image can be distinguished in contrast to the original image. In addition, we should mention in that particular image the CIFAR pretrained classifier **predicted this image as an airplane** with **99% probability**.

Exercise 3.

Introduction

The goal of our Face Mask Detection project is to identify based on an image if an individual wears face mask (medical mask) or not. The last 2 years due to COVID-19 virus, many governments and particularly the World Health Organization suggest that all individuals should wear a face mask (medical mask) in order to protect themselves from the virus. Therefore, a classifier which is trained to classify if an individual wears a mask is very useful and has many applications nowadays, for example places which is required the face mask such as Malls or supermarkets.

Data Ingestion – Image Collection

The classifier has been trained from three different public datasets and combing all the images together we reach to 12000 unique images.

The target of our classifier it is a binary classification, therefore all the images of individuals (of the below public datasets) who have “incorrectly face mask” are neglected.

The first dataset is from Kaggle (https://www.kaggle.com/andrewmvd/face-mask-detection) and in total there are 3949 images, in which 3232 individuals have a face mask and 717 are without a facemask. In order to extract only the faces from an image we followed the xml files from the annotations directory. **In more detail, we focus only for unique faces per image and not in the whole image.** In this way we "crop" a face based on the boundaries from the xml file. A sample from the dataset is shown below.

Fig. 10 Sample from Kaggle Face Mask Detection Dataset (andrewmvd)

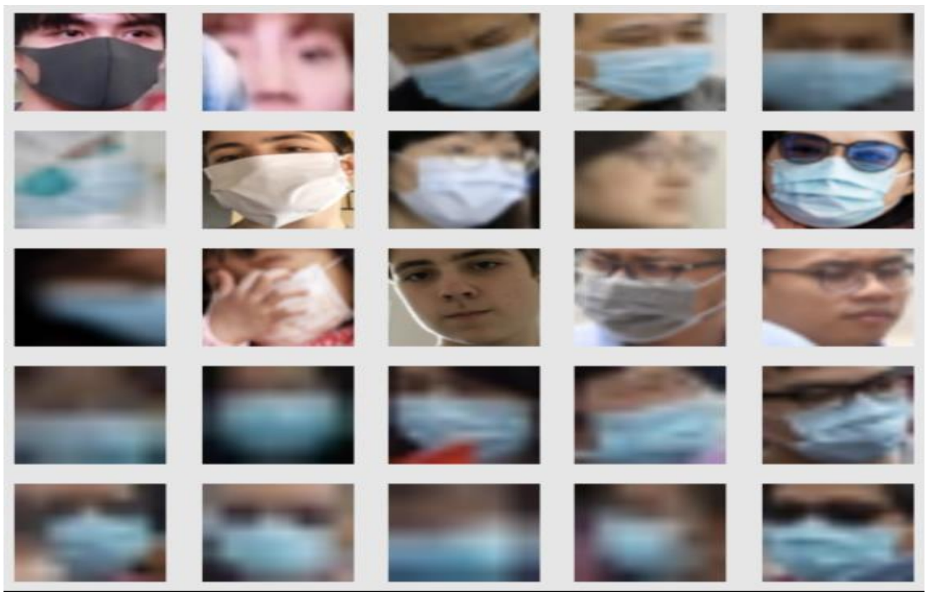
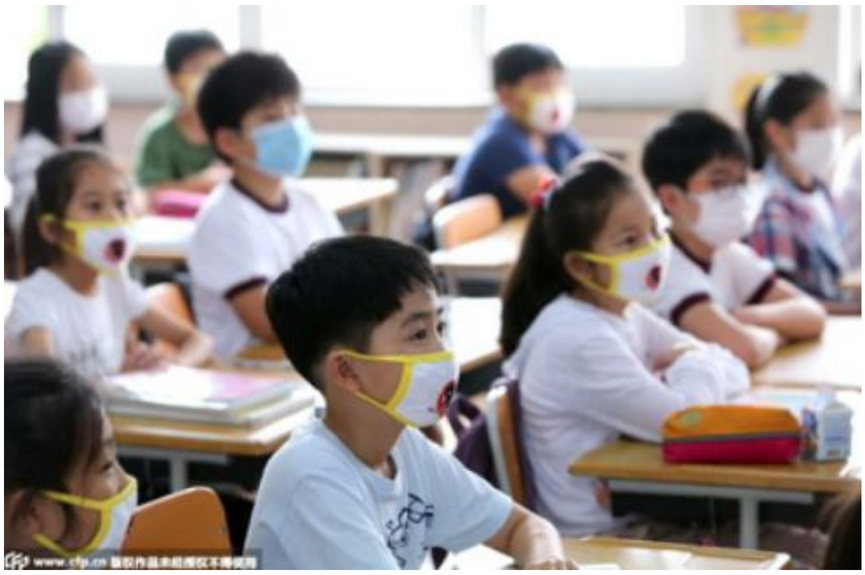
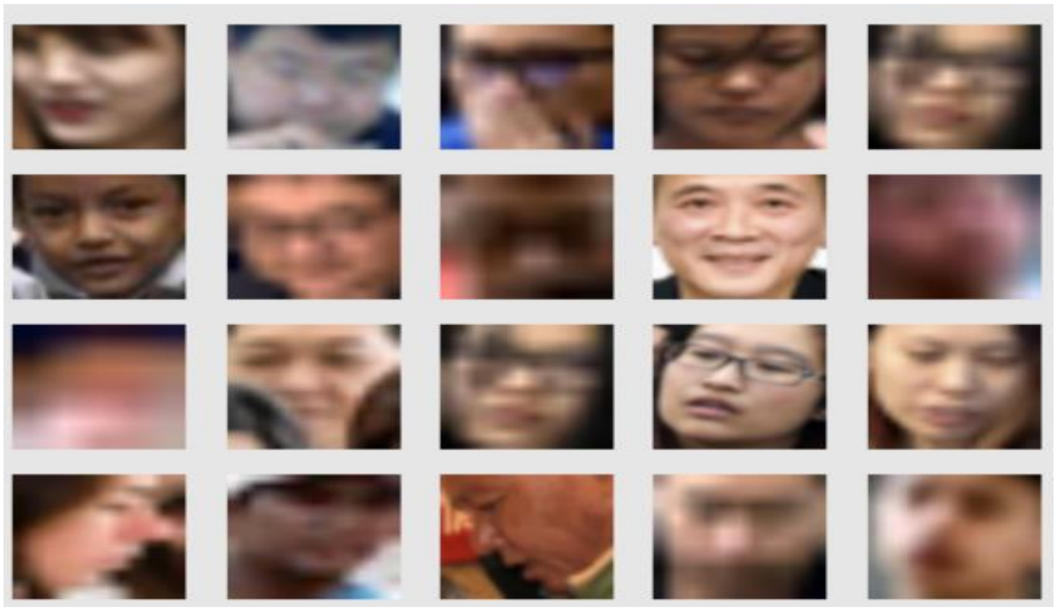


Fig. 11 An original image from Kaggle Dataset with many individuals per image



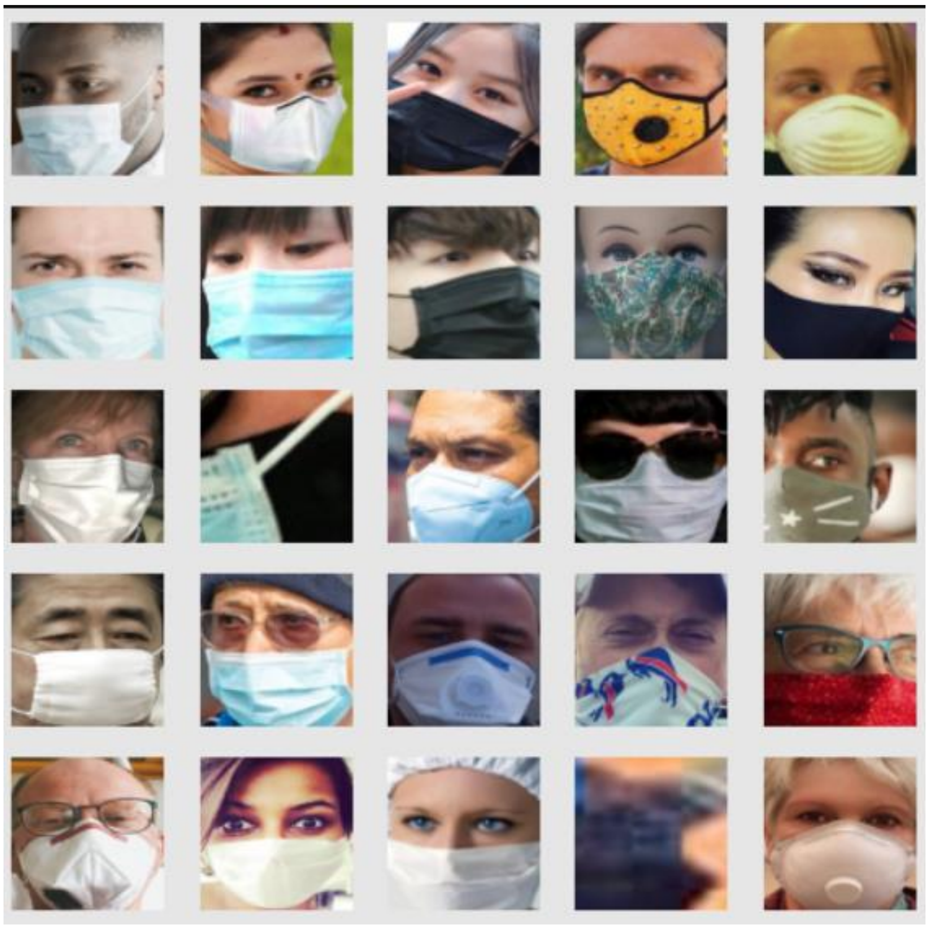
Obviously this Kaggle dataset is **imbalanced**, with a huge difference between the class of "people with face mask" and the class of "people with no face mask». In order to tackle this problem, we used **Data Augmentation to generated images only for the class "people with no face mask".** A sample from the augmented dataset is shown below.

Fig. 12 Synthetic Data based on Kaggle Face Mask Detection Dataset.



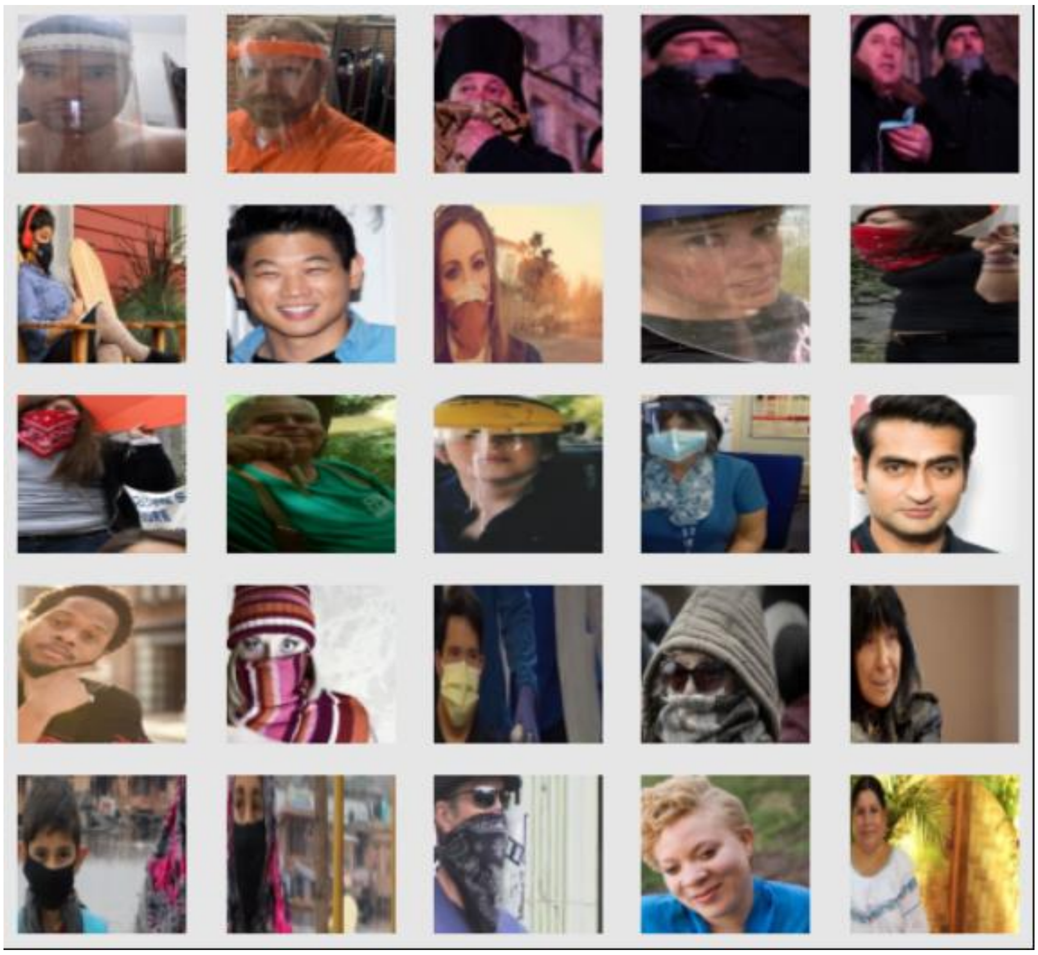
Additionally, with the first Kaggle dataset for face mask detection, **we used another dataset from Kaggle** (https://www.kaggle.com/prithwirajmitra/covid-face-mask-detection-dataset). However, in this dataset the structure of the data is different, and we have only an individual per image. **Therefore, we used a Python library, which is called MTCNN (Multi-task Cascaded Convolutional Networks), for face detection**. In this way the face detector detects the faces per image and returns the boundaries or the bounding box of a face. Having the bounding box, you can easily crop the image with respect to the dimensions. The total images from this dataset sum to 791. A sample from this dataset can be shown below.

Fig. 13 Sample from the second Kaggle Dataset (prithwirajmitra)



Finally, we added 5000 images from the humans in the loop organization (https://humansintheloop.org/resources/datasets/medical-mask-dataset). The humans in the loop is an organization which provides free dataset under request. After a short communication through email, they provided us the dataset. The images had json files per image in the annotations directory in order to extract only the faces of the individuals per image exactly as the process we mentioned before. We should also mention that the humans in the loop images in contrast to Kaggle images are very **high resolution. Therefore, our classifier was trained with both low- and high-resolution images, in order to achieve the highest performance**.

Fig.14 High resolution images from humans in the loop Dataset



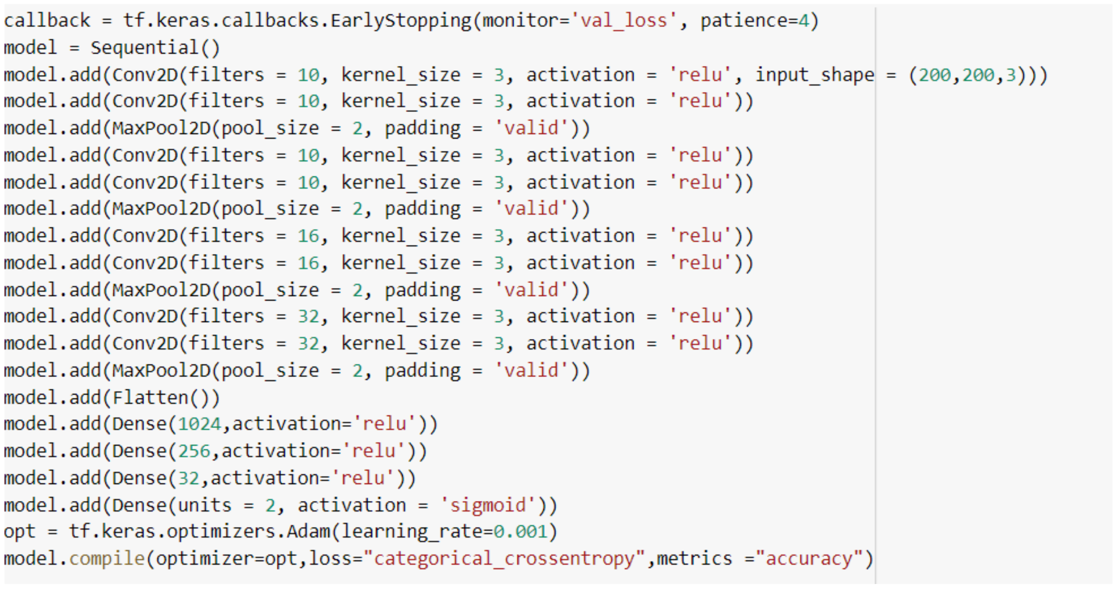
Data Transformation - Normalization

Combining all the images from the three datasets above we have a total of 12243 images. Obviously cropping only, the faces of the images will lead to have many different dimensions of the images in your dataset. In order to tackle this problem, we transform all the images to **200 x 200 pixels and of course keeping the 3 channels (RGB)**. The normalization step is important also in our pipeline, dividing each image with the 255, which is the biggest value of a certain pixel can get. We applied a custom function as train – test split keeping only 2000 images as test set and 10243 as training set.

Model Architecture – Performance

The building of the model is based on the **Keras Framework**. **The Architecture of the model is inspired from the VGG network, and it has 4 VGG blocks**. **The input shape** is of course **(200,200,3)** as mentioned above and the VGG block, consists of sequence of convolution layers followed by a maximum pool. In more detail, the first block has two convolution layers of 10 filters with 3 x 3 size and with non-linear ReLu activation function, followed by a max pooling of 2x2 size. The second block is the same as the first, the third block has 16 filters of 3x3 size (instead of 10) and the fourth block has 32 filters size 3x3. After the convolution blocks, we flatten the network and we apply 3 dense layers of 1024, 256, 32 units respectively. As final step we apply a sigmoid function. We should mention that as an optimizer we applied **Adam** (which is a combination of RMSprop and momentum) with a **learning rate of 0.001** and as a loss function, we applied categorical cross entropy. Finally, we should mention that we applied also **Early Stopping**, monitoring the validation loss, and stopping the training after no improvement in the validation loss in 4 epochs.

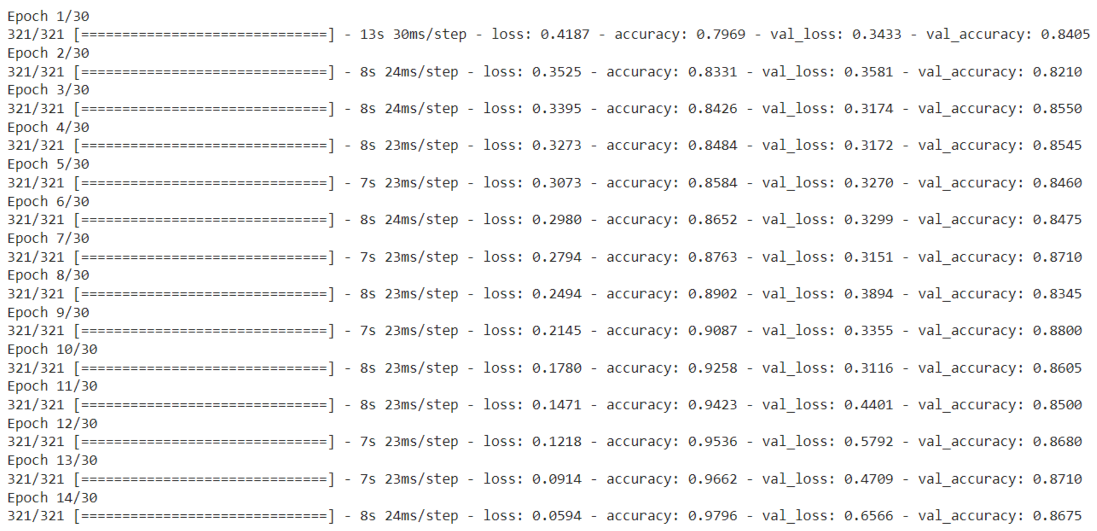
Fig. 15 The structure of the Keras Model



```python
callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=4)
model = Sequential()
model.add(Conv2D(filters = 10, kernel_size = 3, activation = 'relu', input_shape = (200,200,3)))
model.add(Conv2D(filters = 10, kernel_size = 3, activation = 'relu'))
model.add(MaxPool2D(pool_size = 2, padding = 'valid'))
model.add(Conv2D(filters = 10, kernel_size = 3, activation = 'relu'))
model.add(Conv2D(filters = 10, kernel_size = 3, activation = 'relu'))
model.add(MaxPool2D(pool_size = 2, padding = 'valid'))
model.add(Conv2D(filters = 16, kernel_size = 3, activation = 'relu'))
model.add(Conv2D(filters = 16, kernel_size = 3, activation = 'relu'))
model.add(MaxPool2D(pool_size = 2, padding = 'valid'))
model.add(Conv2D(filters = 32, kernel_size = 3, activation = 'relu'))
model.add(Conv2D(filters = 32, kernel_size = 3, activation = 'relu'))
model.add(MaxPool2D(pool_size = 2, padding = 'valid'))
model.add(Flatten())
model.add(Dense(1024,activation='relu'))
model.add(Dense(256,activation='relu'))
model.add(Dense(32,activation='relu'))
model.add(Dense(units = 2, activation = 'sigmoid'))
opt = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=opt,loss="categorical_crossentropy",metrics ="accuracy")
```

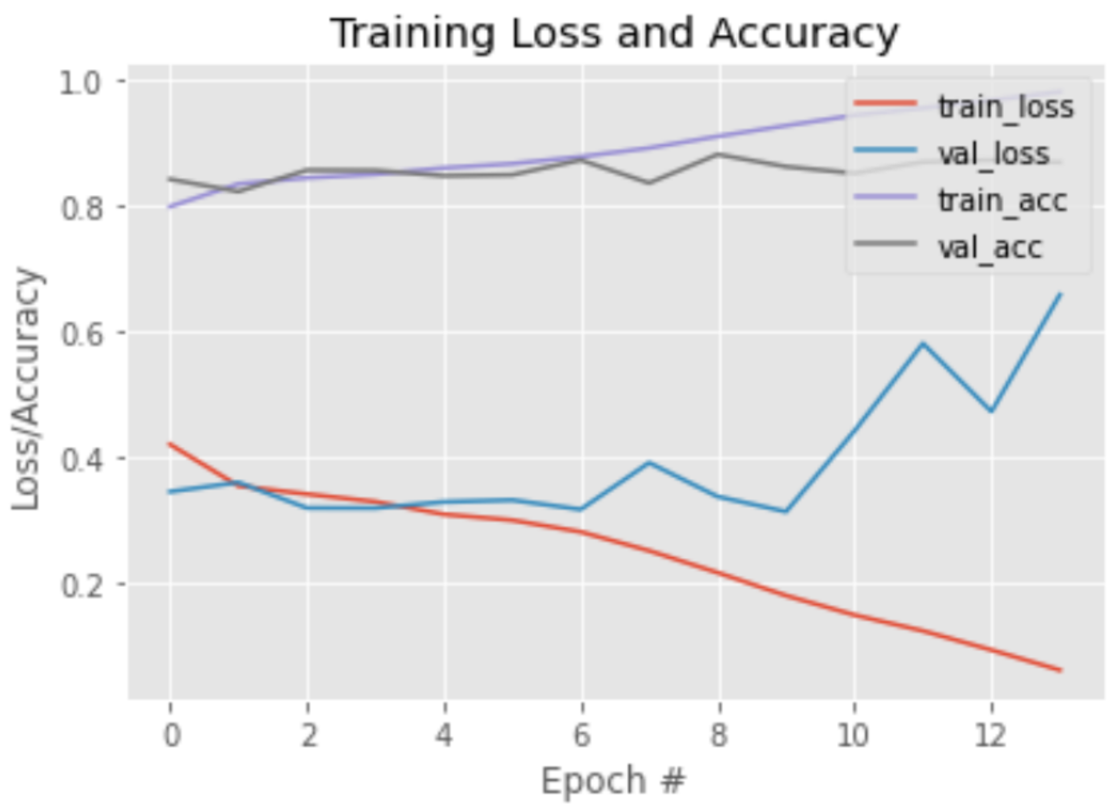
The training of the model is predefined with 30 epochs and a batch size of 32 images therefore we will have 321 updates in our weights in order to complete 1 epoch. The Accuracy in the training set starts with 79% in the 1st epoch and ends with a 97% in the 14th epoch, in which the straining stopped due to Early Stopping. The Accuracy in the testing set starts with 84% in the 1st epoch and ends with 86% in the 14th epoch. As it is shown below in the graph the training loss was reducing consecutively per epoch whereas the validation loss had fluctuations and that's why the call back of Early Stopping took place in the 14th epoch.

Fig.16 The training output of the Keras Model



```
Epoch 1/30
321/321 [==============================] - 13s 30ms/step - loss: 0.4187 - accuracy: 0.7969 - val_loss: 0.3433 - val_accuracy: 0.8405
Epoch 2/30
321/321 [==============================] - 8s 24ms/step - loss: 0.3525 - accuracy: 0.8331 - val_loss: 0.3581 - val_accuracy: 0.8210
Epoch 3/30
321/321 [==============================] - 8s 24ms/step - loss: 0.3395 - accuracy: 0.8426 - val_loss: 0.3174 - val_accuracy: 0.8550
Epoch 4/30
321/321 [==============================] - 8s 23ms/step - loss: 0.3273 - accuracy: 0.8484 - val_loss: 0.3172 - val_accuracy: 0.8545
Epoch 5/30
321/321 [==============================] - 7s 23ms/step - loss: 0.3073 - accuracy: 0.8584 - val_loss: 0.3270 - val_accuracy: 0.8460
Epoch 6/30
321/321 [==============================] - 8s 24ms/step - loss: 0.2980 - accuracy: 0.8652 - val_loss: 0.3299 - val_accuracy: 0.8475
Epoch 7/30
321/321 [==============================] - 7s 23ms/step - loss: 0.2794 - accuracy: 0.8763 - val_loss: 0.3151 - val_accuracy: 0.8710
Epoch 8/30
321/321 [==============================] - 8s 23ms/step - loss: 0.2494 - accuracy: 0.8902 - val_loss: 0.3894 - val_accuracy: 0.8345
Epoch 9/30
321/321 [==============================] - 7s 23ms/step - loss: 0.2145 - accuracy: 0.9087 - val_loss: 0.3355 - val_accuracy: 0.8800
Epoch 10/30
321/321 [==============================] - 8s 23ms/step - loss: 0.1780 - accuracy: 0.9258 - val_loss: 0.3116 - val_accuracy: 0.8605
Epoch 11/30
321/321 [==============================] - 8s 23ms/step - loss: 0.1471 - accuracy: 0.9423 - val_loss: 0.4401 - val_accuracy: 0.8500
Epoch 12/30
321/321 [==============================] - 7s 23ms/step - loss: 0.1218 - accuracy: 0.9536 - val_loss: 0.5792 - val_accuracy: 0.8680
Epoch 13/30
321/321 [==============================] - 7s 23ms/step - loss: 0.0914 - accuracy: 0.9662 - val_loss: 0.4709 - val_accuracy: 0.8710
Epoch 14/30
321/321 [==============================] - 8s 24ms/step - loss: 0.0594 - accuracy: 0.9796 - val_loss: 0.6566 - val_accuracy: 0.8675
```

Fig.17 A graph of the model Performance



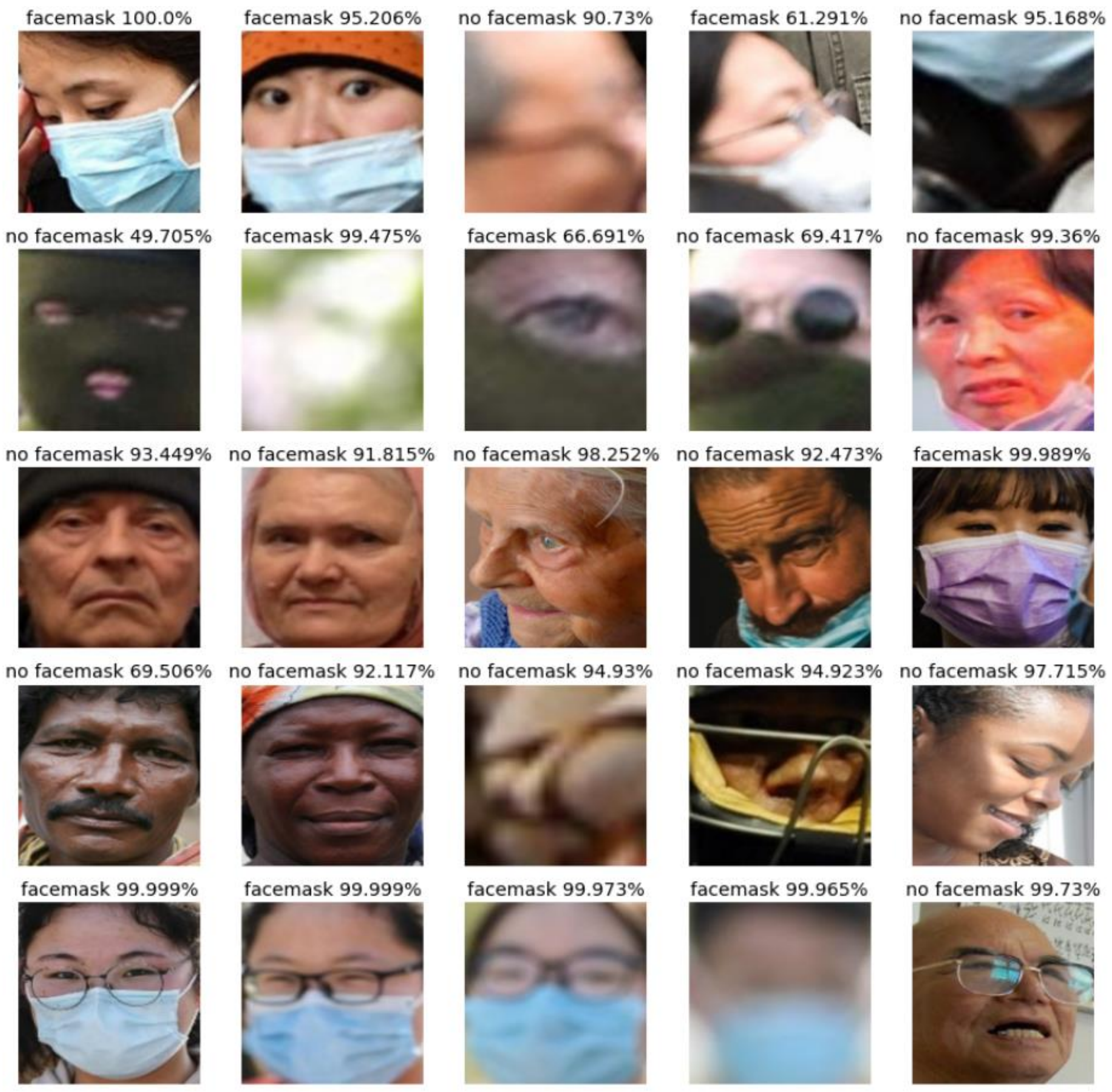
Important*

We should mention that only the training of the model was successfully finished in 5 minutes and with a Python 3 Google Compute Engine with **GPU** and **25.46 GB RAM** provided **by Google Colab PRO**. The conventional Google Engine of Colab **run out of memory** processing NumPy arrays of 12000 images.

Model Performance in Unknown Images / Score Dataset

As a Score Dataset, in other words a dataset which the model **has never “seen” before**, we kept **1000** images from the human in the loop dataset. We evaluated a sample of 25 of those images into our trained network and we kept the predictions and the probabilities in order to display them afterwards as titles for the respective image. The figures below show the predictions of the model with respect to the score dataset.

Fig.18 Plotting 25 score images along with the predictions & probabilities



Adding a powerful Visualization

In order to “expose” our model to images that have never seen before we downloaded and added **certain images from the Google**. We also added a **visualization** in order to keep the **original image**, the face detected images and the final predicted image as well. The performance of the model in the Google images was more than satisfying.

Fig.19 The visualization, focusing on the highlighted rectangle face images along with the predictions & probabilities

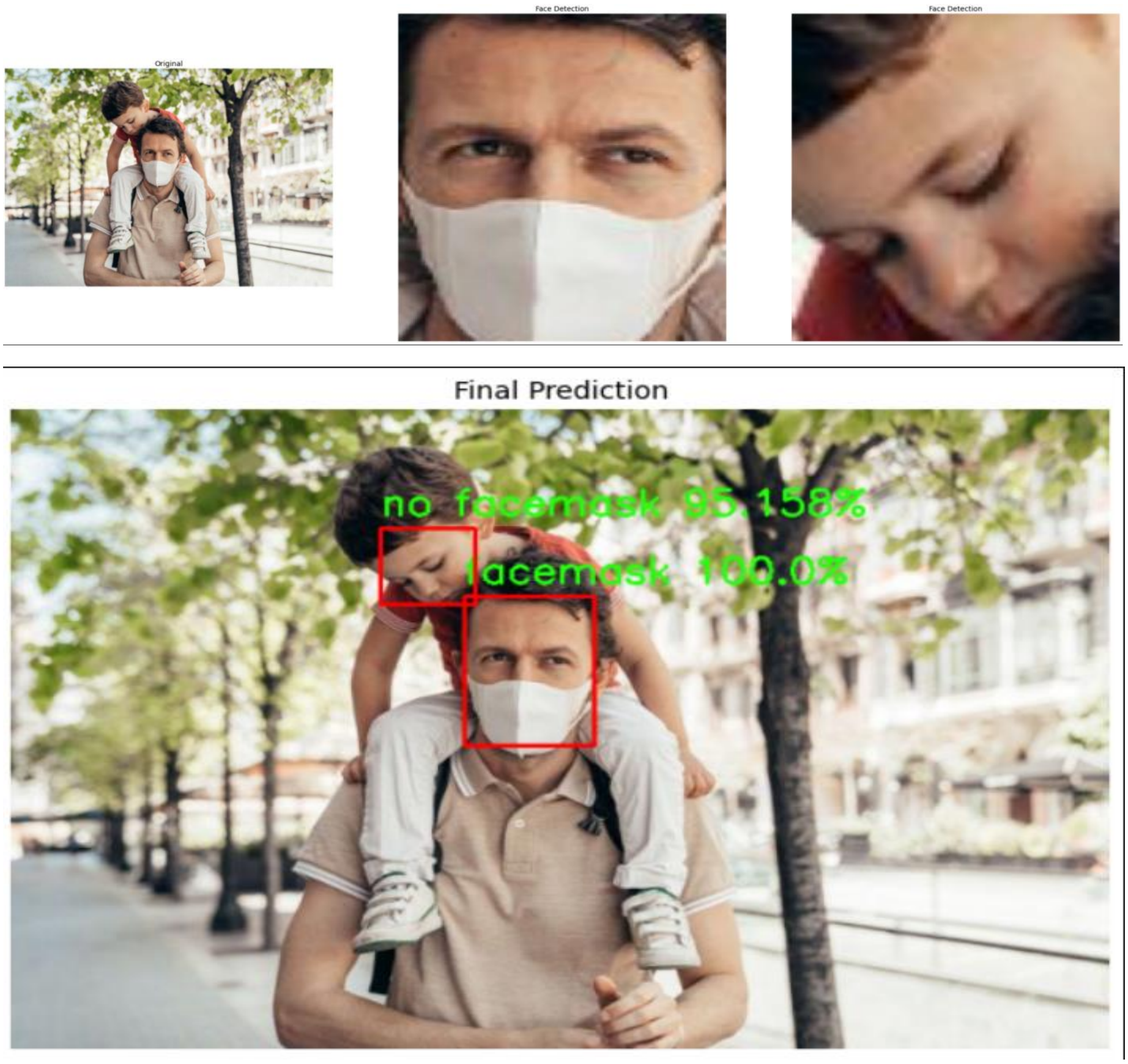
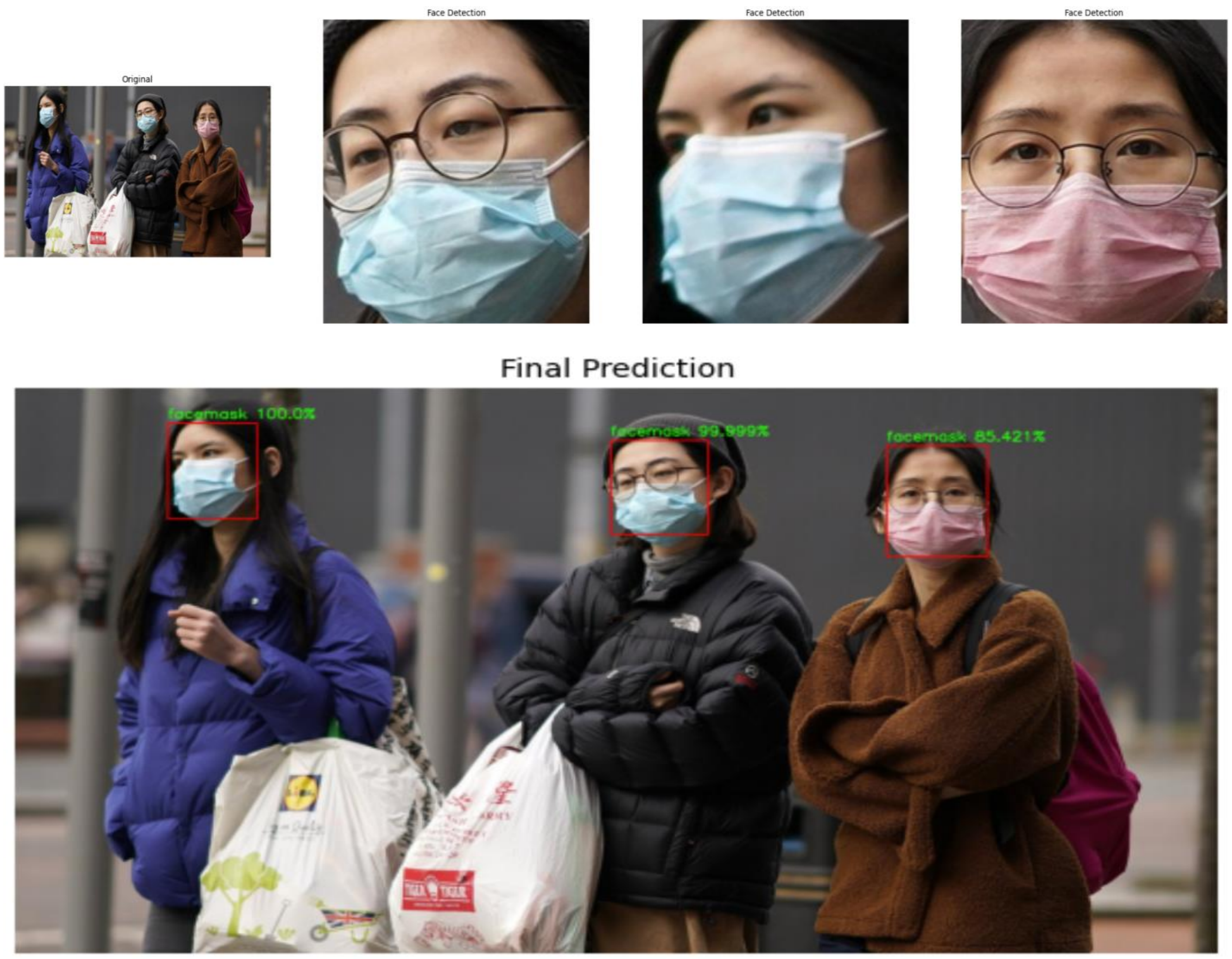


Fig. 20 The visualization, focusing on the highlighted rectangle face images along with the predictions & probabilities



Retrain the Model

The last step in our pipeline is to **retrain** the model. We have to take **advantage** the **2000 test images that were in memory and the model has not trained with them**. Therefore, we applied a final training of the model only for those 2000 images with a predefined epoch of 5 and a batch size of 32 images. Finally, we saved the last Keras model as "final_face_detection.h5" in h5 format.

Guide for supplementary files

- **Source code:**
  1) ML_Exercise_1.ipynb (python notebook)
  2) ML_Exercise_2.ipynb (python notebook)
  3) ML_Exercise_3.ipynb (python notebook)
  4) CIFAR.ipynb (python notebook)

- **Requirements:**
  The txt file requirements.txt with all the dependencies.

- **Model:**

  The file “final_face_detection_model.h5” which is the saved keras model in h5 format.