Code Solution To Authenticate Into An Application Using Deep Facial Recognition

Technical Report · July 2022

DOI: 10.13140/RG.2.2.13239.60327

1 author:

Chethine Liyanarachchi
Sri Lanka Institute of Information Technology
5 PUBLICATIONS 0 CITATIONS
SEE PROFILE

Code Solution To Authenticate Into An Application Using Deep Facial Recognition

Helle Liyanarachchige Chethine Lakvindu Liyanarachchi
Faculty of Computing
Sri Lanka Institute of Information Technology
Colombo, Sri Lanka
it20250256@my.sliit.lk

***Abstract*— The vast majority of people who use the internet have at least a basic comprehension of the concepts of online privacy and application security. The majority of the time, apps will employ some sort of authentication method in order to verify the identities of their users. Methods such as faceID, biometric authentication, passwords, and tokens are examples. Facial recognition technology is one of them that is gaining more and more attention. The most fundamental aspect of facial recognition is the process of comparing a human face captured in an electronic photograph or a video frame to a database containing other people's faces. It is well known that facial recognition is one of the most secure and reliable techniques of authentication that can be found everywhere around the globe.**

***Keywords*—TensorFlow, Deep learning, faceID, OpenCV, Biometrics**

I. Introduction

The primary purpose of this paper is to discuss the implementation of a deep face recognition application, which is something that can be used to authenticate users when users sign into an application. Python is the primary programming language utilized in the development of this facial recognition application. Additionally, the development approach included the utilization of transflow and kivy.

Applications that use face recognition are susceptible to a wide variety of security flaws, including the theft of the numerical code and unconscious facial recognition. Over the course of the past few years, several of these vulnerabilities have been exploited. Some of the vulnerabilities that have been exploited include CVE-2015-7897, CVE-2014-2983, and CVE-2019-0657. The implementation of this code solution will assist in removing certain vulnerabilities.

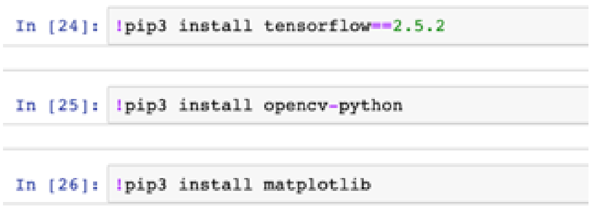
The deep facial recognition application is implemented in seven steps, as shown below.

1. Setup dependencies and structures
2. Collect image samples
3. Fill and preprocess collected images
4. Model Engineering
5. Custome data training using loops
6. Evaluate model
7. Real-time verification with OpenCV integration

II. Methodology

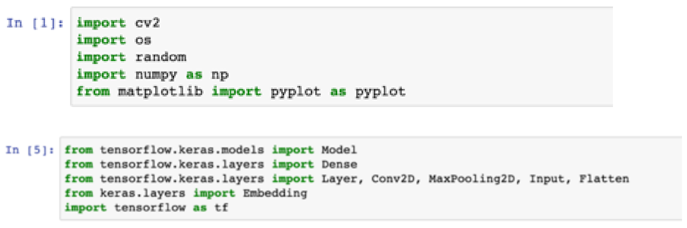
I. Setup dependencies and structures

The main objectives are to install dependencies, import transflow layers, set up GPU growth, and build data folders in this stage. The codes below are used to install dependencies such as transflow, openCV and matplotlib.



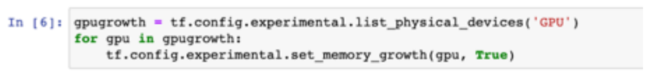
```
In [24]: !pip3 install tensorflow==2.5.2

In [25]: !pip3 install opencv-python

In [26]: !pip3 install matplotlib
```

The next step is to import dependencies including tensorflow dependencies, which is done using the code segments below.



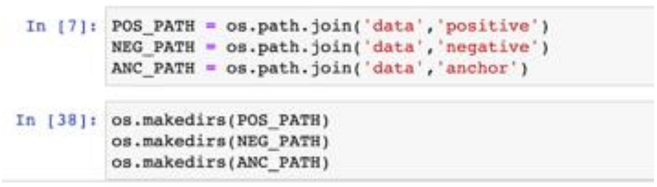
```
In [1]: import cv2
        import os
        import random
        import numpy as np
        from matplotlib import pyplot as pyplot

In [5]: from tensorflow.keras.models import Model
        from tensorflow.keras.layers import Dense
        from tensorflow.keras.layers import Layer, Conv2D, MaxPooling2D, Input, Flatten
        from keras.layers import Embedding
        import tensorflow as tf
```

The following code segment is used to manage GPU memory consumption, which can result in an out of memory error.



```
In [6]: gpugrowth = tf.config.experimental.list_physical_devices('GPU')
        for gpu in gpugrowth:
            tf.config.experimental.set_memory_growth(gpu, True)
```

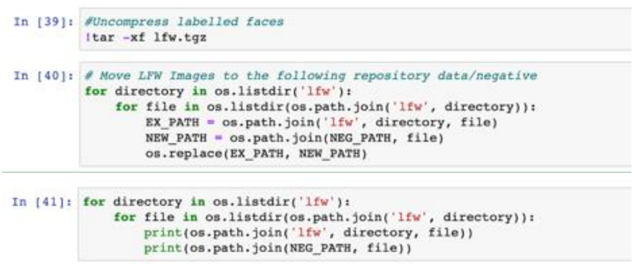
Then, for identifying purposes, folder structures must be created to contain positive and negative photos.



```
In [7]: POS_PATH = os.path.join('data','positive')
        NEG_PATH = os.path.join('data','negative')
        ANC_PATH = os.path.join('data','anchor')

In [38]: os.makedirs(POS_PATH)
         os.makedirs(NEG_PATH)
         os.makedirs(ANC_PATH)
```

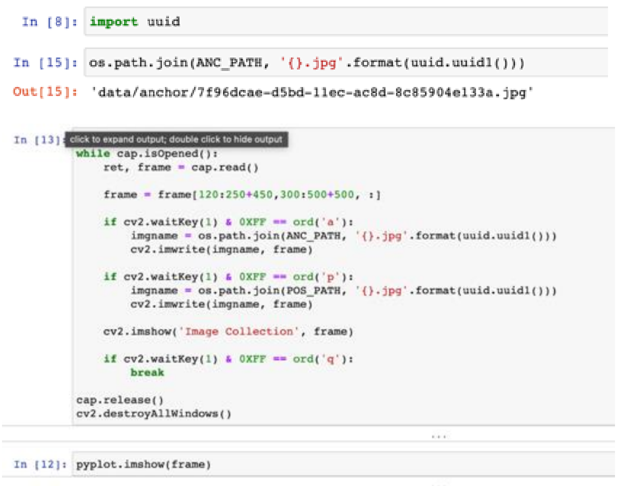
II. Collect image samples

During this stage of the process, It will be tasked for gathering positive and anchor images for use in the verification process. The database at the University of Massachusetts contains tagged faces, which can be utilized for the purpose of implementation. Once finished downloading those images, those images were moved in to negative images.



```
In [39]: #Uncompress labelled faces
         !tar -xf lfw.tgz

In [40]: # Move LFW Images to the following repository data/negative
         for directory in os.listdir('lfw'):
             for file in os.listdir(os.path.join('lfw', directory)):
                 EX_PATH = os.path.join('lfw', directory, file)
                 NEW_PATH = os.path.join(NEG_PATH, file)
                 os.replace(EX_PATH, NEW_PATH)

In [41]: for directory in os.listdir('lfw'):
             for file in os.listdir(os.path.join('lfw', directory)):
                 print(os.path.join('lfw', directory, file))
                 print(os.path.join(NEG_PATH, file))
```

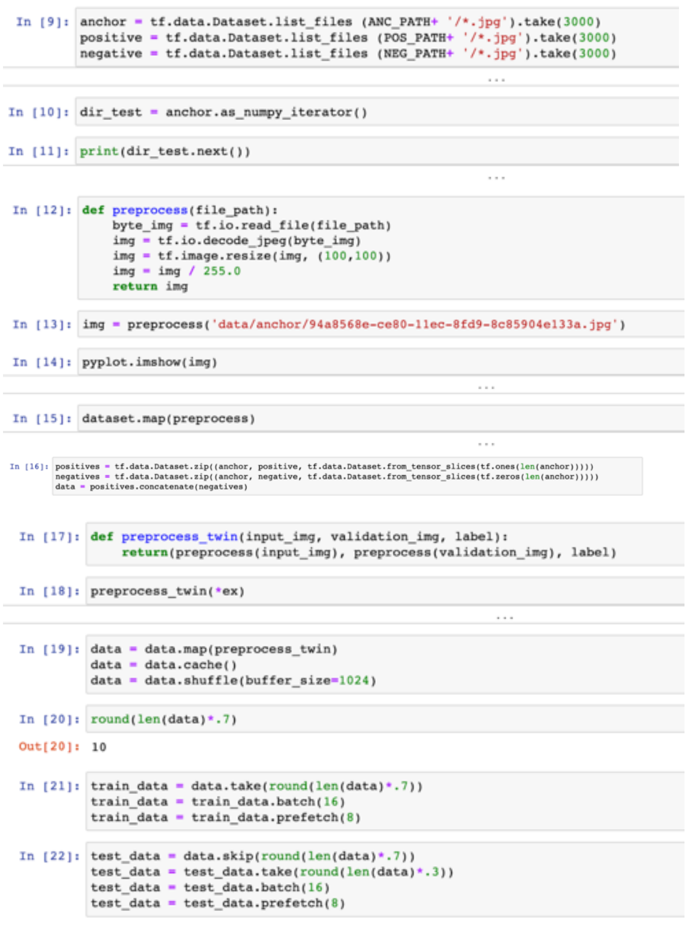
After having gathered negative faces, one must next move on to collecting positive faces. Importing the uuid library is the first step in accomplishing that goal. Establishing a connection to the webcam is required before any positive images collected. After then, one may gather anchor images by pressing the letter "a," and one can collect positive photos by hitting the letter "p."



```
In [8]: import uuid

In [15]: os.path.join(ANC_PATH, '{}.jpg'.format(uuid.uuid1()))
Out[15]: 'data/anchor/7f96dcae-d5bd-11ec-ac8d-8c85904e133a.jpg'

In [13]: click to expand output; double click to hide output
         while cap.isOpened():
             ret, frame = cap.read()

             frame = frame[120:250+450,300:500+500, :]

             if cv2.waitKey(1) & 0XFF == ord('a'):
                 imgname = os.path.join(ANC_PATH, '{}.jpg'.format(uuid.uuid1()))
                 cv2.imwrite(imgname, frame)

             if cv2.waitKey(1) & 0XFF == ord('p'):
                 imgname = os.path.join(POS_PATH, '{}.jpg'.format(uuid.uuid1()))
                 cv2.imwrite(imgname, frame)

             cv2.imshow('Image Collection', frame)

             if cv2.waitKey(1) & 0XFF == ord('q'):
                 break

         cap.release()
         cv2.destroyAllWindows()
                                                        ...

In [12]: pyplot.imshow(frame)
                                                        ...
```

III. Fill and preprocess collected images

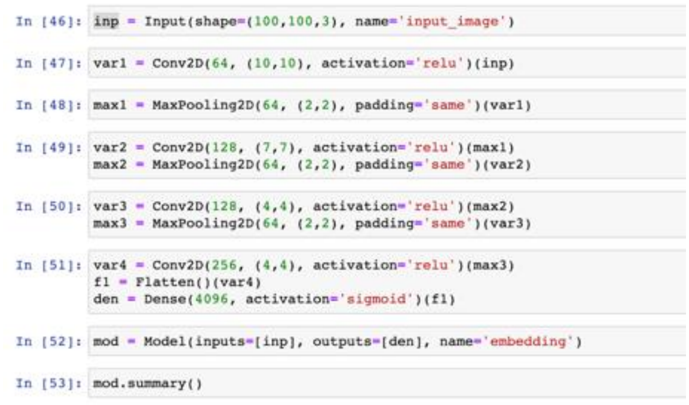
The loading of images into the TensorFlow data loader is the primary responsibility of this stage. In addition, the preprocessing of images, as well as the installation of both positive and negative data samples, will take place during this step. The code segment below is the one that should be utilized for that.



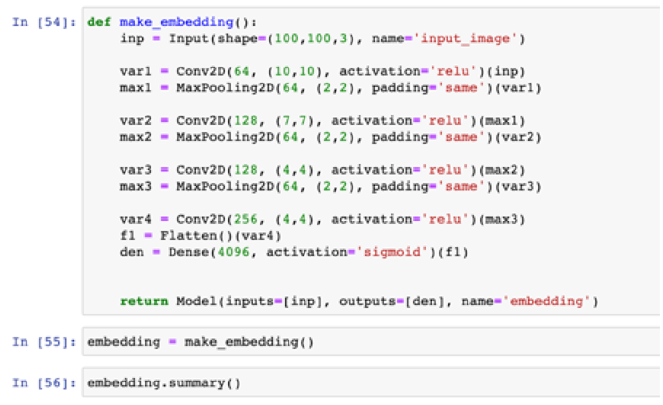
```
In [9]: anchor = tf.data.Dataset.list_files (ANC_PATH+ '/*.jpg').take(3000)
        positive = tf.data.Dataset.list_files (POS_PATH+ '/*.jpg').take(3000)
        negative = tf.data.Dataset.list_files (NEG_PATH+ '/*.jpg').take(3000)
                                                        ...

In [10]: dir_test = anchor.as_numpy_iterator()

In [11]: print(dir_test.next())
                                                        ...

In [12]: def preprocess(file_path):
             byte_img = tf.io.read_file(file_path)
             img = tf.io.decode_jpeg(byte_img)
             img = tf.image.resize(img, (100,100))
             img = img / 255.0
             return img

In [13]: img = preprocess('data/anchor/94a8568e-ce80-11ec-8fd9-8c85904e133a.jpg')

In [14]: pyplot.imshow(img)
                                                        ...

In [15]: dataset.map(preprocess)
                                                        ...

In [16]: positives = tf.data.Dataset.zip((anchor, positive, tf.data.Dataset.from_tensor_slices(tf.ones(len(anchor)))))
         negatives = tf.data.Dataset.zip((anchor, negative, tf.data.Dataset.from_tensor_slices(tf.zeros(len(anchor)))))
         data = positives.concatenate(negatives)

In [17]: def preprocess_twin(input_img, validation_img, label):
             return(preprocess(input_img), preprocess(validation_img), label)

In [18]: preprocess_twin(*ex)
                                                        ...

In [19]: data = data.map(preprocess_twin)
         data = data.cache()
         data = data.shuffle(buffer_size=1024)

In [20]: round(len(data)*.7)
Out[20]: 10

In [21]: train_data = data.take(round(len(data)*.7))
         train_data = train_data.batch(16)
         train_data = train_data.prefetch(8)

In [22]: test_data = data.skip(round(len(data)*.7))
         test_data = test_data.take(round(len(data)*.3))
         test_data = test_data.batch(16)
         test_data = test_data.prefetch(8)
```

IV. Model Engineering

During this stage, the implementation of the embedding layer, followed by the construction of the distance layer and the compilation of the siamese network, will take place. The following line of code was used to implement the embedding layer.

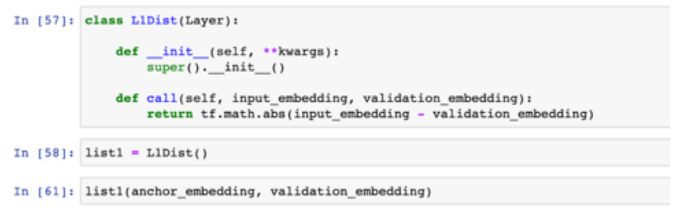


```
In [46]: inp = Input(shape=(100,100,3), name='input_image')

In [47]: var1 = Conv2D(64, (10,10), activation='relu')(inp)

In [48]: max1 = MaxPooling2D(64, (2,2), padding='same')(var1)

In [49]: var2 = Conv2D(128, (7,7), activation='relu')(max1)
         max2 = MaxPooling2D(64, (2,2), padding='same')(var2)

In [50]: var3 = Conv2D(128, (4,4), activation='relu')(max2)
         max3 = MaxPooling2D(64, (2,2), padding='same')(var3)

In [51]: var4 = Conv2D(256, (4,4), activation='relu')(max3)
         f1 = Flatten()(var4)
         den = Dense(4096, activation='sigmoid')(f1)

In [52]: mod = Model(inputs=[inp], outputs=[den], name='embedding')

In [53]: mod.summary()
```



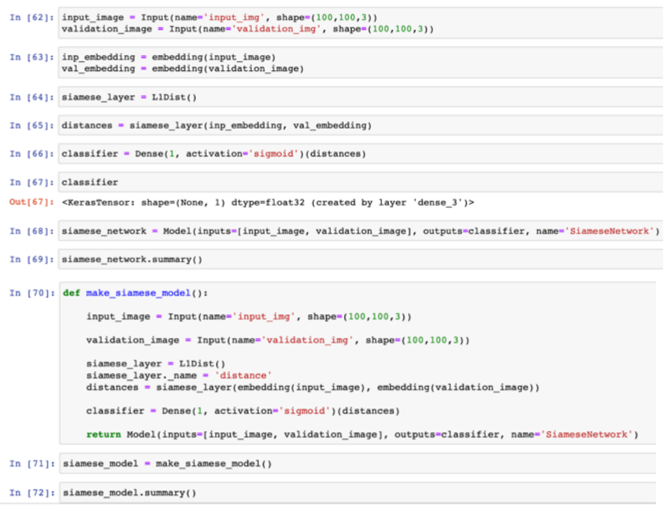
```
In [54]: def make_embedding():
             inp = Input(shape=(100,100,3), name='input_image')

             var1 = Conv2D(64, (10,10), activation='relu')(inp)
             max1 = MaxPooling2D(64, (2,2), padding='same')(var1)

             var2 = Conv2D(128, (7,7), activation='relu')(max1)
             max2 = MaxPooling2D(64, (2,2), padding='same')(var2)

             var3 = Conv2D(128, (4,4), activation='relu')(max2)
             max3 = MaxPooling2D(64, (2,2), padding='same')(var3)

             var4 = Conv2D(256, (4,4), activation='relu')(max3)
             f1 = Flatten()(var4)
             den = Dense(4096, activation='sigmoid')(f1)


             return Model(inputs=[inp], outputs=[den], name='embedding')

In [55]: embedding = make_embedding()

In [56]: embedding.summary()
```

The implementation of the distance layer is the next task. The line of code below was utilized for that purpose.



```
In [57]: class L1Dist(Layer):

             def __init__(self, **kwargs):
                 super().__init__()

             def call(self, input_embedding, validation_embedding):
                 return tf.math.abs(input_embedding - validation_embedding)

In [58]: list1 = L1Dist()

In [61]: list1(anchor_embedding, validation_embedding)
```

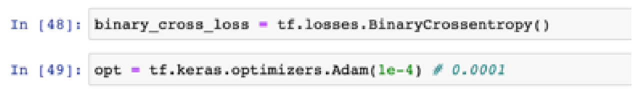
The next step in creating a siamese neural network is to combine the embedding layer with the distance layer.



```
In [62]: input_image = Input(name='input_img', shape=(100,100,3))
         validation_image = Input(name='validation_img', shape=(100,100,3))

In [63]: inp_embedding = embedding(input_image)
         val_embedding = embedding(validation_image)

In [64]: siamese_layer = L1Dist()

In [65]: distances = siamese_layer(inp_embedding, val_embedding)

In [66]: classifier = Dense(1, activation='sigmoid')(distances)

In [67]: classifier

Out[67]: <KerasTensor: shape=(None, 1) dtype=float32 (created by layer 'dense_3')>

In [68]: siamese_network = Model(inputs=[input_image, validation_image], outputs=classifier, name='SiameseNetwork')

In [69]: siamese_network.summary()

In [70]: def make_siamese_model():

             input_image = Input(name='input_img', shape=(100,100,3))

             validation_image = Input(name='validation_img', shape=(100,100,3))

             siamese_layer = L1Dist()
             siamese_layer._name = 'distance'
             distances = siamese_layer(embedding(input_image), embedding(validation_image))

             classifier = Dense(1, activation='sigmoid')(distances)

             return Model(inputs=[input_image, validation_image], outputs=classifier, name='SiameseNetwork')

In [71]: siamese_model = make_siamese_model()

In [72]: siamese_model.summary()
```

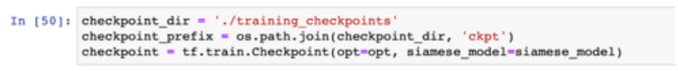
V. Custome data training using loops

The phases of data training can be further subdivided into four more steps, which are the establishment of checkpoints, the setting up of a loss and optimizer, the implementation of the train setup function, and the evaluation of the training loop. The code segmant that may be found below is used to set up loss and optimizer.



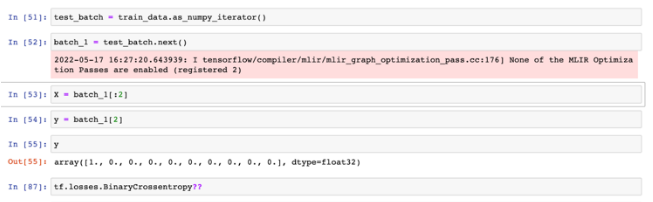
```
In [48]: binary_cross_loss = tf.losses.BinaryCrossentropy()

In [49]: opt = tf.keras.optimizers.Adam(1e-4) # 0.0001
```

The establishment of checkpoints for the purpose of training is the second substep in the process.

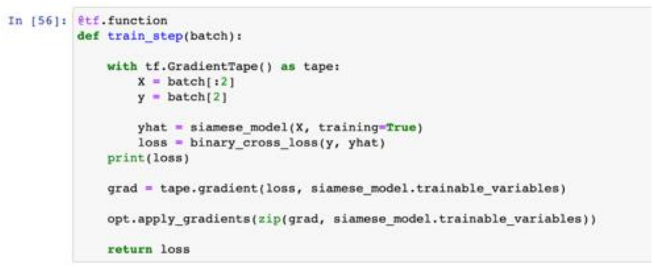


```
In [50]: checkpoint_dir = './training_checkpoints'
         checkpoint_prefix = os.path.join(checkpoint_dir, 'ckpt')
         checkpoint = tf.train.Checkpoint(opt=opt, siamese_model=siamese_model)
```

The next step is to put in place the function for setting up the train. At this stage, the implementation of the fundamental flow for training on one batch of data takes place.

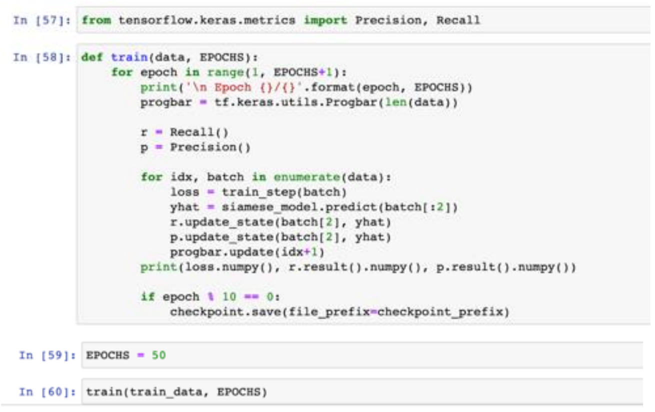


```
In [51]: test_batch = train_data.as_numpy_iterator()

In [52]: batch_1 = test_batch.next()

         2022-05-17 16:27:20.643939: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the MLIR Optimiza
         tion Passes are enabled (registered 2)

In [53]: X = batch_1[:2]

In [54]: y = batch_1[2]

In [55]: y

Out[55]: array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)

In [87]: tf.losses.BinaryCrossentropy??
```



```
In [56]: @tf.function
         def train_step(batch):

             with tf.GradientTape() as tape:
                 X = batch[:2]
                 y = batch[2]

                 yhat = siamese_model(X, training=True)
                 loss = binary_cross_loss(y, yhat)
             print(loss)

             grad = tape.gradient(loss, siamese_model.trainable_variables)

             opt.apply_gradients(zip(grad, siamese_model.trainable_variables))

             return loss
```

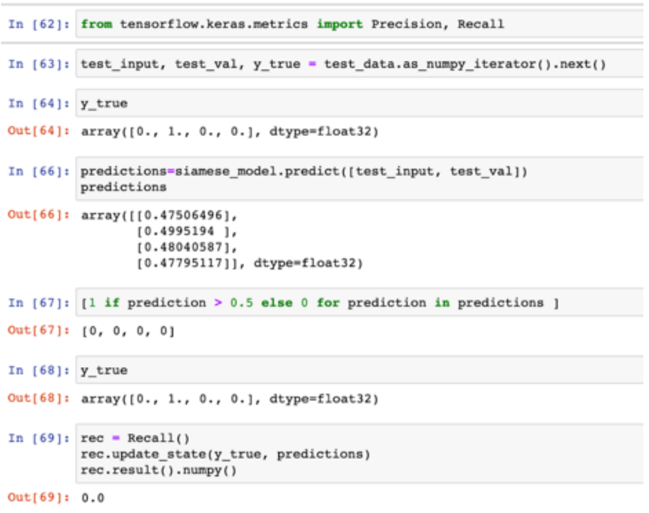
After that, training loops are put into place for the purpose of data training.



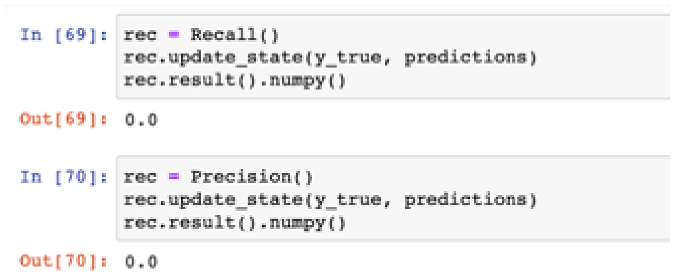
```
In [57]: from tensorflow.keras.metrics import Precision, Recall

In [58]: def train(data, EPOCHS):
             for epoch in range(1, EPOCHS+1):
                 print('\n Epoch {}/{}'.format(epoch, EPOCHS))
                 progbar = tf.keras.utils.Progbar(len(data))

                 r = Recall()
                 p = Precision()

                 for idx, batch in enumerate(data):
                     loss = train_step(batch)
                     yhat = siamese_model.predict(batch[:2])
                     r.update_state(batch[2], yhat)
                     p.update_state(batch[2], yhat)
                     progbar.update(idx+1)
                 print(loss.numpy(), r.result().numpy(), p.result().numpy())

                 if epoch % 10 == 0:
                     checkpoint.save(file_prefix=checkpoint_prefix)

In [59]: EPOCHS = 50

In [60]: train(train_data, EPOCHS)
```

VI. Evaluate model

At this stage, the primary responsibility is to test the model on various photos. The primary goals of this step are to test the model, assess its performance, and save the model in preparation for deployment. During the testing phase of the model, metrics are imported, and then metrics are calculated along with the predictions.

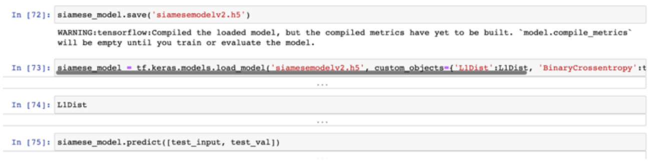


```
In [62]: from tensorflow.keras.metrics import Precision, Recall

In [63]: test_input, test_val, y_true = test_data.as_numpy_iterator().next()

In [64]: y_true

Out[64]: array([0., 1., 0., 0.], dtype=float32)

In [66]: predictions=siamese_model.predict([test_input, test_val])
         predictions

Out[66]: array([[0.47506496],
                [0.4995194 ],
                [0.48040587],
                [0.47795117]], dtype=float32)

In [67]: [1 if prediction > 0.5 else 0 for prediction in predictions ]

Out[67]: [0, 0, 0, 0]

In [68]: y_true

Out[68]: array([0., 1., 0., 0.], dtype=float32)

In [69]: rec = Recall()
         rec.update_state(y_true, predictions)
         rec.result().numpy()

Out[69]: 0.0
```

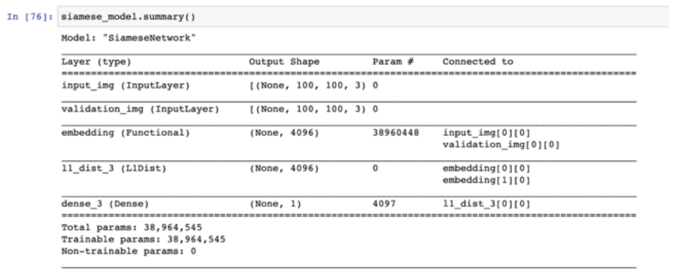


```
In [69]: rec = Recall()
         rec.update_state(y_true, predictions)
         rec.result().numpy()

Out[69]: 0.0

In [70]: rec = Precision()
         rec.update_state(y_true, predictions)
         rec.result().numpy()

Out[70]: 0.0
```

The next code segment is utilized after that in order to save the model prior to its deployment.



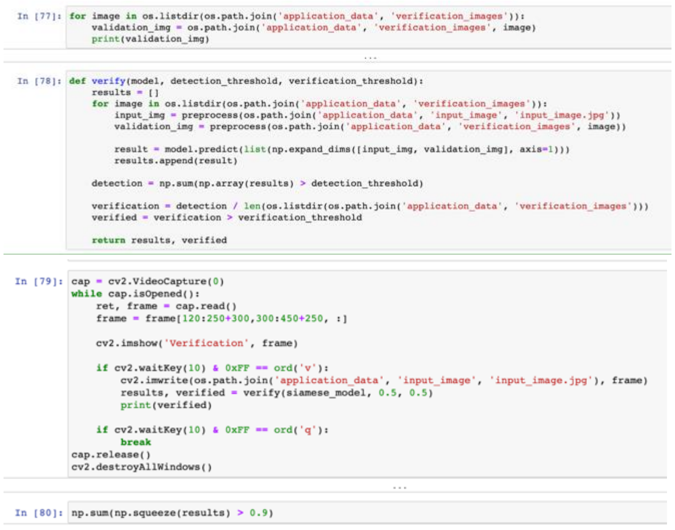
```
In [72]: siamese_model.save('siamesemodelv2.h5')

         WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile_metrics`
         will be empty until you train or evaluate the model.

In [73]: siamese_model = tf.keras.models.load_model('siamesemodelv2.h5', custom_objects={'L1Dist':L1Dist, 'BinaryCrossentropy':t
                                                   ...

In [74]: L1Dist
                                                   ...

In [75]: siamese_model.predict([test_input, test_val])
```



```
In [76]: siamese_model.summary()

         Model: "SiameseNetwork"
         __________________________________________________________________________________________________
         Layer (type)                    Output Shape         Param #     Connected to
         ==================================================================================================
         input_img (InputLayer)          [(None, 100, 100, 3) 0
         __________________________________________________________________________________________________
         validation_img (InputLayer)     [(None, 100, 100, 3) 0
         __________________________________________________________________________________________________
         embedding (Functional)          (None, 4096)         38960448    input_img[0][0]
                                                                          validation_img[0][0]
         __________________________________________________________________________________________________
         l1_dist_3 (L1Dist)              (None, 4096)         0           embedding[0][0]
                                                                          embedding[1][0]
         __________________________________________________________________________________________________
         dense_3 (Dense)                 (None, 1)            4097        l1_dist_3[0][0]
         ==================================================================================================
         Total params: 38,964,545
         Trainable params: 38,964,545
         Non-trainable params: 0
         __________________________________________________________________________________________________
```

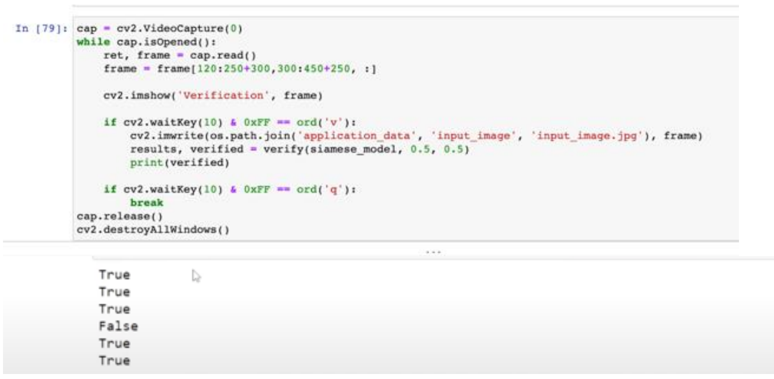
VII. **Real-time verification with OpenCV integration**

The real-time verification process may be further broken down into three substeps, which are the following: setting up the verification images; putting the verification function into action; and doing the verification in real time. Opencv is what is utilized in order to connect to and open the device's webcam. After that, an input image is acquired by using the webcam. Comparisons between that input image and samples taken from the positive directory are made so that verification can be performed. The result of the verification is then displayed on the screen.



```
In [77]: for image in os.listdir(os.path.join('application_data', 'verification_images')):
             validation_img = os.path.join('application_data', 'verification_images', image)
             print(validation_img)
                                                   ...

In [78]: def verify(model, detection_threshold, verification_threshold):
             results = []
             for image in os.listdir(os.path.join('application_data', 'verification_images')):
                 input_img = preprocess(os.path.join('application_data', 'input_image', 'input_image.jpg'))
                 validation_img = preprocess(os.path.join('application_data', 'verification_images', image))

                 result = model.predict(list(np.expand_dims([input_img, validation_img], axis=1)))
                 results.append(result)

             detection = np.sum(np.array(results) > detection_threshold)

             verification = detection / len(os.listdir(os.path.join('application_data', 'verification_images')))
             verified = verification > verification_threshold

             return results, verified

In [79]: cap = cv2.VideoCapture(0)
         while cap.isOpened():
             ret, frame = cap.read()
             frame = frame[120:250+300,300:450+250, :]

             cv2.imshow('Verification', frame)

             if cv2.waitKey(10) & 0xFF == ord('v'):
                 cv2.imwrite(os.path.join('application_data', 'input_image', 'input_image.jpg'), frame)
                 results, verified = verify(siamese_model, 0.5, 0.5)
                 print(verified)

             if cv2.waitKey(10) & 0xFF == ord('q'):
                 break
         cap.release()
         cv2.destroyAllWindows()
                                                   ...

In [80]: np.sum(np.squeeze(results) > 0.9)
```

The user needs to press the letter 'v' in order to authenticate the face in this solution. If the user has been verified, the output is "True," but if they have not been validated, the result is "False." The output can be seen as below.



```
In [79]: cap = cv2.VideoCapture(0)
         while cap.isOpened():
             ret, frame = cap.read()
             frame = frame[120:250+300,300:450+250, :]

             cv2.imshow('Verification', frame)

             if cv2.waitKey(10) & 0xFF == ord('v'):
                 cv2.imwrite(os.path.join('application_data', 'input_image', 'input_image.jpg'), frame)
                 results, verified = verify(siamese_model, 0.5, 0.5)
                 print(verified)

             if cv2.waitKey(10) & 0xFF == ord('q'):
                 break
         cap.release()
         cv2.destroyAllWindows()
                                                   ...

         True
         True
         True
         False
         True
         True
```

CONCLUSION

Python is the computer programming language that is utilized for this facial authentication solution. Tools like as tensorflow, OpenCV and matplotlib are utilized, in addition to Python. This approach for biometric authentication using facial features is compatible with all operating systems.

REFERENCE

[1]for, "Graph regularization for document classification using natural graphs | Neural Structured Learning | TensorFlow," *TensorFlow*, 2018. xhttps://www.tensorflow.org/neural_structured_lea

[2] "Open CV Cheat Sheet," *Cheatography*, 2022. https://cheatography.com/thatguyandy27/cheat-sheets/open-cv/ (accessed March 19, 2022).

[3] "Tensor Analysis Face Verification in the Wild and Kinship," *ResearchGate*, Jun. 21, 2017. https://www.researchgate.net/project/Tensor-Analysis-Face-Verification-in-the-Wild-and-Kinship (accessed April 21, 2022).

[4] N. Kose and J. Dugelay, "On the vulnerability of face recognition systems to spoofing mask attacks," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 2357-2361, doi: 10.1109/ICASSP.2013.6638076.

[5] S. Nemmaoui and S. Elhammani, "A New Approach Based on Steganography to Face Facial Recognition Vulnerabilities Against Fake Identities," *Business Intelligence*, pp. 269–283, 2021, doi: 10.1007/978-3-030-76508-8_19.

[6] U. Scherhag, R. Raghavendra, K. B. Raja, M. Gomez-Barrero, C. Rathgeb, and C. Busch, "On the vulnerability of face recognition systems towards morphed face attacks," *IEEE Xplore*, Apr. 01, 2017.

https://ieeexplore.ieee.org/abstract/document/7935088/ (accessed Jul. 26, 2020).

[7] R. Ramachandra and C. Busch, “Presentation Attack Detection Methods for Face Recognition Systems,” *ACM Computing Surveys*, vol. 50, no. 1, pp. 1–37, Mar. 2017, doi: 10.1145/3038924.

[8] S. Nemmaoui and S. Elhammani, “A New Approach Based on Steganography to Face Facial Recognition Vulnerabilities Against Fake Identities,” *Business Intelligence*, pp. 269–283, 2021, doi: 10.1007/978-3-030-76508-8_19.