Design a Siamese Architecture to compare the similarity between two images

Name: Shrinidhi Anil Varna
Roll No: 171CO145
Course: Computer Vision (CO468)

The assignment consists of the following parts. It begins with a brief description of one-shot learning, followed by Siamese Network for one-shot learning. The dataset is described with the problem statement. The code with brief explanations follows. The code ends with results obtained by training the Siamese network architecture.

One-shot learning

One-shot learning is a classification task where one or a few examples are used to classify many new examples in the future. Typically, classification involves fitting a model given many examples of each class, then using the fit model to make predictions on each class's many examples. One-shot learning is a classification task where one example (or a very small number of samples) is given for each class, which is used to prepare a model, which must make predictions about many unknown examples in the future. For example, a model can be built to recognise/identify 50 staff members working in a school. This model is prepared to predict these 50 staff members based on some images of their past. It must predict a staff member correctly for different moods and states at which they might arrive at school on a particular day. This should be

distinguished from zero-shot learning, in which the model cannot look at any examples from the target classes. In the case of face verification, a model or system may only have one example of a person's face on record and must correctly verify new photos of that person, perhaps each day.

Siamese Network for One-Shot Learning

A Siamese network is an architecture with two parallel neural networks, each taking a different input and whose outputs are combined to provide some prediction. Two identical networks are used, one taking the known signature for the person and another taking a candidate signature. Both networks' outputs are combined and scored to indicate whether the candidate signature is real or a forgery. The deep CNNs are first trained to discriminate between examples of each class. The idea is to have the models learn feature vectors that effectively extract abstract features from the input images. The models are then re-purposed for verification to predict whether new examples match a template for each class. The Siamese Network is interesting for its approach to solving one-shot learning by learning feature representations (feature vectors) compared to verification tasks. Dimensionality reduction is the approach that Siamese networks use to address one-shot learning. Unlike other loss functions that may evaluate a model's performance across all input examples in the training dataset, a contrastive loss is calculated between pairs of inputs, such as between the two inputs provided to a Siamese network. Pairs of examples are provided to the network, and the loss function penalises the model differently based on whether the samples' classes are the same or different. Specifically, if the classes are the same, the loss function encourages the models to output feature vectors that are more similar. In contrast, if the classes differ, the loss function encourages the models to output feature vectors that are less similar.

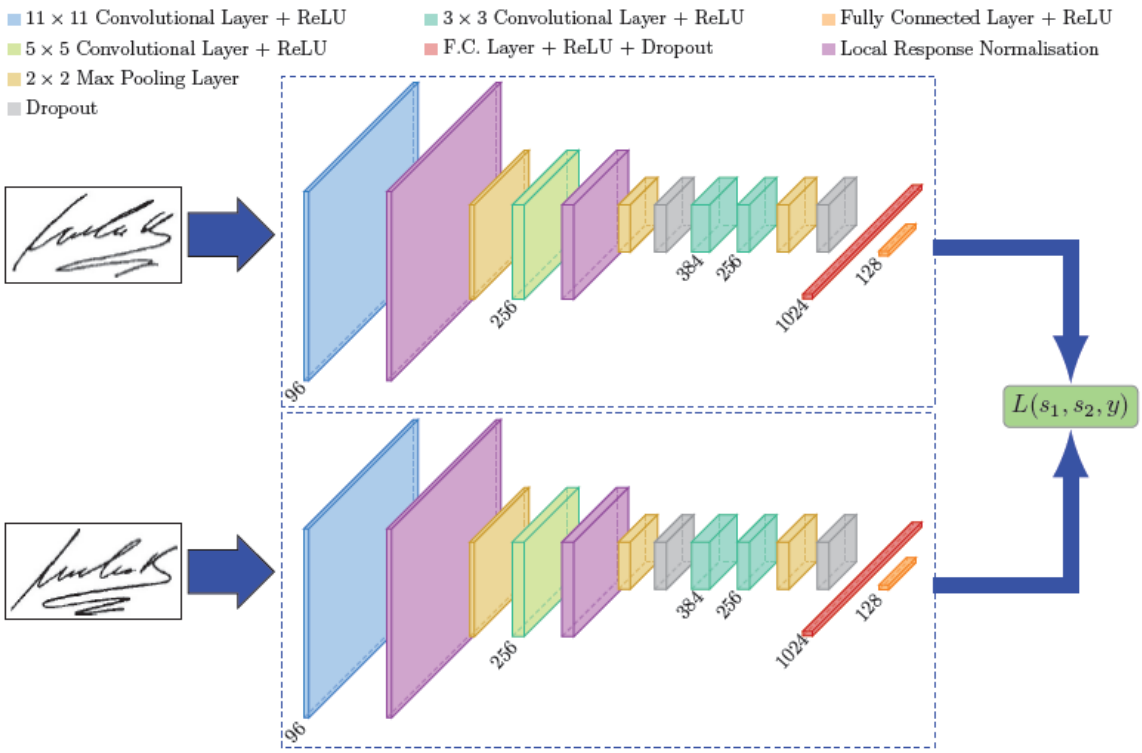


Image credits: SigNet

Problem Statement

To verify signatures of Dutch users for checking its forgery. To solve this problem, we can make use of the Siamese Network. The approach to solving this problem has been explained in the previous section.

Dataset

Link to the dataset: https://www.kaggle.com/robinreni/signature-verification-dataset

This dataset contains the signatures of Dutch users, both genuine and fraud. The dataset includes 64 distinct user signatures in the training set and 21 different user signatures in the testing set. Under each user, we have multiple signatures that are put inside two directories. For example, if there is a user with user_id = “049”, the “049/”

consists of signatures that are genuine, whereas “049_forg” consists of fraudulent signatures. For every genuine signature in the former directory, there is an equivalent fraudulent signature in the latter directory. Two CSV files contain the address of the genuine signatures mapped with the forged signatures alongside. One CSV file includes this for the training set and another for the testing set.

Code snippets

Importing libraries

Keras is the library used to perform deep learning tasks with libraries like cv2 for image processing purposes.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import cv2
from keras import backend as K
from keras.layers import Dense, Dropout, Flatten, Conv2D, Dropout
from keras.layers import BatchNormalization, Activation, Input, Lambda
from keras.layers import Convolution2D, MaxPooling2D, Flatten, Dense,
from keras.models import model_from_json, Sequential, Model, load_model
from keras.optimizers import RMSprop
from keras import optimizers, callbacks
from keras.callbacks import ModelCheckpoint, LearningRateScheduler
from keras.callbacks import EarlyStopping, ReduceLROnPlateau, TensorBoard
import os
import json
import warnings
from keras.utils.vis_utils import plot_model
from sklearn.metrics import accuracy_score
```

Loading the dataset

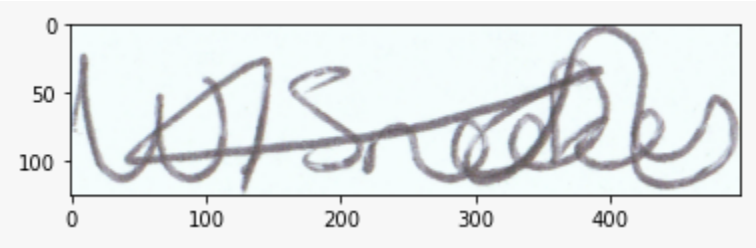
The dataset images are stored in train/ and test/ directories, whereas the image paths of a genuine image and its forge are present in the CSV files. We are using *pandas* to load the CSV files, whereas cv2 (openCV) loads the images.

```
train_dir="../signature-verification-dataset/sign_data/train/"
train_csv="../signature-verification-dataset/sign_data/train_data.csv"
test_csv="../signature-verification-dataset/sign_data/test_data.csv"
test_dir="../signature-verification-dataset/sign_data/test/"
df_train = pd.read_csv(train_csv)
df_test = pd.read_csv(test_csv)
```

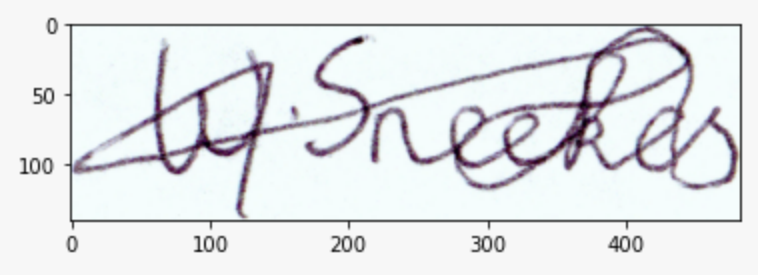
Checking the image sample in the dataset

Here we preview the images to check how they look. The first and second image is of the same person. The first one is the original (genuine) image, whereas the second one is its forge.

```
img = plt.imread(train_dir+df_train.iat[1,0])
plt.imshow(img)
```




```
img = plt.imread(train_dir+df_train.iat[1,1])
plt.imshow(img)
```



Data preprocessing

Here the first step is to read the image path from a row in the data frame and load it using cv2 (OpenCV). First, we do the following steps for images present in the test set. The image is initially in BGR format, which is converted to grayscale. It is then resized to 100x100 dimension and appended to a list (array). The same operation is done to the forged image as well. The images are then normalised to 0 to 1 range from [0, 255] domain. The exact process is repeated for training set images.

```
# testing set images
test_images1 = []
test_images2 = []
for j in range(0,len(df_test)):
    img1 = cv2.imread(test_dir+df_test.iat[j,0])
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img1 = cv2.resize(img1, (100, 100))
    test_images1.append(img1)
    img2 = cv2.imread(test_dir+df_test.iat[j,1])
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    img2 = cv2.resize(img2, (100, 100))
    test_images2.append(img2)


test_images1 = np.array(test_images1)/255.0
test_images2 = np.array(test_images2)/255.0

# training set images
train_images1 = []
train_images2 = []
```

```
train_labels = []
for i in range(len(df_train)):
    img1 = cv2.imread(train_dir+df_train.iat[i,0])
    img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    img1 = cv2.resize(img1, (100, 100))
    train_images1.append(img1)
    img2 = cv2.imread(train_dir+df_train.iat[i,1])
    img2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    img2 = cv2.resize(img2, (100, 100))
    train_images2.append(img2)
    train_labels.append(df_train.iat[i,2])
train_images1 = np.array(train_images1)/255.0
train_images2 = np.array(train_images2)/255.0
train_labels = np.array(train_labels)
```

```
# making a list item into a list (Ex: [1, 2] --> [[1], [2]])
train_images1 = np.expand_dims(train_images1, -1)
train_images2 = np.expand_dims(train_images2, -1)
test_images1 = np.expand_dims(test_images1, -1)
test_images2 = np.expand_dims(test_images2, -1)
```

Siamese network

Here we define the similarity function, which is the Euclidean distance.
The first function computes the same. The second function returns the shape of the output.
Lastly, we have a function that defines the Siamese neural network architecture. The dropout functions are used to avoid overfitting of the model. The activation function and the rest of the architecture are per a standard Siamese network widely accepted.

```
def euclidean_distance(vects):
    x, y = vects
    return K.sqrt(K.sum(K.square(x - y), axis=1, keepdims=True))
def eucl_dist_output_shape(shapes):
    shape1, shape2 = shapes
    return (shape1[0], 1)
def build_base_network(input_shape):
    model = Sequential()
    kernel_size = 3


    # conv layer 1
    model.add(Convolution2D(64, (kernel_size, kernel_size),
                input_shape=input_shape))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(.25))

    model.add(Convolution2D(32, (kernel_size, kernel_size)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(.25))

    # conv layer 2
    model.add(Convolution2D(32, (kernel_size, kernel_size)))
    model.add(Activation('relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(.25))
```

```
    # flatten
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(50, activation='relu'))
    return model
```

Training the model

Here we set the image dimensions of the input images to the neural network. It is the same as we have done earlier during resizing of images. The neural network is loaded, and Lambda, Dense layers are appended to it, which gives the final output (prediction). The dense layer does the binary classification here, and hence 'sigmoid' function is used. Early stopping is set to validation loss parameter. When the validation loss increases, a callback function stops the training process. The loss function used here is the binary-cross entropy:

$$E = -(t \log y + 1 - t \log(1 - y))$$

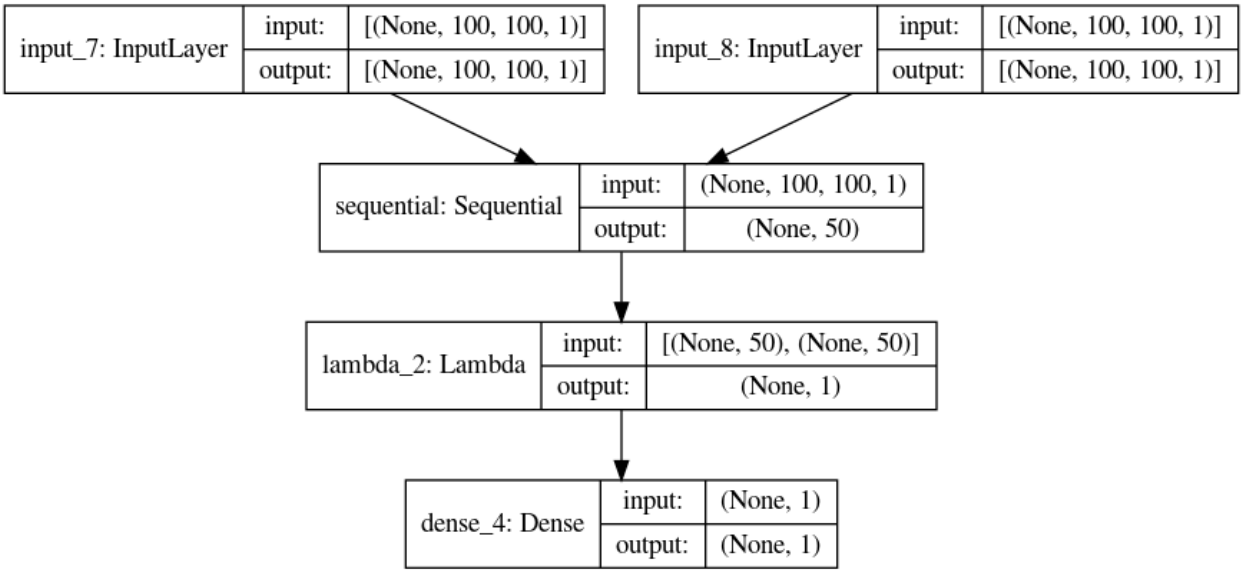
Since sigmoid neurons are present in the fully connected layers that give probabilistic output, depending on which one of the two categories has to be predicted, a loss function must be chosen, which targets two labels. If you consider binary cross-entropy and mean square error as loss functions, mean square error (MSE) assumes that the underlying data has been generated to form a normal distribution. But the ground truth here is a Bernoulli distribution. Therefore binary cross-entropy is preferable for this specific task. The model is fitted with the Adam optimiser with 0.0001 as its learning rate is a hyper-parameter that works fine for this problem. We train the model for eight epochs and get the following result. The training set is divided into further training and validation sets, with the latter taking out 10% of the images from the

former for itself before the training process begins. The train and validation metrics are obtained during the training process and are later used to plot the graphs.

```
input_dim = (100,100,1)

base_network = build_base_network(input_dim)
img_a = Input(shape=input_dim)
img_b = Input(shape=input_dim)
feat_vecs_a = base_network(img_a)
feat_vecs_b = base_network(img_b)
distance = Lambda(euclidean_distance,
                  output_shape=eucl_dist_output_shape)
                  ([feat_vecs_a, feat_vecs_b])
prediction = Dense(1,activation='sigmoid')(distance)
earlyStopping = EarlyStopping(monitor='val_loss',
                              min_delta=0,
                              patience=3,
                              verbose=1)
callback_early_stop_reduceLROnPlateau=[earlyStopping]
model = Model([img_a, img_b],prediction)

plot_model(model, to_file='model_plot.png', show_shapes=True,
show_layer_names=True)
```



Here the *Sequential* layer represents the CNN network defined in build_based_network()

```
model.compile(loss="binary_crossentropy",
optimizer=optimizers.Adam(lr=0.0001), metrics=["accuracy"])

history = model.fit([train_images1,train_images2], train_labels,
validation_split=0.10, batch_size= 32, verbose=1, epochs=8,
callbacks=callback_early_stop_reduceLROnPlateau)
```

```
Epoch 1/8
653/653 - 12s 17ms/step - loss: 0.7224 - acc: 0.4684 - val_loss: 0.6929 - val_acc: 0.5118
Epoch 2/8
653/653 - 10s 16ms/step - loss: 0.6917 - acc: 0.5467 - val_loss: 0.6929 - val_acc: 0.5118
Epoch 3/8
653/653 - 10s 16ms/step - loss: 0.6911 - acc: 0.5439 - val_loss: 0.6929 - val_acc: 0.5118
Epoch 4/8
653/653 - 10s 16ms/step - loss: 0.6902 - acc: 0.5479 - val_loss: 0.6931 - val_acc: 0.5118
Epoch 5/8
653/653 - 10s 16ms/step - loss: 0.6525 - acc: 0.6183 - val_loss: 0.4255 - val_acc: 0.9138
```

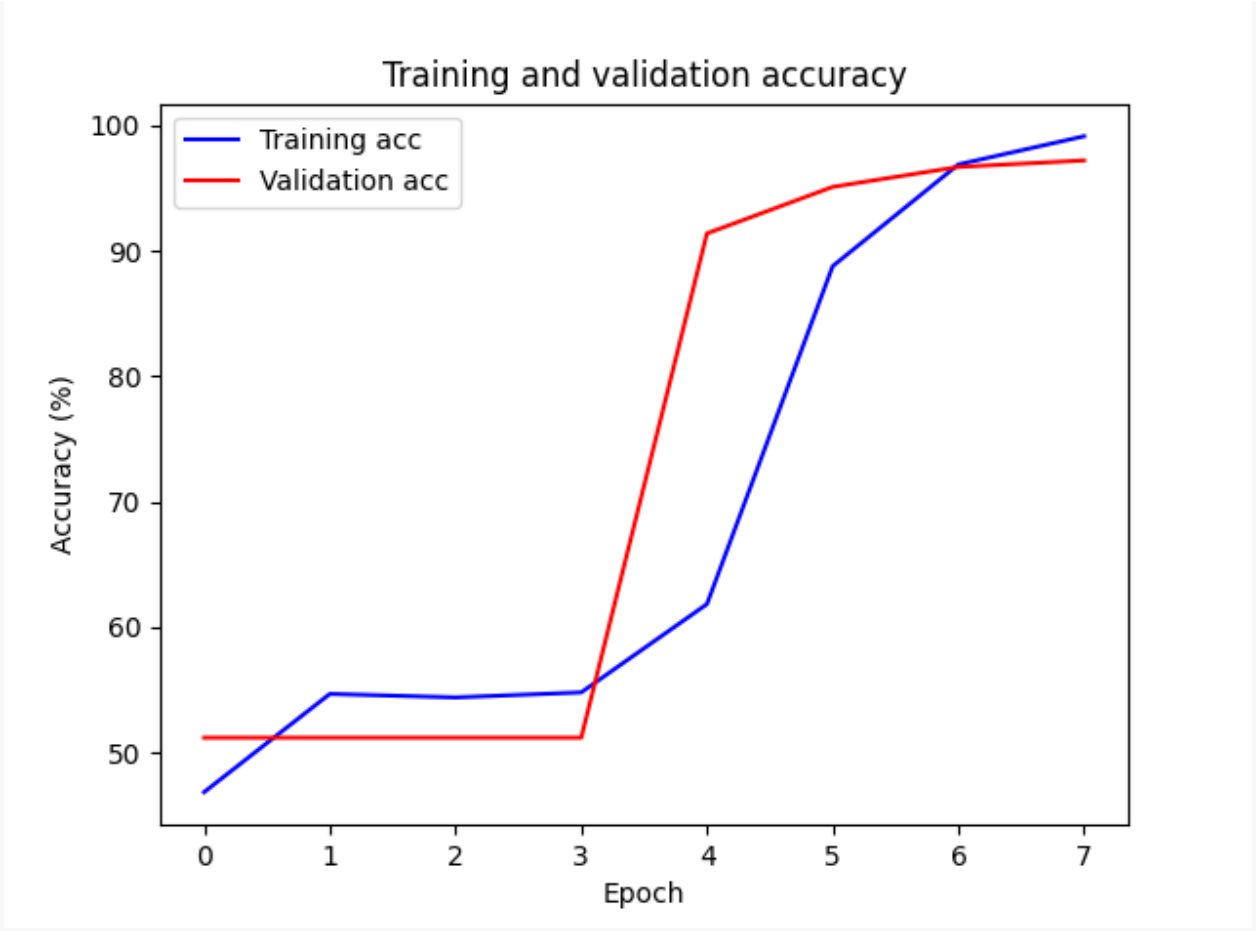
```
Epoch 6/8
653/653 - 10s 16ms/step - loss: 0.4451 - acc: 0.8877 - val_loss: 0.3508 - val_acc: 0.9509
Epoch 7/8
653/653 - 10s 16ms/step - loss: 0.3553 - acc: 0.9688 - val_loss: 0.3171 - val_acc: 0.9668
Epoch 8/8
653/653 - 10s 16ms/step - loss: 0.3117 - acc: 0.9913 - val_loss: 0.2918 - val_acc: 0.9720
```

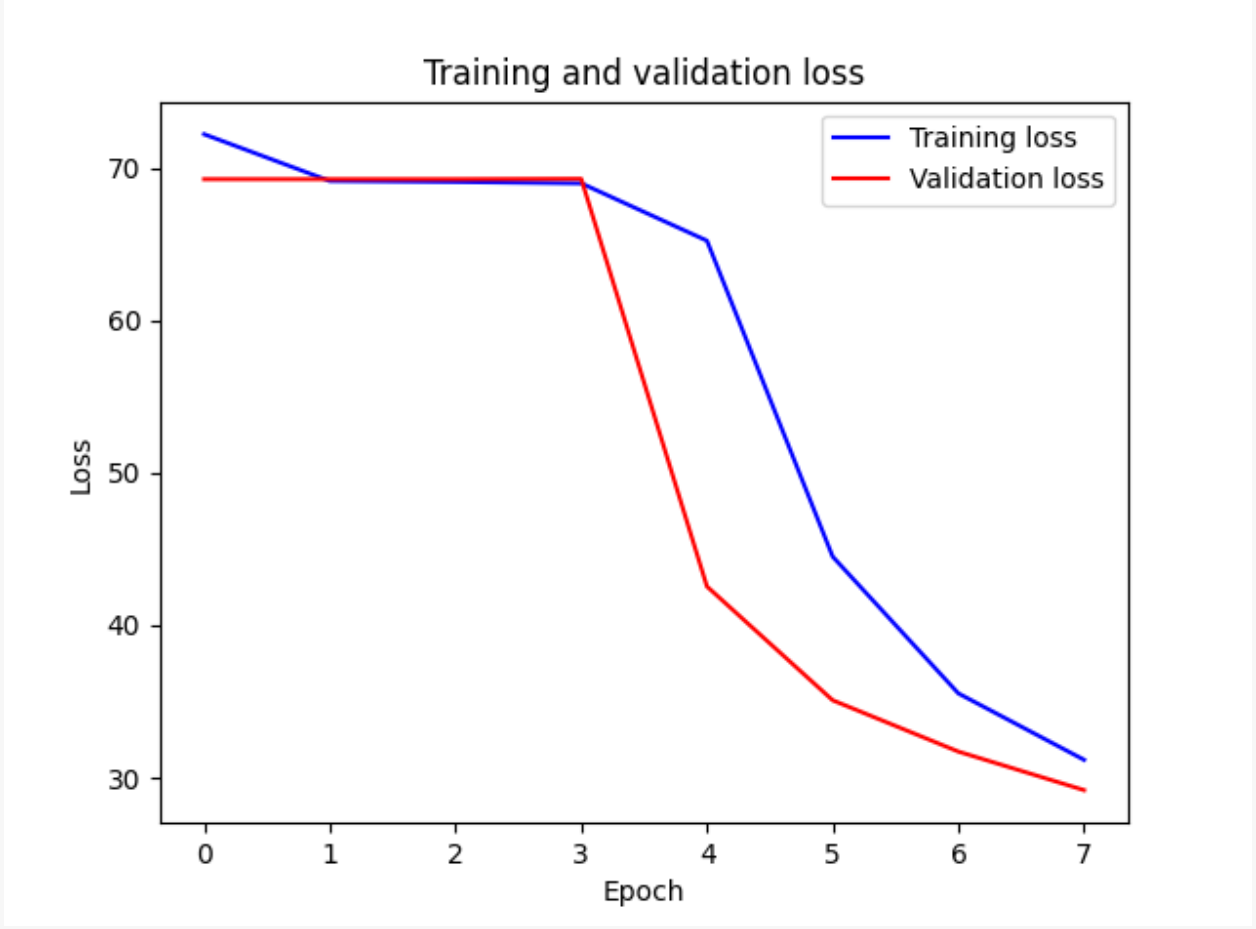
Plotting graphs

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(len(acc))
plt.plot(epochs, acc, 'b', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Training and validation accuracy')
plt.legend()
plt.show()
plt.figure()


loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
plt.plot(epochs, loss, 'b', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and validation loss')
plt.legend()
```

```
plt.show()
plt.figure()
```

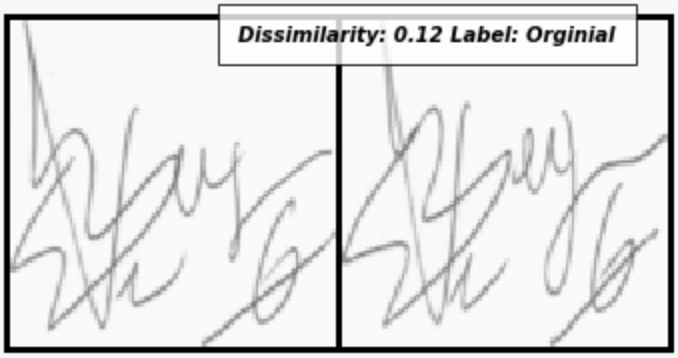

Training and validation accuracy
Training acc
Validation acc
Accuracy (%)
100
90
80
70
60
50
0
1
2
3
4
5
6
7
Epoch

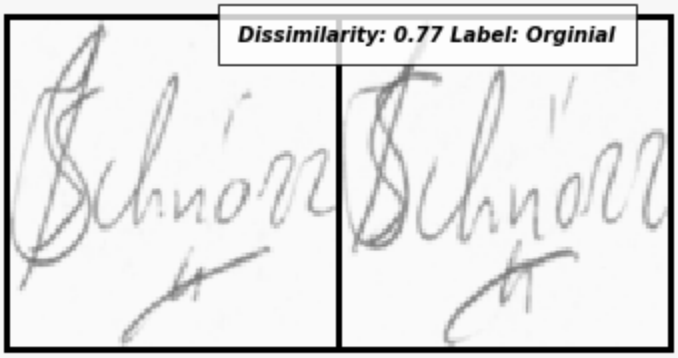


Predictions

Here are some predictions that were made after training the model. The similarity measure, along with the classification, is returned as the output. The label original indicates that the signature is genuine, whereas forged would suggest that it is fraudulent.


Dissimilarity: 0.79 Label: Forged


Dissimilarity: 0.12 Label: Orginial


Dissimilarity: 0.77 Label: Orginial


Dissimilarity: 0.32 Label: Forged

==========