Trusted Authentication Token

Intern Documentation

(June 2022 - August 2022)

Core Objectives **2**

Process Workflow **2**

Phases Of Development **3**

Phase 1: Signature Authentication **3**

Module 1: Automatic Signature Extraction 3

Module 2: Scikit's SSIM To Compare Structural Similarity 4

Module 3 - Siamese Neural Networks To Compare Image Vectors 8

Phase 2: Document Similarity Analysis **9**

Module 1: Extracting Text From Files 9

Module 2: Analyzing Document Similarity (Cosine Similarity) 10

Module 3: Text Parsing To Identify Text Forgery 11

REPLACED MODULES **12**

Signature Extraction Using signature-detect 12

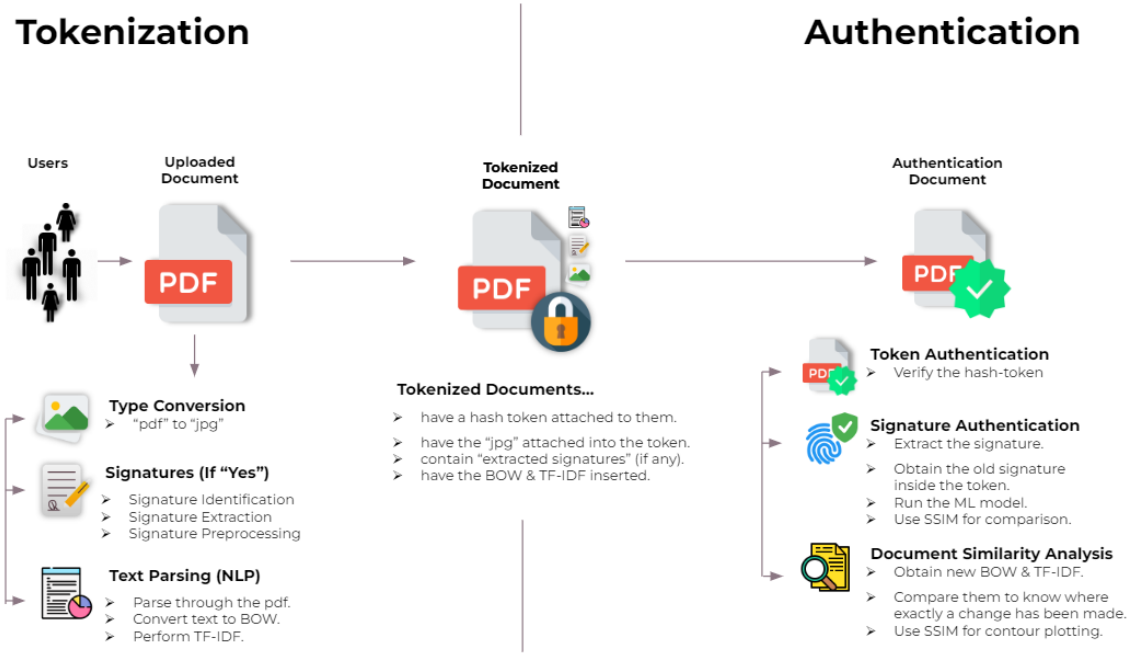
Transfer Learning (VGG & ResNet) 14

I. Core Objectives

- To identify, extract and authenticate signatures in the user-uploaded documents and offer a predictive score that categorizes the signatures into genuine and forged.

- To parse through the document's text both before and after tokenization to predict precisely where a change has been made to the document. This includes words, sentences, and punctuations.

II. Process Workflow

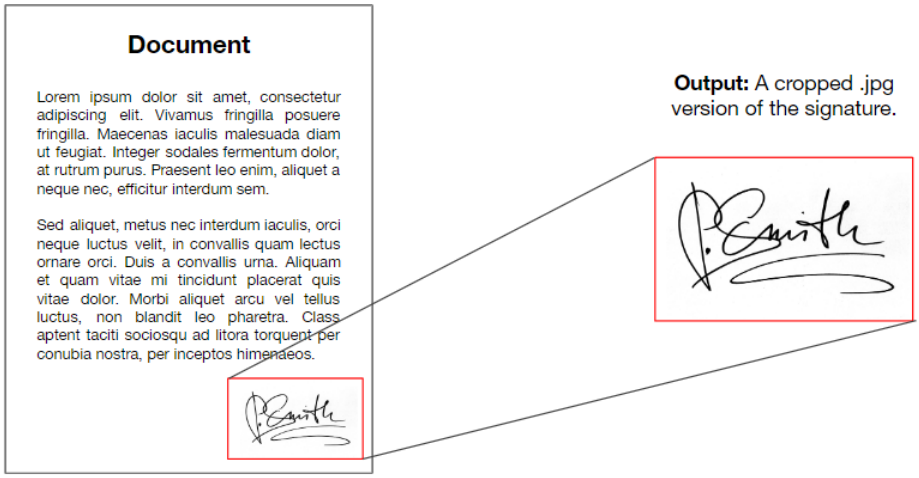


III. Phases Of Development

Phase 1: Signature Authentication

Authenticates the signature (if present) in the document and provides the probability of the given signature being genuine/forged.

Module 1: Automatic Signature Extraction



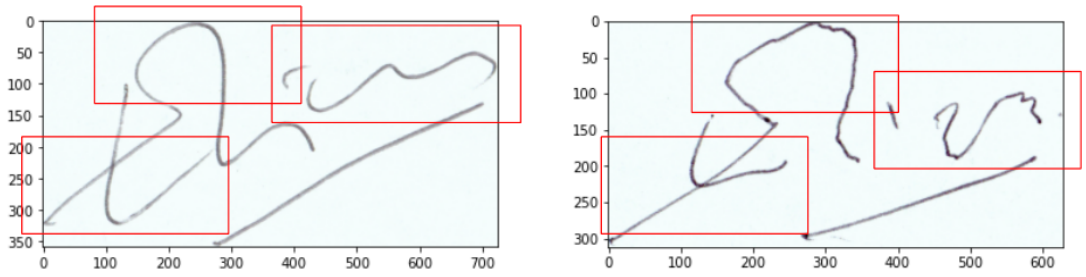
Built Using: [OpenCV]

Package Used: [cv2.findContours()]

Short Description:

- With OpenCV's contour detection capability, one can identify the boundaries of a signature and plot a rectangular contour box around it.
- This contour box would then be used as a mask.
- The dimensions of this mask along with its position on the given document would be used to crop the signature from the background using `cv2.boundingRect()`.
- The cropped signature would be stored temporarily in the local host for further processing and authentication.

Module 2: Scikit's SSIM To Compare Structural Similarity



Genuine Image
(Before Tokenization)

Image Under Consideration
(After Tokenization)

Built Using: SSIM Package (Scikit-Image)

Usage: `from skimage.metrics import structural_similarity as ssim`

Base code: Scikit-Image SSIM (Underlying Base Code)

Input: `match(path1, path2, contour = True)`

The inputs would be two signatures (extracted before and after Tokenization)

Output:

1. **A similarity score** (For example, "The two signatures are 100% similar")
2. **A side-by-side comparison with contour boxes** (If `contour = True`)

Short Description:

The Structural Similarity Index (SSIM) is a perceptual metric that quantifies the image quality degradation or changes made due to image modifications. Scikit-Image's SSIM package allows us to calculate the mean structural similarity index between two images. Similarity calculations are done independently for each channel and then averaged.

Plotting Contours:

Apart from analyzing the structural similarity and offering a "percentage" to which the images are similar, the code module can also draw contours (or boxes) around the

regions that differ between two images (in our case, signatures). However, this is not included in the standard package and has been coded separately.

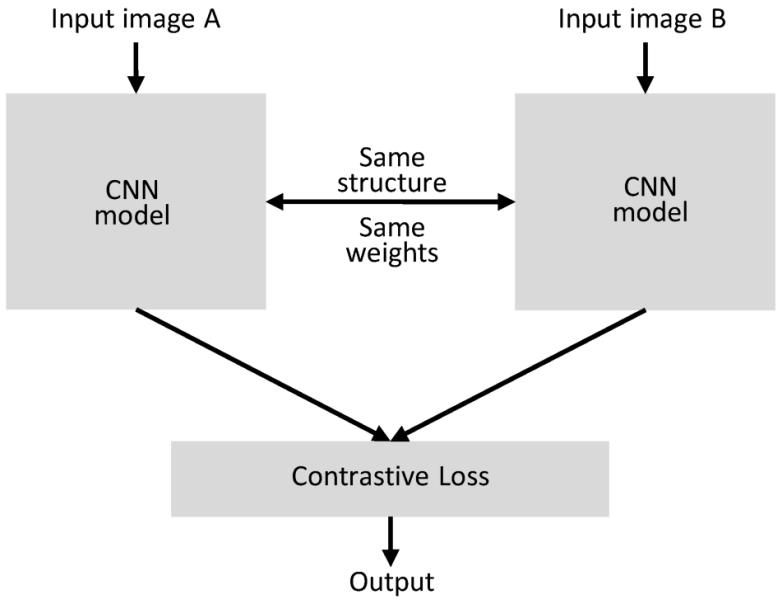
The code below allows us to draw contour boxes around the regions that differ,

```
if countour == True:
 thresh = cv2.threshold(diff, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
 cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
 cnts = imutils.grab_contours(cnts)

 for c in cnts:
   # compute the bounding box of the contour and then draw the
   # bounding box on both input images to represent where the two
   # images differ
   (x, y, w, h) = cv2.boundingRect(c)
   cv2.rectangle(img1, (x, y), (x + w, y + h), (0, 0, 255), 2)
   cv2.rectangle(img2, (x, y), (x + w, y + h), (0, 0, 255), 2)
```

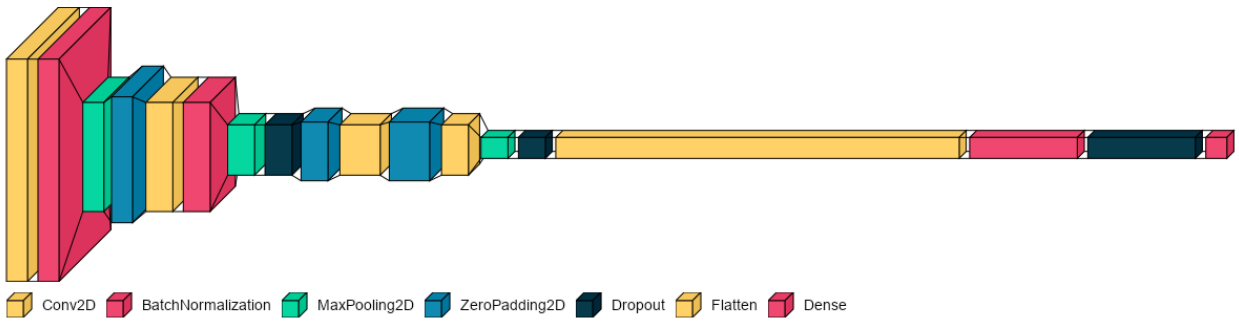
Module 3 - Siamese Neural Networks To Compare Image Vectors

Objective:

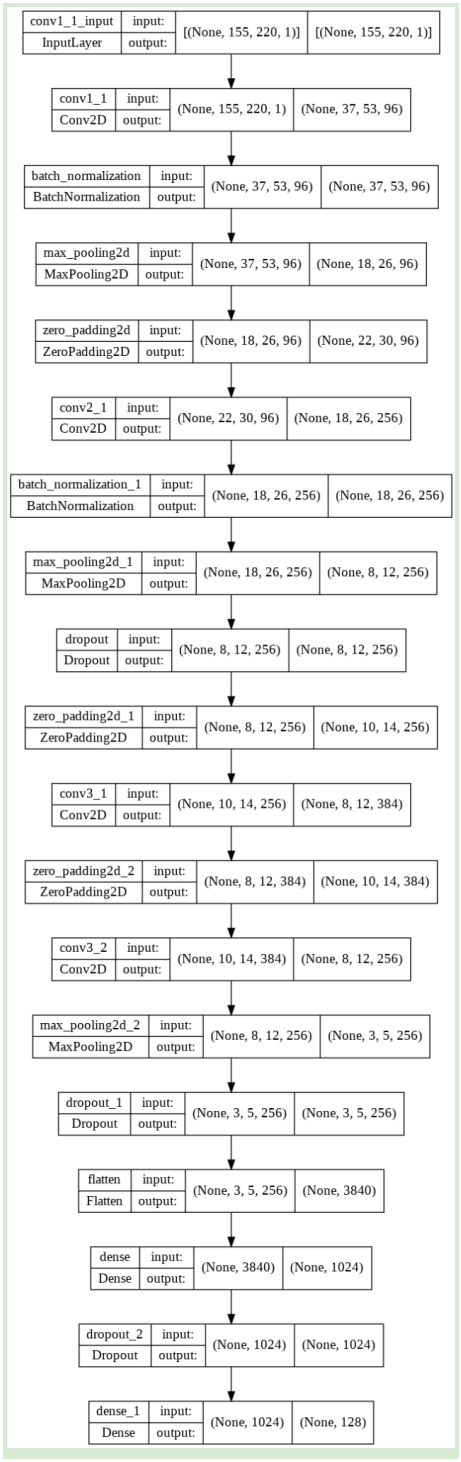
Most CNN classification models (for processing images) are only capable of accepting one single input. The objective of utilizing a Siamese Neural Network is to feed two inputs (two signatures, before and after tokenization) to a CNN model and compare how similar/different the two images are.



Base Model:



Structure Of The Model:



Dataset:
https://www.kaggle.com/datasets/divyanshrai/handwritten-signatures

→ The dataset is further divided into 4 different dataset folders.
→ Each dataset folder has a folder pair of real and forged signatures.
→ Dataset 1 & Dataset 3 - Training Data
→ Dataset 2 - Validation Data
→ Dataset 4 - Test Data

Base Model (Code):
Google Colab

Threshold: 0.5474644088745119

→ This is the base threshold value (capability) of the model that has been generated by passing the test set.
→ If the difference score is above this threshold, it is categorized as a forged signature and if it's below the threshold, it is categorized as a genuine signature.

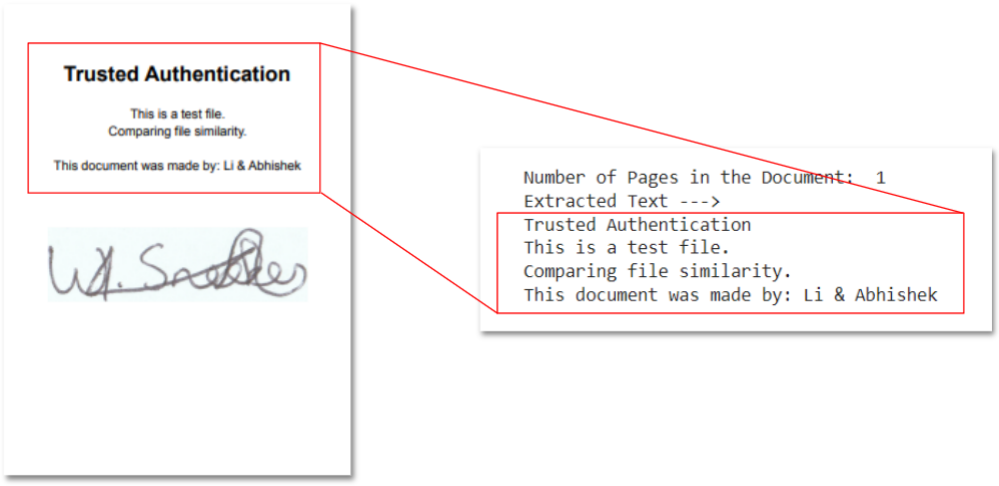
Output:

Difference Score = 0.93089944
It's a Forged Signature

Phase 2: Document Similarity Analysis

Predicts whether the document has been modified before and after tokenization and pinpoints the user to exactly where the change has been made.

Module 1: Extracting Text From Files



Built Using: [PyPDF2 Python Package]
Installation: `pip install PyPDF2`

Short Description:

```
path1='/content/testdocReal.pdf'
path2='/content/testdocForg.pdf'
text1=text_extract(path1)
text2=text_extract(path2)
```

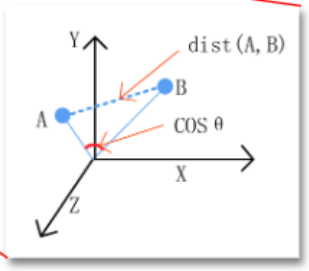
The package is pre-built and only requires the input path of the files from where we need to extract the text from.

Module 2: Analyzing Document Similarity (Cosine Similarity)

Built Using:

- ➔ `PorterStemmer()` from the `nltk` package.
- ➔ `defaultdict()` from the `nltk` python package.
- ➔ Linear Algebra functions from NumPy.
- ➔ Dot Product functions from Numpy.

```
def cos_sim(a,b):
    dot_product=np.dot(a,b)
    norm_a=np.linalg.norm(a)
    norm_b=np.linalg.norm(b)
    return dot_product/(norm_a * norm_b)
```



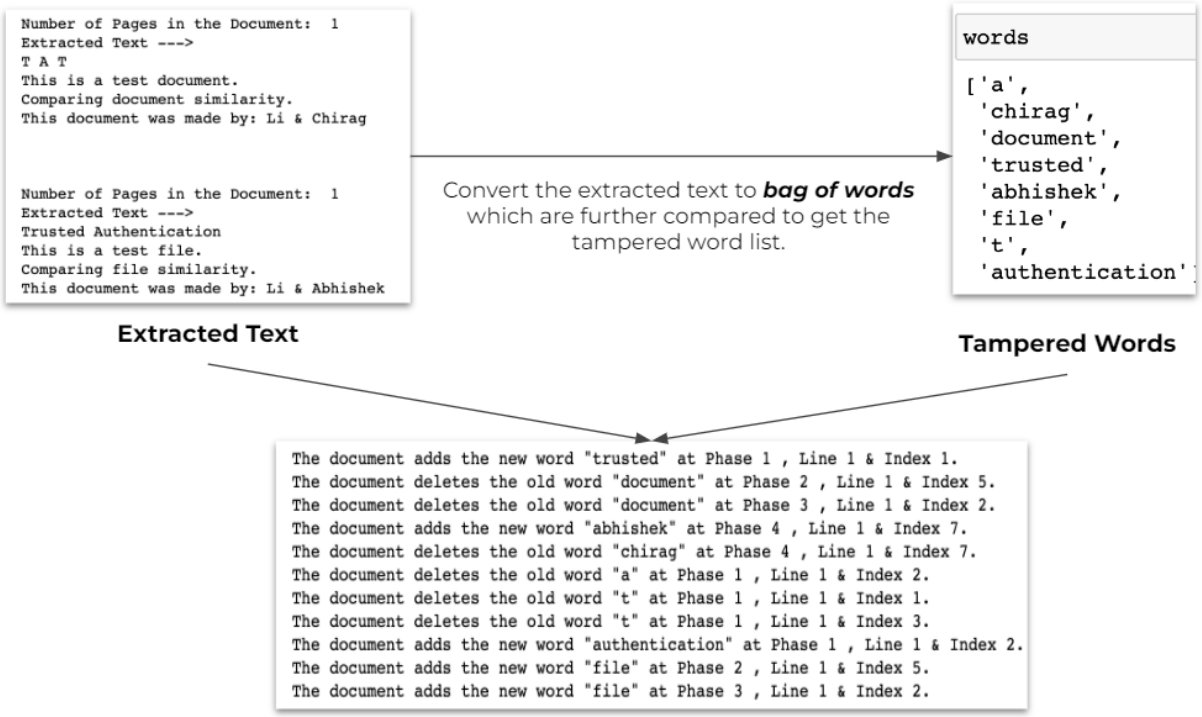

```
dict1=process(text1)
dict2=process(text2)
print("Similarity between two text documents", getSimilarity(dict1,dict2))
```

```
Similarity between two text documents 0.8295150620062532
```

Short Description:

- ➔ The principles of **Cosine Similarity** is the most important piece of this module.
- ➔ Cosine Similarity helps to predict how similar or different two sequences of numbers are by plotting the arrays on a multi-dimensional axis.
- ➔ The process( ) function in submitted Final Code Module converts the extracted text into **Bag Of Words (BOW)** which here is a python dictionary with key: value pairs of all the unique words along with their frequency of occurrence.
- ➔ The getSimilarity( ) function then uses these BOWs to find their cosine similarity.
- ➔ The more closer it is to 1, the more similar the documents are. The closer to 0 it gets, the more dissimilar (the more forged) the documents might be.

Module 3: Text Parsing To Identify Text Forgery



Short Description:

- ➔ The Bag Of Words from the previous module (both before and after tokenization) is compared against each other to identify the words that misalign.
- ➔ These words have either been erased in the old document or added in the document after tokenization.
- ➔ These tampered are stored in a list.
- ➔ This list of words are compared against the extracted text from both of these documents to identify the Phase, Line and Index in which the change was made.

REPLACED MODULES

The modules that are mentioned below when integrated with the existing framework were either found to be significantly underperforming or not well-suited for the requirement when compared to their current counterparts.

Signature Extraction Using `signature-detect`

Reason for discontinuation:	This was a pre-existing python package that was developed by a private contributor. Although it was really lightweight both in terms of required packages and the time taken for function execution, it fails to identify the contours of the signature every single time especially when the given document doesn't have a clear white background.
Replacement	A new module was then developed using plain OpenCV's syntax taking full advantage of its contour identification ability. Once the contours of a signature was identified, it's then used as a mask and cropped out from the background.

Built Using: signature-detect 0.1.4

Installation: pip install signature-detect

Input: `extractor(inputPath = 'inputfile', outputPath = 'outputfile.png')`

The functions mentioned below have been compiled into a single function that takes in an input document (a .pdf, .txt, etc., that has been converted to an image), extracts the signature, and saves it locally for further processing.

Short Description:

The Loader, Extractor, and Cropper functions are utilized inside the signature-detect python module. Any given file is first converted to an image file which in turn gets converted to NumPy arrays. The arrays are labeled and any pixels that are empty or insufficient get removed. It also uses the ImageMagick open-source conversion tool which helps convert the given document into more than 200 different file formats.

1. **Loader:** It returns a list of the masks. Each mask is a NumPy 2 dimensions array. Its element's value is 0 or 255.

```
loader = Loader(
    low_threshold=(0, 0, 250),
    high_threshold=(255, 255, 255))
mask = loader.get_masks(inputPath)[0]
```

2. **Extractor:** The extractor reads a mask, labels the regions in the mask, and removes both small and big regions. The signature is a region of middle size.

```
extractor = Extractor(
      outlier_weight=1,
      outlier_bias=100,
      amplfier=100,
      min_area_size=0.1)
  labeled_mask = extractor.extract(mask)
```

3. **Cropper:** Cropper crops the regions in the labeled mask.

```
cropper = Cropper(
      min_region_size=100,
      border_ratio=0.01)
results = cropper.run(labeled_mask)
signature = results[0]["cropped_mask"]
```

4. **Judger:** Judger decides whether a region is a signature.

Transfer Learning (VGG & ResNet)

Reason For Discontinuation	Transfer Learning worked great. It yielded a really good accuracy of 98 - 99%. Although the VGG & ResNet models developed were capable of classifying one single signature along with the probability of the given signature being genuine and forged, they wouldn't be able to take in two inputs (two signatures) and compare their differences.
Replacement	In order to offer two input images to a Machine Learning model, the signature authentication module now uses a Siamese Neural Network. The Siamese Architecture uses a CNN model but runs it twice using two input images. The output arrays are then passed to a loss function (contrastive loss) to understand the differences between them.

What Is Transfer Learning?

Reference (Timestamp: 10:11 - 13:40) https://youtu.be/GVsUOuSjvcg?t=610

Transfer Learning is when you use a pre-trained neural network along with all the weights that it comes with across all the layers except the last fully-connected layer. One can introduce their own classification layer at the end on top of the network layers imported from the pre-trained model.

These pre-trained models are made up of multiple layers of network architecture (Sometimes over 100 layers) with each layer consisting of multiple neurons and were proven to yield considerable accuracies for image classification tasks.

Why Use Transfer Learning?

Major Advantage:

The CNN architectures that we can import through Transfer Learning (for example, ResNet50) particularly address the **VANISHING GRADIENT** problem.

THE VANISHING GRADIENT PROBLEM

When the model is nearing convergence, the fluctuations in weights are not significant enough to make a difference in the learning curve which makes the curvature of the learning curve constant. Even when the model still has room left for improvement, it assumes that it has reached its limit since the weight fluctuations aren't significant enough.

Other Advantages:

- Does not require a large dataset of labeled training images.
- Utilize the existing Neural Network Architectures which have proven results.
- Reduced training time (Since a lot of the layers already have acquired weights)

Popular Pre-Trained Neural Network Architectures:

- ☐ AlexNet
- ☐ VGGNet
- ☐ ResNet
- ☐ Inception

Implementation Of Transfer Learning For TAT

How is it useful for TAT?

Since we already have a signature extraction module that extracts handwritten signatures from the documents (if present), we need a way of authenticating the signatures to verify if it's genuine or forged in any way.

Data Preprocessing

Utilizing pre-trained networks still needs the training data to be preprocessed before training. This preprocessing involves reading the image using OpenCV `cv2.imread(data)`, image color conversions `cv2.cvtColor(img, cv2.COLOR_BGR2RGB)` and resizing the images `cv2.resize(img, (SIZE,SIZE))`

```
SIZE = 224
train_data = []
train_labels = []

for folder in os.listdir(train_dir):
    for data in glob.glob(train_dir+folder+'/*.*'):
        img = cv2.imread(data)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = cv2.resize(img, (SIZE,SIZE))
        train_data.append([img])
        if "forg" in folder:
            train_labels.append(np.array(1))
        else:
            train_labels.append(np.array(0))
```

Preprocessing The Training Data

The code above shows how the images in the training data (stored locally) were preprocessed. Since the input images were already separated into “genuine” and “forged” folders, we labeled the image as “1” if it is from the “forged” folder and “0” if it is from the “genuine” folder.

Importing The Pre-Trained Model:

- Pre-Trained models are effective but not ready-to-use.
- They have to be modified (at least at the output layer) to address a particular classification problem.
- In our particular problem, the output layer has been flattened to suit our input shape and a few dense layers have been added.
- While importing the pre-trained model **it is important to import the weights** along with the convoluted layers.
- If a pre-trained model is imported without the weights, it is similar to training a new model with a completely new dataset for which there’s no need for transfer learning.

▾ **Loading The Pre-Trained Model**

▾ **Base Model:** Imported CNN Layers From VGG16

Output Model: Few Dense Layers With "Softmax" (industry standard for classification)

```
[3]  base_model = applications.vgg16.VGG16(weights="imagenet", include_top=False, input_shape=(224,224,3))
     # base_model.summary()

     add_model = Sequential()
     add_model.add(Flatten(input_shape = base_model.output_shape[1:]))
     add_model.add(Dense(256, activation='relu'))
     add_model.add(Dense(2, activation='softmax'))

     model = Model(inputs=base_model.input, outputs=add_model(base_model.output))
     model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=1e-4),
                   metrics=['accuracy'])

     # model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58892288/58889256 [==============================] - 0s 0us/step
58900480/58889256 [==============================] - 0s 0us/step
```

Implementation Using VGG16

▾ **Loading The Pre-Trained Model**

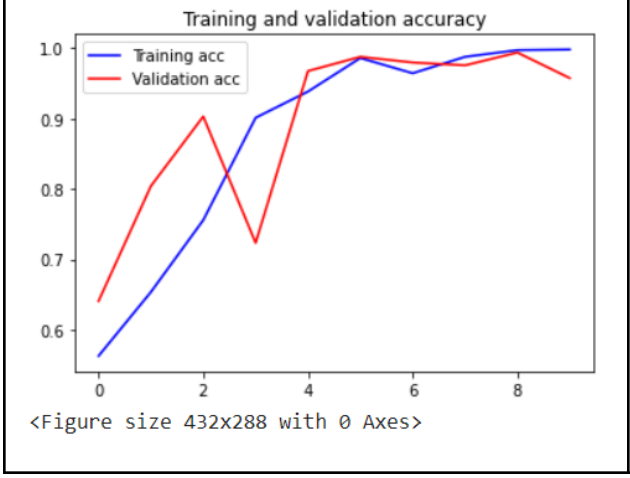
▾ **Base Model:** Imported CNN Layers From ResNet50

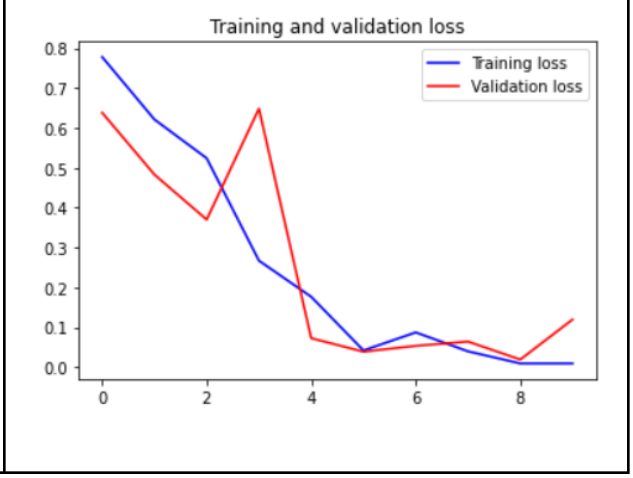
Output Model: Few Dense Layers With "Softmax" (industry standard for classification)

```
[ ]  ip = 224
     input_shape = [ip, ip, 3]

     in_1 = tf.keras.layers.Input((None, None, 3))
     in_2 = tf.keras.layers.Input((None, None, 3))

     base_model = Sequential()

     pretrained_model= tf.keras.applications.ResNet50(include_top=False, input_shape=(ip,ip,3), pooling='avg',classes=2, weights='imagenet')
     for layer in pretrained_model.layers:
            layer.trainable=False

     base_model.add(pretrained_model)
     base_model.add(Flatten())
     base_model.add(Dense(512, activation='relu'))
     base_model.add(Dense(2, activation='softmax'))
     base_model.summary()
```

Implementation Using ResNet50

Plots For Accuracy & Loss (Both Training & Testing):





The trained models gave an accuracy of 98% - 99%.

Transfer Learning - VGG16: Link To The Code

Transfer Learning - ResNet50: Link To The Code