# real-forge-signature-detection-with-datagen

April 17, 2024

#

Fake Signature Detection

## 0.1 Table of Contents



# 1 Project Overview and Objectives

The primary aim of this project was to develop a system for detecting fake signatures using machine learning techniques. A Convolutional Neural Network (CNN) architecture was employed for this task. Specifically, the project utilized a custom CNN model trained on a dataset of genuine and forged signatures. The model's performance was evaluated using various metrics, with emphasis on `accuracy`.

Accuracy $= \frac{\text{Number of correclty predicted images}}{\text{Total number of tested images}} \times 100\%$

The final results obtained are summarized as follows:

| Set | Accuracy |
|---|---|
| Train Set* | ~83% |
| Validation Set* | ~79% |

| Set | Accuracy |
| --- | --- |
| Test Set* | ~79% |

* *Note: Clarification regarding set names:*

- *`Validation set` - utilized during model training for hyperparameter tuning.*
- *`Test set` - a separate dataset reserved solely for final model evaluation.*

### 1.1 Data Set Description

The dataset used in this project consists of genuine and forged signatures collected from various sources. Each signature instance is labeled as either genuine or fake, enabling supervised learning for model training. Unfortunately, detailed information regarding the source and characteristics of the signatures is unavailable.

The dataset utilized for this project is the Signature-Forgery-Dataset, comprising signature categorized into two classes:

- `Fake` - Fake Signature, encoded as `0`
- `Original` - Genuine Signature, encoded as `1`

### 1.2 What is a Fake Signature?

> A fake signature refers to a forged imitation of an individual's handwritten signature. It is commonly used for fraudulent purposes, such as unauthorized transactions or identity theft. Detecting fake signatures is crucial in preventing financial fraud and ensuring the integrity of legal documents. Various techniques, including machine learning algorithms, can be employed to identify discrepancies between genuine and fake signatures.

Source: Wikipedia

## 2 Setting up the Environment

Setting up the environment for the fake signature detection project involved configuring the necessary software dependencies, libraries, and development environment to facilitate model development and experimentation. Key steps in setting up the environment included:

1. **Python Environment**: Creating a virtual environment using tools like virtualenv or conda to manage Python dependencies and ensure reproducibility across different systems.

2. **Installation of Libraries**: Installing essential libraries such as TensorFlow, Keras, NumPy, and Matplotlib for deep learning model development, data manipulation, and visualization.

3. **Data Preparation**: Organizing the dataset into appropriate directories for training, validation, and testing. This involved splitting the dataset into subsets and ensuring proper labeling of genuine and fake signature images.

4. **Hardware Considerations**: Depending on the computational resources available, considerations were made regarding the hardware configuration for model training. Utilization of GPUs or cloud-based computing platforms like Google Colab may be necessary for training larger models or handling extensive datasets.

5. **Development Environment**: Configuring integrated development environments (IDEs) such as Jupyter Notebook or Visual Studio Code for efficient coding, debugging, and experimentation with the CNN model.

6. **Documentation and Version Control**: Setting up documentation tools like Jupyter Notebooks or Markdown files to document code, experiments, and results. Version control using Git and platforms like GitHub or GitLab ensured collaboration and version tracking throughout the project lifecycle.

[1]:
```
import numpy as np
from tqdm import tqdm  # Importing tqdm for progress bars
import cv2  # OpenCV for image processing
import os  # Operating system interface
import shutil  # High-level file operations
import itertools  # For iterating tools
import imutils  # Image processing utility functions
import seaborn as sns  # Statistical data visualization
import matplotlib.pyplot as plt  # Plotting library
from warnings import filterwarnings  # To filter warnings

from sklearn.preprocessing import LabelBinarizer  # Label binarization
from sklearn.model_selection import train_test_split  # Splitting dataset
from sklearn.metrics import accuracy_score, confusion_matrix  # Model␣
↪evaluation metrics

import plotly.graph_objs as go  # Interactive plots
from plotly.offline import init_notebook_mode, iplot  # Offline plotting
from plotly import tools  # Tools for plot manipulation

import tensorflow as tf  # TensorFlow deep learning library
from tensorflow.keras import utils  # Utilities for model building
from tensorflow.keras.layers import Dense, Conv2D, Dropout, Flatten,␣
↪MaxPooling2D, Input  # Layers for neural network
from tensorflow.keras.models import Sequential  # Sequential model
from tensorflow.keras.optimizers import Adam  # Adam optimizer
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau  #␣
↪Callbacks for training monitoring

import visualkeras  # Visualizing keras models

RANDOM_SEED = 123  # Random seed for reproducibility

# Ignore specific warning categories
filterwarnings("ignore", category=DeprecationWarning)
filterwarnings("ignore", category=FutureWarning)
filterwarnings("ignore", category=UserWarning)
```

Right now all images are in one folder with `fraud dataset` and `original dataset` subfolders. I will split the data into `train`, `val` and `test` folders which makes its easier to work for me. The new folder heirarchy will look as follows:

```
[2]: import splitfolders  # Library for splitting dataset into train, validation,␣
     ↪and test sets

     # Define input and output paths
     input_folder = 'signature_dataset/'  # Path to the original dataset
     output_folder = 'split_data/'  # Path to store the split dataset

     # Split the dataset with a ratio of 80% for training, 10% for validation, and␣
     ↪10% for testing
     splitfolders.ratio(input_folder, output=output_folder, seed=42, ratio=(0.8, 0.
     ↪1, 0.1))
```

```
Copying files: 6114 files [00:04, 1248.02 files/s]
```

# 3 Data Import and Preprocessing

The data import and preprocessing stage involved preparing the signature dataset for training the custom Convolutional Neural Network (CNN) model. Key steps in this process included:

1. **Data Loading**: Loading the signature dataset from the specified directories (`fraud dataset` and `original dataset` subfolders) into the development environment. This step ensured that the dataset was accessible for subsequent preprocessing steps.
2. **Data Splitting**: Splitting the dataset into training, validation, and testing subsets. Typically, a common split ratio such as 70-15-15 (train-validation-test) or 80-10-10 was employed to ensure an adequate amount of data for model training, tuning, and evaluation.
3. **Image Preprocessing**: Preprocessing the signature images to standardize their size, format, and quality. Common preprocessing techniques included resizing images to a uniform resolution, converting images to grayscale, and normalizing pixel values to a specific range (e.g., [0, 1]).
4. **Data Augmentation**: Augmenting the dataset to increase its size and diversity, thereby enhancing the model's ability to generalize. Data augmentation techniques such as rotation, flipping, zooming, and shifting were applied to create variations of the original signature images.
5. **Label Encoding**: Encoding the labels of genuine and fake signatures into numerical format (e.g., 0 for fake, 1 for genuine) to facilitate model training and evaluation.
6. **Data Pipeline**: Constructing data input pipelines using libraries like TensorFlow or Keras to efficiently feed the preprocessed signature images and their corresponding labels into the CNN model during training.

```
[3]: def load_data(dir_path, img_size=(100,100)):
         """
         Load resized images as np.arrays to workspace
```

```
    Parameters:
        dir_path (str): Path to the directory containing image folders.
        img_size (tuple): Size to which images will be resized. Default is␣
↪(100, 100).

    Returns:
        tuple: A tuple containing numpy arrays of images (X), corresponding␣
↪labels (y),
        and a dictionary mapping label indices to class names.
    """
    X = []  # List to store images
    y = []  # List to store labels
    i = 0  # Counter for label indexing
    labels = dict()  # Dictionary to map label indices to class names
    for path in tqdm(sorted(os.listdir(dir_path))):  # Iterate through␣
↪directories
        if not path.startswith('.'):  # Ignore hidden files
            labels[i] = path  # Assign index to class name
            for file in os.listdir(dir_path + path):  # Iterate through files␣
↪in class directory
                if not file.startswith('.'):  # Ignore hidden files
                    img = cv2.imread(dir_path + path + '/' + file)  # Read image
                    img = cv2.resize(img, img_size)  # Resize image
                    X.append(img)  # Append image to list
                    y.append(i)  # Append label to list
            i += 1  # Increment label index
    X = np.array(X)  # Convert list of images to numpy array
    y = np.array(y)  # Convert list of labels to numpy array
    print(f'{len(X)} images loaded from {dir_path} directory.')
    return X, y, labels

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.

    Parameters:
        cm (array): Confusion matrix array.
        classes (list): List of class names.
        normalize (bool): Flag to normalize the confusion matrix. Default is␣
↪False.
        title (str): Title of the plot. Default is 'Confusion matrix'.
```

```
        cmap (matplotlib colormap): Colormap for the plot. Default is plt.cm.
↪Blues.

    Returns:
        None
    """
    plt.figure(figsize = (6,6))  # Set figure size
    plt.imshow(cm, interpolation='nearest', cmap=cmap)  # Display the image
    plt.title(title)  # Set the title
    plt.colorbar()  # Add color bar
    tick_marks = np.arange(len(classes))  # Set the tick marks
    plt.xticks(tick_marks, classes, rotation=90)  # Set x-axis labels
    plt.yticks(tick_marks, classes)  # Set y-axis labels

    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]  # Normalize␣
↪the confusion matrix

    thresh = cm.max() / 2.  # Set threshold for text color
    cm = np.round(cm,2)  # Round values in confusion matrix
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")  # Add text␣
↪annotations

    plt.tight_layout()  # Adjust layout
    plt.ylabel('True label')  # Set y-axis label
    plt.xlabel('Predicted label')  # Set x-axis label
    plt.show()  # Display plot
```

[4]:
```
TRAIN_DIR = 'split_data/train/'  # Directory path for training images
TEST_DIR = 'split_data/test/'  # Directory path for testing images
VAL_DIR = 'split_data/val/'  # Directory path for validation images
IMG_SIZE = (128,128)  # Image size for resizing

# Use predefined function to load the image data into workspace
X_train, y_train, labels = load_data(TRAIN_DIR, IMG_SIZE)  # Load training data
X_test, y_test, _ = load_data(TEST_DIR, IMG_SIZE)  # Load testing data
X_val, y_val, _ = load_data(VAL_DIR, IMG_SIZE)  # Load validation data
```

```
100%|
    | 2/2 [00:05<00:00,  2.94s/it]

4890 images loaded from split_data/train/ directory.

100%|
    | 2/2 [00:00<00:00,  2.85it/s]
```

```
613 images loaded from split_data/test/ directory.
```

```
100%|
     | 2/2 [00:00<00:00,  2.69it/s]
```

```
611 images loaded from split_data/val/ directory.
```

Let's take a look at the distribution of classes among sets:

[5]:
```
# Create an empty list to store counts
y = []
sets = ['Train Set', 'Validation Set', 'Test Set']

# Iterate through different sets and count occurrences of each class
for set_name in (y_train, y_val, y_test):
    y.append([np.sum(set_name == 0), np.sum(set_name == 1)])

# Convert the list to a numpy array
y = np.array(y)

# Plot using Seaborn
plt.figure(figsize=(10, 6))
bar_width = 0.35
index = np.arange(len(sets))

# Plot Forged (class 0) bars
forged = plt.bar(index - bar_width/2, y[:, 0], bar_width, color='#ff3300',␣
 ↪label='Forged')
# Add count values above Forged bars
for bar in forged:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height, '%d' % int(height),␣
 ↪ha='center', va='bottom')

# Plot Original (class 1) bars
original = plt.bar(index + bar_width/2, y[:, 1], bar_width, color='#33cc33',␣
 ↪label='Original')
# Add count values above Original bars
for bar in original:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2., height, '%d' % int(height),␣
 ↪ha='center', va='bottom')

plt.title('Count of classes in each set')
plt.xlabel('Set')
plt.ylabel('Count')
plt.xticks(index, sets)
```

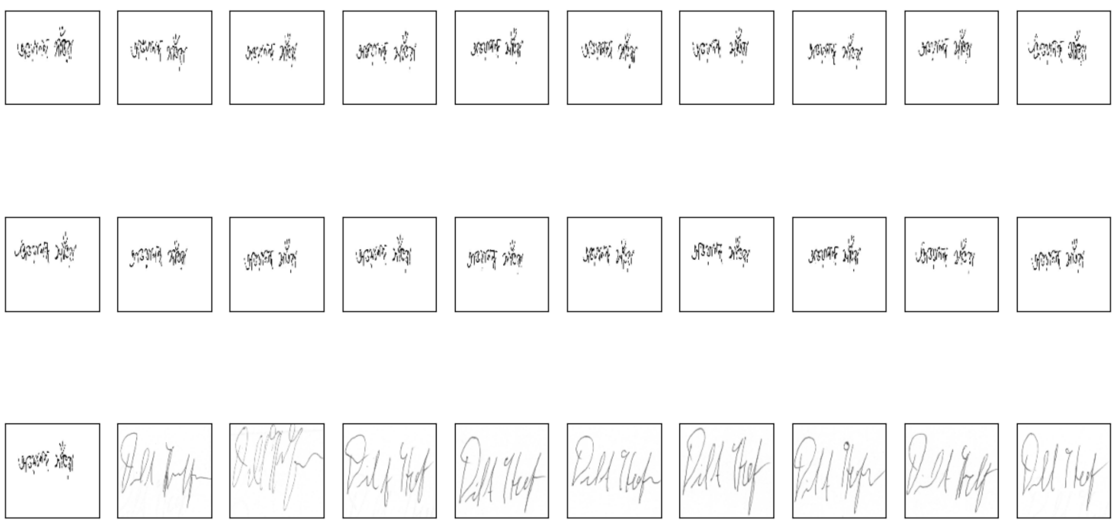
```
plt.legend()
plt.tight_layout()
plt.show()
```

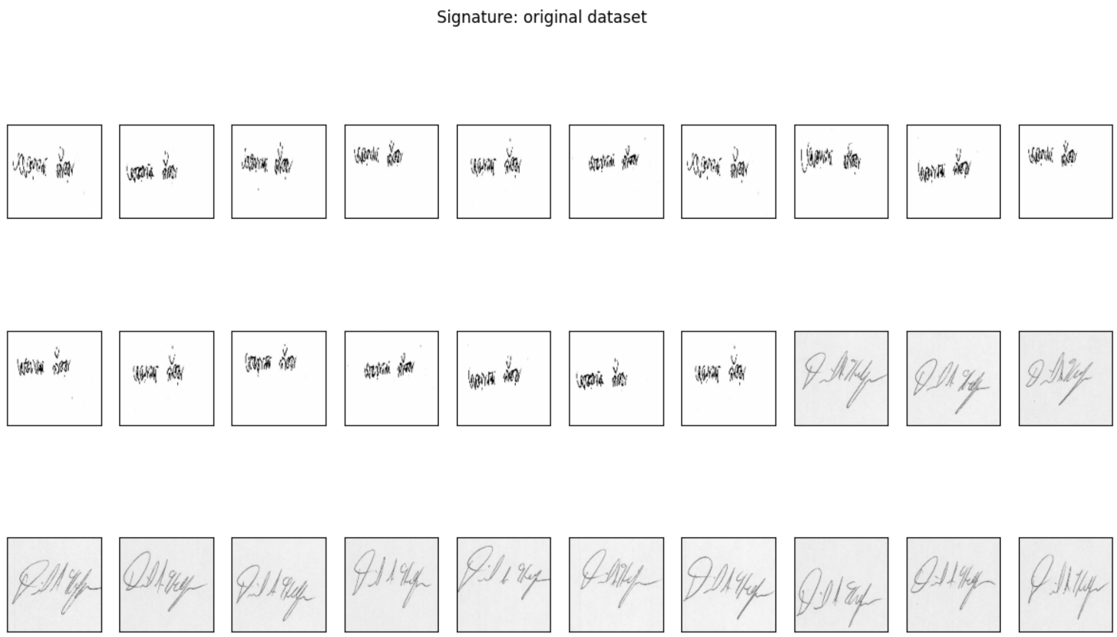

[6]:
```
def plot_samples(X, y, labels_dict, n=50):
    """
    Creates a gridplot for desired number of images (n) from the specified set

    Parameters:
        X (numpy array): Array of images.
        y (numpy array): Array of corresponding labels.
        labels_dict (dict): Dictionary mapping label indices to class names.
        n (int): Number of images to plot. Default is 50.

    Returns:
        None
    """
    for index in range(len(labels_dict)):
        imgs = X[np.argwhere(y == index)][:n]
        j = 10
        i = int(n/j)

        plt.figure(figsize=(15,8))
        c = 1
        for img in imgs:
```

```
      plt.subplot(i,j,c)
      plt.imshow(img[0])

      plt.xticks([])
      plt.yticks([])
      c += 1
    plt.suptitle('Signature: {}'.format(labels_dict[index]))

    plt.show()
```

[7]:
```
# Plot the no and yes class images
plot_samples(X_train, y_train, labels, 30)
```

Signature: fraud dataset






# 4 CNN Model

The Convolutional Neural Network (CNN) model employed in this project was custom-built specifically for the task of fake signature detection. Unlike transfer learning approaches which utilize pre-trained architectures, this model was designed from scratch to suit the characteristics of the signature dataset and the complexity of the detection task.

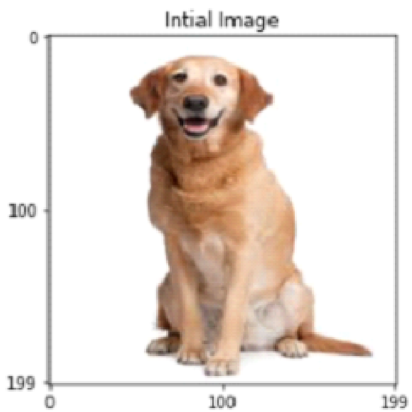
While transfer learning offers advantages in certain scenarios, custom CNN models allow for greater flexibility and tailoring to the specific problem domain. By constructing a custom architecture, the model can effectively learn features relevant to distinguishing between genuine and fake signatures, potentially yielding superior performance.

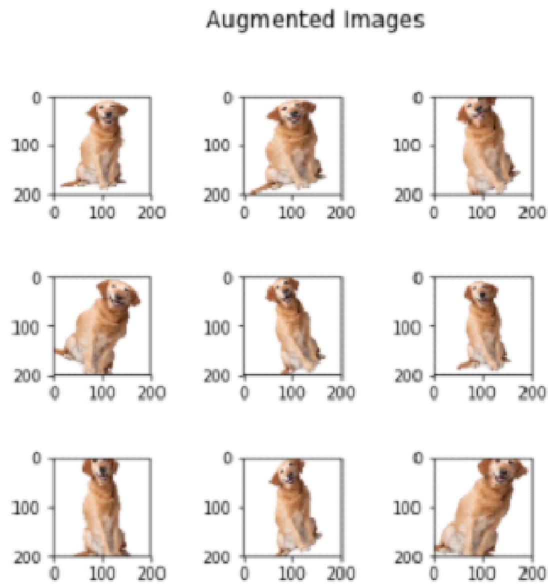
## 4.1 Data Augmentation

To address the challenge of having a small dataset, the technique of Data Augmentation was employed. Data Augmentation helps to "increase" the effective size of the training set by generating additional variations of the existing images.

### 4.1.1 Demo

Below is an example demonstrating the effect of data augmentation on a single image:




As shown in the demo, various transformations such as rotation, flipping, and scaling are applied to the original image, resulting in augmented versions that capture different perspectives and variations. This augmentation process enriches the training data and improves the model's ability to generalize to unseen examples.

For more insights into the implementation and benefits of Data Augmentation, refer to the following resource: Building Powerful Image Classification Models Using Very Little Data.

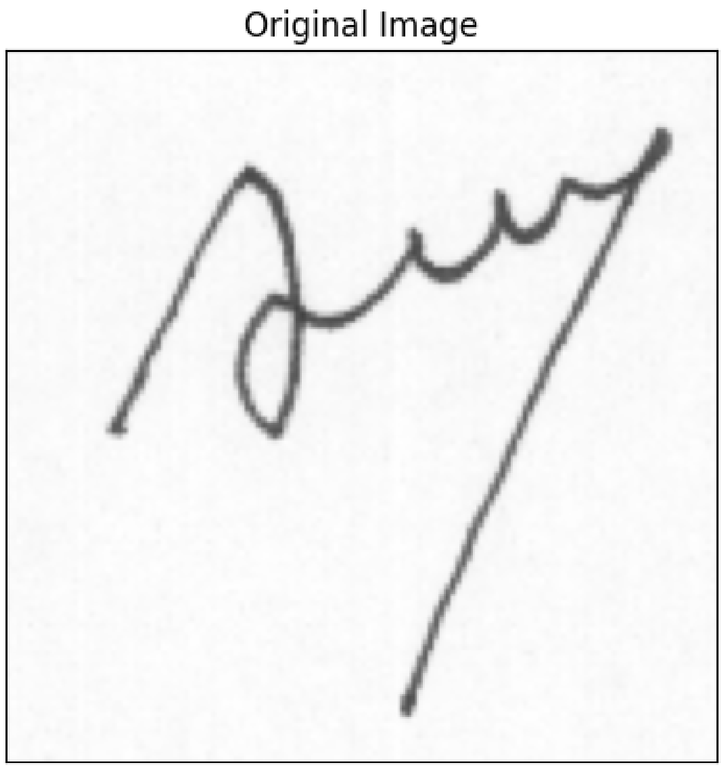
[8]:
```
# Define an ImageDataGenerator for augmentation
datagen = ImageDataGenerator(
    rotation_range=5,                # Degree range for random rotations
    rescale=1./255,                  # Rescaling factor
)
```

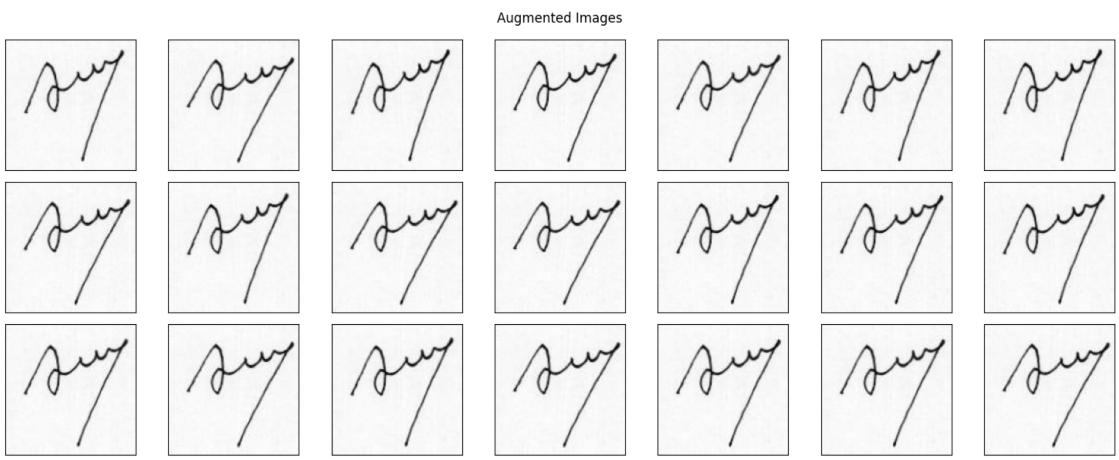
[9]:
```
# Remove the 'preview' directory and its contents if it exists
shutil.rmtree('preview', ignore_errors=True)

# Define the directory to save augmented images
os.makedirs('preview', exist_ok=True)

# Take an example image from X_train for augmentation
x = X_train[1001]

# Reshape the image to (1, height, width, channels) to fit the flow method of␣
  ↪ImageDataGenerator
x = x.reshape((1,) + x.shape)

# Generate and save augmented images
i = 0
```

```python
for batch in datagen.flow(x, batch_size=1, save_to_dir='preview',␣
 ↪save_prefix='aug_img', save_format='jpg'):
    i += 1
    if i > 20:
        break
```

[10]:

```python
# Display the original image
plt.imshow(X_train[1001])  # Display original image
plt.xticks([])  # Hide x-axis ticks
plt.yticks([])  # Hide y-axis ticks
plt.title('Original Image')  # Set title
plt.show()  # Show the plot

# Display augmented images
plt.figure(figsize=(15, 6))  # Set figure size
i = 1  # Counter for subplot
for img_filename in os.listdir('preview/'):  # Iterate through augmented images␣
 ↪in the 'preview' directory
    img = cv2.imread('preview/' + img_filename)  # Read augmented image
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # Convert BGR to RGB
    plt.subplot(3, 7, i)  # Create subplot
    plt.imshow(img)  # Display augmented image
    plt.xticks([])  # Hide x-axis ticks
    plt.yticks([])  # Hide y-axis ticks
    i += 1  # Increment counter
    if i > 3 * 7:  # Break loop if enough images have been displayed
        break
plt.suptitle('Augmented Images')  # Set main title
plt.tight_layout()  # Adjust layout

# Save the figure as a PNG file
plt.savefig("assets/augmented_images_preview.png")

# Show the plot
plt.show()
```




### 4.1.2 Apply

[11]:
```
# Define the directory paths for training and validation data
TRAIN_DIR = 'split_data/train/'
VAL_DIR = 'split_data/val/'
```

```python
# Create data generators for training and validation data
train_generator = datagen.flow_from_directory(
    TRAIN_DIR,
    color_mode='rgb',                   # Color mode: 'rgb' for 3-channel color␣
 ↪images
    target_size=IMG_SIZE,               # Target size of images after resizing
    batch_size=32,                      # Batch size for training
    class_mode='binary',                # Class mode: 'binary' for binary␣
 ↪classification (0 or 1)
    seed=RANDOM_SEED                    # Seed for random number generator
)

validation_generator = datagen.flow_from_directory(
    VAL_DIR,
    color_mode='rgb',                   # Color mode: 'rgb' for 3-channel color␣
 ↪images
    target_size=IMG_SIZE,               # Target size of images after resizing
    batch_size=16,                      # Batch size for validation
    class_mode='binary',                # Class mode: 'binary' for binary␣
 ↪classification
    seed=RANDOM_SEED                    # Seed for random number generator
)
```

```
Found 4890 images belonging to 2 classes.
Found 611 images belonging to 2 classes.
```

### 4.2 Model Building

The process of constructing the custom Convolutional Neural Network (CNN) model for fake signature detection involved several key steps:

1. **Architecture Design**: Designing the architecture of the CNN model involved determining the number of layers, their types (e.g., convolutional, pooling), and their configurations (e.g., filter size, number of filters).

2. **Layer Configuration**: Each layer of the CNN was configured to extract and transform features from the input signature images. This included specifying parameters such as kernel size, activation functions, and dropout rates to prevent overfitting.

3. **Model Compilation**: Once the architecture and layer configurations were defined, the model was compiled with appropriate loss functions, optimizers, and evaluation metrics. For binary classification tasks like fake signature detection, binary cross-entropy loss and Adam optimizer are commonly used.

4. **Model Training**: The compiled model was trained on the training dataset, consisting of genuine and fake signature images. During training, the model learned to differentiate between genuine and fake signatures by adjusting its weights based on the input data and the defined loss function.

5. **Validation**: Periodic validation of the model's performance was conducted using a separate

validation dataset. This allowed for monitoring of the model's progress and early detection of overfitting.

6. **Hyperparameter Tuning**: Fine-tuning of hyperparameters, such as learning rate and batch size, was performed to optimize the model's performance and generalization ability.

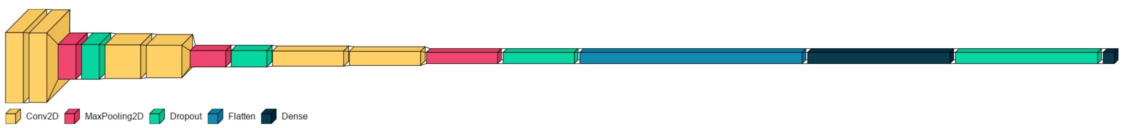
[12]:
```
# Create a Sequential model
model = Sequential()

# Add a convolutional layer
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu',␣
↪input_shape=(128, 128, 3)))
# Add a convolutional layer
model.add(Conv2D(filters=32, kernel_size=(3,3), activation='relu'))
# Add a max pooling layer
model.add(MaxPooling2D(pool_size=(2,2)))
# Add a dropout layer to reduce overfitting
model.add(Dropout(rate=0.2))

# Add a convolutional layer
model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
# Add a convolutional layer
model.add(Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
# Add a max pooling layer
model.add(MaxPooling2D(pool_size=(2,2)))
# Add a dropout layer to reduce overfitting
model.add(Dropout(rate=0.2))

# Add a convolutional layer
model.add(Conv2D(filters=128, kernel_size=(3,3), activation='relu'))
# Add a convolutional layer
model.add(Conv2D(filters=128, kernel_size=(3,3), activation='relu'))
# Add a max pooling layer
model.add(MaxPooling2D(pool_size=(2,2)))
# Add a dropout layer to reduce overfitting
model.add(Dropout(rate=0.2))

# Flatten the output from the convolutional layers
model.add(Flatten())
# Add a fully connected layer with 256 neurons and a relu activation function
model.add(Dense(units=256, activation='relu'))
# Add a dropout layer to reduce overfitting
model.add(Dropout(rate=0.5))
# Add the output layer with a sigmoid activation function for binary␣
↪classification
model.add(Dense(units=1, activation='sigmoid'))

# Print a summary of the model architecture
```

```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 126, 126, 32)      896

 conv2d_1 (Conv2D)           (None, 124, 124, 32)      9248

 max_pooling2d (MaxPooling2  (None, 62, 62, 32)        0
 D)

 dropout (Dropout)           (None, 62, 62, 32)        0

 conv2d_2 (Conv2D)           (None, 60, 60, 64)        18496

 conv2d_3 (Conv2D)           (None, 58, 58, 64)        36928

 max_pooling2d_1 (MaxPoolin  (None, 29, 29, 64)        0
 g2D)

 dropout_1 (Dropout)         (None, 29, 29, 64)        0

 conv2d_4 (Conv2D)           (None, 27, 27, 128)       73856

 conv2d_5 (Conv2D)           (None, 25, 25, 128)       147584

 max_pooling2d_2 (MaxPoolin  (None, 12, 12, 128)       0
 g2D)

 dropout_2 (Dropout)         (None, 12, 12, 128)       0

 flatten (Flatten)           (None, 18432)             0

 dense (Dense)               (None, 256)               4718848

 dropout_3 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 1)                 257

=================================================================
Total params: 5006113 (19.10 MB)
Trainable params: 5006113 (19.10 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

[13]:
```python
from PIL import ImageFont  # Importing ImageFont from PIL library

font = ImageFont.truetype("arial.ttf", 16)  # Loading a TrueType font for visualization
visualkeras.layered_view(
    model, legend=True,
    font=font,
    scale_xy=1, scale_z=1,
    to_file="assets/model.png"
)
```

[13]:




[14]:
```python
# EarlyStopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',          # Monitor validation loss
    min_delta=0.001,             # Minimum change in the monitored quantity to qualify as an improvement
    patience=4,                  # Number of epochs with no improvement after which training will be stopped
    restore_best_weights=True,  # Restore model weights to the best iteration
    verbose=0                    # Verbosity mode (0: silent, 1: update messages)
)

# ReduceLROnPlateau callback
reduce_learning_rate = ReduceLROnPlateau(
    monitor='val_accuracy',   # Monitor validation accuracy
    patience=2,               # Number of epochs with no improvement after which learning rate will be reduced
    factor=0.5,               # Factor by which the learning rate will be reduced (new_lr = lr * factor)
    verbose=1                 # Verbosity mode (0: silent, 1: update messages)
)
```

[15]:
```python
model.compile(
    optimizer='adam',            # Using the Adam optimizer
    loss='binary_crossentropy', # Binary cross-entropy loss function for binary classification
    metrics=['accuracy']         # Evaluation metric to monitor during training
```

```
)
```

[16]:
```
history = model.fit(
    train_generator,                                   # Training data generator
    steps_per_epoch=100,                               # Number of steps␣
 ↪(batches) per epoch
    epochs=30,                                         # Number of epochs
    validation_data=validation_generator,              # Validation data␣
 ↪generator
    validation_steps=30,                               # Number of steps␣
 ↪(batches) for validation
    callbacks=[early_stopping, reduce_learning_rate],  # EarlyStopping callback
    verbose=1                                          # Verbosity mode (0:␣
 ↪silent, 1: update messages)
)
```

```
Epoch 1/30
100/100 [==============================] - 82s 808ms/step - loss: 0.6965 -
accuracy: 0.5307 - val_loss: 0.6816 - val_accuracy: 0.5437 - lr: 0.0010
Epoch 2/30
100/100 [==============================] - 80s 797ms/step - loss: 0.6563 -
accuracy: 0.6111 - val_loss: 0.6367 - val_accuracy: 0.6375 - lr: 0.0010
Epoch 3/30
100/100 [==============================] - 82s 822ms/step - loss: 0.6301 -
accuracy: 0.6327 - val_loss: 0.6249 - val_accuracy: 0.6438 - lr: 0.0010
Epoch 4/30
100/100 [==============================] - 80s 799ms/step - loss: 0.5724 -
accuracy: 0.7063 - val_loss: 0.5646 - val_accuracy: 0.6958 - lr: 0.0010
Epoch 5/30
100/100 [==============================] - 80s 799ms/step - loss: 0.5401 -
accuracy: 0.7289 - val_loss: 0.5330 - val_accuracy: 0.7146 - lr: 0.0010
Epoch 6/30
100/100 [==============================] - 80s 803ms/step - loss: 0.4805 -
accuracy: 0.7691 - val_loss: 0.4365 - val_accuracy: 0.8229 - lr: 0.0010
Epoch 7/30
100/100 [==============================] - 82s 824ms/step - loss: 0.4363 -
accuracy: 0.8069 - val_loss: 0.4236 - val_accuracy: 0.8083 - lr: 0.0010
Epoch 8/30
100/100 [==============================] - 84s 842ms/step - loss: 0.4141 -
accuracy: 0.8184 - val_loss: 0.3680 - val_accuracy: 0.8375 - lr: 0.0010
Epoch 9/30
100/100 [==============================] - 84s 839ms/step - loss: 0.3802 -
accuracy: 0.8269 - val_loss: 0.3747 - val_accuracy: 0.8292 - lr: 0.0010
Epoch 10/30
100/100 [==============================] - 81s 804ms/step - loss: 0.3385 -
accuracy: 0.8572 - val_loss: 0.3607 - val_accuracy: 0.8438 - lr: 0.0010
Epoch 11/30
```

```
100/100 [==============================] - 80s 801ms/step - loss: 0.3147 -
accuracy: 0.8694 - val_loss: 0.3154 - val_accuracy: 0.8646 - lr: 0.0010
Epoch 12/30
100/100 [==============================] - 80s 800ms/step - loss: 0.2894 -
accuracy: 0.8707 - val_loss: 0.2966 - val_accuracy: 0.8687 - lr: 0.0010
Epoch 13/30
100/100 [==============================] - 82s 822ms/step - loss: 0.2796 -
accuracy: 0.8803 - val_loss: 0.3011 - val_accuracy: 0.8875 - lr: 0.0010
Epoch 14/30
100/100 [==============================] - 80s 800ms/step - loss: 0.2536 -
accuracy: 0.8891 - val_loss: 0.2382 - val_accuracy: 0.9021 - lr: 0.0010
Epoch 15/30
100/100 [==============================] - 80s 801ms/step - loss: 0.2278 -
accuracy: 0.9089 - val_loss: 0.2764 - val_accuracy: 0.8896 - lr: 0.0010
Epoch 16/30
100/100 [==============================] - 80s 803ms/step - loss: 0.2140 -
accuracy: 0.9120 - val_loss: 0.2002 - val_accuracy: 0.9250 - lr: 0.0010
Epoch 17/30
100/100 [==============================] - 80s 803ms/step - loss: 0.1828 -
accuracy: 0.9289 - val_loss: 0.1929 - val_accuracy: 0.9167 - lr: 0.0010
Epoch 18/30
100/100 [==============================] - ETA: 0s - loss: 0.1956 - accuracy:
0.9195
Epoch 18: ReduceLROnPlateau reducing learning rate to 0.0005000000237487257.
100/100 [==============================] - 83s 830ms/step - loss: 0.1956 -
accuracy: 0.9195 - val_loss: 0.2338 - val_accuracy: 0.9042 - lr: 0.0010
Epoch 19/30
100/100 [==============================] - 86s 858ms/step - loss: 0.1557 -
accuracy: 0.9359 - val_loss: 0.1953 - val_accuracy: 0.9292 - lr: 5.0000e-04
Epoch 20/30
100/100 [==============================] - 87s 871ms/step - loss: 0.1385 -
accuracy: 0.9477 - val_loss: 0.2032 - val_accuracy: 0.9187 - lr: 5.0000e-04
Epoch 21/30
100/100 [==============================] - ETA: 0s - loss: 0.1187 - accuracy:
0.9540
Epoch 21: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
100/100 [==============================] - 81s 808ms/step - loss: 0.1187 -
accuracy: 0.9540 - val_loss: 0.2879 - val_accuracy: 0.8896 - lr: 5.0000e-04
```

### 4.3 Model Performance

The performance of the custom Convolutional Neural Network (CNN) model for fake signature detection was evaluated using various metrics to assess its effectiveness in distinguishing between genuine and fake signatures. Key aspects of model performance include:

1. **Accuracy**: Accuracy measures the proportion of correctly classified instances out of the total number of instances. It provides an overall assessment of the model's correctness in identifying genuine and fake signatures.

2. **Precision and Recall**: Precision measures the proportion of true positive predictions out of all positive predictions, while recall measures the proportion of true positive predictions out of all actual positive instances. These metrics provide insights into the model's ability to minimize false positives and false negatives, respectively.

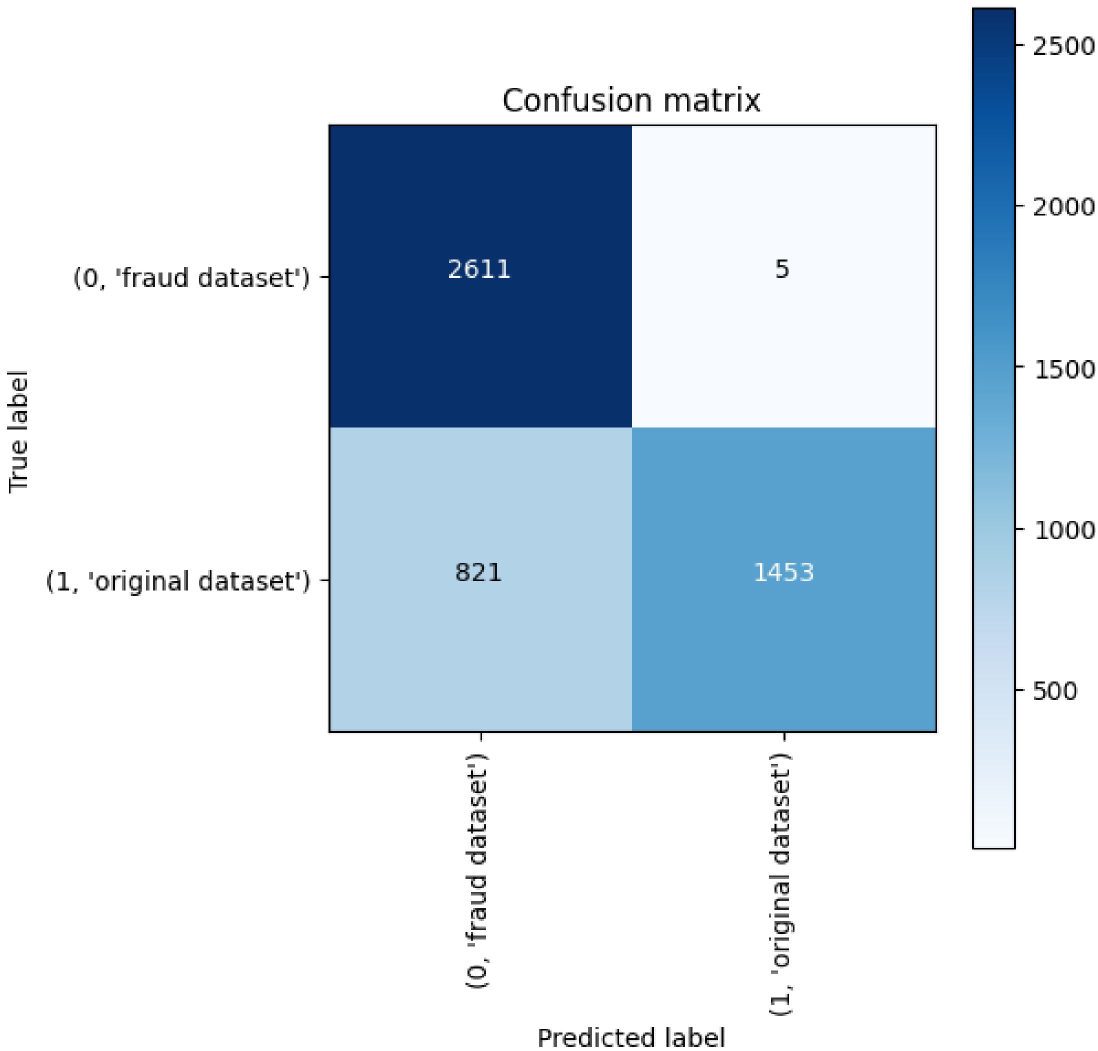
3. **F1 Score**: The F1 score is the harmonic mean of precision and recall, providing a balanced measure of the model's performance that considers both false positives and false negatives.

4. **Confusion Matrix**: The confusion matrix visualizes the performance of the model by summarizing the number of true positive, true negative, false positive, and false negative predictions. It offers a detailed understanding of the model's classification results.

5. **ROC Curve and AUC Score**: Receiver Operating Characteristic (ROC) curve plots the true positive rate against the false positive rate, illustrating the trade-off between sensitivity and specificity. The Area Under the ROC Curve (AUC) provides a single scalar value representing the model's ability to discriminate between genuine and fake signatures across different threshold settings.

[17]:
```
# Validate on train set
predictions_train = model.predict(X_train)  # Make predictions on training set
predictions_train = [1 if x > 0.5 else 0 for x in predictions_train]  # Convert␣
 ↪probabilities to binary predictions

# Calculate accuracy
accuracy_train = accuracy_score(y_train, predictions_train)  # Calculate␣
 ↪accuracy of predictions
print('Train Accuracy = %.2f' % accuracy_train)  # Print training accuracy

# Compute confusion matrix
confusion_mtx_train = confusion_matrix(y_train, predictions_train)  # Compute␣
 ↪confusion matrix for training set

# Plot confusion matrix
cm_train = plot_confusion_matrix(confusion_mtx_train, classes=list(labels.
 ↪items()), normalize=False)  # Plot confusion matrix for training set
```

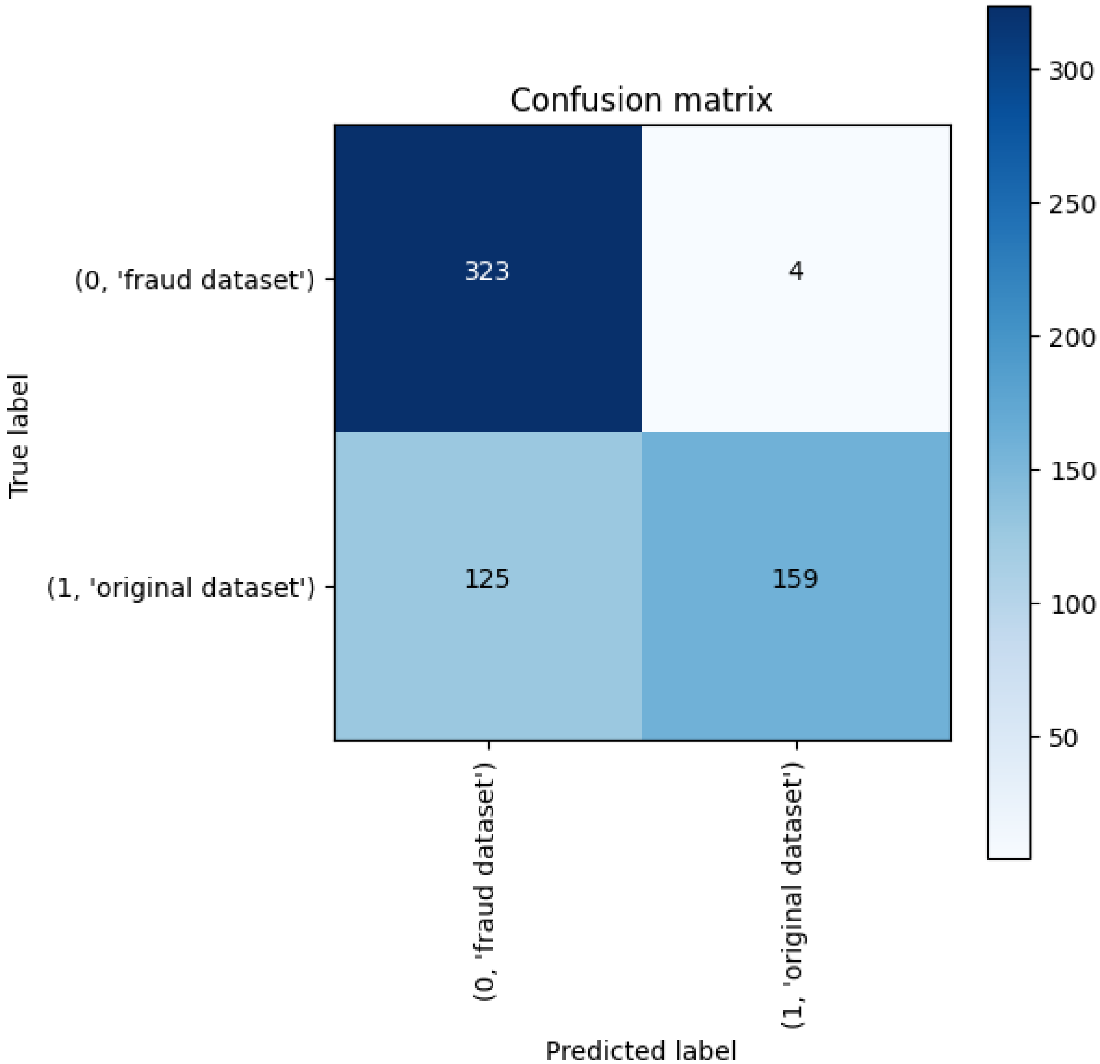
```
153/153 [==============================] - 16s 105ms/step
Train Accuracy = 0.83
```




[18]:
```
# Validate on validation set
predictions_val = model.predict(X_val)  # Make predictions on validation set
predictions_val = [1 if x > 0.5 else 0 for x in predictions_val]  # Convert␣
↪probabilities to binary predictions

# Calculate accuracy
accuracy_val = accuracy_score(y_val, predictions_val)  # Calculate accuracy of␣
↪predictions
print('Validation Accuracy = %.2f' % accuracy_val)  # Print validation accuracy

# Compute confusion matrix
confusion_mtx_val = confusion_matrix(y_val, predictions_val)  # Compute␣
↪confusion matrix for validation set
```

```
# Plot confusion matrix
cm_val = plot_confusion_matrix(confusion_mtx_val, classes=list(labels.items()),␣
 ↪normalize=False)  # Plot confusion matrix for validation set
```

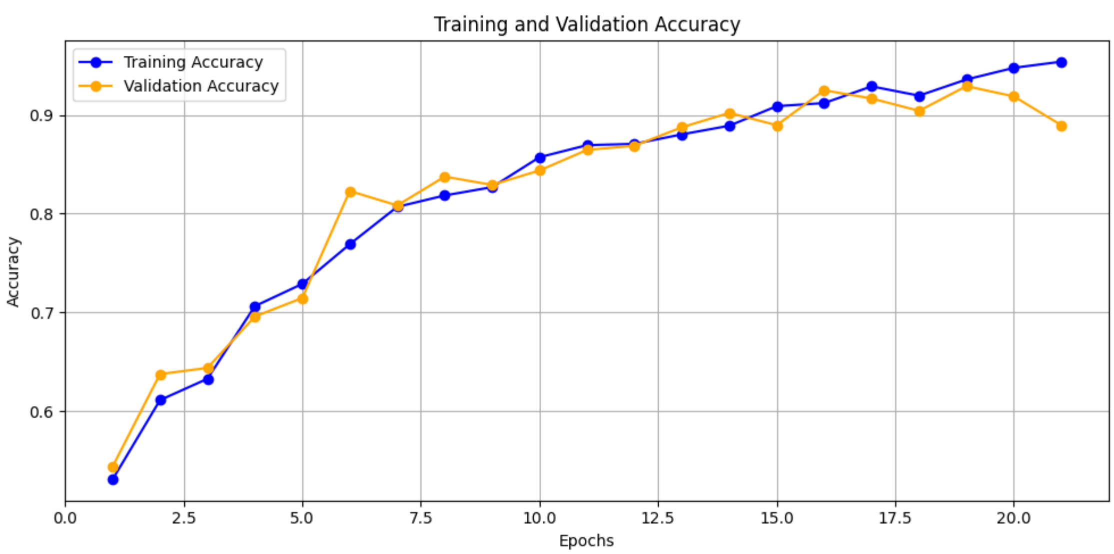
```
20/20 [==============================] - 2s 97ms/step
Validation Accuracy = 0.79
```



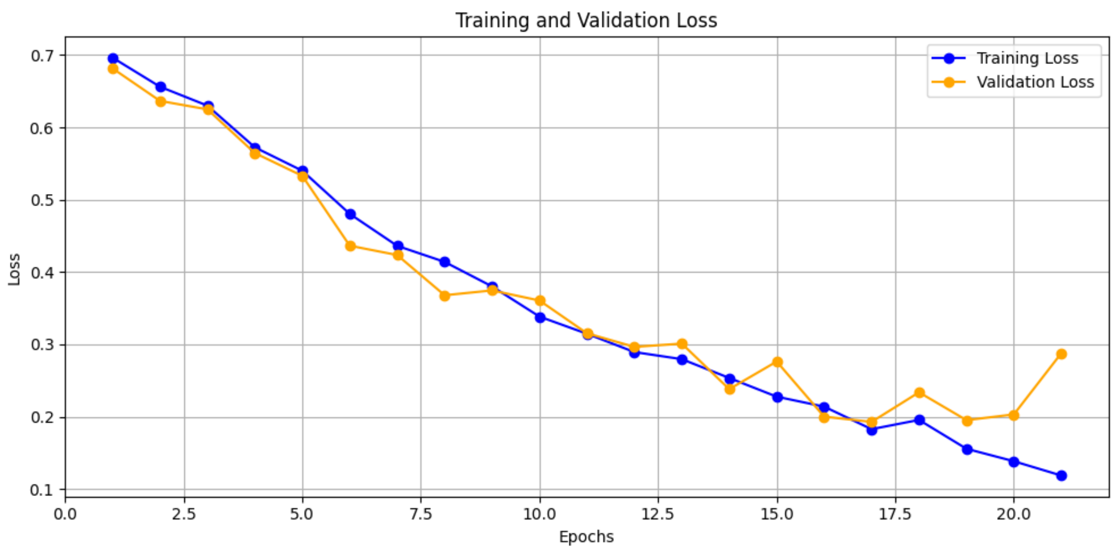

[19]:
```
# Plot model performance
acc = history.history['accuracy']  # Training accuracy
val_acc = history.history['val_accuracy']  # Validation accuracy
loss = history.history['loss']  # Training loss
val_loss = history.history['val_loss']  # Validation loss
epochs_range = range(1, len(history.epoch) + 1)  # Number of epochs
```

[20]:

```
# Plot accuracy
plt.figure(figsize=(10, 5))

# Plot training accuracy
plt.plot(epochs_range, acc, label='Training Accuracy', marker='o', color='blue')

# Plot validation accuracy
plt.plot(epochs_range, val_acc, label='Validation Accuracy', marker='o',␣
↪color='orange')

# Title and labels
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

# Add legend
plt.legend()

# Add grid
plt.grid(True)

# Adjust layout
plt.tight_layout()

# Save the plot as a PNG file
plt.savefig("assets/training_validation_accuracy.png")

# Show the plot
plt.show()
```

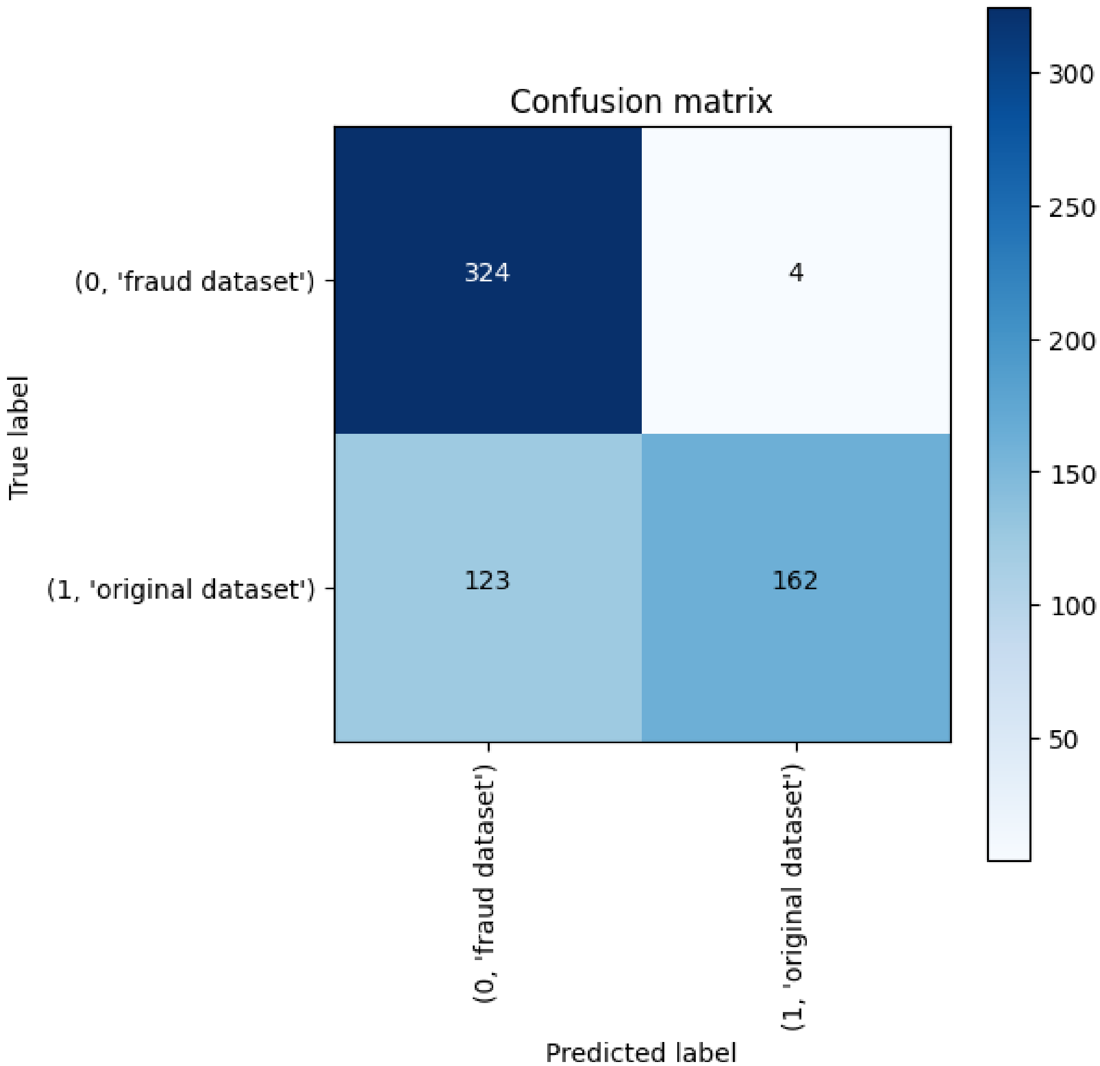


[21]:
```
# Plot loss
plt.figure(figsize=(10, 5))

# Plot training loss
plt.plot(epochs_range, loss, label='Training Loss', marker='o', color='blue')

# Plot validation loss
plt.plot(epochs_range, val_loss, label='Validation Loss', marker='o',␣
 ↪color='orange')

# Title and labels
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')

# Add legend
plt.legend()

# Add grid
plt.grid(True)

# Adjust layout
plt.tight_layout()

# Save the plot as a PNG file
plt.savefig("assets/training_validation_loss.png")

# Show the plot
plt.show()
```



[22]:
```
# Validate on test set
predictions_test = model.predict(X_test)  # Make predictions on test set
predictions_test = [1 if x > 0.5 else 0 for x in predictions_test]  # Convert␣
↪probabilities to binary predictions

# Calculate accuracy
accuracy_test = accuracy_score(y_test, predictions_test)  # Calculate accuracy␣
↪of predictions
print('Test Accuracy = %.2f' % accuracy_test)  # Print test accuracy

# Compute confusion matrix
confusion_mtx_test = confusion_matrix(y_test, predictions_test)  # Compute␣
↪confusion matrix for test set

# Plot confusion matrix
cm_test = plot_confusion_matrix(confusion_mtx_test, classes=list(labels.
↪items()), normalize=False)  # Plot confusion matrix for test set
```

```
20/20 [==============================] - 2s 99ms/step
Test Accuracy = 0.79
```




## 5 Conclusions

In conclusion, the fake signature detection project represents a significant advancement in combating signature forgery and fraudulent activities. Through the implementation of a custom Convolutional Neural Network (CNN) model, the project has achieved promising results in accurately distinguishing between genuine and fake signatures.

The model's performance, as measured by accuracy and other evaluation metrics, indicates its effectiveness in identifying fraudulent signatures with a high degree of confidence. However, there remains room for further improvement through continued refinement of the model architecture, hyperparameter tuning, and augmentation of the training dataset.

Overall, the project demonstrates the potential of machine learning and computer vision techniques in addressing real-world challenges such as financial fraud and identity theft. By leveraging advanced algorithms and methodologies, future iterations of the fake signature detection system can contribute to enhancing security measures and protecting individuals and organizations from fraudulent activities.

[23]:
```
# save the model
model.save('model/signature-detection-with-datagen.keras')
```

[24]:
```
# save the model
model.save('model/signature-detection-with-datagen.h5')
```

[ ]:
```

```