K L UNIVERSITY

COMPUTER SCIENCE ENGINEERING DEPARTMENT

A Project Based Lab Report

On

Classifying Audio Samples using convolutional neural networks.

Object Detection Using OpenCV and TensorFlow

Categorizing Images of Clothing with Convolutional Neural Networks

SUBMITTED BY:

I.D NUMBER	NAME
2000032182	M SAI GANGADHAR
2000031554	N LOHITH SAI

UNDER THE ESTEEMED GUIDANCE OF

Dr. Vishnuvardhan Mannava

<Associate Professor>

KL UNIVERSITY

Green fields, Vaddeswaram – 522 502
Guntur Dt., AP, India.

DEPARTMENT OF BASIC ENGINEERING SCIENCES

.

CERTIFICATE

This is to certify that the project based laboratory report entitled

"Classifying Audio Samples using convolutional neural networks, "Object Detection Using OpenCV and TensorFlow", "Categorizing Images of Clothing with Convolutional Neural Networks" submitted by **M SAI GANGADHAR** bearing Regd.No.200032182, **N LOHITH SAI** bearing Regd.No.2000031554 to the **Department of Computer Science Engineering Sciences, KL University** in partial fulfillment of the requirements for the completion of a project in "AIDS" course in II B Tech II Semester, is a bonafide record of the work carried out by him/her under my supervision during the academic year 2021-22.

PROJECT SUPERVISOR

Dr. Vishnuvardhan Mannava

HEAD OF THE DEPARTMENT

Dr. Hari Kiran Vege

Professor

ACKNOWLEDGEMENTS

It is great pleasure for me to express my gratitude to our honorable President **Sri. Koneru Satyanarayana**, for giving the opportunity and platform with facilities in accomplishing the project based laboratory report.

I express the sincere gratitude to our director **Dr. A Jagadeesh** for his administration towards our academic growth.

I express sincere gratitude to our Coordinator and HOD-CSE **Dr.Hari Kiran Vege** for her leadership and constant motivation provided in successful completion of our academic semester. I record it as my privilege to deeply thank for providing us the efficient faculty and facilities to make our ideas into reality.

I express my sincere thanks to our project supervisor **Dr VishnuVardhan Mannava** for his/her novel association of ideas, encouragement, appreciation and intellectual zeal which motivated us to venture this project successfully.

Finally, it is pleased to acknowledge the indebtedness to all those who devoted themselves directly or indirectly to make this project report success.

	NAME	REGD.NO
1.	M SAI GANGADHAR	2000032182
2.	N LOHITH SAI	2000031554

ABSTRACT

Purpose

This notebook serves as an introduction to working with audio data for classification problems; it is meant as a learning resource rather than a demonstration of the state-of-the-art. The techniques mentioned in this notebook apply not only to classification problems, but to regression problems and problems dealing with other types of input data as well. I focus particularly on feature engineering techniques for audio data and provide an in-depth look at the logic, concepts, and properties of the Multilayer Perceptron (MLP) model, an ancestor and the origin of deep neural networks (DNNs) today. I also provide an introduction to a few key machine learning models and the logic in choosing their hyperparameters. These objectives are framed by the task of recognizing emotion from snippets of speech audio.

1) Classifying Audio Samples using convolutional neural networks.

Data cleansing and feature engineering comprise the most crucial aspect of preparing machine and deep learning models alike and is often the difference between success and failure. We can drastically improve the performance of a model with proper attention paid to feature engineering. This stands for input data which is already useable for predictions; even such data can be transformed in myriad ways to improve predictive performance. For features to be useful in classification they must encompass sufficient variance between different classes. We can further improve the performance of our models by understanding the influence of and precisely tuning their hyperparameters, for which there are algorithmic aids such as Grid Search.

Network architecture is a critical factor in determining the computational complexity of DNNs; often, however, simpler models with just one hidden layer perform better than more complicated models. The importance of proper model evaluation cannot be overstressed: training data should be used strictly for training a model, validation data strictly

for tuning a model, and test data strictly to evaluate a model once it is tuned - a model should never be tuned to perform better on test data. To this end, K-Fold Cross Validation is a staple tool. Finally, the Random Forest ensemble model makes a robust benchmark model suitable to less-than-clean data with unkown distribution, especially when strapped for time and wishing to evaluate the useability of features extracted from a dataset.

Intro: Speech Emotion Recognition on the RAVDESS dataset

In this notebook we explore the most common machine learning models, specifically those available off the shelf in scikit-learn. After benchmarking performance with machine learning models, we train a Multilayer Perceptron (MLP) model for classification in an attempt to recognize the emotion conveyed in a speech audio snippet. MLP classifiers are a good DNN model to start with because they are simple, flexible, and suited when inputs are assigned a label - in our case, emotion.

I'm going to use the RAVDESS dataset (Ryerson Audio-Visual Database of Emotional Speech and Song dataset), created by Steven Livingstone and Frank Russo of Ryerson University.
Details of the RAVDESS dataset
Download the dataset used in this notebook
Scroll half-way down the page and find "Audio_Speech_Actors_01-24"

We're going to use the audio-only speech portion of the RAVDESS dataset, ~200MB. Audio is sourced from 24 actors (12 male, 12 female) repeating two sentences with a variety of emotions and intensity. We get 1440 speech files (24 actors * 60 recordings per actor). Each audio sample has been rated by a human 10 times for emotional quality.

CODE:

Classifying Audio Samples using convolutional neural networks.

SECTION -2

PROJECT NUMBER -30

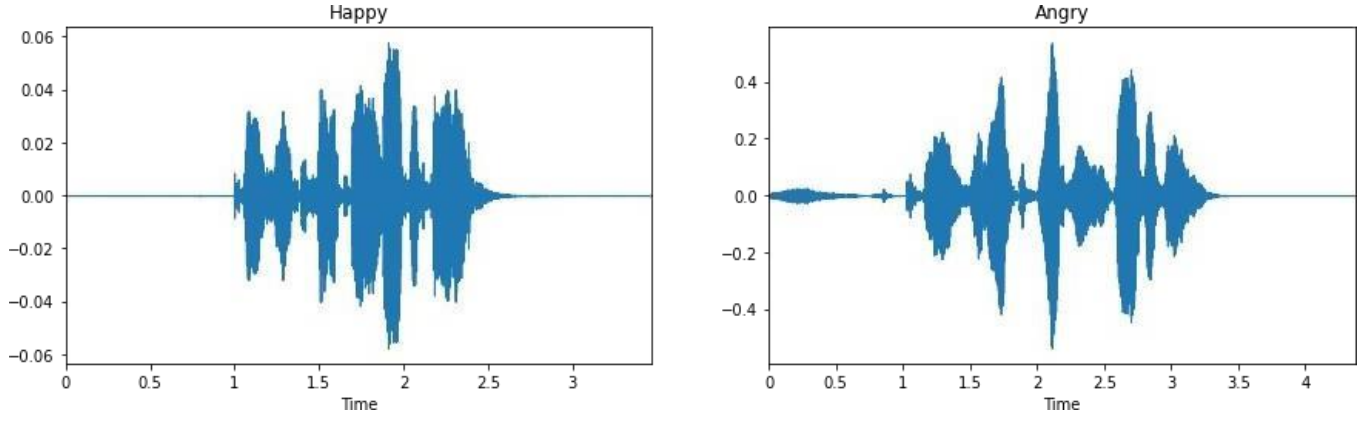
In [2]:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import librosa.display
import soundfile
import os
import warnings; warnings.filterwarnings('ignore')
from IPython.core.display import HTML

HTML("""
<style>
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
</style>
""")
```

Out[2]:

In [3]:

```
with soundfile.SoundFile('.\\RAVDESS dataset\\actor_01\\03-01-03-01-01-01-01.wav') as audio:
    waveform = audio.read(dtype="float32")
    sample_rate = audio.samplerate
    plt.figure(figsize=(15,4))
    plt.subplot(1, 2, 1)
    librosa.display.waveplot(waveform, sr=sample_rate)
    plt.title('Happy')

with soundfile.SoundFile('.\\RAVDESS dataset\\actor_01\\03-01-05-02-01-02-01.wav') as audio:
    waveform = audio.read(dtype="float32")
    sample_rate = audio.samplerate
    plt.subplot(1, 2, 2)
    librosa.display.waveplot(waveform, sr=sample_rate)
    plt.title('Angry')
```



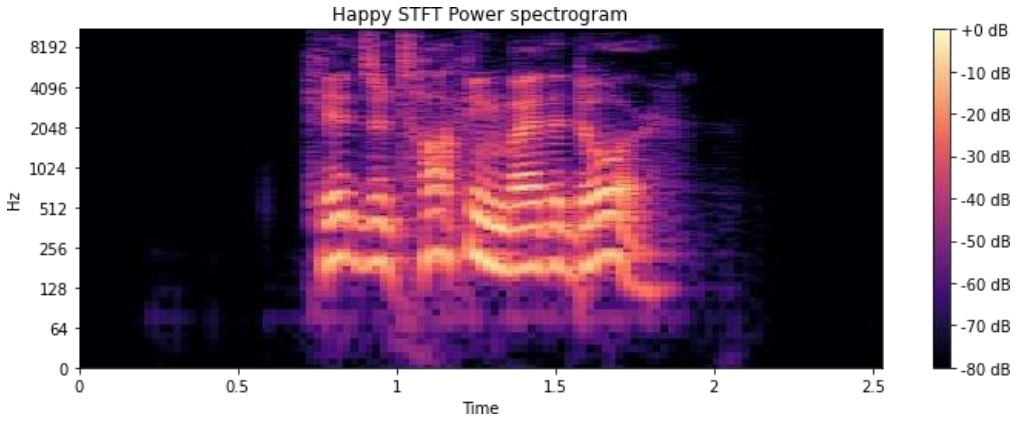
In [4]:

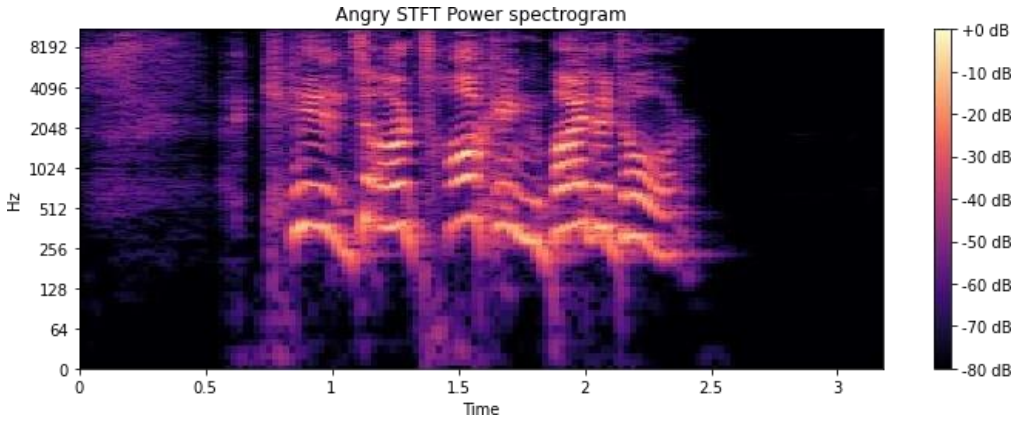
```
with soundfile.SoundFile('.\\RAVDESS dataset\\actor_01\\03-01-03-01-01-01-01.wav') as audio:
    happy_waveform = audio.read(dtype="float32")
    sample_rate = audio.samplerate

with soundfile.SoundFile('.\\RAVDESS dataset\\actor_01\\03-01-05-02-01-02-01.wav') as audio:
    angry_waveform = audio.read(dtype="float32")
    # same sample rate
```

In [5]:

```
stft_spectrum_matrix = librosa.stft(happy_waveform)
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.amplitude_to_db(np.abs(stft_spectrum_matrix), ref=np.max),y_axis='log', x_axis=
plt.title('Happy STFT Power spectrogram')
plt.colorbar(format='%+2.0f dB')
plt.tight_layout()
```

```
stft_spectrum_matrix = librosa.stft(angry_waveform)
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.amplitude_to_db(np.abs(stft_spectrum_matrix), ref=np.max),y_axis='log', x_axis=
plt.title('Angry STFT Power spectrogram')
plt.colorbar(format='%+2.0f dB')
plt.tight_layout()
```

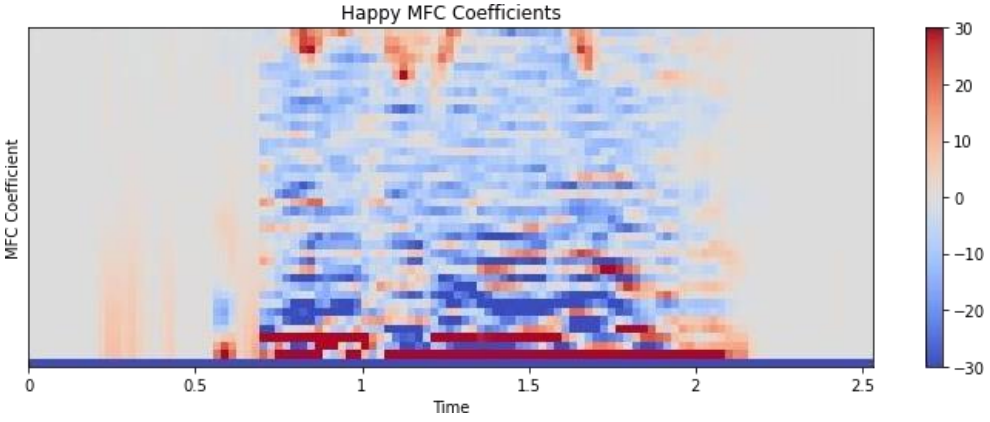


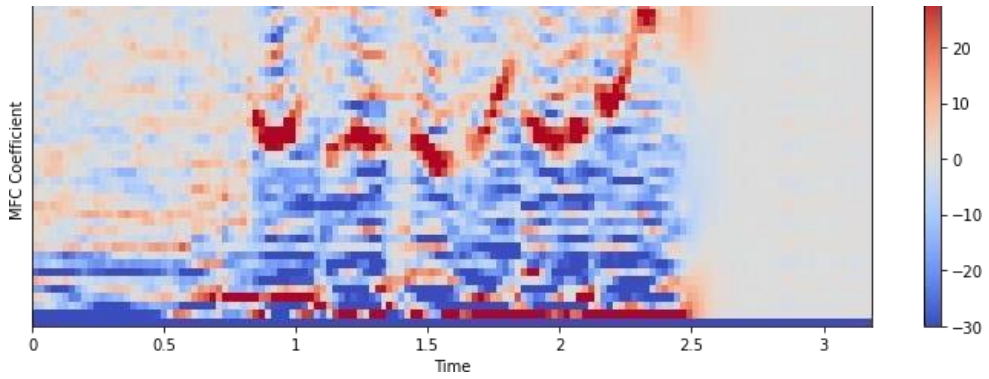


Visualize our sample's MFC coefficients w.r.t time:

In [142…

```
from matplotlib.colors import Normalize
mfc_coefficients = librosa.feature.mfcc(y=happy_waveform, sr=sample_rate, n_mfcc=40)
plt.figure(figsize=(10, 4))
librosa.display.specshow(mfc_coefficients, x_axis='time',norm=Normalize(vmin=-30,vmax=30))
plt.colorbar()
plt.yticks(())
plt.ylabel('MFC Coefficient')
plt.title('Happy MFC Coefficients')
plt.tight_layout()

mfc_coefficients = librosa.feature.mfcc(y=angry_waveform, sr=sample_rate, n_mfcc=40)
plt.figure(figsize=(10, 4))
librosa.display.specshow(mfc_coefficients, x_axis='time',norm=Normalize(vmin=-30,vmax=30))
plt.colorbar()
plt.yticks(())
plt.ylabel('MFC Coefficient')
plt.title('Angry MFC Coefficients')
plt.tight_layout()
```





Positive MFCCs correspond to low-frequency regions of the cepstrum, and negative MFFCs to high-frequency.

We see the Angry voice has a much greater proportion of positive MFFCs, corresponding to a lower voice pitch compared to the Happy voice. Makes sense - a happy voice carries a lighter tone and a higher pitch.

Quick MFCC Derivation

1. Take the square of magnitudes in spectrograms produced by STFT to produce audio power spectrograms for short overlapping frames of the audio signal,
2. Apply a mel-scale-based transformation, mel filterbanks (triangular window functions) to each STFT power spectrogram and sum the power in each filterbank
3. Take the log of each filterbank power
4. Take the discrete cosine transform of each log power in each STFT frame, giving us the MFC coefficients - our measure of power at various mel frequencies, corresponding to audible pitch.

Wikipedia has a pretty straight-forward explanation of MFCCs.

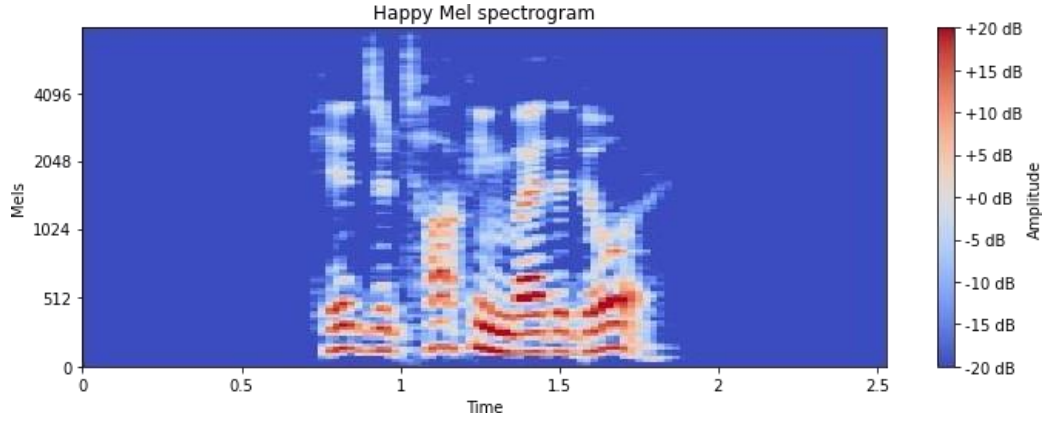
Mel Spectrograms and Mel-Frequency Cepstrums

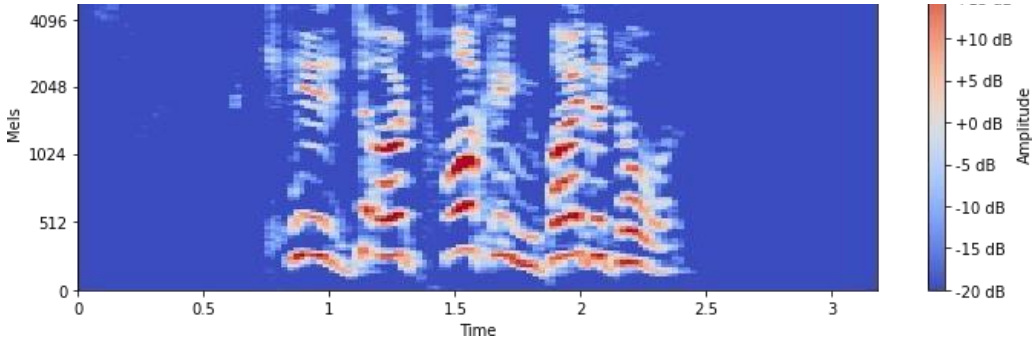
In deriving our MFCCs, we have also produced an additional feature we can make use of. When we mapped the frequencies of a power spectrogram to the mel scale, we produced a Mel Frequency Spectrogram - a simple analog of the power spectrogram with the frequency scale in mels. We're going to use the Mel Specotrgrogram as a feature of its own.

Visualize our sample's Mel spectrogram w.r.t time:

In [143…

```
melspectrogram = librosa.feature.melspectrogram(y=happy_waveform, sr=sample_rate, n_mels=128, fmax=8000)
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.power_to_db(S=melspectrogram, ref=np.mean),y_axis='mel',fmax=8000, x_axis='time
plt.colorbar(format='%+2.0f dB',label='Amplitude')
plt.ylabel('Mels')
plt.title('Happy Mel spectrogram')
plt.tight_layout()

melspectrogram = librosa.feature.melspectrogram(y=angry_waveform, sr=sample_rate, n_mels=128, fmax=8000)
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.power_to_db(S=melspectrogram, ref=np.mean),y_axis='mel',fmax=8000, x_axis='time
plt.colorbar(format='%+2.0f dB',label='Amplitude')
plt.ylabel('Mels')
plt.title('Angry Mel spectrogram')
plt.tight_layout()
```

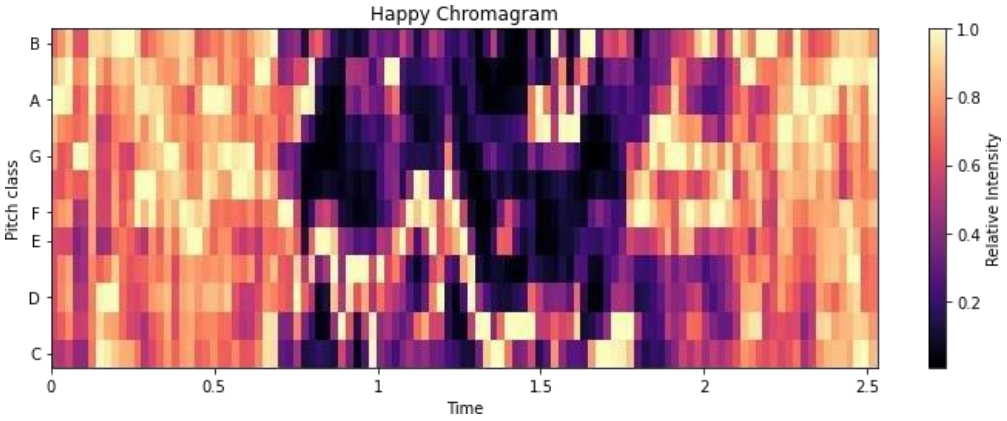


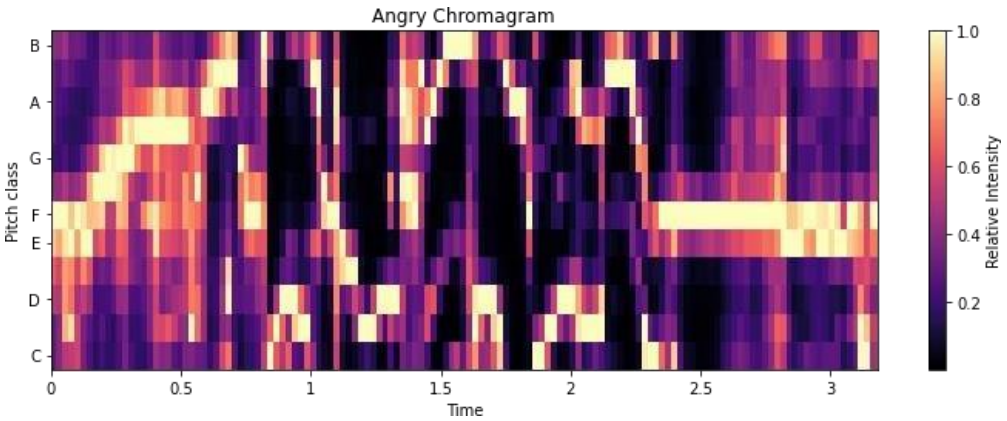


Visualize the chromagram for our sample audio:

In [144…

```
chromagram = librosa.feature.chroma_stft(y=happy_waveform, sr=sample_rate)
plt.figure(figsize=(10, 4))
librosa.display.specshow(chromagram, y_axis='chroma', x_axis='time')
plt.colorbar(label='Relative Intensity')
plt.title('Happy Chromagram')
plt.tight_layout()

chromagram = librosa.feature.chroma_stft(y=angry_waveform, sr=sample_rate)
plt.figure(figsize=(10, 4))
librosa.display.specshow(chromagram, y_axis='chroma', x_axis='time')
plt.colorbar(label='Relative Intensity')
plt.title('Angry Chromagram')
plt.tight_layout()
```





In [6]:

```
import librosa

def feature_chromagram(waveform, sample_rate):
        stft_spectrogram=np.abs(librosa.stft(waveform))

    chromagram=np.mean(librosa.feature.chroma_stft(S=stft_spectrogram, sr=sample_rate).T,axis=0)
    return chromagram

def feature_melspectrogram(waveform, sample_rate):
    melspectrogram=np.mean(librosa.feature.melspectrogram(y=waveform, sr=sample_rate, n_mels=128, fmax=8000).T,a
    return melspectrogram

def feature_mfcc(waveform, sample_rate):
```

```
    mfc_coefficients=np.mean(librosa.feature.mfcc(y=waveform, sr=sample_rate, n_mfcc=40).T, axis=0)
    return mfc_coefficients
```

We're going to wrap our feature extraction functions so we only have to load each audio file once. After extracting our 3 audio features as NumPy arrays representing a time series, we're going to stack them horizontally to create a single feature array.

In [7]:

```
def get_features(file):
    # load an individual soundfile
     with soundfile.SoundFile(file) as audio:
        waveform = audio.read(dtype="float32")
        sample_rate = audio.samplerate
        # compute features of soundfile
        chromagram = feature_chromagram(waveform, sample_rate)
        melspectrogram = feature_melspectrogram(waveform, sample_rate)
        mfc_coefficients = feature_mfcc(waveform, sample_rate)

        feature_matrix=np.array([])
        # use np.hstack to stack our feature arrays horizontally to create a feature matrix
        feature_matrix = np.hstack((chromagram, melspectrogram, mfc_coefficients))

        return feature_matrix
```

Load the Dataset and Compute Features

We have to understand the labelling of the RAVDESS dataset to find the ground truth emotion for each sample. Each file is labelled with 7 numbers delimited by a "-". Most of the numbers describe metadata about the audio samples such as their format (video and/or audio), whether the audio is a song or statement, which of two statements is being read and by which actor.

The third and fourth numbers pertain to the emotional quality of each sample. The third number is in the range of 1-8 with each number representing an emotion. The fourth number is either 1 or 2, representing normal (1) or strong (2) emotional intensity.

We're going to define a dictionary based on the third number (emotion) and assign an emotion to each number as specified by the RAVDESS dataset:

In [8]:

```
#Emotions in the RAVDESS dataset
emotions ={
  '01':'neutral',
  '02':'calm',
  '03':'happy',
  '04':'sad',
  '05':'angry',
  '06':'fearful',
  '07':'disgust',
  '08':'surprised'
}
```

Finally, let's load our entire dataset and compute the features of each audio file:

In [9]:

```
import os, glob

def load_data():
    X,y=[],[]
    count = 0
    for file in glob.glob(".\\RAVDESS dataset\\actor_*\\*.wav"):
        file_name=os.path.basename(file)
        emotion=emotions[file_name.split("-")[2]]
        features = get_features(file)
        X.append(features)
        y.append(emotion)
        count += 1
        # '\r' + end='' results in printing over same line
        print('\r' + f' Processed {count}/{1440} audio samples',end=' ')
    # Return arrays to plug into sklearn's cross-validation algorithms
    return np.array(X), np.array(y)
```

Compute the feature matrix and read the emotion labels for the entire dataset. Note that our regressor (independent/explanatory variable), usually denoted X, is named 'features', and our regressand (dependent variable), usually denoted y, is named 'emotions'.

In [10]:

```
features, emotions = load_data()
```

```
 Processed 1440/1440 audio samples
```

Let's see what the features we extracted look like:

In [150…

```
print(f'\nAudio samples represented: {features.shape[0]}')
print(f'Numerical features extracted per sample: {features.shape[1]}')
features_df = pd.DataFrame(features) # make it pretty for display
features_df
```

```
Audio samples represented: 1440
Numerical features extracted per sample: 180
```

Out[150…

	0	1	2	3	4	5	6	7	8	9	...	170	171	172	
0	0.633752	0.648747	0.621731	0.634556	0.660326	0.660486	0.700930	0.731556	0.746887	0.723435	...	-3.130589	-2.700900	-1.888213	-
1	0.660875	0.705912	0.703556	0.672467	0.693959	0.728635	0.716797	0.758791	0.785981	0.786871	...	-3.480977	-2.376257	-0.995905	-
2	0.716729	0.705105	0.707905	0.727850	0.695296	0.698327	0.743120	0.790097	0.815669	0.742015	...	-2.590883	-2.652148	-1.988785	-
3	0.725767	0.712236	0.700385	0.719994	0.744948	0.750953	0.744565	0.791981	0.806350	0.761700	...	-3.747972	-2.862656	-1.471904	-
4	0.603845	0.626068	0.696292	0.688255	0.679438	0.692962	0.699785	0.720104	0.734387	0.747514	...	-3.700608	-0.653383	-1.610423	-
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1435	0.598480	0.521324	0.491678	0.529918	0.559808	0.571009	0.620753	0.623964	0.632223	0.627766	...	-0.329260	-1.489990	-2.404569	-
1436	0.668894	0.686021	0.639824	0.610624	0.594836	0.562757	0.561363	0.609690	0.617460	0.612550	...	-1.158591	0.621641	-0.323034	-
1437	0.632550	0.612255	0.547708	0.537014	0.537187	0.581168	0.586129	0.561292	0.579341	0.593762	...	-1.348669	0.198006	-0.209769	-
1438	0.574530	0.544936	0.543283	0.532215	0.539510	0.599558	0.623495	0.597271	0.567244	0.591442	...	-1.365634	0.276205	0.135981	
1439	0.617133	0.565528	0.568652	0.559601	0.592577	0.633922	0.660309	0.689957	0.672458	0.687257	...	-0.113733	1.638723	-0.236629	

1440 rows × 180 columns

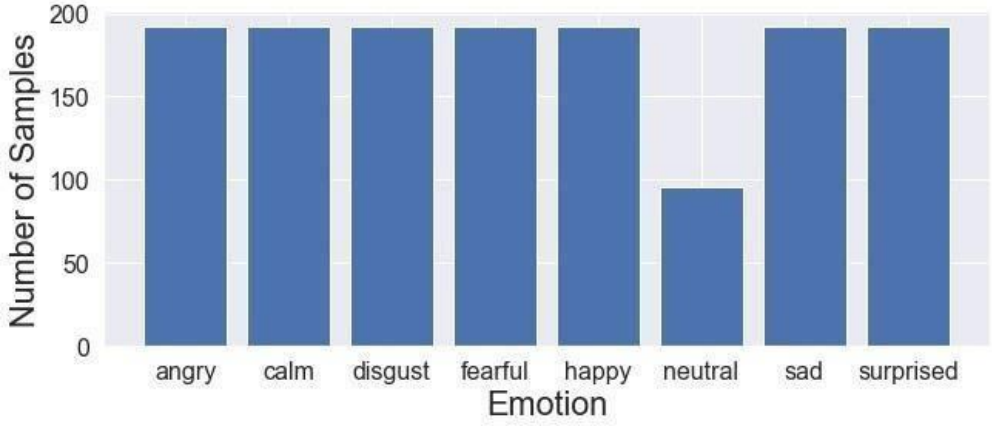
We have a matrix of dim 1440 x 180. Looks good - 1440 audio samples, one per row, with a series of 180 numerical features for each sample.

Each of the 1440 feature arrays has 180 features composed of 12 chromagram pitch classes + 128 mel spectrogram bands + 40 MFC coefficients.

Let's see the class balance of our dataset:

In [327…

```
plt.figure(figsize=(35,4))
plt.subplot(1,3,1)
emotion_list, count = np.unique(emotions, return_counts=True)
plt.bar(x=range(8), height=count)
plt.xticks(ticks=range(8), labels = [emotion for emotion in emotion_list],fontsize=10)
plt.xlabel('Emotion')
plt.tick_params(labelsize=16)
plt.ylabel('Number of Samples')
plt.show()
```



Great, the classes appear to be balanced. That makes the task easier. All emotions *except* the neutral class have a "strong" intensity so there are half as many neutral samples. That might have an impact.

2) Object Detection Using OpenCV and TensorFlow

The Model

We are going to use a model from the **"Tensorflow Hub"** library, which has multiple ready to deploy models trained in all kinds of datasets and to solve all kinds of problems. For our use, I filtered models trained for object detection tasks and models in the TFLite format. This format is usually used for IoT applications, for its small size and faster performance than bigger models

The chosen model was the EfficientDet-Lite2 Object detection model. It was trained on the COCO17 dataset with 91 different labels and optimized for the TFLite application. This model returns:

1. The box boundaries of the detection;
2. The detection scores (probabilities of a given class);
3. The detection classes;
4. The number of detections.

Detecting Objects

I'm going to divide this section into two parts: Detections on static images and detection on live webcam video.

Static Images

We will start by detecting objects in this image from Unsplash:

Basically, we used OpenCV to load and do a couple of transformations on the raw image to an RGB tensor in the model format.

Now we can load the model and the labels

Object Detection Using OpenCV and TensorFlow

SECTION -02

PROJECT NO:30

Live Camera

In [ ]:

```
import tensorflow_hub as hub
import cv2
import numpy
import tensorflow as tf
import pandas as pd

# Carregar modelos
detector = hub.load("https://tfhub.dev/tensorflow/efficientdet/lite2/detection/1")
labels = pd.read_csv('labels.csv',sep=';',index_col='ID')
labels = labels['OBJECT (2017 REL.)']

cap = cv2.VideoCapture(0)

width = 512
height = 512

while(True):
    #Capture frame-by-frame
    ret, frame = cap.read()

    #Resize to respect the input_shape
    inp = cv2.resize(frame, (width , height ))

    #Convert img to RGB
    rgb = cv2.cvtColor(inp, cv2.COLOR_BGR2RGB)

    #Is optional but i recommend (float convertion and convert img to tensor image)
    rgb_tensor = tf.convert_to_tensor(rgb, dtype=tf.uint8)

    #Add dims to rgb_tensor
    rgb_tensor = tf.expand_dims(rgb_tensor , 0)

    boxes, scores, classes, num_detections = detector(rgb_tensor)

    pred_labels = classes.numpy().astype('int')[0]

    pred_labels = [labels[i] for i in pred_labels]
    pred_boxes = boxes.numpy()[0].astype('int')
    pred_scores = scores.numpy()[0]
    #loop throughout the faces detected and place a box around it

    for score, (ymin,xmin,ymax,xmax), label in zip(pred_scores, pred_boxes, pred_labels):
        if score < 0.5:
            continue

        score_txt = f'{100 * round(score,0)}'
        img_boxes = cv2.rectangle(rgb,(xmin, ymax),(xmax, ymin),(0,255,0),1)
        font = cv2.FONT_HERSHEY_SIMPLEX
        cv2.putText(img_boxes,label,(xmin, ymax-10), font, 0.5, (255,0,0), 1, cv2.LINE_AA)
        cv2.putText(img_boxes,score_txt,(xmax, ymax-10), font, 0.5, (255,0,0), 1, cv2.LINE_AA)



    #Display the resulting frame
    cv2.imshow('black and white',img_boxes)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# When everything done, release the capture
cap.release()
cv2.destroyAllWindows()
```

In [ ]:

Static Image

In [25]:

```
import tensorflow_hub as hub
```

```
import cv2
import numpy
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
```

In [26]:
```
# Apply image detector on a batch of image.
detector = hub.load("https://tfhub.dev/tensorflow/efficientdet/lite2/detection/1")
```

In [27]:
```
width = 1080
height = 1028

#Load image by Opencv2
img = cv2.imread('image.jpg')
#Resize to respect the input_shape
inp = cv2.resize(img, (width , height ))

#Convert img to RGB
rgb = cv2.cvtColor(inp, cv2.COLOR_BGR2RGB)

#Is optional but i recommend (float convertion and convert img to tensor image)
rgb_tensor = tf.convert_to_tensor(rgb, dtype=tf.uint8)

#Add dims to rgb_tensor
rgb_tensor = tf.expand_dims(rgb_tensor , 0)

#Now you can use rgb_tensor to predict label for exemple :
```

In [29]:
```
plt.figure(figsize=(10,10))
plt.imshow(rgb)
```

Out[29]:
```
<matplotlib.image.AxesImage at 0x218df09d5e0>
```

In [30]:
```
boxes, scores, classes, num_detections = detector(rgb_tensor)
```

In [31]:
```
labels = pd.read_csv('labels.csv',sep=';',index_col='ID')
labels = labels['OBJECT (2017 REL.)']
```

In [32]:
```
labels.head()
```

Out[32]:
```
ID
```

```
1         person
2        bicycle
3            car
4     motorcycle
5       airplane
Name: OBJECT (2017 REL.), dtype: object
```

In [33]:

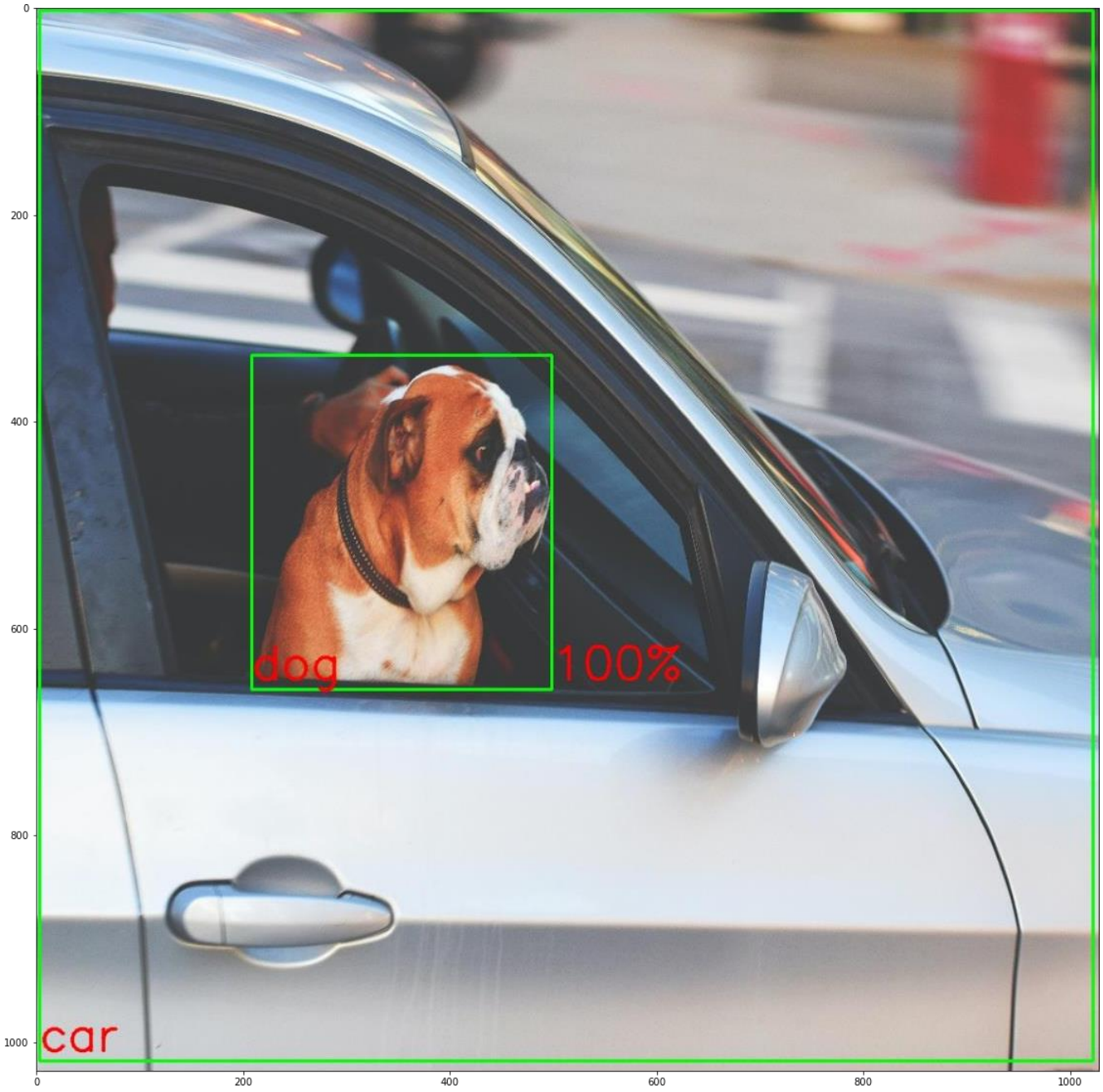
```
pred_labels = classes.numpy().astype('int')[0]
pred_labels = [labels[i] for i in pred_labels]
pred_boxes = boxes.numpy()[0].astype('int')
pred_scores = scores.numpy()[0]
```

In [34]:

```
for score, (ymin,xmin,ymax,xmax), label in zip(pred_scores, pred_boxes, pred_labels):
        if score < 0.5:
            continue

        score_txt = f'{100 * round(score)}%'
        img_boxes = cv2.rectangle(rgb,(xmin, ymax),(xmax, ymin),(0,255,0),2)
        font = cv2.FONT_HERSHEY_SIMPLEX
        cv2.putText(img_boxes, label,(xmin, ymax-10), font, 1.5, (255,0,0), 2, cv2.LINE_AA)
        cv2.putText(img_boxes,score_txt,(xmax, ymax-10), font, 1.5, (255,0,0), 2, cv2.LINE_AA)
```

In [35]:

```
plt.figure(figsize=(20,20))
plt.imshow(img_boxes)

plt.savefig('image_pred.jpg',transparent=True, )
```

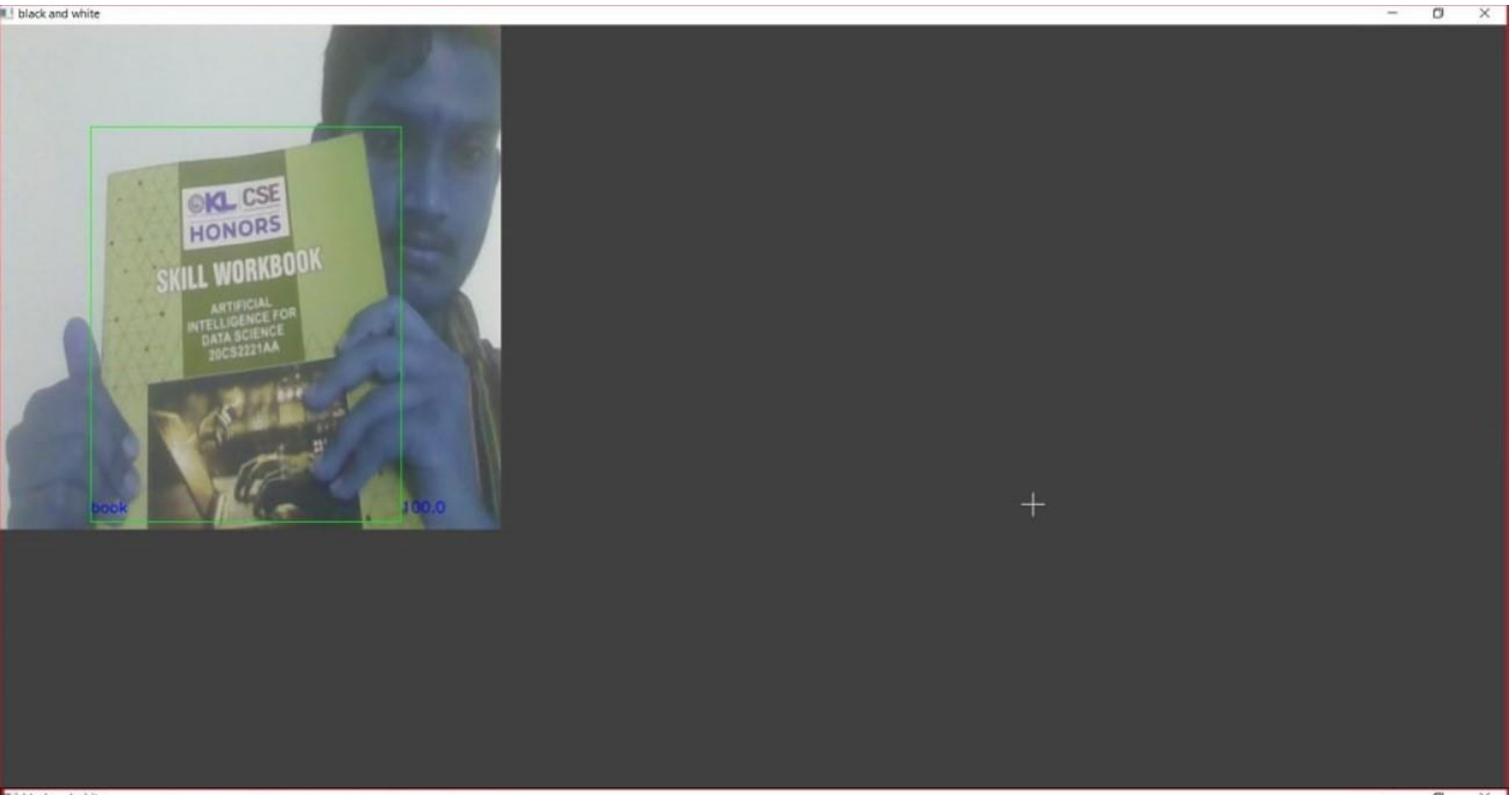


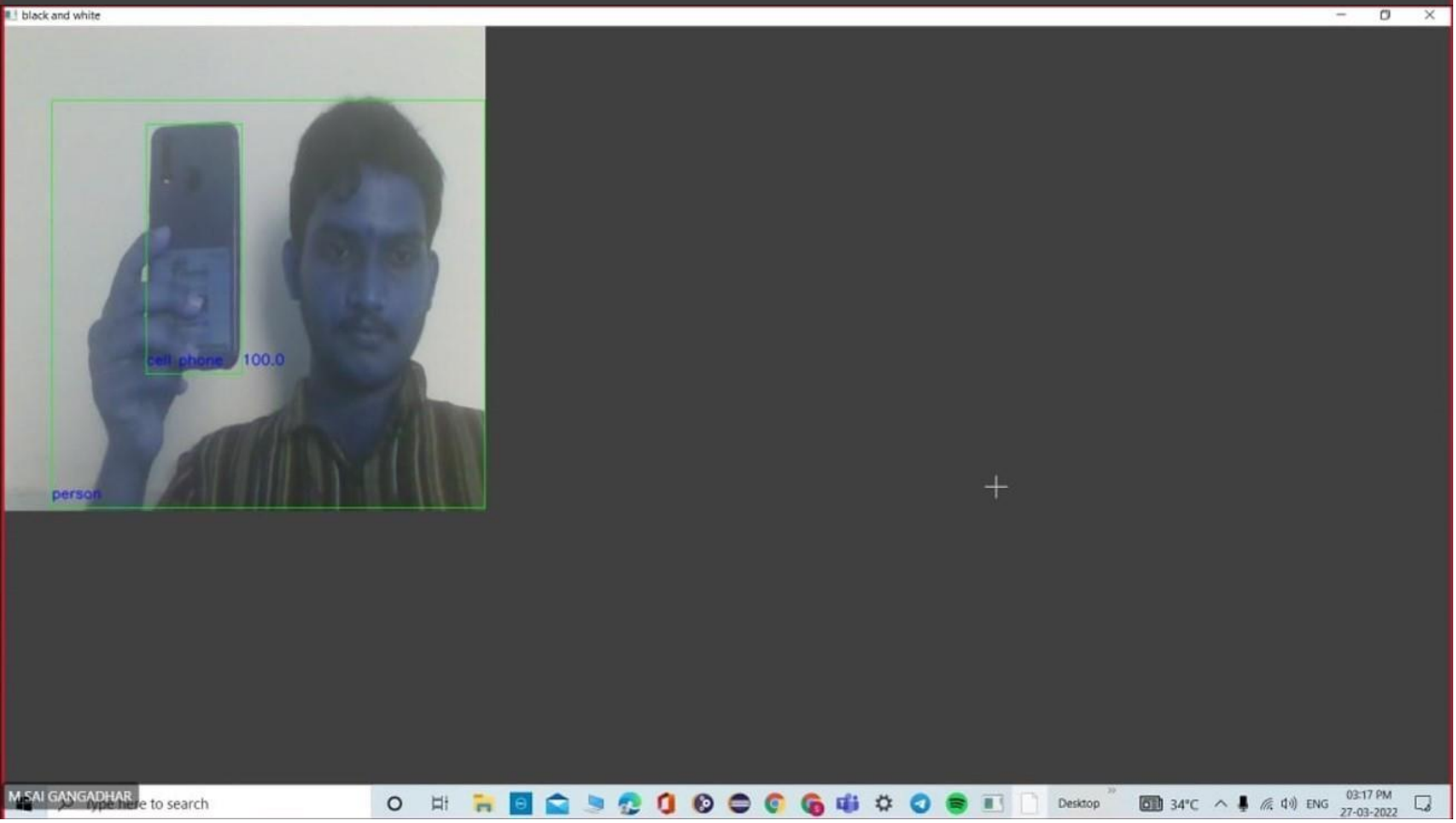
Live Webcam Video

Now we can move on to detecting objects live using the webcam on your pc.

This part is not as hard as it seems, we just have to insert the code we used for one image in a loop:

After Executing The Above Code Of Live Camera I got Output like this:





3) Categorizing Images of Clothing with Convolutional Neural Networks

Exploring the Dataset

We are going to use the Fashion MNIST dataset, which contains 70,000 greyscale images in 10 categories. The images represents individual clothing with 28 x 28 pixels of resolution.

Each image has a single class in the range [0, 9]. We can see in the table below the classes in the dataset.

An image is just a matrix of numbers, in our problem a 28 x 28 matrix. Each value is in the range [0, 255], which defines the color and intensity of each pixel.

Here we can see the first 5 rows from the first sample of our training dataset.

Preprocessing the Data

Normalizing

The value of each pixel in the image, an integer in the range [0, 255], needs to be normalized for the model to work properly. We can create a function that divides each value by 255.0. When applying this function to our dataset, we will get normalized values in the range [0,1].

Reshaping the Image

The Convolutional Neural Network expects 4 dimensions as input: number of samples (60,000), pixels (28 x 28), and color channel. Since we are working with greyscale images there is only a single channel. However, as we have seen before, the shape of our X_train dataset is (6000, 28, 28) and we need (6000, 28, 28, 1) as input. Thus, we need to reshape our datasets.

One-Hot Encoding

Our class data has labels in the range [0, 9], which is called *Integer Encoding*. However, there is no ordinal relationship between the labels and the corresponding class.

In this case, using the integer encoding allows the model to assume a natural ordering between categories, which may result in poor performance or unexpected results from the Deep Learning model. To solve this problem, we can use a one-hot encode, which creates a new binary variable for each unique integer value[2].

Now, let's check how are our label data. Each label changed from a single value to a vector with value "1" in the respective position.

Creating the Model

Building the Layers

The first step in the model creation is to define the layers of our network. The CNN has at least one convolutional layer and also includes other types of layers, such as pooling layers and fully

connected layers (dense). For this project, we are going to use a typical CNN architecture represented in the image below.

we will include a convolutional and a pooling layers, then another convolutional and pooling layers. Then, we are going to add a flatten layer to transform our 2d-array image in a 1d-array and add some dense layers. We can add some dropout layers to reduce overfitting. For the last layer, we add a dense layer with the number of classes from our problem (10) and a softmax activation, which creates the probability distribution for each class.

Compiling the Model

The next step is to compile the model. Here we pass the optimizer, which adjusts the weights to minimize the loss, the loss function, which measures the disparity between the true and predicted values, and the metrics, a function used to measure the performance of the model.

Training the Model

The last step is to train our model. Here we need to pass the input data, the target data and the number of epochs, which defines the number of full iterations of the training dataset. We will also pass a parameter to split our data in training (70%) and validation (30%) and a parameter to define the batch_size, which is the number of training examples in each pass.

We will save the results of our training in the variable *model history*

Evaluating the Loss

The fit method returns a history object with the results for each epoch. We can plot a chart with the loss and accuracy for the training and validation datasets. From this chart, it is possible to see how the loss goes down and the accuracy goes up over the epochs. This chart is also used to identify evidence of overfitting and underfitting. For our model, it doesn't seem we have strong evidence of these problems. Thus, let's move on and make some predictions with our test dataset.

Making Predictions and Evaluating the Results

Evaluating the Accuracy in the Test Dataset

Now, let's see how the model performs with our test dataset.

Making Predictions

We can use our model to predict a class for each example in our test database.

Now, let's plot a few images from our test dataset with the true and the predicted labels. When the model predicts right, the text will be displayed in blue, if the prediction is wrong, it will be displayed in red. Also, it will be displayed the calculated probability for the predicted class.

From these 20 examples, we can see that our model made wrong predictions for one coat and one shirt. However, it is not feasible to

analyze the predictions for 10,000 examples using this plot. Thus, let's plot a crosstab to analyze what our model predicted right and wrong for each class.

Crosstab

Analyzing our crosstab, we can notice that our best accuracy was achieved for the Sandals classification (99,1%), while our lowest accuracy was for the Shirts (76,9%). The crosstab provides a great way to visualize the quantities predicted by our model for each class. We can easily see, for example, that it predicted Shirt as T-shirt/Top for 74 examples or that it didn't predict any Bag as Ankle boot.

Classification Report

Now, let's plot a summary with the precision, recall and, f1-score for each class using the classification report from the scikit-learn library. Here it is possible to see that we didn't have a considerate disparity between precision and recall for any classes. As we noticed in the crosstab, our worst result is for the Shirt class.

Conclusions

In this project, it was presented how to train a Convolutional Neural Network to classify images of clothing from the Fashion MNIST dataset using TensorFlow and Keras. Using this model, we got an overall accuracy of 91,55% in our test dataset, which is a good result. However, specifically for our *Shirt* class we got an accuracy of only

76,90%. We could try to improve the accuracy of this class using some data augmentation techniques.

CODE:

Classifying Images of Clothing Using TensorFlow

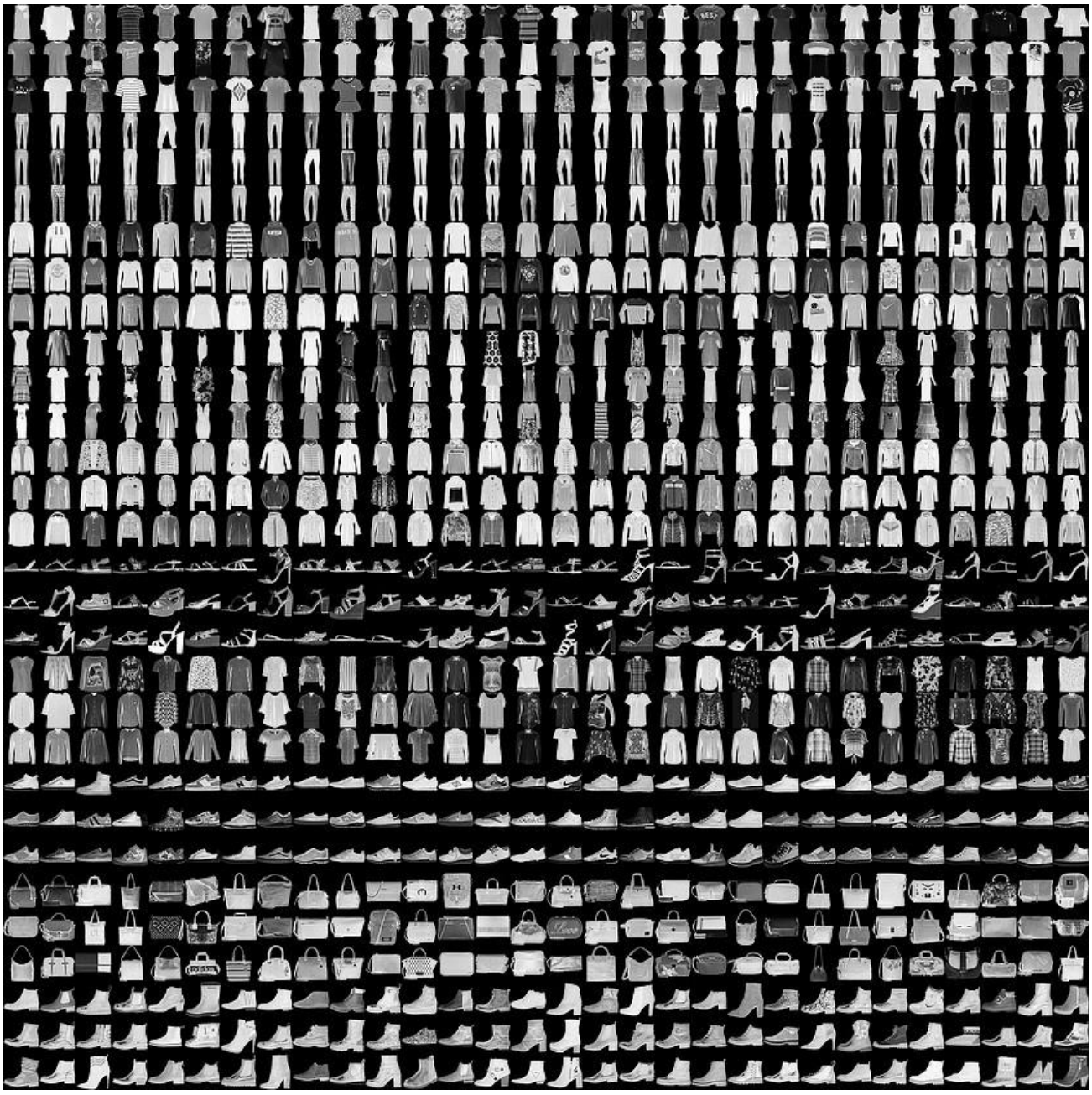
How to train a Deep Learning model to classify images of clothing using Convolutional Neural Networks in TensorFlow.

Deep Learning is a subfield of machine learning that uses multi-layered neural networks to extracts patterns from data. My objective within this project is to present how to apply Deep Learning concepts to an image classification problem. For this, we are going to train a Convolutional Neural Network (CNN) to classify a dataset of clothing using the TensorFlow library in Python.

In the last few years, Convolutional Neural Networks has been achieving superhuman performance on some complex visual tasks. They power image search services, self-driving cars, automatic video classification systems, and more1.

Understanding and Importing the Dataset

We are going to use the Fashion MNIST dataset, which contains 70,000 greyscale images of clothing in 10 categories, like shirts, dresses, and sandals. The images represent individual clothing with 28 x 28 pixels of resolution. You can see some examples of them in the picture below.



Each image has a single class in the range [0, 9]. We can see in the table below the classes in the dataset.

In [1]:

```
# importing the libraries
import tensorflow  as  tf
from tensorflow import keras
import matplotlib.pyplot as plt
```

```
import numpy as np
import pandas as pd
from sklearn.metrics import classification_report
```

The dataset can be loaded directly from Keras, using the Datasets API. It is split in train (60,000 samples) and test (10,000 samples)

In [2]:
```
(X_train_raw, y_train_raw), (X_test_raw, y_test_raw) = keras.datasets.fashion_mnist.load_data()
```

We can check the shape of the datasets. As expected, we have 60,000 samples in the training dataset and 10,000 in the test dataset, each of them with 28 x 28 pixels (X_train) and 1 class (y_train)

In [3]:
```
print("X_train: ", X_train_raw.shape)
print("y_train: ", y_train_raw.shape)
print("X_test: ", X_test_raw.shape)
print("y_test:  ", y_test_raw.shape)
```

```
X_train:  (60000, 28, 28)
y_train:  (60000,)
X_test:   (10000, 28, 28)
y_test:   (10000,)
```

In the class datasets, we have only the label for each class. We can define a new variable with the names for each label, as presented in the table above. This will be used when plotting some images with the corresponding class.

In [4]:
```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankl
```

Exploring the Dataset

An image is just a matrix of numbers, in our problem a 28 x 28 matrix. Each value is in the range [0, 255], which defines the color and intensity of each pixel.

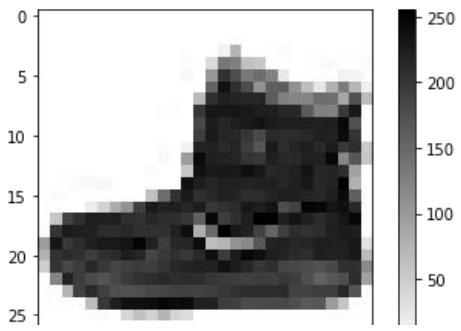
Here we can see the first 5 rows from the first sample of our training dataset.

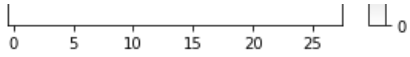
In [5]:
```
X_train_raw[0][:5]
```

Out[5]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,
          0,   0,  13,  73,   0,   0,   1,   4,   0,   0,   0,   0,   1,
          1,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
          0,  36, 136, 127,  62,  54,   0,   0,   0,   1,   3,   4,   0,
          0,   3]], dtype=uint8)
```

Now, let's plot this image using the matplotlib library. Since our image is in greyscale, we can pass a binary argument for the color map.

In [6]:
```
plt.imshow(X_train_raw[0], cmap=plt.cm.binary)
plt.colorbar()
plt.show()
```





Then, we can confirm which class our image represents.

In [7]:
```
print("Class label:  ",  y_train_raw[0])
print("Class name:  ", class_names[y_train_raw[0]])
```

```
Class label:  9
Class name:   Ankle boot
```

We can display a few examples from our train dataset with the respective class. Plotting the first 24 samples, it is possible to see at least one example of each class.

In [8]:
```
plt.figure(figsize=(16,6))
for i in range(24):
    plt.subplot(3,8,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(X_train_raw[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train_raw[i]])
plt.show()
```

Preprocessing the Data

Normalizing

The value of each pixel in the image, an integer in the range [0, 255], needs to be normalized for the model to work properly. We can create a function that divides each value by 255.0. When applying this function to our dataset, we will get normalized values in the range [0,1]. Furthermore, we can transform it to 'float32' to reduce memory usage.

In [9]:
```
def normalize(X):
  X = (X / 255.0).astype('float32')
  return X
```

In [10]:
```
X_train = normalize(X_train_raw)
X_test = normalize(X_test_raw)
```

Reshaping the Image

The Convolutional Neural Network expects 4 dimensions as input: number of samples (60,000), pixels (28 x 28), and color channel. Since we are working with greyscale images there is only a single channel. However, as we have seen before, the shape of our X_train dataset is (6000, 28, 28) and we need (6000, 28, 28, 1) as input. Thus, we need to reshape our datasets.

In [11]:
```
# redimensionar as imagens
```

```
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], X_train.shape[2], 1))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], X_test.shape[2], 1))

print("X_train: ", X_train.shape)
print("X_test:  ", X_test.shape)
```

```
X_train:  (60000, 28, 28, 1)
X_test:   (10000, 28, 28, 1)
```

One-Hot Encoding

Our class data has labels in the range [0, 9], which is called Integer Encoding.

In [12]:
```
np.unique(y_train_raw)
```

Out[12]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

However, there is no ordinal relationship between the labels and the corresponding class.

In this case, using the integer encoding allows the model to assume a natural ordering between categories, which may result in poor performance or unexpected results from the Deep Learning model. To solve this problem, we can use a one-hot encode, which creates a new binary variable for each unique integer value2.

In [13]:
```
y_train = keras.utils.to_categorical(y_train_raw,10)
y_test = keras.utils.to_categorical(y_test_raw,10)
```

Now, let's check how are our label data. Each label changed from a single value to a vector with value "1" in the respective position.

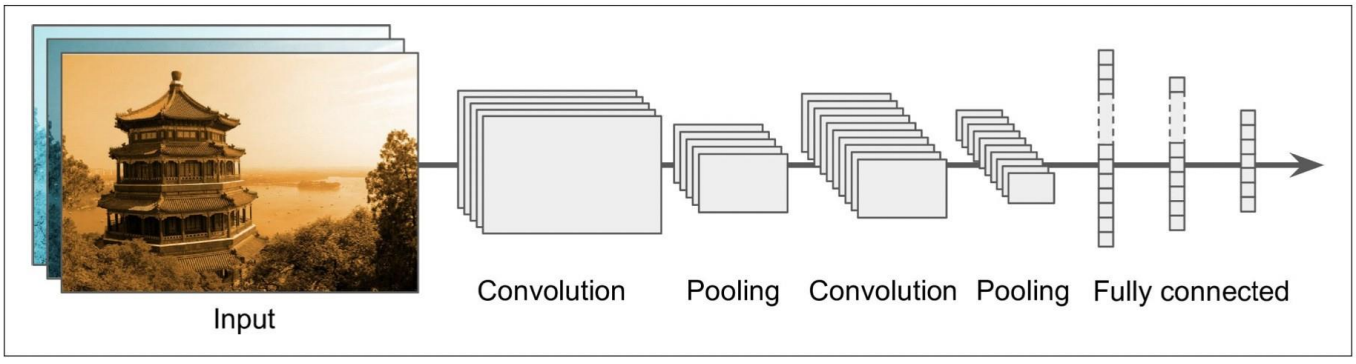
In [14]:
```
print("First Label Before One-Hot Encoding: ", y_train_raw[0])
print("First Label After One-Hot Encoding:  ", y_train[0])
```

```
First Label Before One-Hot Encoding:  9
First Label After One-Hot Encoding:   [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

Creating the Model

Building the Layers

The first step in the model creation is to define the layers of our network. The CNN has at least one convolutional layer and also includes other types of layers, such as pooling layers and fully connected layers (dense). For this project, we are going to use a typical CNN architecture represented in the image below



As we have in the image, we will include a convolutional and a pooling layers, then another convolutional and pooling layers. Then, we are going to add a flatten layer to transform our 2d-array image in a 1d-array and add some dense layers. We can add some dropout layers to reduce overfitting. For the last layer, we add a dense layer with the number of classes from our problem (10) and a softmax activation, which creates the probability distribution for each class.

In [15]:
```
model = keras.models.Sequential([
  keras.layers.Conv2D(filters=64, kernel_size=3, activation='relu',
                      padding='same', input_shape=[28, 28, 1]),
```

```
  keras.layers.MaxPool2D(pool_size=2),
  keras.layers.Conv2D(filters=128, kernel_size=3, activation='relu',
                      padding='same'),
  keras.layers.MaxPool2D(pool_size=2),
  keras.layers.Flatten(),
  keras.layers.Dense(units=128, activation='relu'),
  keras.layers.Dropout(0.25),
  keras.layers.Dense(units=64, activation='relu'),
  keras.layers.Dropout(0.25),
  keras.layers.Dense(units=10, activation='softmax'),
])
```

Compiling the Model

The next step is to compile the model. Here we pass the optimizer, which adjusts the weights to minimize the loss, the loss function, which measures the disparity between the true and predicted values, and the metrics, a function used to measure the performance of the model.

In [16]:
```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Training the Model

The last step is to train our model. Here we need to pass the input data, the target data and the number of epochs, which defines the number of full iterations of the training dataset. We will also pass a parameter to split our data in training (70%) and validation (30%) and a parameter to define the batch_size, which is the number of training examples in each pass.

We will save the results of our training in the variable model_history.

Obs.: if you are running this notebook, I recommend you change the runtime type to GPU. It will train the model really faster than using the CPU (No hardware accelerator).

In [17]:
```
model_history = model.fit(X_train, y_train, batch_size=50, epochs=10, validation_split=0.3)
```

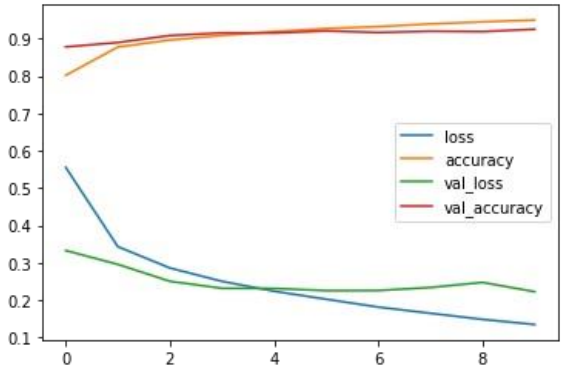
```
Epoch 1/10
840/840 [==============================] - 55s 64ms/step - loss: 0.5552 - accuracy: 0.8021 - val_loss: 0.3327 - v
al_accuracy: 0.8782
Epoch 2/10
840/840 [==============================] - 58s 69ms/step - loss: 0.3429 - accuracy: 0.8779 - val_loss: 0.2955 - v
al_accuracy: 0.8900
Epoch 3/10
840/840 [==============================] - 62s 74ms/step - loss: 0.2857 - accuracy: 0.8967 - val_loss: 0.2501 - v
al_accuracy: 0.9087
Epoch 4/10
840/840 [==============================] - 66s 79ms/step - loss: 0.2501 - accuracy: 0.9090 - val_loss: 0.2314 - v
al_accuracy: 0.9157
Epoch 5/10
840/840 [==============================] - 63s 75ms/step - loss: 0.2235 - accuracy: 0.9192 - val_loss: 0.2310 - v
al_accuracy: 0.9156
Epoch 6/10
840/840 [==============================] - 64s 76ms/step - loss: 0.2022 - accuracy: 0.9273 - val_loss: 0.2252 - v
al_accuracy: 0.9211
Epoch 7/10
840/840 [==============================] - 63s 75ms/step - loss: 0.1810 - accuracy: 0.9326 - val_loss: 0.2255 - v
al_accuracy: 0.9168
Epoch 8/10
840/840 [==============================] - 64s 76ms/step - loss: 0.1645 - accuracy: 0.9398 - val_loss: 0.2336 - v
al_accuracy: 0.9202
Epoch 9/10
840/840 [==============================] - 65s 77ms/step - loss: 0.1482 - accuracy: 0.9453 - val_loss: 0.2472 - v
al_accuracy: 0.9192
Epoch 10/10
840/840 [==============================] - 63s 75ms/step - loss: 0.1344 - accuracy: 0.9503 - val_loss: 0.2224 - v
al_accuracy: 0.9256
```

Evaluating the Loss

The fit method returns a history object with the results for each epoch. We can plot a chart with the loss and accuracy for the training and validation datasets. From this chart, it is possible to see how the loss goes down and the accuracy goes up over the epochs. This chart is also used to identify evidence of overfitting and underfitting. For our model, it doesn't seem we have strong evidence of these problems. Thus, let's move on and make some predictions with our test dataset

In [24]:
```
pd.DataFrame(model_history.history).plot()
```

```
plt.show()
```



Making Predictions and Evaluating the Results

Evaluating the Accuracy in the Test Dataset

Now, let's see how the model performs with our test dataset.

In [25]:
```
model.evaluate(X_test, y_test);
```

```
313/313 [==============================] - 5s 17ms/step - loss: 0.2507 - accuracy: 0.9201
```

The accuracy on the training dataset is smaller than the accuracy on the test and validation datasets. However, this is still a good result with 91,55% of accuracy.

Making Predictions

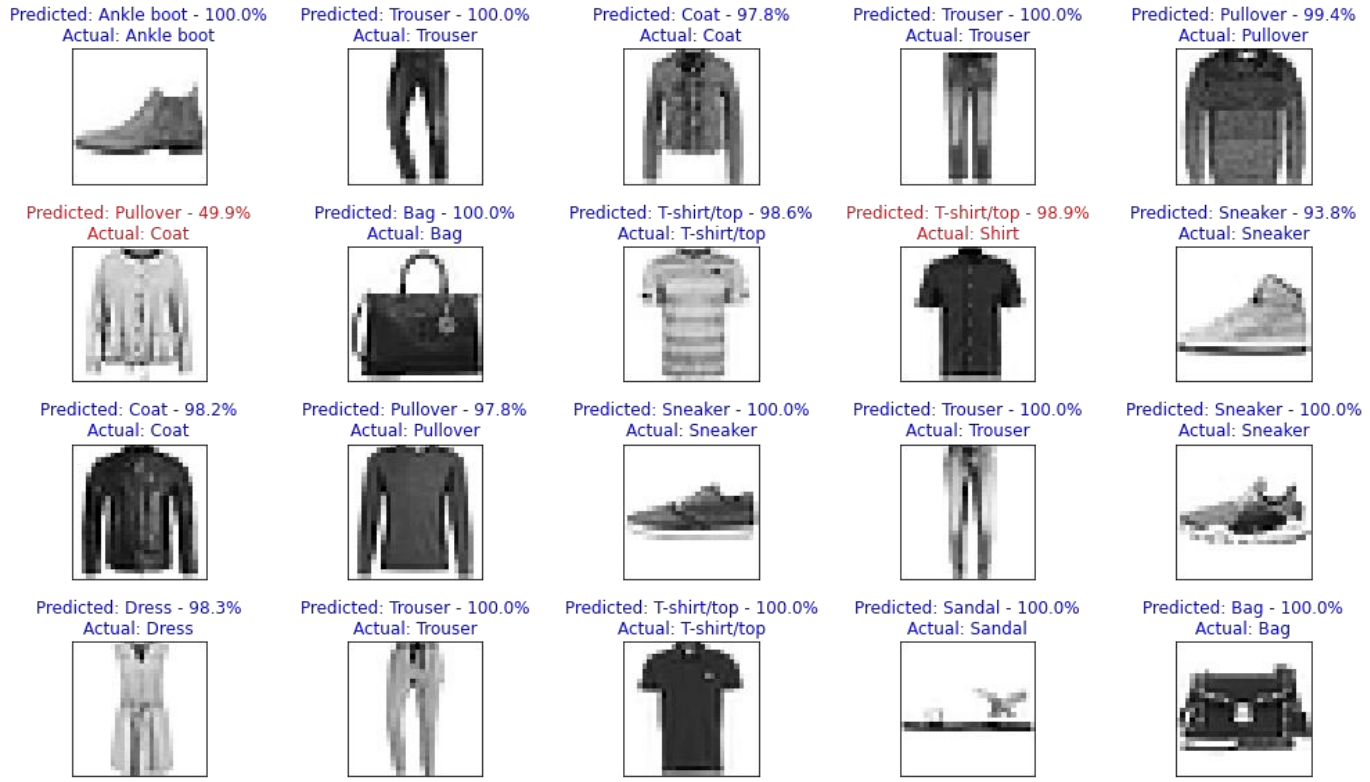
We can use our model to predict a class for each example in our test database and store the results in the variable predictions.

In [26]:
```
predictions = model.predict(X_test)
```

Now, let's plot a few images from our test dataset with the true and the predicted labels. When the model predicts right, the text will be displayed in blue, if the prediction is wrong, it will be displayed in red. Also, it will be displayed the calculated probability for the predicted class.

In [27]:
```
def plot_img_label(img, pred_class, pred_percentage, true_class):

  plt.imshow(img,cmap=plt.cm.binary)

  if pred_class == true_class:
    color = 'blue'
  else:
    color = 'red'

  plt.title(label= f"Predicted: {pred_class} - {pred_percentage:2.1f}%\nActual: {true_class}",
            fontdict={'color': color})
```

In [28]:
```
plt.figure(figsize=(14,10))
for i in range(20):

    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)

    i = i * 5

    img = X_test[i].reshape(28,28)
    pred_class = class_names[np.argmax(predictions[i])]
    pred_percentage = np.max(predictions[i])*100
    true_class = class_names[np.argmax(y_test[i])]

    plot_img_label(img, pred_class, pred_percentage, true_class)

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



From these 20 examples, we can see that our model made wrong predictions for one coat and one shirt. However, it is not feasible to analyze the predictions for 10,000 examples using this plot. Thus, let's plot a crosstab to analyze what our model predicted right and wrong for each class.

Crosstab

First, let's create our crosstab. Here we will have the columns and index with our labels [0 - 9]. After creating it, we can rename the columns and indexes to display the name of the classes instead of the labels.

In [29]:
```
predicted_label = np.argmax(predictions,axis = 1)
true_label = np.argmax(y_test, axis = 1)
```

In [30]:
```
crosstab = pd.crosstab(true_label, predicted_label, rownames=["True"], colnames=["Predicted"], margins=True)
```

In [31]:
```
classes = {}
for item in zip(range(10), class_names):
  classes[item[0]] = item[1]
```

In [32]:
```
crosstab.rename(columns=classes, index=classes, inplace=True)
crosstab
```

Out[32]:

Predicted	T-shirt/top	Trouser	Pullover	Dress	Coat	Sandal	Shirt	Sneaker	Bag	Ankle boot	All
True											
T-shirt/top	872	0	15	12	3	1	87	0	10	0	1000
Trouser	0	985	0	11	1	0	2	0	1	0	1000
Pullover	14	1	860	10	52	0	63	0	0	0	1000
Dress	10	1	10	925	29	0	23	0	2	0	1000
Coat	1	1	50	11	897	0	38	0	2	0	1000
Sandal	0	0	0	0	0	981	0	15	1	3	1000
Shirt	111	0	47	29	60	0	745	0	8	0	1000
Sneaker	0	0	0	0	0	3	0	983	0	14	1000
Bag	3	0	0	1	2	1	0	4	989	0	1000
Ankle boot	0	0	0	0	0	4	0	31	1	964	1000
All	1011	988	982	999	1044	990	958	1033	1014	981	10000

Analyzing our crosstab, we can notice that our best accuracy was achieved for the Sandals classification (99,1%), while our lowest accuracy was for the Shirts (76,9%). The crosstab provides a great way to visualize the quantities predicted by our model for each class. We can easily see, for example, that it predicted Shirt as T-shirt/Top for 74 examples or that it didn't predict any Bag as Ankle boot.

Classification Report

Now, let's plot a summary with the precision, recall and, f1-score for each class using the classification report from the scikit-learn library. Here it is possible to see that we didn't have a considerate disparity between precision and recall for any classes. As we noticed in the crosstab, our worst result is for the Shirt class.

In [33]:
```
print(classification_report(true_label, predicted_label, target_names=class_names))
```

```
              precision    recall  f1-score   support

 T-shirt/top       0.86      0.87      0.87      1000
     Trouser       1.00      0.98      0.99      1000
    Pullover       0.88      0.86      0.87      1000
       Dress       0.93      0.93      0.93      1000
        Coat       0.86      0.90      0.88      1000
      Sandal       0.99      0.98      0.99      1000
       Shirt       0.78      0.74      0.76      1000
     Sneaker       0.95      0.98      0.97      1000
         Bag       0.98      0.99      0.98      1000
  Ankle boot       0.98      0.96      0.97      1000

    accuracy                           0.92     10000
   macro avg       0.92      0.92      0.92     10000
weighted avg       0.92      0.92      0.92     10000
```

Conclusion

In this project, it was presented how to train a Convolutional Neural Network to classify images of clothing from the Fashion MNIST dataset using TensorFlow and Keras. Using this model, we got an overall accuracy of 91,55% in our test dataset, which is a good result. However, specifically for our Shirt class we got an accuracy of only 76,90%. We could try to improve the accuracy of this class using some data augmentation techniques. Furthermore, in case you want to get a model with higher accuracy, you could try changing some hyperparameters or using different network architectures.