CS 464

Introduction to Machine Learning

Project Final Report

Group 26

Group Members:

Onurcan Ataç 22002194

Oğuz Kuyucu 21902683

Kutay Şenyiğit 21902377

Alperen Utku Yalçın 22002187

Table of Contents

1. Introduction

1.1. Project Description

Since the emergence of deep fake audio, differentiating real voices and AI-generated audio has become a significant problem. Humans cannot differentiate those perfectly [1].In this project, we will perform audio analysis by using two deep learning models for two differently split datasets. Our primary objective is to differentiate between real and AI-generated 'deepfake' audio. Our project begins by carefully processing raw stereo audio data. We start by breaking down each audio segment to closely examine its amplitude and frequency details. This thorough analysis involves separating and evaluating each 5-second audio segment to understand its distinct sound wave patterns and spectral properties. By generating visual representations such as amplitude graphs and frequency spectrograms for both left and right channels in five-second intervals, we transform complex audio information into a more understandable visual format. These visualizations serve as the cornerstone for training a Convolutional Neural Network (CNN) and ResNet. Our CNN and ResNet models will be designed to analyze these spectrograms to differentiate the subtle discrepancies between real human speech and AI-generated deep fake speech.

1.2. Coding Environment Summary

We are using Python programming language because of its useful libraries such as Pytorch, Pandas, Numpy, and Matplotlib. We used librose in order to obtain spectrograms from raw audio data.

- PyTorch is an open-source machine learning library that was developed by Facebook's AI Research lab. It's famous for, particularly deep learning and artificial intelligence applications. It provides opportunities for building and training neural networks. It supports GPU acceleration and that takes a significant role when vast data is used. We will use PyTourch for constructing our model [6].
- Pandas is a data manipulation and analysis library. NumPy (Numerical Python) is an important library for scientific computing in Python. Pandas and Numpy were used for the data set preparation phase of our project. While Pandas was helpful in data manipulation tasks such as merging datasets, Numpy was crucial for numerical calculations and array manipulations [2][3].
- Matplotlib is a plotting library for Python. Matplotlib was used for data visualization [4].
- Librosa: Librosa is a library which is designed for audio analysis. It has useful methods to convert raw sound data to amplitude and frequency graphs [5].

In order to execute the Python codes and train our model, we decided to use Google Colab. It provides an opportunity to execute our code on GPU. It also helps us synchronously collaborate on the same program [7].

2. Preprocessing

2.1. Our Dataset

We used “DEEP-VOICE: DeepFake Voice Recognition” dataset from kaggle. It has 8 people’s original voices and the voices which converted to each other.

2.2. Deriving Spectrograms

In the preprocessing phase of our project, we meticulously handled the raw audio data to extract meaningful data for further analysis. Initially, we segregated the audio into discrete five-second segments, ensuring a consistent and manageable analysis window for each sample. For each of these segments, we conducted a thorough channel-wise dissection, extracting both the left and right audio channels separately. For each channel in every segment, we generated two types of graphical representations: amplitude graphs and frequency spectrograms. The amplitude graphs were instrumental in visualizing the sound wave's loudness variations over time. These visualizations were transforming the abstract and complex raw audio data into analyzable visual formats.

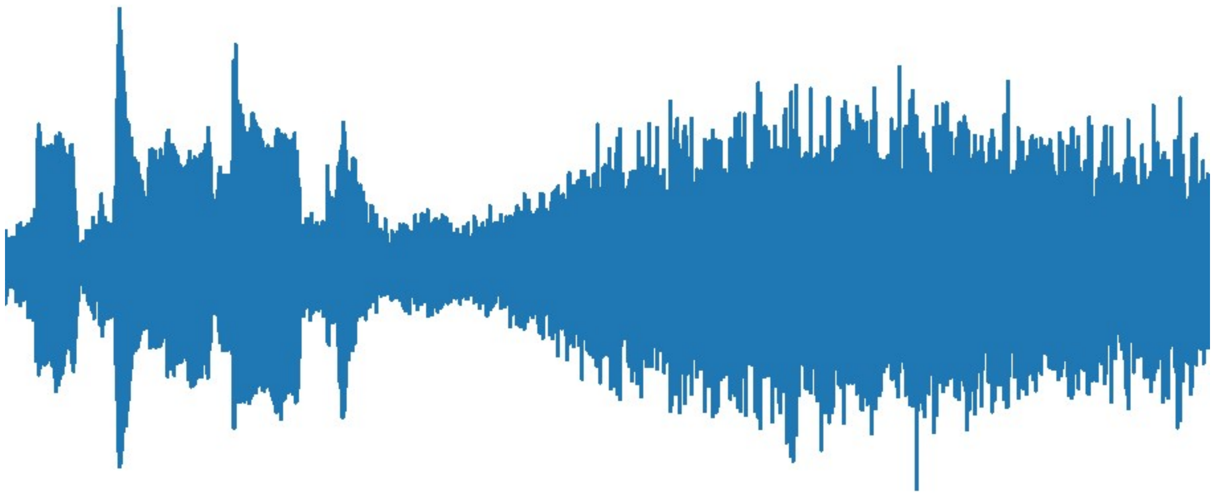


Figure 1: Sample left channel amplitude

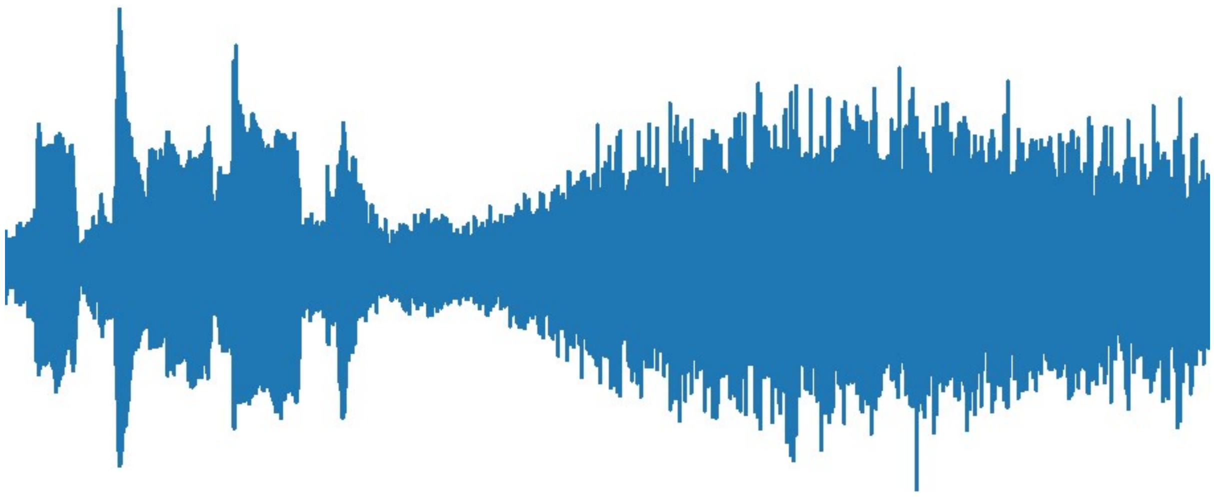


Figure 2: Sample right channel amplitude

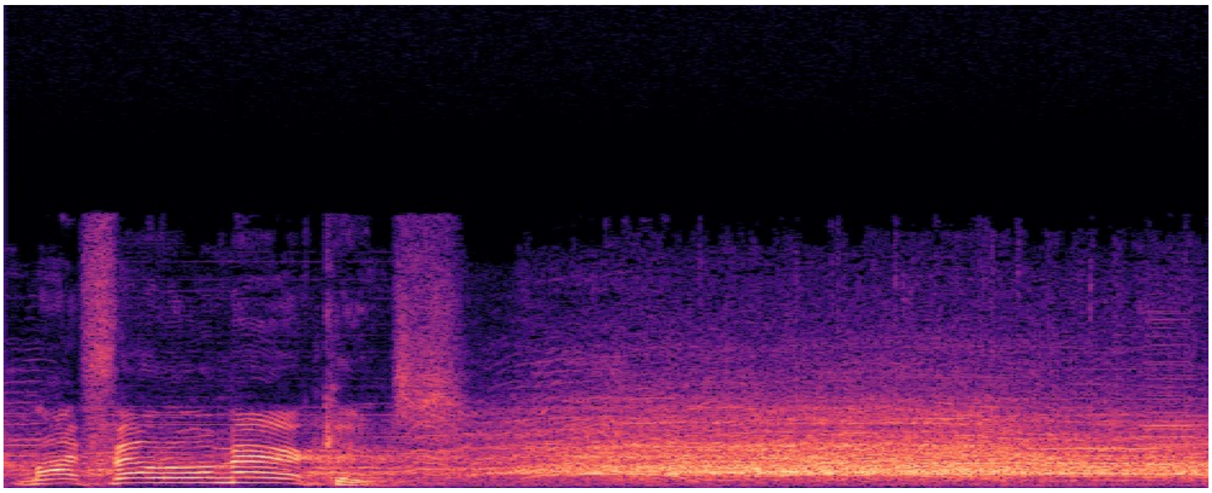


Figure 3: Sample left channel frequency

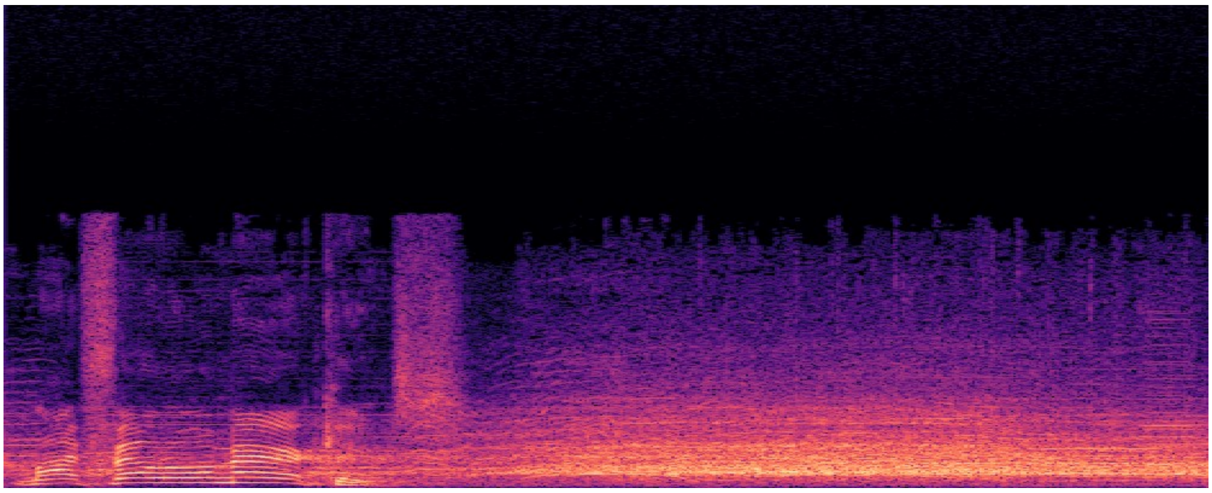


Figure 4: Sample right channel frequency

In our dataset, the right channel and the left channel were identical. And frequency spectrograms carry more information. Therefore, we only used right spectrograms in our models.

3. Dataset Splits

3.1. First Split: Detecting Deepfake Speech by Knowing A Person's Voice but not the Transcript

For our first type of dataset split, we will aim for our models to detect deepfake speech while not encountering the transcripts in the validation and test datasets during the training process. We will split each audio file in our dataset into 80%, 10%, and 10% ratios as train, validation, and test data. However, there is a downside to the first split. The voice characteristics of people whom our dataset consists of will be encountered by the model in the training process. Therefore, while using this split, our model is expected to capture the voice characteristics of famous people in our dataset, besides capturing the patterns that deep fake and original audios possess.

At the initiation of the dataset split process, after the preprocessing, we split our data into 5-second intervals and we extracted the amplitude and frequency plots for each 5-second interval.

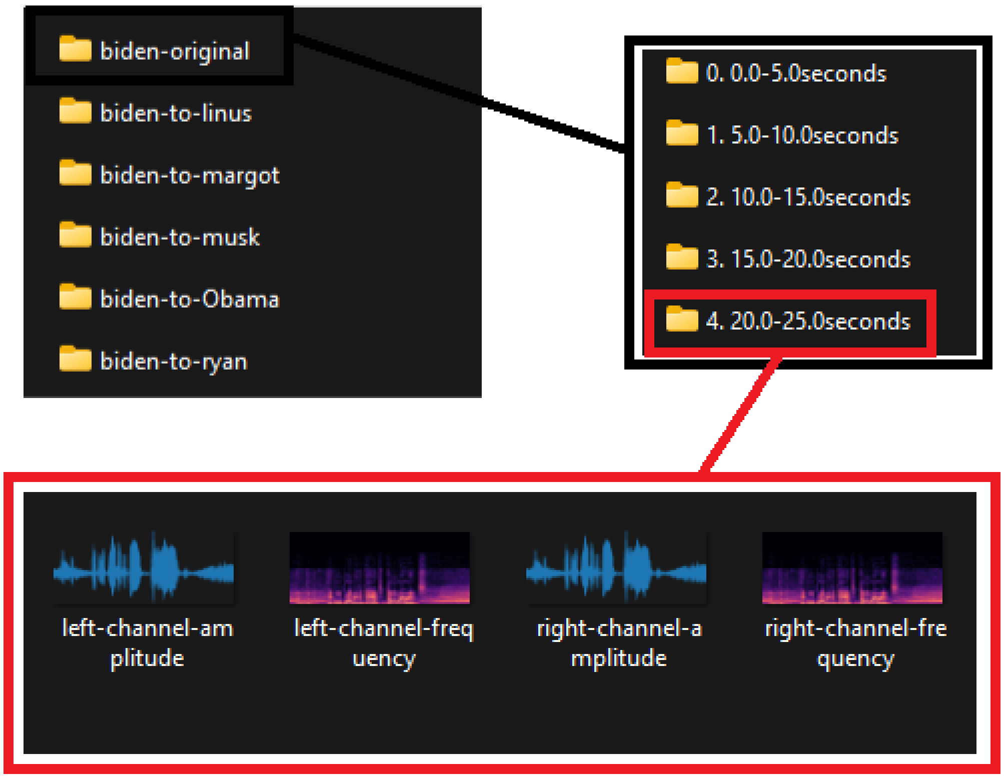


Figure 5: The directory structure of the data after preprocessing

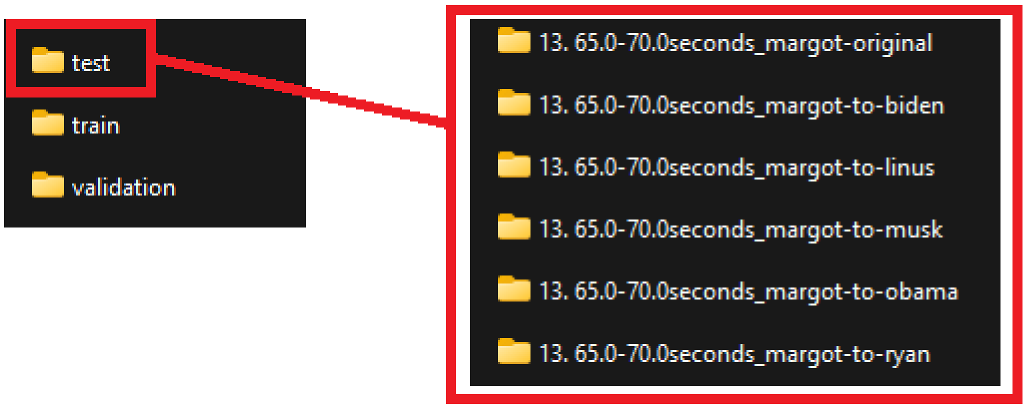
Our aim for the first split is to split the data in 80%, 10%, and 10% ratios as train, validation, and test. To achieve that, we implemented two functions; one to split the data in required ratios and one to copy and carry the data to new directories. The code for those functions is as follows:

```
# Split data into train/validation/test sets
def split_data(data_dir, train_ratio=0.8, val_ratio=0.1):
    files = os.listdir(data_dir)

    files = sorted(files, key=lambda x: int(x.split('.')[0]))  # Sort using numerical order

    num_files = len(files)
    num_train = int(train_ratio * num_files)
    num_val = int(val_ratio * num_files)

    train_files = files[:num_train]
    val_files = files[num_train:num_train+num_val]
    test_files = files[num_train+num_val:]
    return train_files, val_files, test_files

# Copy files to train/validation/test folders
def copy_files(source_dir, dest_dir, piece_folders, upper_folder_name):
    os.makedirs(dest_dir, exist_ok=True)

    for folder in piece_folders:
        source_folder = os.path.join(source_dir, folder)
        dest_folder_name = f"{folder}_{upper_folder_name}"
        dest_folder = os.path.join(dest_dir, dest_folder_name)

        os.makedirs(dest_folder, exist_ok=True)

        # Iterate over files in the 5-second folders and copy them to the destination folder
        for file in os.listdir(source_folder):
            source_path = os.path.join(source_folder, file)
            dest_filename = f"{file}"
            dest_path = os.path.join(dest_folder, dest_filename)

            # Perform the copy operation
            shutil.copy(source_path, dest_path)
```

Then we call these functions accordingly for each sound directory, split their subdirectories accordingly, and copy the files to the new directory for the first split.

```
# Get a list of subdirectories in all_spectogram_folders
subfolders = [f.path for f in os.scandir(all_spectrogram_folders) if f.is_dir()]

for folder in subfolders:
    print(f"Processing folder: {folder}")

    # Get the list of files and split them
    train_files, val_files, test_files = split_data(folder)

    # Copy the files to train/validation/test folders with upper folder name
    upper_folder_name = os.path.basename(folder)

    copy_files(folder, os.path.join(save_dir, 'train'), train_files, upper_folder_name)
    copy_files(folder, os.path.join(save_dir, 'validation'), val_files, upper_folder_name)
    copy_files(folder, os.path.join(save_dir, 'test'), test_files, upper_folder_name)
```

After the dataset is split and carried over to the new directories, the new directory structure for the first split becomes as follows:



13. 65.0-70.0seconds_margot-original
13. 65.0-70.0seconds_margot-to-biden
13. 65.0-70.0seconds_margot-to-linus
13. 65.0-70.0seconds_margot-to-musk
13. 65.0-70.0seconds_margot-to-obama
13. 65.0-70.0seconds_margot-to-ryan

Figure 6: The directory structure for the first split

Train, validation, and test data are separated into distinct directories so that they can be used without complications while training and testing our models.

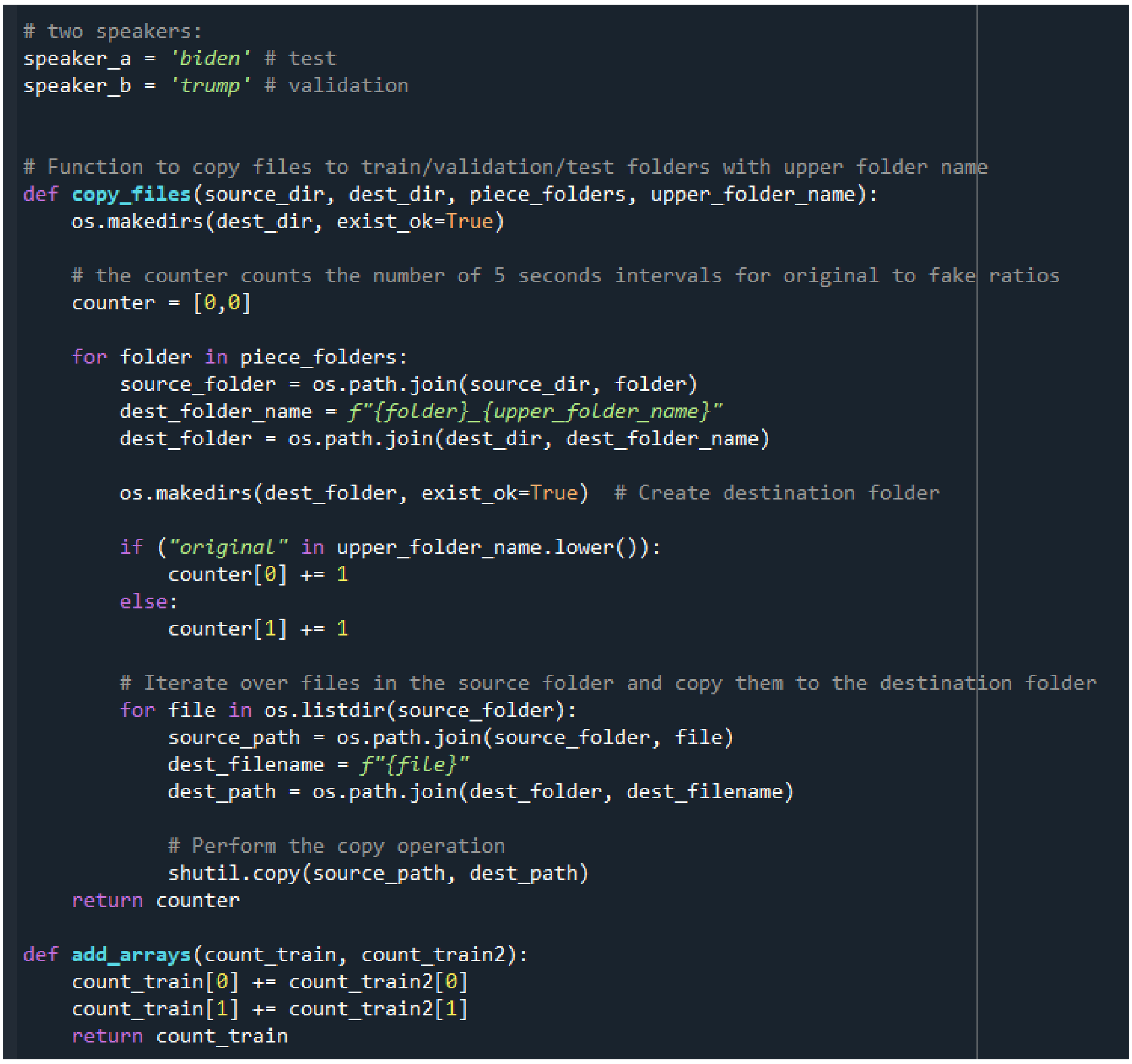
3.2. Second Split: Detecting Deepfake Speech Without Knowing A Person's Voice or the Transcript

The second version of our split aims to consider the case in which we want a model that can be trained to decide between fake and real voices given an unknown (not in the train data) speaker. In the case where a voice that the system has no prior knowledge of is considered, the training data should have no knowledge of the transcript or the voice of the speaker. So, in this split, we should avoid any information regarding a person, while keeping all other speakers in the system. Before explaining our solution, we should have a better understanding of how the data is separated.

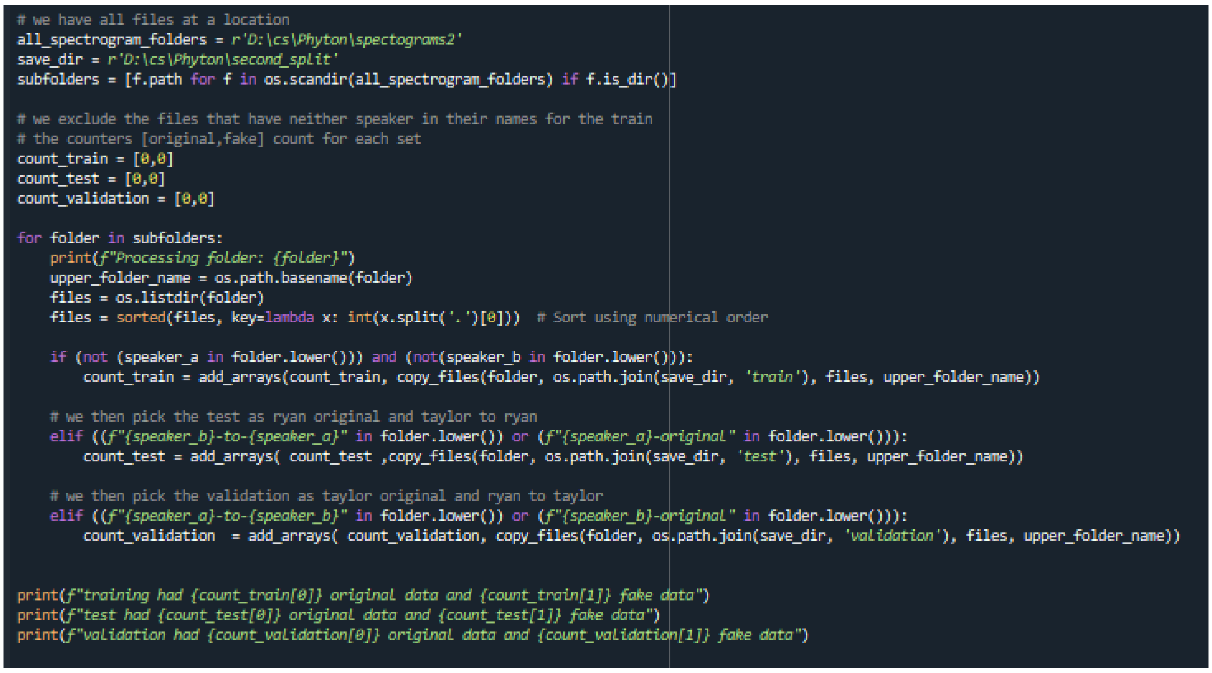
There are two main types of data: A-original and B-to-A. Here, A-original is the voice data which is the original (the non-edited true A), and B-to-A is the voice data where the voice of A was put on the transcript of speaker B (the deepfake voice A). Hence we should exclude these files from the training set. If we were to take "trump" as the test set voice and exclude the false and true voice files of "trump" (trump-original and A-to-trump), we run into the problem that A-to-trump files contain a transcript of person A, and our training set has already memorized the person A's transcript in training. Person A's original soundtrack plays over their transcript and other people in training also talk over person A's transcript. Therefore, the system knows that any voice other than person A's voice is a fake given A's transcript. Hence, in the case that A-to-trump is given as a test, the system will be biased since it already knows person A's transcript.

Another problem occurs with any files of trump-to-B, where person B reads the transcript of "trump". If these files are given in the training set, the system will be likely to recognize the "trump" transcript and give a prediction of "fake" automatically. This happens since all trump-to-B files will be labeled as fake for the training set and the model would possibly memorize the trump transcript as fake for all and any possible input voices. However, if the alternative is chosen where we include this file for the test set, the model will already be trained for the voice of B, and therefore will be guessing over voices it was already trained on and the results would not display the case where we give completely new inputs that have no reliance on prior data.

To avoid these issues, we decided to subtract two people at once for the training set. The "trump" and "biden" people were subtracted from the training data, which means any file name without "trump" or "biden" is within the training dataset. For the test split, we have the "trump-original" and "biden-to-trump". Here, the trained data has no prior knowledge of Trump's voice, Trump's transcript, or Biden's transcript. Therefore, there is no bias for the inputs where we hear the original Trump voice over the original transcript, and for the fake Trump voice over Biden's transcript (which the system doesn't know). For the validation split, we have a similar structure where we only have "biden-original" and "trump-to-biden" (true biden, fake biden), for which the model has no prior information on both "trump" and "biden".



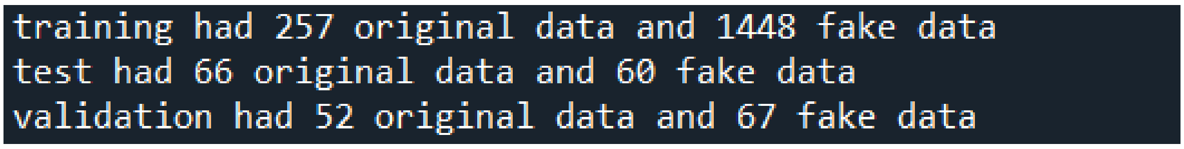
```python
# two speakers:
speaker_a = 'biden' # test
speaker_b = 'trump' # validation


# Function to copy files to train/validation/test folders with upper folder name
def copy_files(source_dir, dest_dir, piece_folders, upper_folder_name):
    os.makedirs(dest_dir, exist_ok=True)

    # the counter counts the number of 5 seconds intervals for original to fake ratios
    counter = [0,0]

    for folder in piece_folders:
        source_folder = os.path.join(source_dir, folder)
        dest_folder_name = f"{folder}_{upper_folder_name}"
        dest_folder = os.path.join(dest_dir, dest_folder_name)

        os.makedirs(dest_folder, exist_ok=True)  # Create destination folder

        if ("original" in upper_folder_name.lower()):
            counter[0] += 1
        else:
            counter[1] += 1

        # Iterate over files in the source folder and copy them to the destination folder
        for file in os.listdir(source_folder):
            source_path = os.path.join(source_folder, file)
            dest_filename = f"{file}"
            dest_path = os.path.join(dest_folder, dest_filename)

            # Perform the copy operation
            shutil.copy(source_path, dest_path)
    return counter

def add_arrays(count_train, count_train2):
    count_train[0] += count_train2[0]
    count_train[1] += count_train2[1]
    return count_train
```



```python
# we have all files at a location
all_spectrogram_folders = r'D:\cs\Phyton\spectograms2'
save_dir = r'D:\cs\Phyton\second_split'
subfolders = [f.path for f in os.scandir(all_spectrogram_folders) if f.is_dir()]

# we exclude the files that have neither speaker in their names for the train
# the counters [original,fake] count for each set
count_train = [0,0]
count_test = [0,0]
count_validation = [0,0]

for folder in subfolders:
    print(f"Processing folder: {folder}")
    upper_folder_name = os.path.basename(folder)
    files = os.listdir(folder)
    files = sorted(files, key=lambda x: int(x.split('.')[0]))  # Sort using numerical order

    if (not (speaker_a in folder.lower())) and (not(speaker_b in folder.lower())):
        count_train = add_arrays(count_train, copy_files(folder, os.path.join(save_dir, 'train'), files, upper_folder_name))

    # we then pick the test as ryan original and taylor to ryan
    elif ((f"{speaker_b}-to-{speaker_a}" in folder.lower()) or (f"{speaker_a}-original" in folder.lower())):
        count_test = add_arrays( count_test ,copy_files(folder, os.path.join(save_dir, 'test'), files, upper_folder_name))

    # we then pick the validation as taylor original and ryan to taylor
    elif ((f"{speaker_a}-to-{speaker_b}" in folder.lower()) or (f"{speaker_b}-original" in folder.lower())):
        count_validation  = add_arrays( count_validation, copy_files(folder, os.path.join(save_dir, 'validation'), files, upper_folder_name))


print(f"training had {count_train[0]} original data and {count_train[1]} fake data")
print(f"test had {count_test[0]} original data and {count_test[1]} fake data")
print(f"validation had {count_validation[0]} original data and {count_validation[1]} fake data")
```

Figure 7: Code written for the splitting according to the folder names

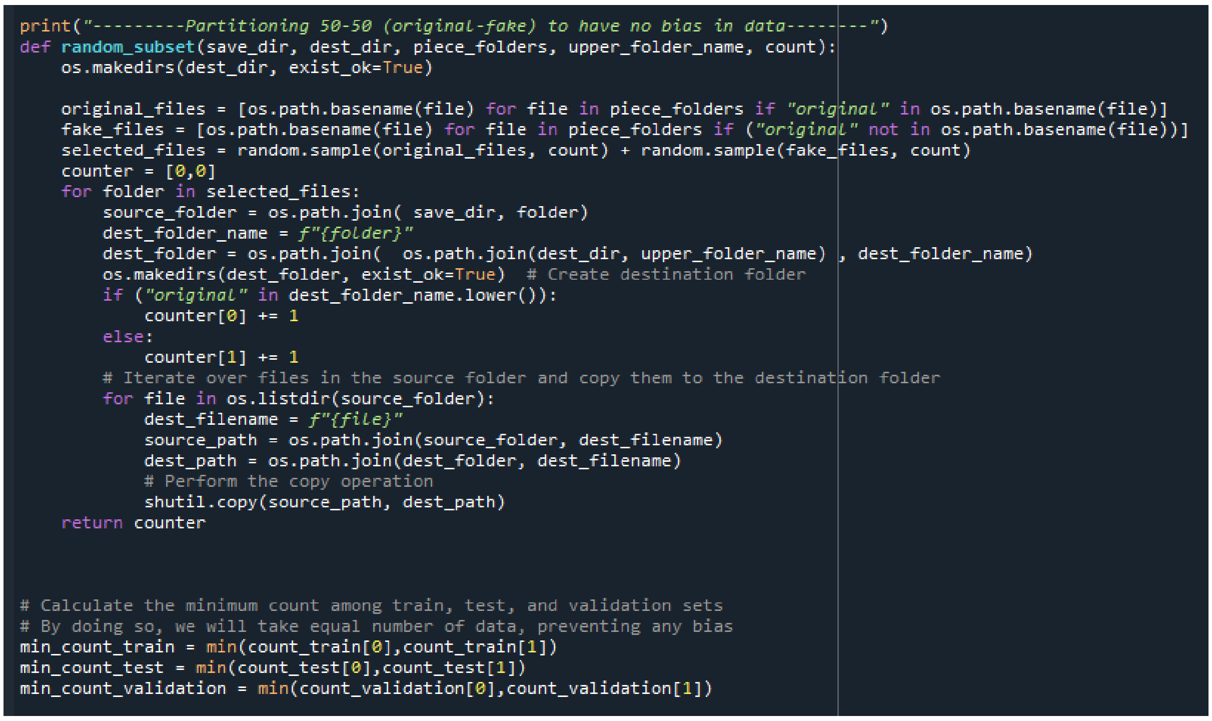
The code we have written enables us to separate the train, test, and validation sets. In the image above, we can observe the copy_files function that enables us to copy the files from one directory to the other. The function returns the number of 5-second intervals of original (real), to fake (deepfake) data. To separate the different sets, we look at the file name, and depending on the content of the name of the file, we separate them into separate folders: train, test, validation. The add_arrays helper function is used in counting the total number of 5-second intervals in the overall train, test, and validation sets; below is the result of the amounts of data (number of pictures of 5-seconds of voice data) we have. The data was printed out.



```
training had 257 original data and 1448 fake data
test had 66 original data and 60 fake data
validation had 52 original data and 67 fake data
```

Figure 8: Code written for number of input data

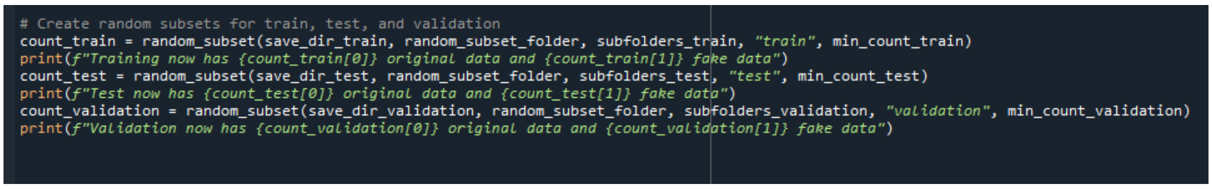
As it is observed, there is some bias in the training data, we observe that the number of fake spectrograms is far greater than the number of correct data. Hence, for a given input, the data is far more likely to be labeled fake. To prevent any bias, we have written additional preprocessing code.



```python
print("---------Partitioning 50-50 (original-fake) to have no bias in data--------")
def random_subset(save_dir, dest_dir, piece_folders, upper_folder_name, count):
    os.makedirs(dest_dir, exist_ok=True)

    original_files = [os.path.basename(file) for file in piece_folders if "original" in os.path.basename(file)]
    fake_files = [os.path.basename(file) for file in piece_folders if ("original" not in os.path.basename(file))]
    selected_files = random.sample(original_files, count) + random.sample(fake_files, count)
    counter = [0,0]
    for folder in selected_files:
        source_folder = os.path.join( save_dir, folder)
        dest_folder_name = f"{folder}"
        dest_folder = os.path.join(  os.path.join(dest_dir, upper_folder_name) , dest_folder_name)
        os.makedirs(dest_folder, exist_ok=True)  # Create destination folder
        if ("original" in dest_folder_name.lower()):
            counter[0] += 1
        else:
            counter[1] += 1
        # Iterate over files in the source folder and copy them to the destination folder
        for file in os.listdir(source_folder):
            dest_filename = f"{file}"
            source_path = os.path.join(source_folder, dest_filename)
            dest_path = os.path.join(dest_folder, dest_filename)
            # Perform the copy operation
            shutil.copy(source_path, dest_path)
    return counter


# Calculate the minimum count among train, test, and validation sets
# By doing so, we will take equal number of data, preventing any bias
min_count_train = min(count_train[0],count_train[1])
min_count_test = min(count_test[0],count_test[1])
min_count_validation = min(count_validation[0],count_validation[1])
```

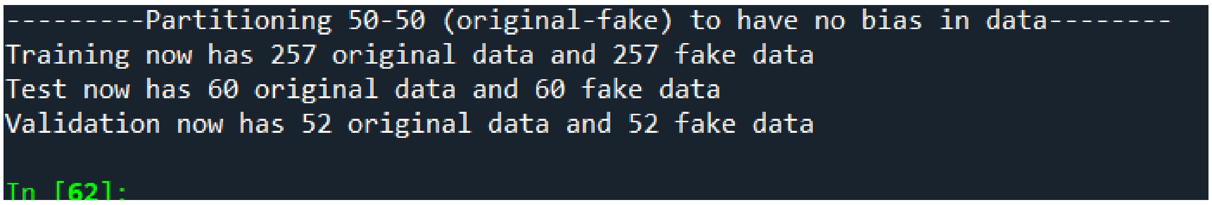
Figure 9: Remaining code for the second split to eliminate bias

We chose to pick the minimum of the total number of data for all three sets. The random_subset function copies the maximum number of original and fake data from the previously created sets and pastes them to a new folder. For the total number of 257 original and 1448 fake data in the training set, it picks the 257 original data and randomly selects 257 data from the 1448 fake data.



```python
# Create random subsets for train, test, and validation
count_train = random_subset(save_dir_train, random_subset_folder, subfolders_train, "train", min_count_train)
print(f"Training now has {count_train[0]} original data and {count_train[1]} fake data")
count_test = random_subset(save_dir_test, random_subset_folder, subfolders_test, "test", min_count_test)
print(f"Test now has {count_test[0]} original data and {count_test[1]} fake data")
count_validation = random_subset(save_dir_validation, random_subset_folder, subfolders_validation, "validation", min_count_validation)
print(f"Validation now has {count_validation[0]} original data and {count_validation[1]} fake data")
```

Figure 10: Continuation of code for calling the function to eliminate bias and for prints



```
---------Partitioning 50-50 (original-fake) to have no bias in data--------
Training now has 257 original data and 257 fake data
Test now has 60 original data and 60 fake data
Validation now has 52 original data and 52 fake data

In [62]:
```

Figure 11: Output of code to eliminate bias

Then, we call the random_subset function and partition our data to have an equal number of fake and original 5-second interval spectrograms. The random_subset function returns the number of each set's fake and real data and we observe that data is now unbiased and fully equal in terms of fake and real. The new randomly picked unbiased data is displayed in the output.

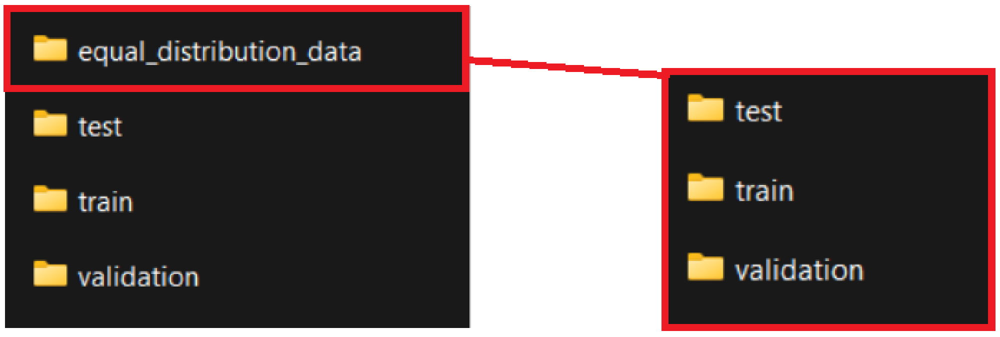


Figure 12: Explanation of folders

The result of our preprocessing is the folders “train”, “test”, “validation”, and “equal_distribution_data” within the folder “second_split”. The “equal_distribution_data” has equally distributed fake and real datasets for the test, training, and validation. The reasoning behind why we decided to have the equal_distribution_data be calculated after we calculate the test, train, and validation on the entire data is because the data we have may prove to be insufficient, and therefore we may need to run our model for the entire dataset. The initial test, train, and validation sets would then come in handy when we wish to run for a biased but high amount of training data model in our implementation. Mainly the test, train, and validation data in “equal_distribution_data” is to be used for the unbiased training data for our purpose of deepfake speech detection in the case where the speaker and the transcript are unknown to the model.

4. Machine Learning Models and Implementations

4.1. Convolutional Neural Networks

4.1.1. Explanation of CNN

Convolutional Neural Networks (CNNs) are a deep learning model that is designed for image recognition. However, it can be used in other domains as audio processing. CNNs can be used in audio processing by training the model by audio visualization. Audio visualization can be made by extracting its spectrogram of the 5-second audio intervals. Spectrograms are a visual representation of the frequency signal over time. It helps the model to capture both temporal and spectral features that are essential for distinguishing between deep fake and real audio. Every 5-second audio part can be transformed into a spectrogram image. It allows us to use it as input to the CNN model. The convolutional layers scan hierarchical features and pooling layers help in spatial down-sampling. The model can distinguish if the audio is generated by deep fake manipulation or an original recording with the help of the ultimate layers. The ultimate layers make predictions by using the acquired features [8].

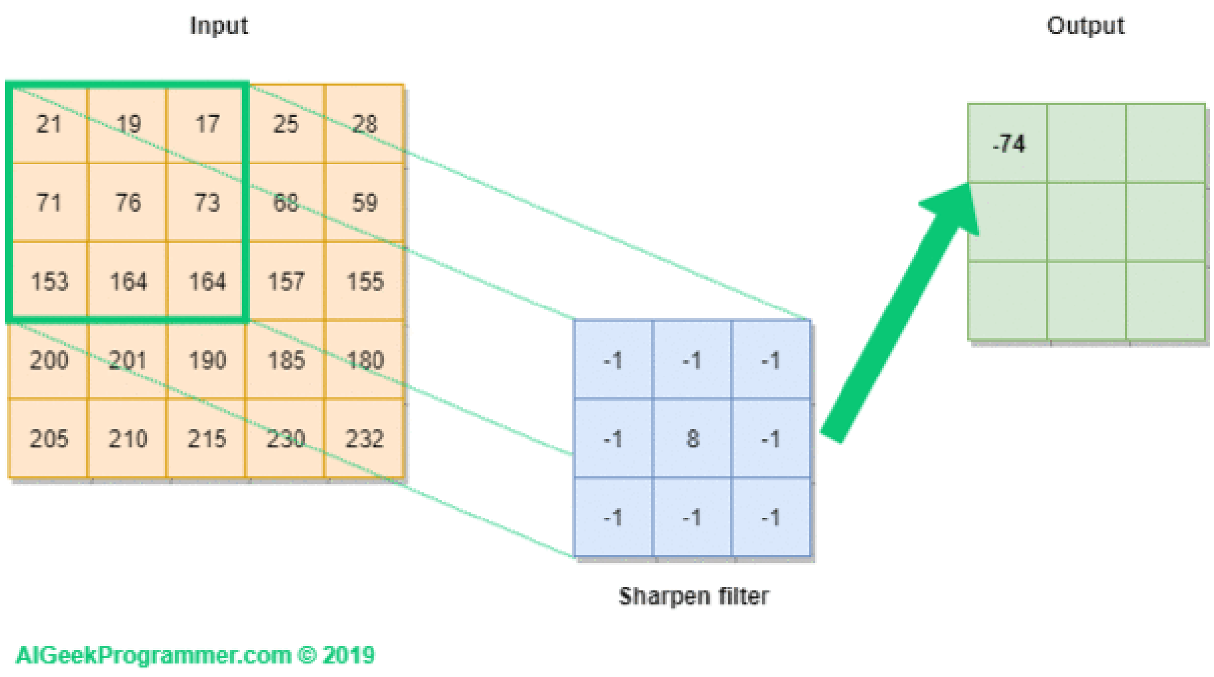


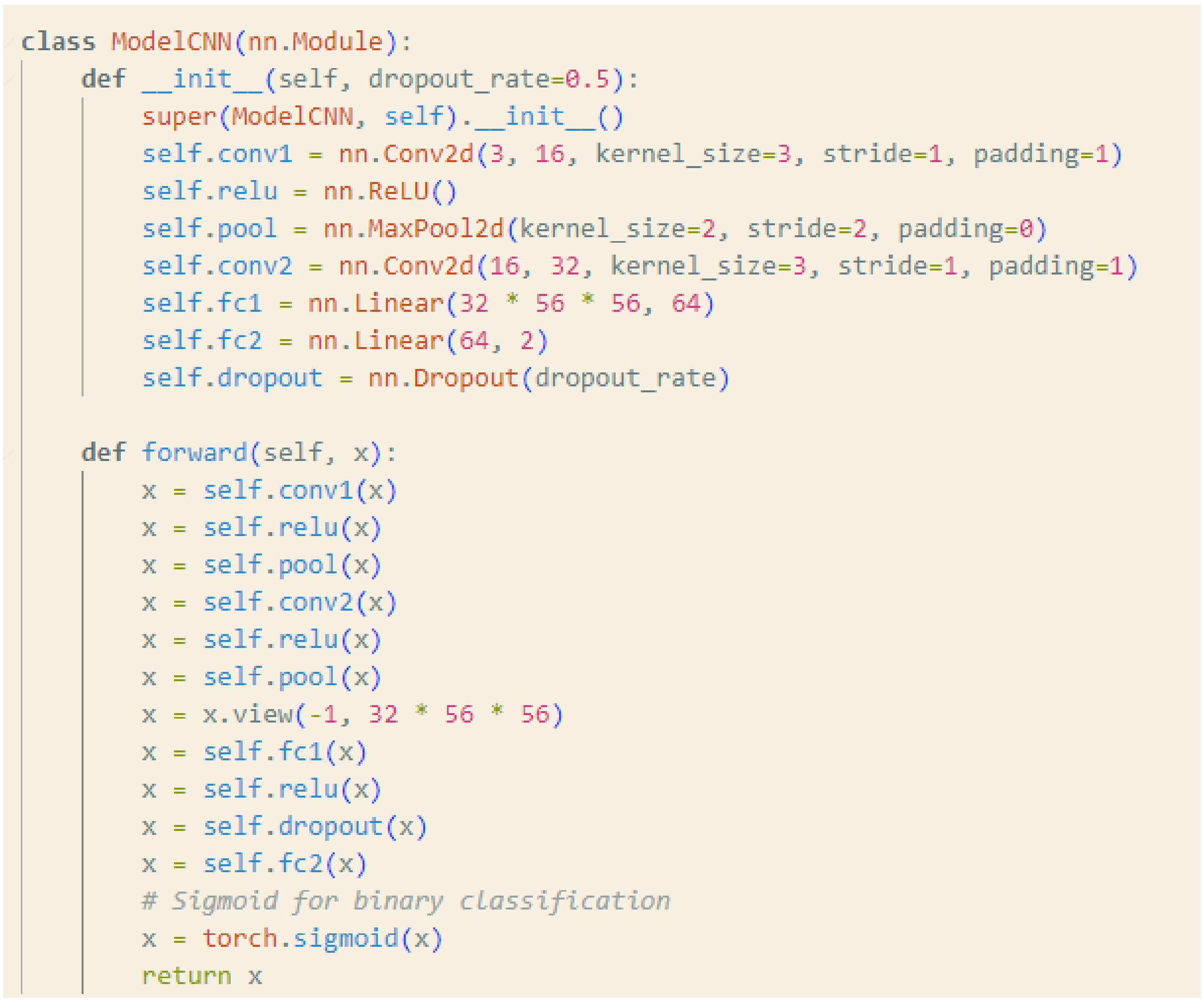
Figure 13: Sample CNN using 3x3 filter.[9]

4.1.2. CNN with First Split

4.1.2.1. Implementation

The implementation of the CNN model with the first data split started with loading and structuring the dataset in order to be used for the model. The model reaches the data from the local folder that the First Split is kept in. Separate datasets are created for train, test, and validation parts. These datasets were already separated in the First Split folder as train, test, and validation. However, our initial dataset contained more deepfake audio data than the original audio data in the training dataset. In order to prevent this from creating a bias in the classification of the model, the deepfake audio data files are selected in the same number as the existing original audio data files in the training dataset. This is done by random selection from the deepfake audio files. By keeping a balanced dataset, we attempted to reduce the bias of the CNN model. The sample data is also checked by printing some of the loaded spectrogram images with their labels.

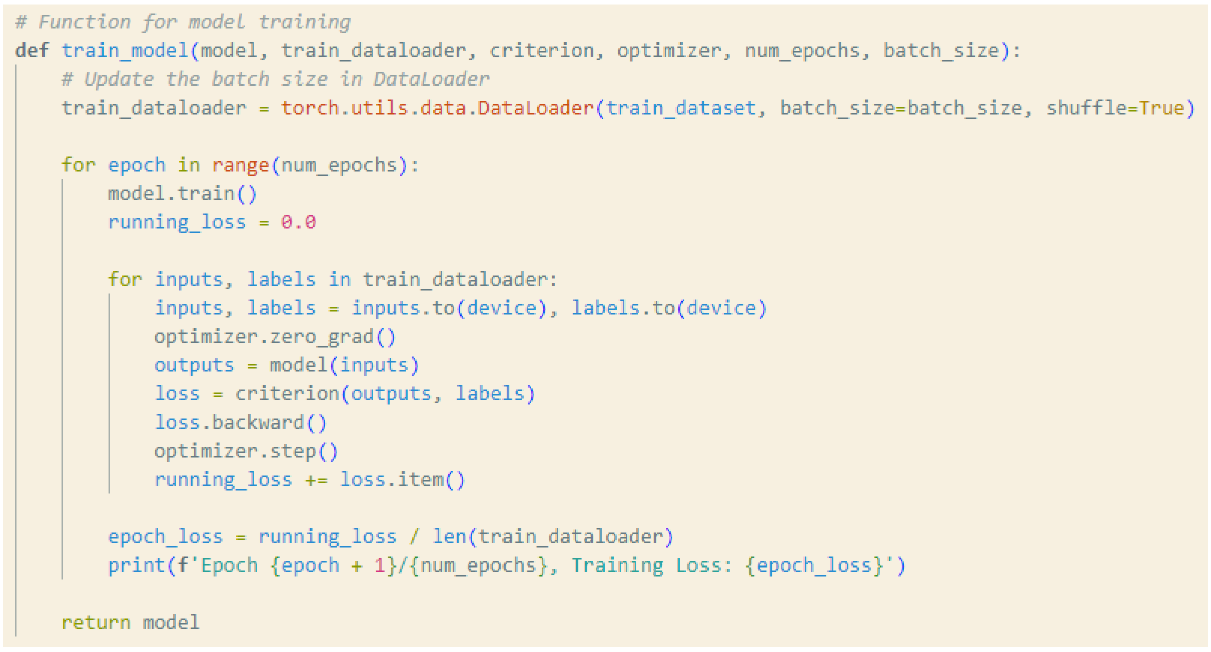
The CNN model is created afterward. The layer structure of the CNN model is as follows:



```python
class ModelCNN(nn.Module):
    def __init__(self, dropout_rate=0.5):
        super(ModelCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(32 * 56 * 56, 64)
        self.fc2 = nn.Linear(64, 2)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.pool(x)
        x = x.view(-1, 32 * 56 * 56)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        # Sigmoid for binary classification
        x = torch.sigmoid(x)
        return x
```

Figure 14

The model is trained with default parameters (learning rate=0.0001, dropout rate=0.5, batch size=32) for 15 epochs first, using the following function:

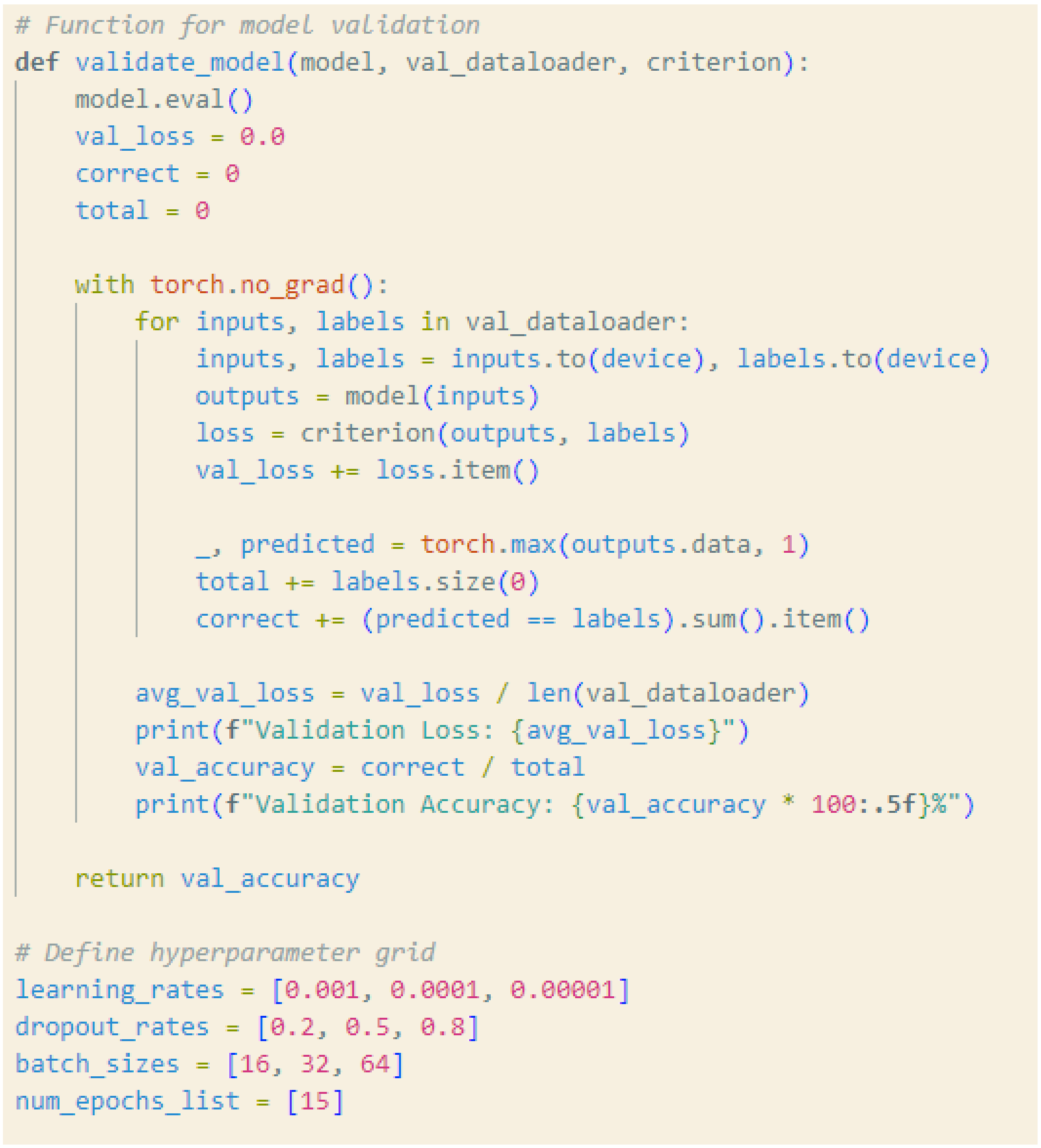


```
# Function for model training
def train_model(model, train_dataloader, criterion, optimizer, num_epochs, batch_size):
    # Update the batch size in DataLoader
    train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0

        for inputs, labels in train_dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        epoch_loss = running_loss / len(train_dataloader)
        print(f'Epoch {epoch + 1}/{num_epochs}, Training Loss: {epoch_loss}')

    return model
```

Figure 15

4.1.2.2. Hyperparameter Tuning

After training and testing the default model, in order to optimize the hyperparameter values, we decided to perform a grid search on the validation set and select the model configurations with the highest accuracy as the optimal model. The validation function and the possible configuration values are as follows:



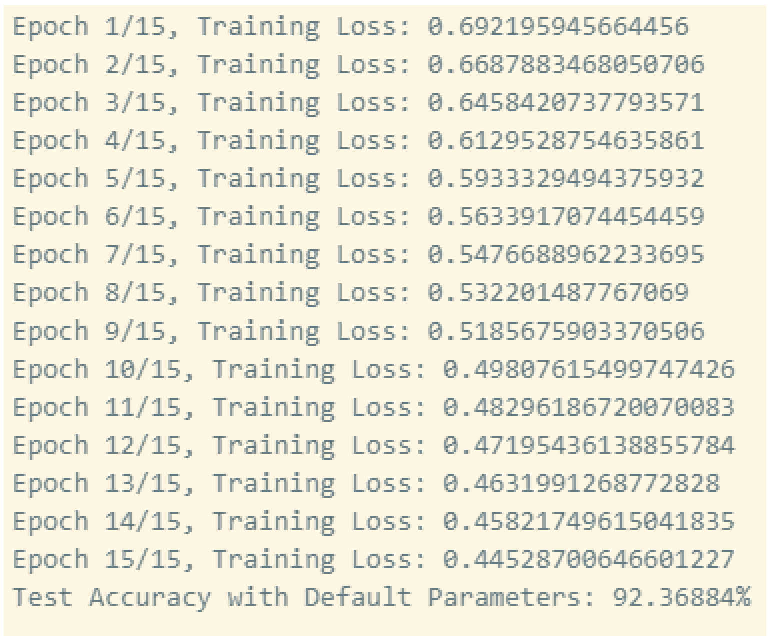
```python
# Function for model validation
def validate_model(model, val_dataloader, criterion):
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, labels in val_dataloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            val_loss += loss.item()

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        avg_val_loss = val_loss / len(val_dataloader)
        print(f"Validation Loss: {avg_val_loss}")
        val_accuracy = correct / total
        print(f"Validation Accuracy: {val_accuracy * 100:.5f}%")

    return val_accuracy

# Define hyperparameter grid
learning_rates = [0.001, 0.0001, 0.00001]
dropout_rates = [0.2, 0.5, 0.8]
batch_sizes = [16, 32, 64]
num_epochs_list = [15]
```

Figure 16

Consequently, the model was tested with 27 different configurations (3*3*3), and the best configuration was determined to be evaluated with the test set.

4.1.2.3. Results

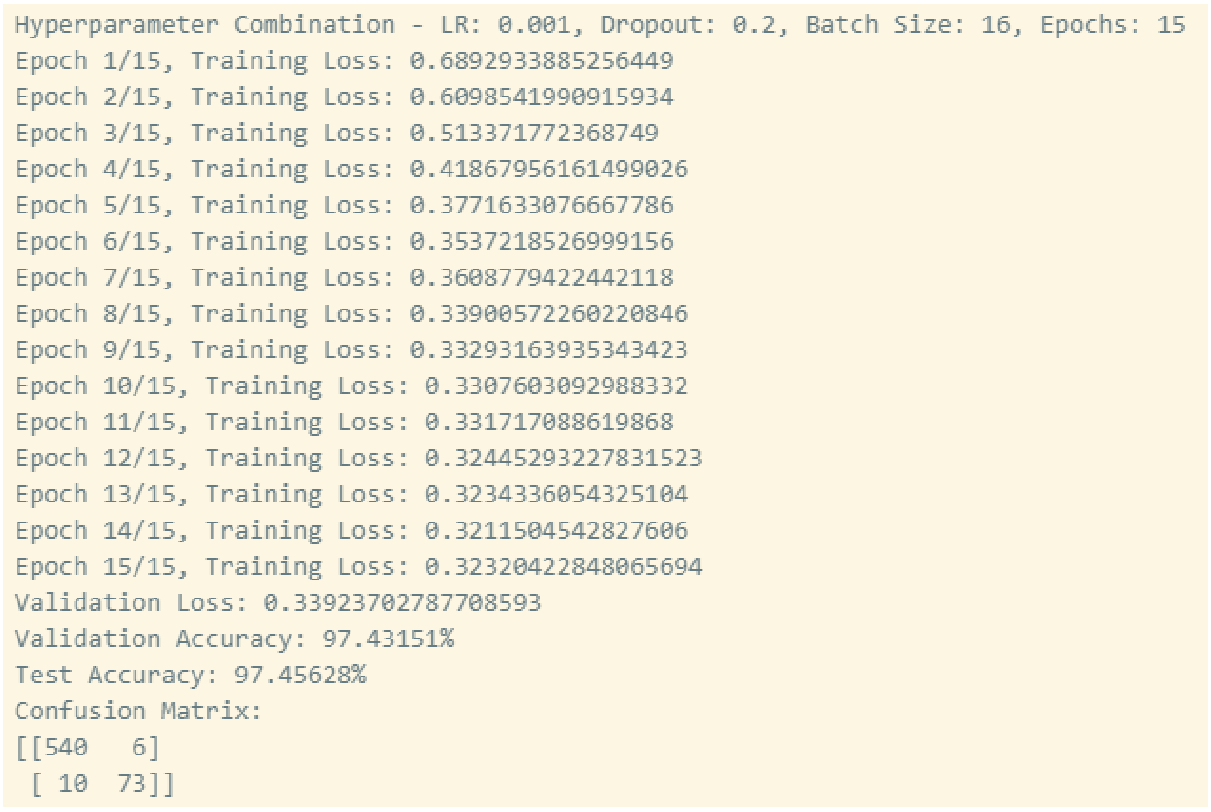
The model trained with the default parameters (learning rate=0.0001, dropout rate=0.5, batch size=32) provided an accuracy of 92.37%.



```
Epoch 1/15, Training Loss: 0.692195945664456
Epoch 2/15, Training Loss: 0.6687883468050706
Epoch 3/15, Training Loss: 0.6458420737793571
Epoch 4/15, Training Loss: 0.6129528754635861
Epoch 5/15, Training Loss: 0.5933329494375932
Epoch 6/15, Training Loss: 0.5633917074454459
Epoch 7/15, Training Loss: 0.5476688962233695
Epoch 8/15, Training Loss: 0.532201487767069
Epoch 9/15, Training Loss: 0.5185675903370506
Epoch 10/15, Training Loss: 0.49807615499747426
Epoch 11/15, Training Loss: 0.48296186720070083
Epoch 12/15, Training Loss: 0.47195436138855784
Epoch 13/15, Training Loss: 0.4631991268772828
Epoch 14/15, Training Loss: 0.45821749615041835
Epoch 15/15, Training Loss: 0.44528700646601227
Test Accuracy with Default Parameters: 92.36884%
```

Figure 17

The best configuration on the validation set provided an accuracy of 97.43% on the validation set, and 97.46% on the test set.



```
Hyperparameter Combination - LR: 0.001, Dropout: 0.2, Batch Size: 16, Epochs: 15
Epoch 1/15, Training Loss: 0.6892933885256449
Epoch 2/15, Training Loss: 0.6098541990915934
Epoch 3/15, Training Loss: 0.513371772368749
Epoch 4/15, Training Loss: 0.41867956161499026
Epoch 5/15, Training Loss: 0.3771633076667786
Epoch 6/15, Training Loss: 0.3537218526999156
Epoch 7/15, Training Loss: 0.3608779422442118
Epoch 8/15, Training Loss: 0.33900572260220846
Epoch 9/15, Training Loss: 0.33293163935343423
Epoch 10/15, Training Loss: 0.3307603092988332
Epoch 11/15, Training Loss: 0.331717088619868
Epoch 12/15, Training Loss: 0.32445293227831523
Epoch 13/15, Training Loss: 0.3234336054325104
Epoch 14/15, Training Loss: 0.3211504542827606
Epoch 15/15, Training Loss: 0.32320422848065694
Validation Loss: 0.33923702787708593
Validation Accuracy: 97.43151%
Test Accuracy: 97.45628%
Confusion Matrix:
[[540   6]
 [ 10  73]]
```

Figure 18

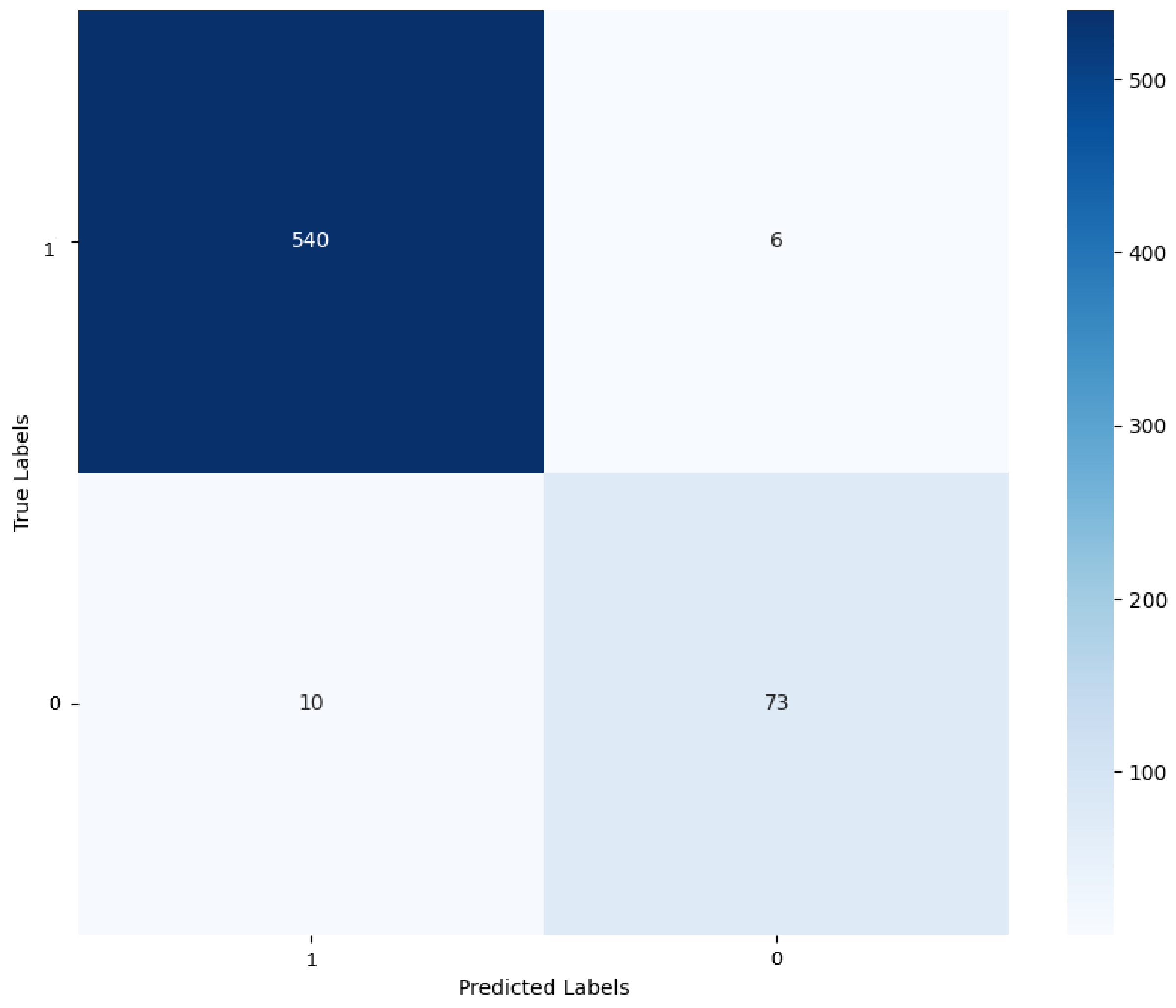


Figure 19

4.1.3. CNN with Second Split

4.1.3.1. Implementation

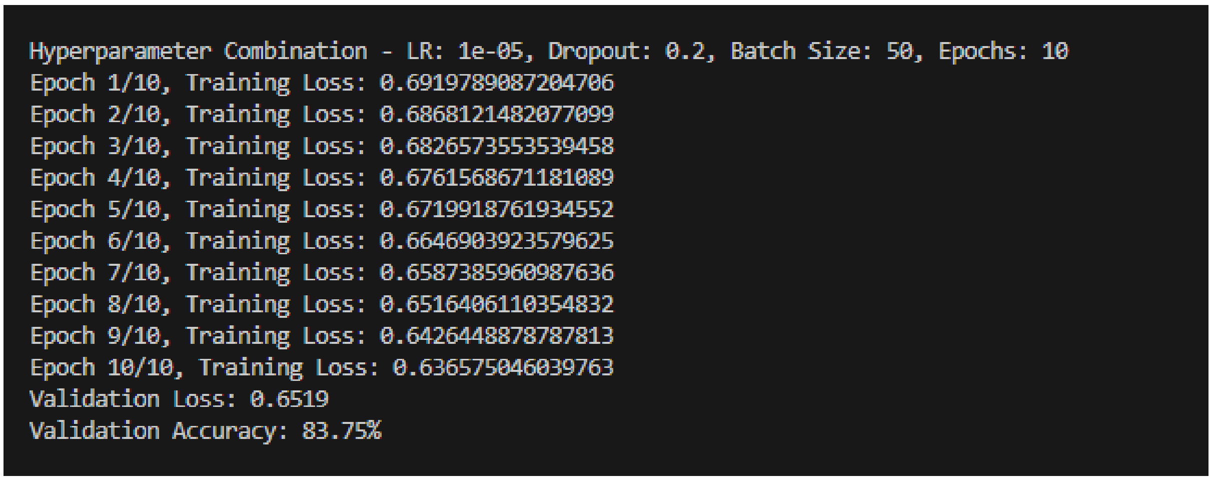
The process of setting up the CNN model using the second dataset division mentioned in section 2.2 began by preparing and organizing the dataset for model use. This involved accessing the data from a local directory where the second Split was stored. Training, testing, and validation datasets were formed separately. These categories were already distinct within the Second Split folder as train, test, and validation. However, the initial dataset had a higher quantity of deepfake audio compared to original audio in the training set. To counter potential bias in the model's classification, an equal number of deepfake audio files were chosen to match the count of original audio files in the training set. This selection was made randomly from the deepfake audio files. We aimed to diminish the CNN model's bias by maintaining a balanced dataset. Additionally, to verify the sample data, we displayed some of the loaded spectrogram images along with their labels.

The CNN model is created afterward. The layer structure of the CNN model is the same as section 4.1.2.1

The local environment and cuda are used in experiments.

4.1.3.2. Hyperparameter Tuning

The model was experimented with different hyperparameters.
One of the results of the experiments are in figure x:



```
Hyperparameter Combination - LR: 1e-05, Dropout: 0.2, Batch Size: 50, Epochs: 10
Epoch 1/10, Training Loss: 0.6919789087204706
Epoch 2/10, Training Loss: 0.6868121482077099
Epoch 3/10, Training Loss: 0.6826573553539458
Epoch 4/10, Training Loss: 0.6761568671181089
Epoch 5/10, Training Loss: 0.6719918761934552
Epoch 6/10, Training Loss: 0.6646903923579625
Epoch 7/10, Training Loss: 0.6587385960987636
Epoch 8/10, Training Loss: 0.6516406110354832
Epoch 9/10, Training Loss: 0.6426448878787813
Epoch 10/10, Training Loss: 0.636575046039763
Validation Loss: 0.6519
Validation Accuracy: 83.75%
```

Figure 20: Sample output from hyperparameter tuning

Then, according to validation set's accuracy, the model with the highest accuracy was selected and it is as such:

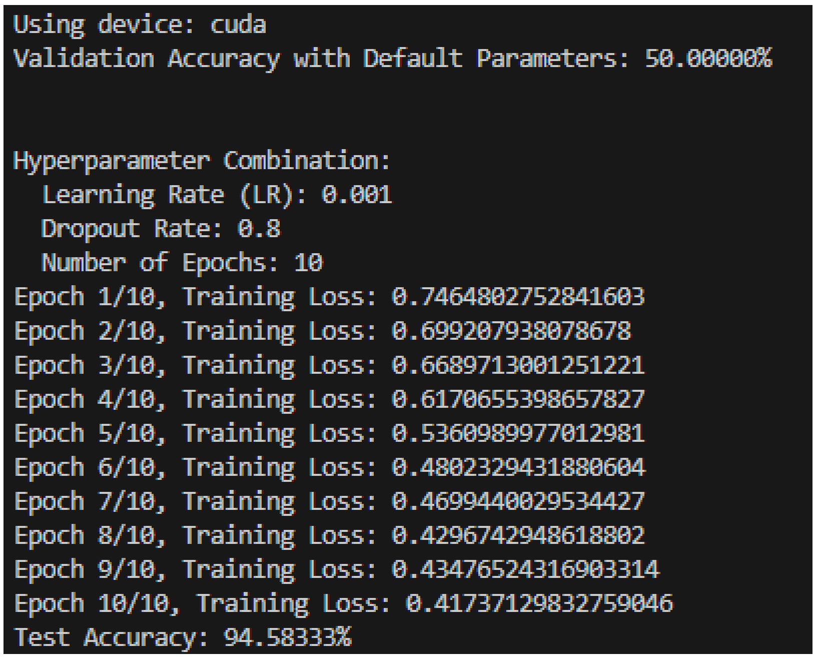
Learning Rate (LR): 0.001
Dropout Rate: 0.5
Number of Epochs: 10

After that, the epoch number was increased to 15. However, it was observed that after 10 epochs, training loss increases instead of decreasing. Therefore, the number of epochs are decreased to 10.

4.1.3.3. Results

Even if the model neither sees the transcript nor the voice, the model gives relatively high accuracy: %93 percent on average in the test set.

The model gives hardly and false positives. It gives 1 or 0 false positives generally. However, it gives some false negatives. Therefore, we can conclude that the model is a bit selective. Here is one of the confusion matrix. In there, accuracy is %94.5



```
Using device: cuda
Validation Accuracy with Default Parameters: 50.00000%


Hyperparameter Combination:
  Learning Rate (LR): 0.001
  Dropout Rate: 0.8
  Number of Epochs: 10
Epoch 1/10, Training Loss: 0.7464802752841603
Epoch 2/10, Training Loss: 0.699207938078678
Epoch 3/10, Training Loss: 0.6689713001251221
Epoch 4/10, Training Loss: 0.6170655398657827
Epoch 5/10, Training Loss: 0.5360989977012981
Epoch 6/10, Training Loss: 0.4802329431880604
Epoch 7/10, Training Loss: 0.4699440029534427
Epoch 8/10, Training Loss: 0.4296742948618802
Epoch 9/10, Training Loss: 0.43476524316903314
Epoch 10/10, Training Loss: 0.41737129832759046
Test Accuracy: 94.58333%
```

Figure 21: Output of test set accuracy experiment

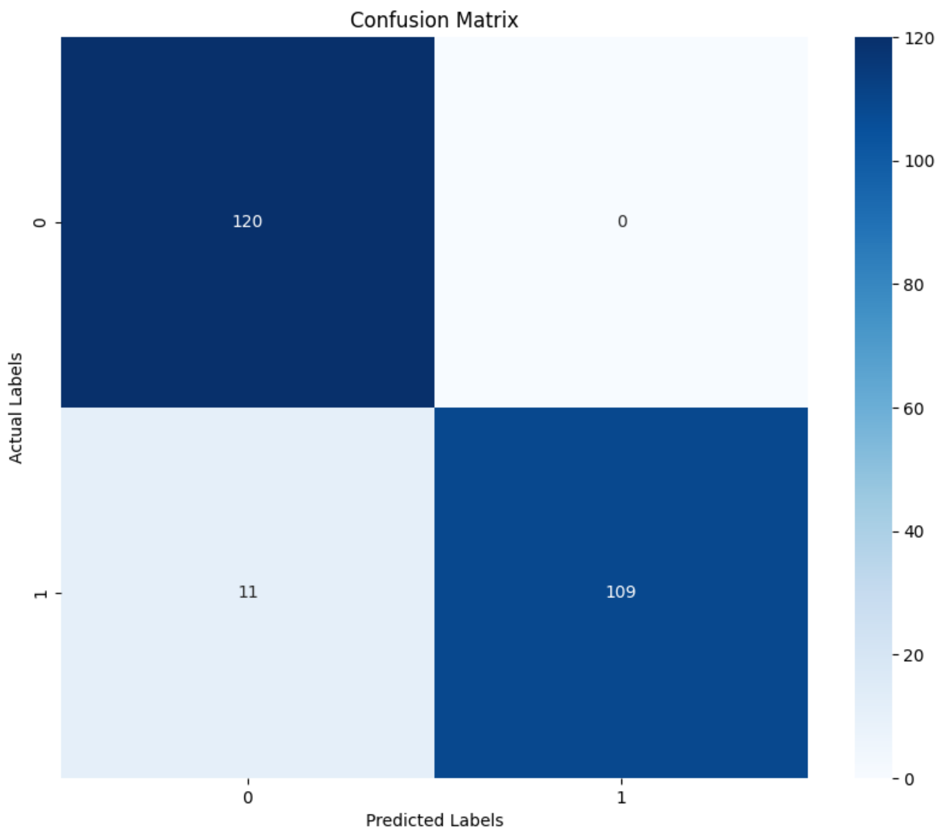


Figure 22: Confusion matrix

4.2. ResNet

4.2.1. Explanation of ResNet

Residual Networks is a deep neural network architecture that uses residual blocks. Residual blocks enable the model to facilitate the training by very deep networks. ResNet can be used for audio classification because it can effectively capture and learn patterns and dependencies within the audio data. Thus, it helps to distinguish the audio from deep fake and real classes. Using the residual connections that enable skipping one or more layers, ResNets helps to cope with the vanishing gradient problem and allows the model to be trained with deeper networks. For our audio classification project, the ResNet can take spectrogram image representations of 5-second audio intervals as input. The residual blocks enable the network to learn residual features and help to understand more accurately the complex patterns that exist in the audio data. This ability of ResNet can be valuable when finding the differences between deep fake and real audio file classification [10].

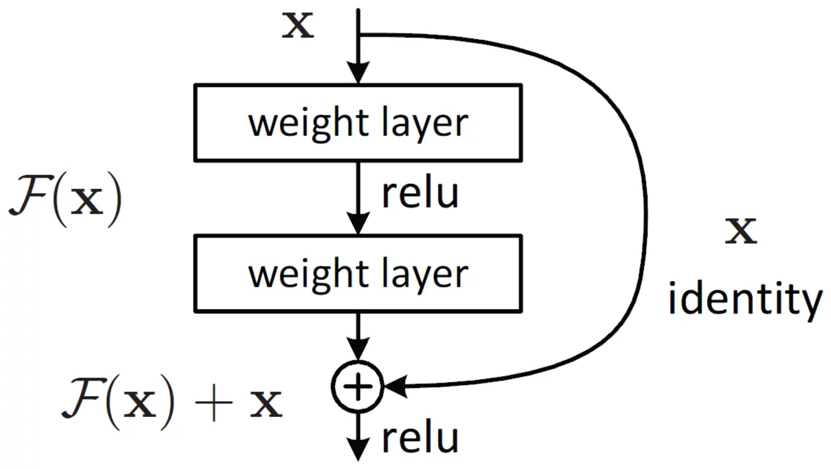

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

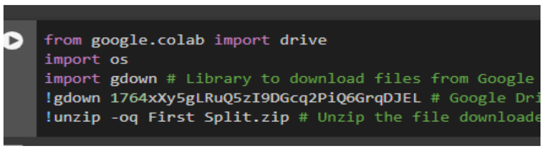
Figure 23: ResNet for Image Classification[11]

The use of ResNet is mainly to counter the degradation caused by increased depth in neural network structure. The addition to the network is the shortcut connections seen as an identity mapping (Figure 14), as an addition to the output, after the ReLu operation between weight layers. This formulation enables high-depth networks to operate with greater accuracy results by adjusting to fit the heightened training error rates which would otherwise go unnoticed [11]. By adjusting error rates according to the past layer inputs, the system is expected to outperform in high-depth networks. ResNet is useful in our research since we are expecting to have some depth due to our operations with the spectrogram images. In order to produce better results, the use of ResNet on images is expected to prove as a useful strategy.

4.2.2. ResNet with First Split

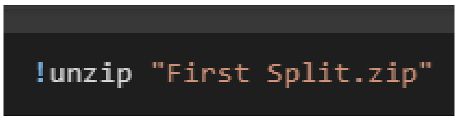
4.2.2.1. Implementation

The data is split into three files in the preprocessing phase. These are “Train,” Validation, and “Test. After retrieving the preprocessed data, they are zipped and uploaded to Google Drive. The data was downloaded and unzipped into Google Collab via the following code blocks:



```
from google.colab import drive
import os
import gdown # Library to download files from Google
!gdown 1764xXy5gLRuQ5zI9DGcq2PiQ6GrqDJEL # Google Dri
!unzip -oq First Split.zip # Unzip the file downloade
```

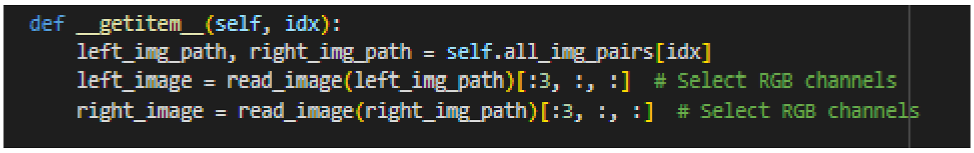
Figure 24: Imports



```
!unzip "First Split.zip"
```

Figure 25: Unzip

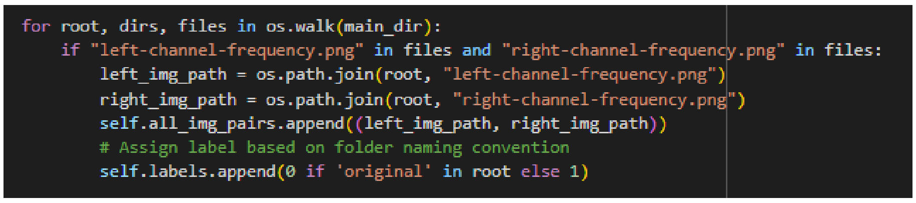
The image consisted of four channels: red, green, blue, and the alfa channel for transparency. The alfa channel was omitted, and only three of the channels are taken for the right and left channels.



```
def __getitem__(self, idx):
    left_img_path, right_img_path = self.all_img_pairs[idx]
    left_image = read_image(left_img_path)[:3, :, :]  # Select RGB channels
    right_image = read_image(right_img_path)[:3, :, :]  # Select RGB channels
```

Figure 26: getitem method

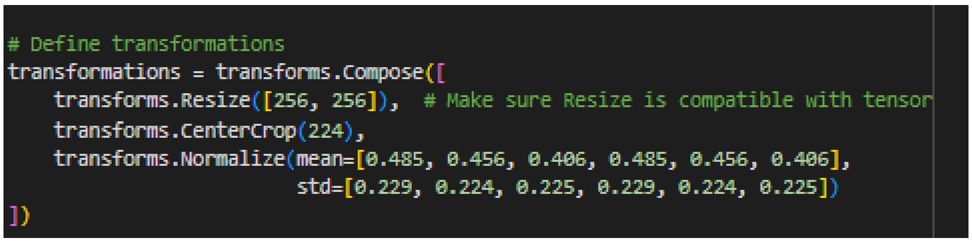
In order to train the model with frequency spectrogram images of two channels, right and left, the images are appended with the following code:



```
for root, dirs, files in os.walk(main_dir):
    if "left-channel-frequency.png" in files and "right-channel-frequency.png" in files:
        left_img_path = os.path.join(root, "left-channel-frequency.png")
        right_img_path = os.path.join(root, "right-channel-frequency.png")
        self.all_img_pairs.append((left_img_path, right_img_path))
        # Assign label based on folder naming convention
        self.labels.append(0 if 'original' in root else 1)
```

Figure 27: train model

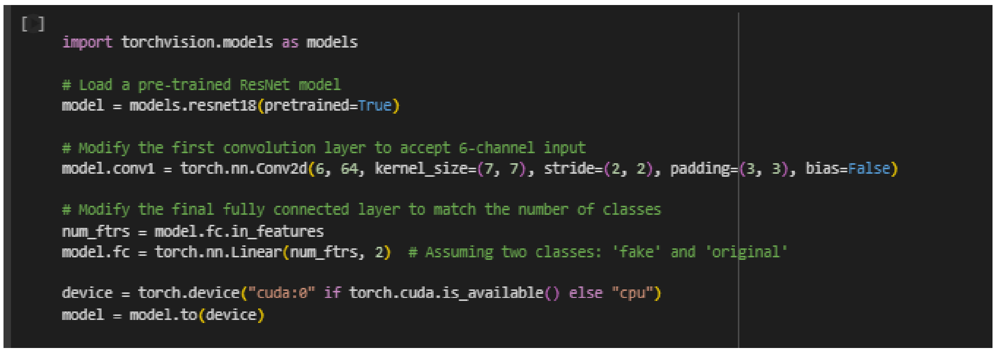
To prepare training the RGB channels of the two images, the transformation is needed. The transformation includes the resize, center crop, and normalization. Since there were two images for left and right spectrograms and three channels for each: red, green, and blue, the normalization is applied to six channels.



```
# Define transformations
transformations = transforms.Compose([
    transforms.Resize([256, 256]),  # Make sure Resize is compatible with tensor
    transforms.CenterCrop(224),
    transforms.Normalize(mean=[0.485, 0.456, 0.406, 0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225, 0.229, 0.224, 0.225])
])
```

Figure 28: transformation

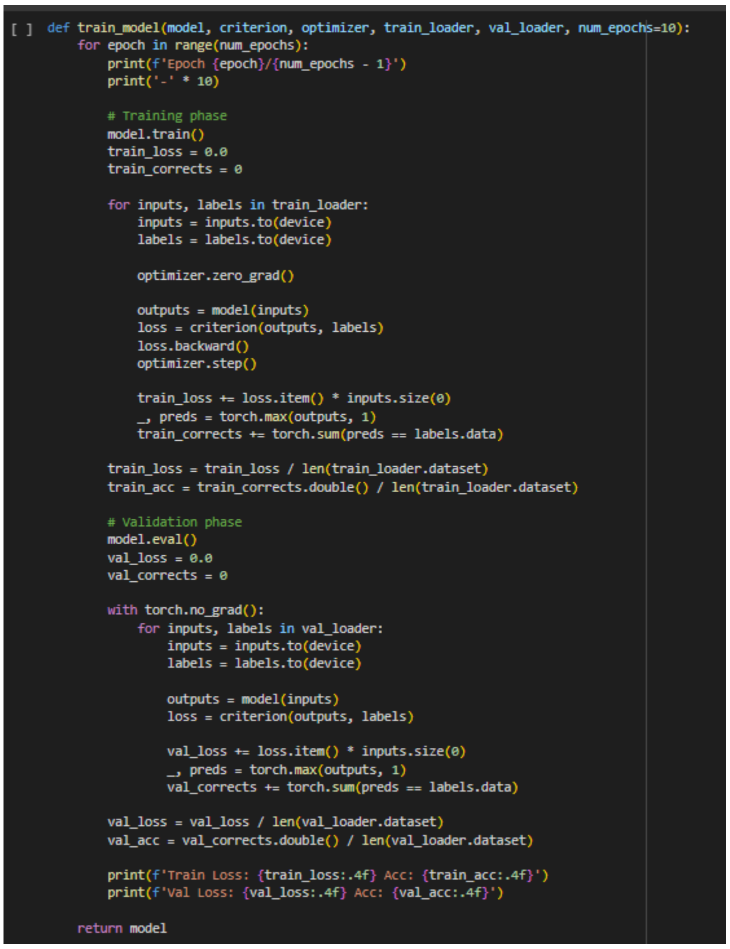
The unzipped three folders are labeled according to their folder names. If any directory exists that includes “original,” it means that the audio record belongs to a real voice record. Otherwise, it is classified as a deepfake. In order to use the resnet18 model, the torch library is utilized. For the image



```python
[ ]
import torchvision.models as models

# Load a pre-trained ResNet model
model = models.resnet18(pretrained=True)

# Modify the first convolution layer to accept 6-channel input
model.conv1 = torch.nn.Conv2d(6, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

# Modify the final fully connected layer to match the number of classes
num_ftrs = model.fc.in_features
model.fc = torch.nn.Linear(num_ftrs, 2)  # Assuming two classes: 'fake' and 'original'

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)
```

Figure 29: resnet model

For training, the model train_model function is written as the following image. It takes model, criterion, optimizer, train_loader, val_loader and num_epochs. The model parameters take the res18 model, criterion is the loss function used for training the model, the optimizer is the algorithm for optimization to update the model's parameters, train_loader is DataLoader for the validation dataset, val_loader the DataLoader for validation dataset and nun_epochs the number of training epochs which is set to 10 as default.



```python
[ ] def train_model(model, criterion, optimizer, train_loader, val_loader, num_epochs=10):
        for epoch in range(num_epochs):
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Training phase
            model.train()
            train_loss = 0.0
            train_corrects = 0

            for inputs, labels in train_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()

                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()

                train_loss += loss.item() * inputs.size(0)
                _, preds = torch.max(outputs, 1)
                train_corrects += torch.sum(preds == labels.data)

            train_loss = train_loss / len(train_loader.dataset)
            train_acc = train_corrects.double() / len(train_loader.dataset)

            # Validation phase
            model.eval()
            val_loss = 0.0
            val_corrects = 0

            with torch.no_grad():
                for inputs, labels in val_loader:
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    outputs = model(inputs)
                    loss = criterion(outputs, labels)

                    val_loss += loss.item() * inputs.size(0)
                    _, preds = torch.max(outputs, 1)
                    val_corrects += torch.sum(preds == labels.data)

            val_loss = val_loss / len(val_loader.dataset)
            val_acc = val_corrects.double() / len(val_loader.dataset)

            print(f'Train Loss: {train_loss:.4f} Acc: {train_acc:.4f}')
            print(f'Val Loss: {val_loss:.4f} Acc: {val_acc:.4f}')

        return model
```

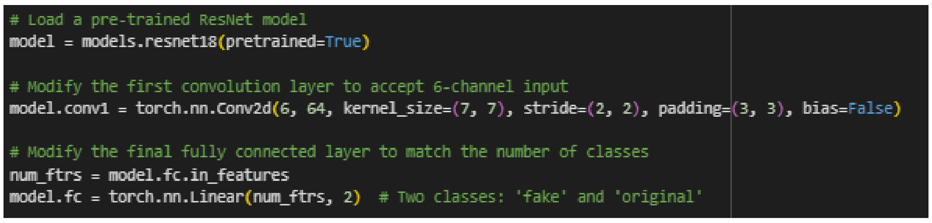
Figure 30

In order to train the model, high computational power is needed. For that reason, the GPU is utilized for the training.

4.2.2.2. Hyperparameter Tuning

Batch size: 32

Image transformations: Resize to [256, 256]
CenterCrop to (224, 224)
They are normalized with specified mean and standard deviation values.
Adam Optimizer Learning rate: 0.001



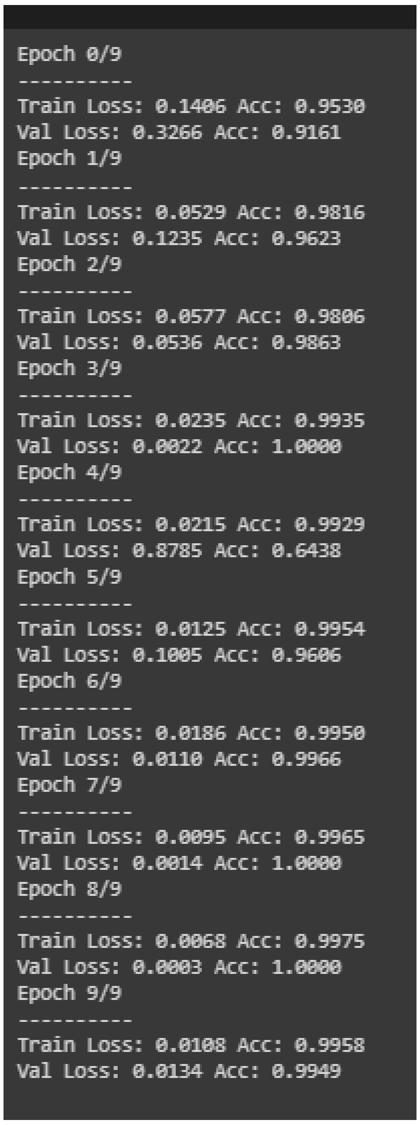
```
# Load a pre-trained ResNet model
model = models.resnet18(pretrained=True)

# Modify the first convolution layer to accept 6-channel input
model.conv1 = torch.nn.Conv2d(6, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)

# Modify the final fully connected layer to match the number of classes
num_ftrs = model.fc.in_features
model.fc = torch.nn.Linear(num_ftrs, 2)  # Two classes: 'fake' and 'original'
```

Figure 31

Number of epochs = 10

4.2.2.3. Results

After training the model with 10 epochs, the results were as follows:



```
Epoch 0/9
----------
Train Loss: 0.1406 Acc: 0.9530
Val Loss: 0.3266 Acc: 0.9161
Epoch 1/9
----------
Train Loss: 0.0529 Acc: 0.9816
Val Loss: 0.1235 Acc: 0.9623
Epoch 2/9
----------
Train Loss: 0.0577 Acc: 0.9806
Val Loss: 0.0536 Acc: 0.9863
Epoch 3/9
----------
Train Loss: 0.0235 Acc: 0.9935
Val Loss: 0.0022 Acc: 1.0000
Epoch 4/9
----------
Train Loss: 0.0215 Acc: 0.9929
Val Loss: 0.8785 Acc: 0.6438
Epoch 5/9
----------
Train Loss: 0.0125 Acc: 0.9954
Val Loss: 0.1005 Acc: 0.9606
Epoch 6/9
----------
Train Loss: 0.0186 Acc: 0.9950
Val Loss: 0.0110 Acc: 0.9966
Epoch 7/9
----------
Train Loss: 0.0095 Acc: 0.9965
Val Loss: 0.0014 Acc: 1.0000
Epoch 8/9
----------
Train Loss: 0.0068 Acc: 0.9975
Val Loss: 0.0003 Acc: 1.0000
Epoch 9/9
----------
Train Loss: 0.0108 Acc: 0.9958
Val Loss: 0.0134 Acc: 0.9949
```

Figure 32

After training the model, the results with the test set had 98.41% accuracy with the following confusion matrix:

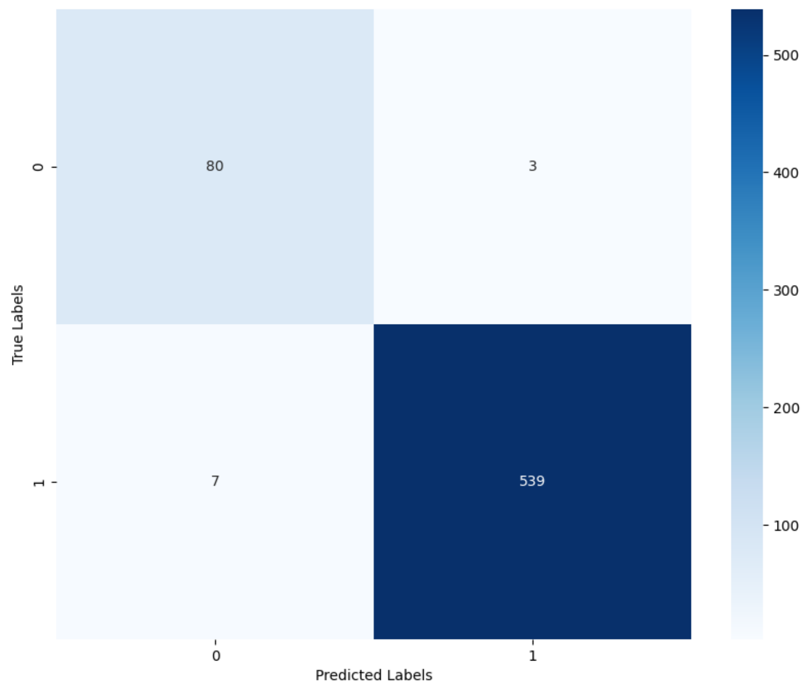


Figure 33

4.2.3. ResNet with Second Split

We have adjusted the ResNet with the first split to the second split and after adjusting the hyperparameters, we have found some results. The second split has unbiased data and has no knowledge of the speaker or their transcript in the test set, hence the expected result is to have a lesser accuracy compared to the ResNet with the first split.

4.2.3.1. Implementation

In our implementation, the training split included all spectrograms that belonged to neither Trump nor Biden's transcript or audio. The test split and validation splits had only one of the transcripts and one of the voices of Trump or Biden. Hence, one set had Biden's original voice on Biden's transcript (original audio), while the other had Biden's voice on Trump's transcript (fake audio). We also have the same amount of spectrograms of originals and fakes, to prevent any data imbalance.
We thus give these files to the ResNet by setting them as DataLoaders in the following code.

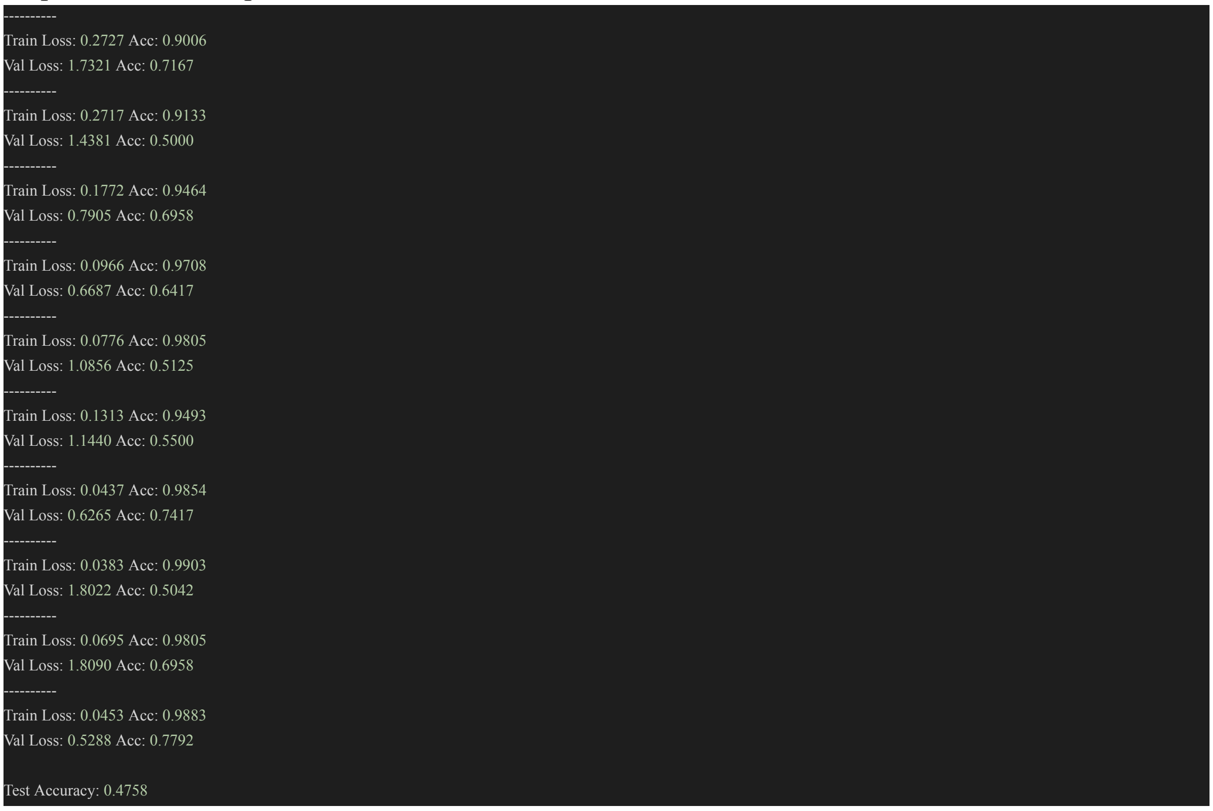
```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=True, num_workers=4)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=True, num_workers=4)
```

We also create CustomSpectogramDataset instances on train, validation, and test sets. This operation checks if the files contain the left and right spectrograms for each set. An earlier error had caused the spectrogram files to be corrupted, causing some of the dataset to disappear, while leaving some files with singular channels of amplitudes. To correct this issue, we check if both channels of right and left audio spectrograms of a stereo voice exist. If one exists and the other is unavailable, we take it as a mono audio and have both channels take the same available input. This mentioned solution could be used if an audio dataset is mixed with mono audio files along with stereo audio, do note that we were able to find the issue of why the spectrogram data was corrupted and why it only showed half of the data. The issue was related to Google Drive, and when someone other than the owner of the file

accessed a large set of files and data and tried to zip download the files, the data was downloaded in a corrupted manner. To solve the issue, we have shared the zipped file directly and the corruption happened no more. Here and in the overall report, we are able to see that the split 2 sets have the same amount of data.
After resizing the spectrogram images and taking them to left, and right channels respectively, and to RGB channels, we transform the images as was stated in the first split's ResNet implementation. Do note that overall, the structure of the code follows similarly to the previous section of ResNet on the first split, the main difference being the change of input sets, as well as the chosen hyperparameters, and received different results accordingly.

4.2.3.2. Hyperparameter Tuning

For our hyperparameters, we have adjusted the learning rate and checked how the training proceeds per each epoch. We have observed that on every learning rate, given enough epochs, the dataset started overfitting and became too accustomed to people's vocal patterns on the training data. Therefore, another parameter we have tuned was where to early stop while training the data, which is equivalent to the number of epochs.
First of all, we have decided to have the learning rate as 0.001, which is the ResNet hyperparameter we had chosen for the ResNet with the first split implementation's best case. The following is our output result on 10 epochs:



```
----------
Train Loss: 0.2727 Acc: 0.9006
Val Loss: 1.7321 Acc: 0.7167
----------
Train Loss: 0.2717 Acc: 0.9133
Val Loss: 1.4381 Acc: 0.5000
----------
Train Loss: 0.1772 Acc: 0.9464
Val Loss: 0.7905 Acc: 0.6958
----------
Train Loss: 0.0966 Acc: 0.9708
Val Loss: 0.6687 Acc: 0.6417
----------
Train Loss: 0.0776 Acc: 0.9805
Val Loss: 1.0856 Acc: 0.5125
----------
Train Loss: 0.1313 Acc: 0.9493
Val Loss: 1.1440 Acc: 0.5500
----------
Train Loss: 0.0437 Acc: 0.9854
Val Loss: 0.6265 Acc: 0.7417
----------
Train Loss: 0.0383 Acc: 0.9903
Val Loss: 1.8022 Acc: 0.5042
----------
Train Loss: 0.0695 Acc: 0.9805
Val Loss: 1.8090 Acc: 0.6958
----------
Train Loss: 0.0453 Acc: 0.9883
Val Loss: 0.5288 Acc: 0.7792

Test Accuracy: 0.4758
```

Here, we observe that the training accuracy starts at 90%, then increases to 98% on the tenth epoch with many fluctuations on each epoch change. However, we see a stranger phenomenon in the validation set. We observe that the model is doing little learning, and the highest value we have is 77% accuracy on the validation set, while the test set has 47% accuracy, which is the same as the 50% rate of random guess. This is completely unusable since a random guess is supposed to theoretically be equal to 50% accuracy. We understand two things, first of all, there are much too high fluctuations in the training set, meaning that our learning rate is too high and is jumping around the minimal

gradient while the gradient descending. Furthermore, either there is overfitting or the model is not learning the right differences between a fake and original audio spectrogram.
In our next hyperparameter, we try with a learning rate of 0.01, which is 10 times higher than our previous case. We would normally try to decrease the hyperparameter since we observed that it is supposedly too high, and thus is causing fluctuations in the learning rate. However, by observing the output on an even higher learning rate, we can see if the issue gets worse- meaning that the issue was possibly caused by the high learning rate. The output on ten epochs is given as:

```
----------
Epoch 1/8
Train Loss: 1.1276 Acc: 0.5916
Val Loss: 20.9049 Acc: 0.5000
Time taken for epoch:6.08 minutes (364.82 seconds)
----------
Epoch 2/8
Train Loss: 0.6614 Acc: 0.6793
Val Loss: 252.6774 Acc: 0.5000
Time taken for epoch:6.05 minutes (363.20 seconds)
----------
Epoch 3/8
Train Loss: 0.5778 Acc: 0.7349
Val Loss: 0.8717 Acc: 0.4333
Time taken for epoch:6.14 minutes (368.18 seconds)
----------
Epoch 4/8
Train Loss: 0.5044 Acc: 0.7914
Val Loss: 1.4914 Acc: 0.4917
Time taken for epoch:6.06 minutes (363.41 seconds)
----------
Epoch 5/8
Train Loss: 0.3891 Acc: 0.8548
Val Loss: 3.2683 Acc: 0.5000
Time taken for epoch:6.01 minutes (360.33 seconds)
----------
Epoch 6/8
Train Loss: 0.3191 Acc: 0.8821
Val Loss: 0.7343 Acc: 0.5542
Time taken for epoch:6.14 minutes (368.47 seconds)
----------
Epoch 7/8
Train Loss: 0.2121 Acc: 0.9308
Val Loss: 154.5649 Acc: 0.5000
Time taken for epoch:5.96 minutes (357.64 seconds)
----------
Epoch 8/8
Train Loss: 0.4723 Acc: 0.8060
Val Loss: 1.0474 Acc: 0.5042
Time taken for epoch:5.96 minutes (357.53 seconds)

Test Accuracy: 0.4958
```

The output here is slightly different since we have changed the print statements slightly, but the code is the same overall. We observe that the problem has gotten worse, since the training accuracy shows to jump to 93 then back to 80, and fluctuates at a more alarming magnitude. Hence, we understand that this issue is most likely related to the learning rate being too high on 0.01 and 0.001 cases, thus we decrease it to 0.0001. The following is the output:

```
----------
Epoch 1/8
                Training Complete
Train Loss: 0.1717 Acc: 0.9259
Val Loss: 0.1525 Acc: 0.9375
Time taken for epoch:5.53 minutes (331.99 seconds)
----------
Epoch 2/8
                Training Complete
Train Loss: 0.0421 Acc: 0.9825
Val Loss: 0.1231 Acc: 0.9667
Time taken for epoch:3.57 minutes (213.94 seconds)
----------
Epoch 3/8
```

```
            Training Complete
Train Loss: 0.0165 Acc: 0.9951
Val Loss: 0.6068 Acc: 0.7167
Time taken for epoch:3.64 minutes (218.52 seconds)
----------
Epoch 4/8
            Training Complete
Train Loss: 0.0197 Acc: 0.9942
Val Loss: 0.7546 Acc: 0.6792
Time taken for epoch:3.60 minutes (215.96 seconds)
----------
Epoch 5/8
            Training Complete
Train Loss: 0.0560 Acc: 0.9795
Val Loss: 0.1920 Acc: 0.9375
Time taken for epoch:3.64 minutes (218.50 seconds)
----------
Epoch 6/8
            Training Complete
Train Loss: 0.0550 Acc: 0.9815
Val Loss: 0.2905 Acc: 0.8875
Time taken for epoch:3.68 minutes (221.10 seconds)
----------
Epoch 7/8
            Training Complete
Train Loss: 0.0618 Acc: 0.9756
Val Loss: 0.3868 Acc: 0.8375
Time taken for epoch:3.60 minutes (216.19 seconds)
----------
Epoch 8/8
            Training Complete
Train Loss: 0.0294 Acc: 0.9893
Val Loss: 0.2170 Acc: 0.9417
Time taken for epoch:3.61 minutes (216.75 seconds)
Test Accuracy: 0.8333
```

This output shows that our model has learned, and the test accuracy jumped to a 83%, much higher than the previous results. Here, we observe that there are some fluctuations on training accuracy, although much smaller compared to the previous results. We thus try to decrease the learning rate further, to 0.00005- which is half of the previous learning rate. The output is as follows:

```
----------
Epoch 1/10
            Training Complete
Train Loss: 0.0399 Acc: 0.9893
Val Loss: 0.5389 Acc: 0.7375
Time taken for epoch:3.71 minutes (222.85 seconds)
----------
Epoch 2/10
            Training Complete
Train Loss: 0.0168 Acc: 0.9951
Val Loss: 0.4816 Acc: 0.7625
Time taken for epoch:3.60 minutes (215.76 seconds)
----------
Epoch 3/10
            Training Complete
Train Loss: 0.0168 Acc: 0.9956
Val Loss: 0.4905 Acc: 0.7321
Time taken for epoch:3.60 minutes (215.76 seconds)
----------
Epoch 4/10
            Training Complete
Train Loss: 0.0197 Acc: 0.9961
Val Loss: 0.4068 Acc: 0.8208
Time taken for epoch:3.63 minutes (217.94 seconds)
----------
Epoch 5/10
            Training Complete
Train Loss: 0.0193 Acc: 0.9961
Val Loss: 0.2913 Acc: 0.8833
Time taken for epoch:3.62 minutes (217.23 seconds)
----------
Epoch 6/10
            Training Complete
```

```
Train Loss: 0.0157 Acc: 0.9971
Val Loss: 0.2714 Acc: 0.8792
Time taken for epoch:3.74 minutes (224.39 seconds)
----------
Epoch 7/10
                Training Complete
Train Loss: 0.0069 Acc: 0.9990
Val Loss: 0.2252 Acc: 0.9042
Time taken for epoch:3.64 minutes (218.44 seconds)
----------
Epoch 8/10
                Training Complete
Train Loss: 0.0035 Acc: 0.9990
Val Loss: 0.4582 Acc: 0.8042
Time taken for epoch:3.64 minutes (218.65 seconds)
----------
Epoch 9/10
                Training Complete
Train Loss: 0.0120 Acc: 0.9981
Val Loss: 0.2720 Acc: 0.8833
Time taken for epoch:3.73 minutes (223.52 seconds)
----------
Epoch 10/10
                Training Complete
Train Loss: 0.0152 Acc: 0.9981
Val Loss: 0.5108 Acc: 0.7833
Time taken for epoch:3.63 minutes (217.79 seconds)

Test Accuracy: 0.7125
```

We observe that the validation accuracy was lesser than the previous learning rate overall, and we can clearly see the overfitting, since the training accuracy keeps increasing until the last epoch, while the validation accuracy seems to decrease after the 7th epoch. The situation seems to have been better overall in the case where the learning rate was 0.0001 instead. Hence, we decide to take 0.0001 as the learning rate and try different epoch numbers to find an epoch with the highest accuracy. Upon experimenting with different epoch sizes, we observe that overall, the model starts overfitting after the 5th epoch, and the accuracy drops towards 50% after the 10th epoch gradually, and after the 20th epoch, the model is guessing mostly “true” as the label, since it can not differentiate. Hence, we decide to perform early stopping. Below is the output to one of the models we have trained:

```
lr=0.0001
----------
Epoch 1/5
                Training Complete
Train Loss: 0.1623 Acc: 0.9366
Val Loss: 0.7103 Acc: 0.5958
Time taken for epoch:3.75 minutes (225.23 seconds)
----------
Epoch 2/5
                Training Complete
Train Loss: 0.0434 Acc: 0.9864
Val Loss: 0.8752 Acc: 0.5708
Time taken for epoch:3.76 minutes (225.50 seconds)
----------
Epoch 3/5
                Training Complete
Train Loss: 0.0321 Acc: 0.9903
Val Loss: 0.3097 Acc: 0.8750
Time taken for epoch:3.66 minutes (219.87 seconds)
----------
Epoch 4/5
                Training Complete
Train Loss: 0.0575 Acc: 0.9834
```

```
Val Loss: 0.6937 Acc: 0.7000
Time taken for epoch:3.66 minutes (219.82 seconds)
----------
Epoch 5/5
                    Training Complete
Train Loss: 0.0261 Acc: 0.9932
Val Loss: 0.1092 Acc: 0.9708
Time taken for epoch:3.74 minutes (224.31 seconds)

Test Accuracy: 0.9417
```

The graph, which we have refrained from showing in the previous outputs since it would otherwise occupy much space, is given below for the best-case hyperparameters:

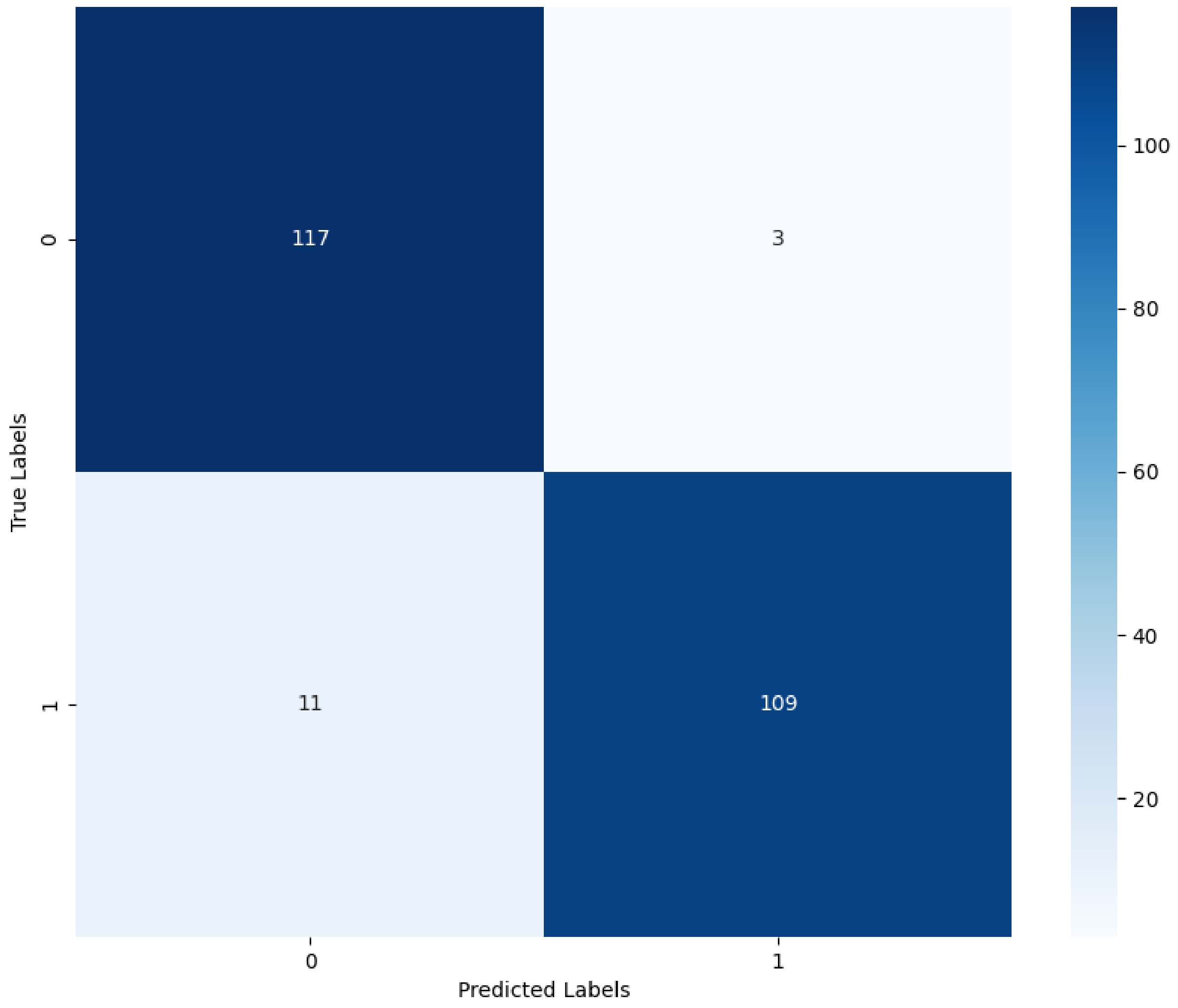


Figure 34: Test-set confusion matrix for the best case of ResNet second split

We observe that we receive a training accuracy of 99.32%, a validation accuracy of 97.08%, and the accuracy on our test set is as high as 94.17%. This is the best result we have obtained, and further changing the number of epochs, batch sizes, and learning rates made little to no improvement to our model accuracy and run time. Please do note that we have further improved our hyperparameters after our presentation, where we received 78% as the best-case performance. The previous falsely calculated result was obtained since half of the data was corrupted and we were only later in our research, were able to find the source of the problem and solve it before writing this report. This change has altered the best-performing hyperparameters for the ResNet second split case, thus affecting the overall discussion derived from our overall study slightly (in the section "Discussion and Comparing the Results" where we have derived our conclusions).

4.2.3.3. Results

Perhaps due to the nature of ResNet, on models where the training set is limited and the validation, test sets have too different data, the models perform poorly since it is too easy to encounter overfitting. There are 18 layers in the ResNet, which is the minimal value given in the libraries we have utilized, perhaps a reason why overfitting happens early. Our data has a handful of people's voices and arguably none of the transcripts nor the voice of Trump has any evident similarity to any of the other voices within our data, hence meaning that it is prone to overfitting in models that learn too reliant to the training dataset. We believe that the amount of data in our training set was insufficient to produce high enough accuracy in a large number of network neurons and layers within the ResNet structure. Given a better training set, we may be able to produce far better results, but according to our current knowledge and available data, the best-performing case is seen to be underperforming compared to our CNN with the second split approach. Nevertheless, an accuracy of 94% can be considered as a quite high success rate, better than what we had in mind before we implemented this part. In a random guess, a model would be able to perform with 50% accuracy, and having 94% is close to a highly accurate distinction on the separation of fake and real audio.

5. Discussion and Comparing the Results

The results we can derive from our study are various, yet we are unable to state only a single best-performing model due to the duality of our splits. For the first split case where we have the bias that the model already knows the person's voice, we had high expectations before beginning the project, and these expectations were fulfilled. The best case here was ResNet, which performed outstandingly and gave us an accuracy as high as 97.53%. This is much higher than all other model performances.

For the second split's case where we have no bias and the model does not have any previous knowledge of the voice of the speaking actor, we had low expectations before we had implemented the model. We thought that for the model to perform the action of successfully identifying original and deepfake voices, it would need to know the person's original voice or it would be close to impossible to identify correctly. However, opposite to our expectations, both CNN and ResNet have given us much higher accuracies than our expectations; (average) 91% and (at most) 94%. This result is particularly high since the trained model would be a powerful tool for finding if a voice is original. especially for this result, we derive that our aim to create an ML model capable of deepfake voice detection was successful. We believe that ResNet has performed better in both cases, with its best models outperforming CNN models. However, CNN follows closely behind for both split cases.

We have observed that our trained models are able to perform well in identifying an unknown human's voice correctly for 5-second voice files at above one out of ten cases of the time when trained with an arguably small dataset. We argue that the dataset was small since we have encountered overfitting several times and had to perform early stopping to counter any less-performing accuracy models that are too accustomed to the training data. This observation meant that our models would need more data to reach their full potential. If the person's original voice and deepfake voice are already known by our model, the model has a much higher overall expected rate of accuracy in finding the correct label for a voice.

Furthermore, the model is able to define if a voice is original or fake, just by observing a 5-second interval, which means, if the model was given as a speech of length 1 minute, we could run the trained model for 18 different spectrograms. This would skyrocket the model's accuracy, nearing 99% accuracy overall. Since we will be taking more than half of the guesses for 18 trials to evaluate a result, we can calculate the exact probability with binomial distribution. Observe that the binomial probability P(X > x) where the probability of more than 9 successes (x) in 18 trials (n) with a probability of success on a single trial (p) is 0.9417 would give a binomial probability of: 0.99999976683561 (found with binomial distribution formula). This is well above 99.99% accuracy, meaning that our model is able to identify between fake and real voices highly accurately for a given 1-minute audio, even if the person's voice or transcript is unknown beforehand. The model is therefore expected to perform outstandingly in real-world applications and could be even further improved with more data. The Deepfake voice files we had in our datasets were varying, some around 2 minutes, and even in a given 1-minute voice, our implementation is expected to have an accuracy of greater than 99.99%, which is as good as the end result could get.

6. Workload Distribution

Oğuz: Pre-processing, progress report, CNN model training with 2nd dataset
Onurcan:1st part of dataset split, progress report, CNN model training with 1st dataset
Alperen: 2nd part of dataset split, progress report, ResNet model training with 2nd dataset.
Kutay: Helped with codes of both dataset splits, progress report, ResNet model training with 1st dataset.

7. References

[1]K. T. Mai, S. D. Bray, T. Davies, and L. D. Griffin, “Warning: Humans Cannot Reliably Detect Speech Deepfakes,” PLOS ONE, vol. 18, no. 8, pp. [Online]. DOI: https://doi.org/10.1371/journal.pone.0285333. [Accessed: 18-Oct-2023].

[2] “Pandas,” pandas. [Online]. Available: https://pandas.pydata.org/. [Accessed: 20-Nov-2023].

[3] ”Numpy,” NumPy. [Online]. Available: https://numpy.org/. [Accessed: 20-Nov-2023].

[4] “Matplotlib 3.7.1 documentation,” Matplotlib documentation - Matplotlib 3.7.1 documentation. [Online]. Available: https://matplotlib.org/stable/index.html. [Accessed: 20-Nov-2023].

[5] “Librosa,” librosa, https://librosa.org/doc/latest/index.html. [[Accessed: Nov-25-2023].

[6] “Pytorch,” PyTorch, https://pytorch.org/. [Accessed: Nov-20-2023].

[7] Google colab, https://colab.research.google.com/. [Accessed: Nov-20-2023].

[8] “What are convolutional neural networks?,” IBM, https://www.ibm.com/topics/convolutional-neural-networks. [Accessed: Nov-20-2023].

[9]Wikimedia Commons, "CNN-filter-animation-1.gif," [Online]. Available: https://commons.wikimedia.org/wiki/File:CNN-filter-animation-1.gif. [Accessed: 24-11-2023].

[10] “Papers with code - resnet explained,” ResNet Explained | Papers With Code, https://paperswithcode.com/method/resnet. [Accessed: Nov-20-2023].

[11] M. ul Hassan, “ResNet (34, 50, 101): Residual CNNS for Image Classification Tasks,” Neurohive, https://neurohive.io/en/popular-networks/resnet/ [Accessed: Nov-25-2023].