

## Multimedia final project

### Audio classification using MLP and CNN architectures

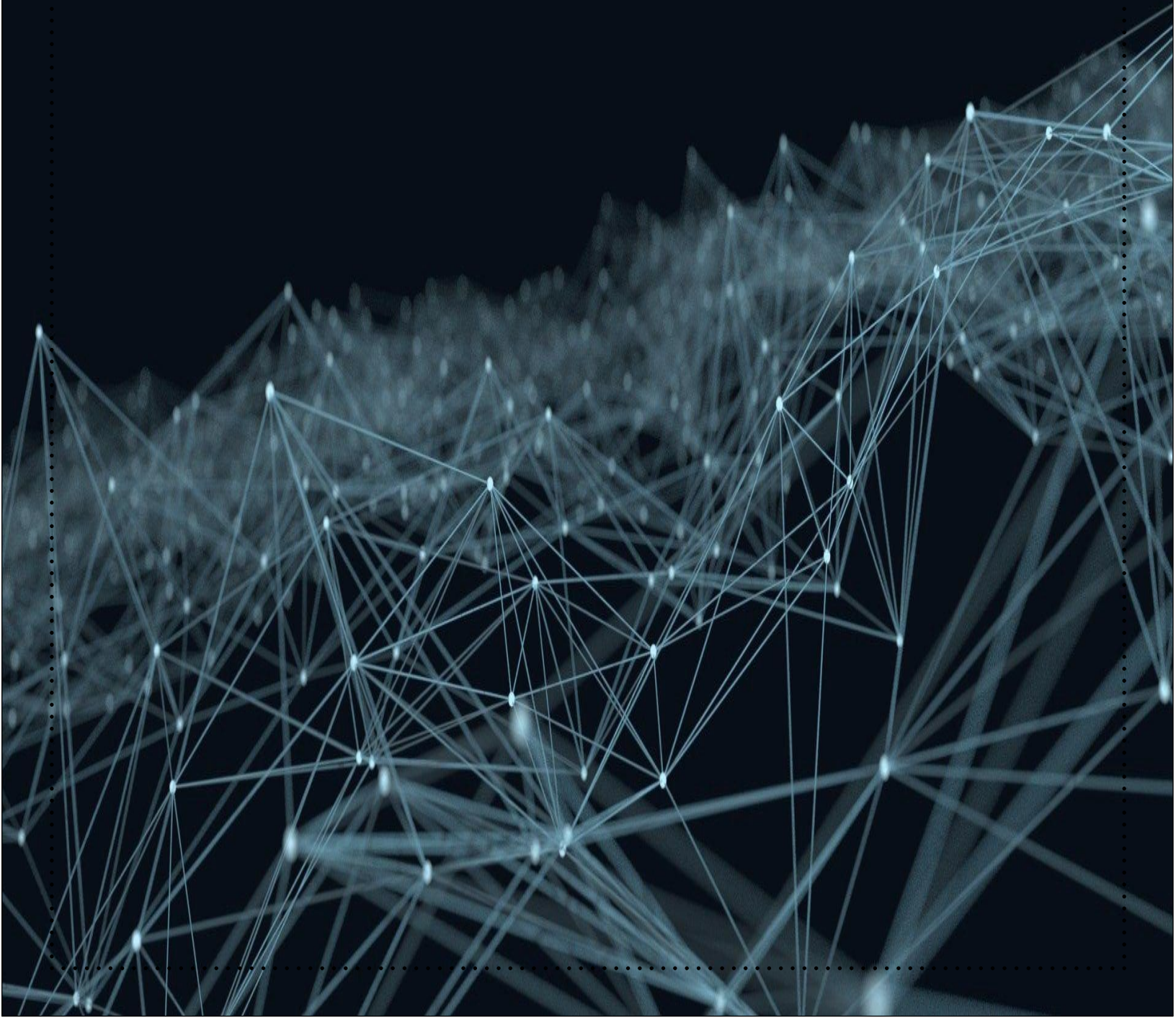
#### Group members:

Mohammad Reza Babaei Mosleh

Abtin Hassan Nezhad

Soheil Abdollahi

Morteza Salimi



In this project we implemented deep neural networks for audio classification problem using two different methods:

- 1- MLP (Muli Layer Perceptron) with MFCC features
- 2- CNN (Convolutional Neural Networks) with spectrum features

## DATASET

---

Our dataset is not one of the well-known open-source datasets and is gathered by our team.

For the first step we recorded about 800 audios with different formats and converted them to .wav format. Next, we doubled our dataset using data augmentation methods with adding noise to our original data. After creating our base dataset, we did some verifications

```
def manipulate(data, noise_factor):  
    pink_noise = cn.powerlaw_psd_gaussian(1, data.shape[0])  
    augmented_data = data + noise_factor * pink_noise  
    # Cast back to same data type  
    augmented_data = augmented_data.astype(type(data[0]))  
    return augmented_data
```

(Function to add noise)

## PRE-PROCESSING

---

At the second step we did some pre-processing

A – cutting all audios to 3 seconds:

our audios are more than 3 second length, we clipped all of our data to 3 second as our entries should be in fixed size

B – down sampling

We down sampled our fixed size data to pick 16000 sample per second and 48000 sample over all

C – train, test, validation split:

we split our data into 70% for training, 20% for test and 10% for validation

D – creating .CSV file

then we created 2 .csv file for our data set for later uses in data loading. The first one is raw .CSV and the second is in One-Hot encoded format

## PREPARING DATA LOADER FOR MLP

---

In this step we prepared a data loader for MLP model to load our data and labels with respect to our pre made .CSV files. to process data and we extracted the MFCC features and added them to our data loader to feed to model and visualizing our features.

```
data_dir = "/mnt/g/onlinelessons/Multimedia/HW/project/dataset/final_data/"

class audio_custom_dataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, transform=None):
        """
        Arguments:
            csv_file (string): Path to the csv file with labels.
            root_dir (string): Directory with all the audios.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """

        self.audio_csv = pd.read_csv(csv_file)
        self.transform = transform

    def __len__(self):
        return len(self.audio_csv)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        audio_name = self.audio_csv.iloc[idx, 1]
        waveform, sample_rate = torchaudio.load(audio_name)
        transform = T.MFCC(
            sample_rate=sample_rate,
            n_mfcc=31,
            melkwargs={"n_fft": 2048, "hop_length": 512, "n_mels": 128, "center": False}
        )
        waveform = transform(waveform)

        waveform = nn.functional.normalize(waveform, dim=-1)

        label = self.audio_csv.iloc[idx, 3:7]
        label = np.array(label, dtype=np.float16)

        # sample = {'audio': waveform, 'Label': label}
        sample = [torch.flatten(waveform), label]

        if self.transform:
            sample = self.transform(sample)

        return sample
```

(Code for data loader)

Then we loaded our data using 8 batches for more training speed (our training works with GPU)

## MODEL ARCHITECTURE FOR MLP

---

then we wrote our training function and specified our model Architecture in this project we trained 4 different models with different accuracy for later use (models are included in this repository). for example, we show one of the models:

in our training we used L2 regularization for preventing from overfitting also we used early stopping with patience value of 15 epoch for this purpose.

In this example model we used an architecture like below

Dense layer (2790 \* 512) followed by batch normalization, dropout (0.5) and ReLU activation function

Dense layer (512 \* 128) followed by batch normalization, dropout (0.5) and ReLU activation function

Dense layer (128 \* 64) followed by batch normalization, dropout (0.5) and ReLU activation function

Dense layer (64 \* 16) followed by batch normalization, dropout (0.5) and ReLU activation function

Dense layer (16 \* 4) followed by SoftMax of 4 neurons

## RESULTS FOR MLP

---

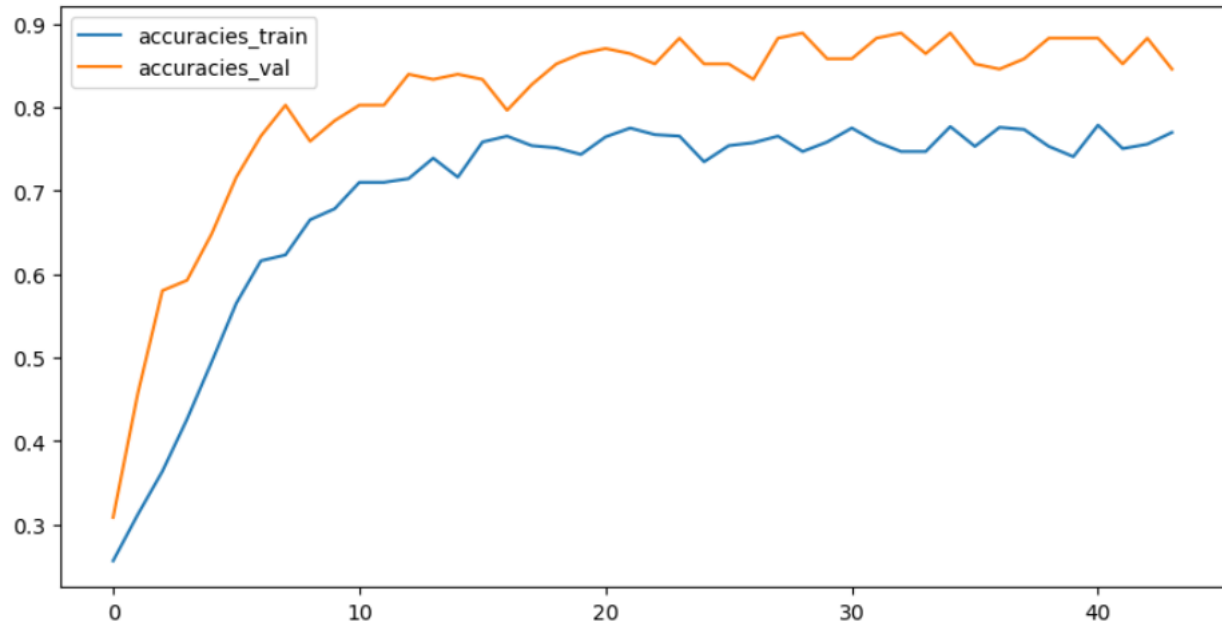
In our last epoch the training accuracy followed by validation is shown below:

```
Epoch 45/300
phase: train
train Loss: 1.1125 Acc: 0.7698
phase: val
val Loss: 1.0502 Acc: 0.8457
15
Early stopping after 44 epochs

Training complete in 6m 39s
Best val Acc: 0.888889
```

It is worth mentioning that our training stopped by early stopping after 45 epochs

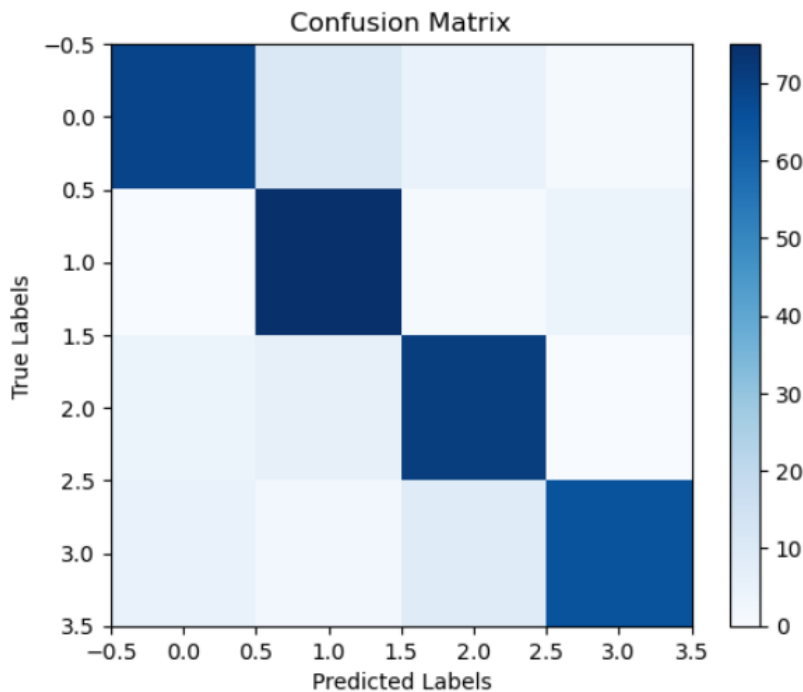
The graph of training process is like below:



After making sure that the model is not overfitted we test it on test data:

```
corrects:: 280  
total: 328  
accuracy: 85.36585235595703
```

Confusion matrix:





## PREPARING DATA LOADER FOR CNN

---

Like the previous part we prepared a data loader for process our data and add it to our data loader with proper format and by proper format we mean spectrum of audios:

```
data_dir = "/mnt/g/onlinelessons/Multimedia/HW/project/dataset/final_data/"

class audio_custom_dataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, transform=None):
        """
        Arguments:
            csv_file (string): Path to the csv file with labels.
            root_dir (string): Directory with all the audios.
            transform (callable, optional): Optional transform to be applied
                on a sample.
        """

        self.audio_csv = pd.read_csv(csv_file)
        self.transform = transform

    def __len__(self):
        return len(self.audio_csv)

    def __getitem__(self, idx):
        if torch.is_tensor(idx):
            idx = idx.tolist()

        audio_name = self.audio_csv.iloc[idx, 1]
        waveform, sample_rate = torchaudio.load(audio_name)

        spectrogram = librosa.feature.melspectrogram(y= waveform.numpy(), sr= sample_rate, n_mels= 128)
        log_spectrogram = librosa.power_to_db(spectrogram, ref= np.max)

        label = self.audio_csv.iloc[idx, 3:7]
        label = np.array(label, dtype=np.float16)

        # sample = {'audio': waveform, 'Label': Label}
        sample = [torch.tensor(log_spectrogram, dtype=torch.float32), label]

        if self.transform:
            sample = self.transform(sample)

        return sample
```

This process will return us features of size (128, 94).

## MODEL ARCHITECTURE FOR CNN

---

then we wrote our training function and specified our model Architecture in this project (model are included in this repository):

in our training we used L2 regularization for preventing from overfitting also we used early stopping with patience value of 15 epoch for this purpose.

Our layers are listed below:

Conv2d(in\_channels=1, out\_channels=32, kernel\_size=5, padding="same")

MaxPool2d(kernel\_size=2)

BatchNorm2d(num\_features=32)

ReLU()

Dropout(p=0.1)

Conv2d(in\_channels=32, out\_channels=32, kernel\_size=3, padding="same")

MaxPool2d(kernel\_size=2)

BatchNorm2d(num\_features=32)

ReLU()

Dropout(p=0.1)

Conv2d(in\_channels=32, out\_channels=64, kernel\_size=3, padding="same")

MaxPool2d(kernel\_size=2)

BatchNorm2d(num\_features=64)

ReLU()

Dropout(p=0.1)

Conv2d(in\_channels=64, out\_channels=64, kernel\_size=3, padding="same")

MaxPool2d(kernel\_size=2)

BatchNorm2d(num\_features=64)

ReLU()

Dropout(p=0.3)

Flatten()

Dense (in\_features=fc\_dim\_in, out\_features=128)

BatchNorm2d(num\_features=128)

ReLU()

Dropout(p=0.5)

Dense (in\_features=128, out\_features=4

Softmax(dim = 1)

## RESULTS

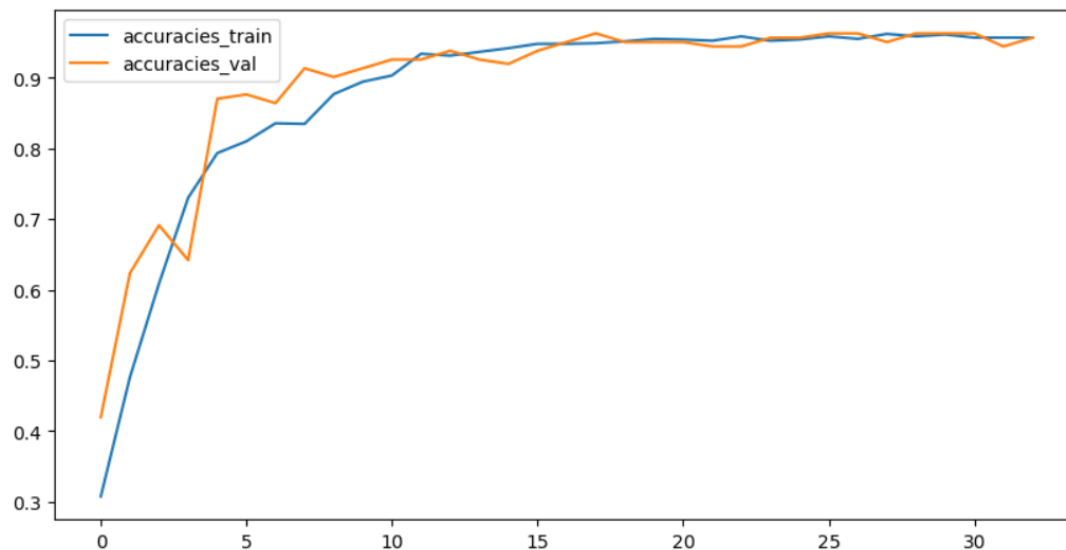
---

Our training accuracy after 33 epochs is:

```
Epoch 33/100  
phase: train  
train Loss: 0.8553 Acc: 0.9569  
phase: val  
val Loss: 0.8517 Acc: 0.9568  
15  
Early stopping after 32 epochs  
  
Training complete in 5m 51s  
Best val Acc: 0.962963
```

It is worth mentioning that we were planned for 300 epochs of training but we had early stopped after 15 epochs of being unimproved in validation accuracy on epoch 34

The graph of training process is:

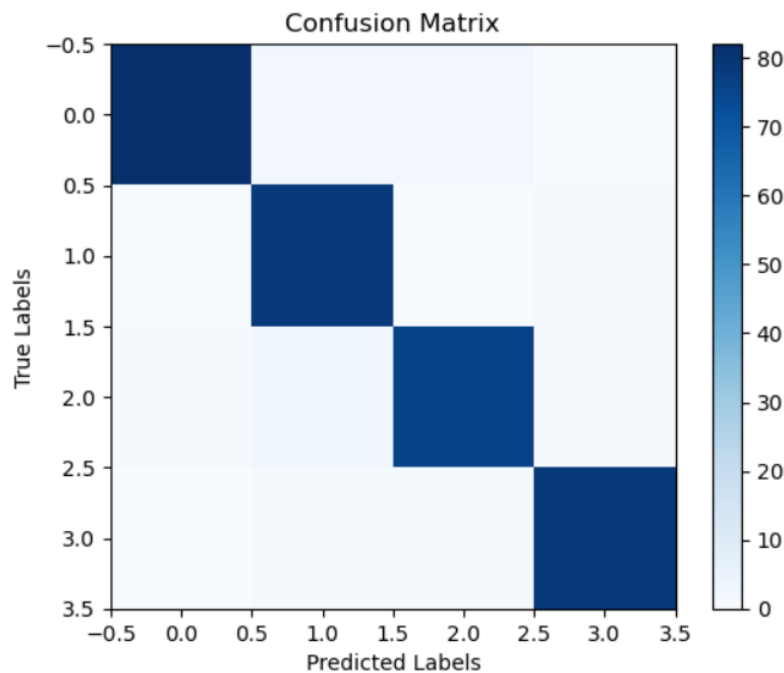




After ensuring that model is not overfitted we tested our data of test dataset and here are the results:

```
corrects:: 316
total: 328
acuuracy: 96.34146118164062
```

Confusion matrix is:



## REAL-TIME PROCESSING OF MLP & CNN

Because our MLP models was not accurate enough to do the classification task individually we used **Ensemble Learning** method by combining models' outputs considering different weights for every model here is how we combined our models:

```
output_1 = test_model_1(tensor)
selected_1 = torch.argmax(output_1.to("cpu"), dim=1)
output_2 = test_model_2(tensor)
selected_2 = torch.argmax(output_2.to("cpu"), dim=1)
output_3 = test_model_3(tensor)
selected_3 = torch.argmax(output_3.to("cpu"), dim=1)

output = (output_1*3 + output_2*6 + output_3)/10

preds = output

print(output_1, "\n", output_2, "\n", output_3, "\n", preds)

predicted_value = torch.argmax(preds.to("cpu"), dim=1)

if(selected_1[0] == 3 or selected_2 == 3):
    predicted_value = torch.tensor([3], dtype=torch.float16)

if output_1[selected_1[0]] > 0.9:
    predicted_value = selected_1
elif output_2[selected_2[0]] > 0.9:
    predicted_value = selected_2
elif output_2[selected_3[0]] > 0.9:
    predicted_value = selected_3
```

(Code for ensemble learning with MLP)

```

tensor = librosa.feature.melspectrogram(y=tensor.numpy(), sr= SAMPLE_RATE, n_mels= 128)
tensor = librosa.power_to_db(tensor, ref= np.max)
tensor = torch.tensor(tensor, dtype=torch.float32)

    tensor = nn.functional.normalize(tensor, dim=-1)

print(tensor.size())

tensor = torch.reshape(tensor, (1, 1, 128, 94))

```

(Code for processing of CNN input audio)

Then we used our system microphone to get audio in 1 second periods. Then we cropped audio to 1 second and padded it to 48000 samples (this reduces accuracy but also reduce the delay between 2 different commands).

```

stream.start_stream()

# Record 3 seconds of audio
recorded_frames = []
for i in range(int(input_sample_rate / chunk_size * output_duration) + 1):
    data = stream.read(chunk_size)
    recorded_frames.append(data)

# Stop the stream
time_elapsed = time.time() - since
print(f"duration: {time_elapsed}")
stream.stop_stream()
stream.close()

```

It is notable that our processor in this method has better accuracy but still not good enough for real-time processing in game so we used CNN model for our game.

# GRADIO

The Gradio library is used to run code with the help of a user interface. In calling this function, we choose the input type (text, audio, image, etc.) and the output type. We set the input as text and the output as a text box. When the user enters "start" as input, the microphone is activated and ready to receive audio. After receiving and processing the audio, it displays the output label in the text box.

```
def transcribe_audio(text):
    if text == "start":
        input_format = pyaudio.paInt16
        input_channels = 1
        input_sample_rate = 16000 # Changed to 16000 samples per second
        chunk_size = 2048
        output_format = 'WAV'
        output_channels = 1
        output_sample_rate = 16000
        output_duration = 3 # seconds
        num_output_frames = output_sample_rate * output_duration

        # Create a PyAudio object
        audio = pyaudio.PyAudio()

        print('Recording...')

        # Open the microphone stream
        stream = audio.open(format=input_format, channels=input_channels,
                             rate=input_sample_rate, input=True, frames_per_buffer=chunk_size)

        # Start the stream
        since = time.time()
        stream.start_stream()

        # Record 3 seconds of audio
        recorded_frames = []
        for i in range(int(input_sample_rate / chunk_size * output_duration) - 15):
            data = stream.read(chunk_size)
            recorded_frames.append(data)

        # Stop the stream
        time_elapsed = time.time() - since
        print(f'duration: {time_elapsed}')
        stream.stop_stream()
        stream.close()

        print('Processing...')

        # Convert the recorded audio to a NumPy array
        recorded_data = np.frombuffer(b''.join(recorded_frames), dtype=np.int16)
        recorded_data = recorded_data[:16000]
        sd.play(recorded_data, 16000)
        recorded_data = recorded_data.astype(np.float16)
        print(recorded_data.shape)

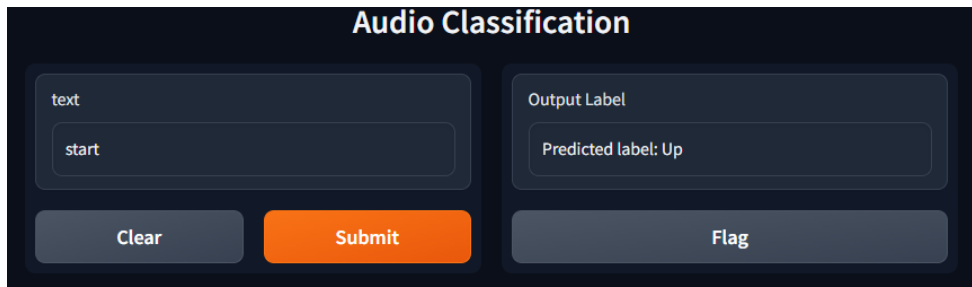
        # Reshape the audio to (1, 48000) size
        audio_tensor = torch.from_numpy(recorded_data).reshape(1, -1)
        print(audio_tensor.size())

        # Process the audio with the pre-trained model
        with torch.no_grad():
            output = process_cnn(audio_tensor)

        return f'Predicted label: {output}'
    else:
        return "invalid input"

# define the inputs and outputs for the Gradio interface
audio_output = gr.outputs.Textbox(label="Output Label")

# create the Gradio interface
gr.Interface(fn=transcribe_audio, inputs="text", outputs=audio_output, title="Audio Classification").launch()
```



# GAME

---

## Game:

This game is programmed using the Pygame library, and constants such as dimensions and scores have been defined. Additionally, we define classes and methods for the airplane, object(bomb), bullet, and explosion. In the airplane class, the init methods are for constructing the airplane, the update method is for its position based on speed, and the shoot method is for firing bullet s. The bomb object class also has an init method for constructing it and an update method for changing its position and explosion. The explosion and bullet classes are also similar to the airplane class.

The spawn function adds bombs to the screen. The endgame function displays a message and results when the game ends. After creating the above, the game starts. As previously mentioned, sound is received and processed in real-time according to the given label, and the speed of the airplane is adjusted. The update method of the airplane is called each time, and the airplane goes to its new position.

Furthermore, we use the explosion class for the collision of a bullet with a bomb. Every 5 seconds, a bomb is added to the screen.

The number of lives and the sharpness of the score are displayed in the corner of the screen. When the number of lives reaches zero, we exit the game and call the endgame function 07:22 PM

Above explanations could be seen in below code:

```
class Airplane(pygame.sprite.Sprite):
    def __init__(self):
        super().__init__()
        self.image = airplane_image
        self.rect = self.image.get_rect()
        self.rect.centerx = SCREEN_WIDTH // 2
        self.rect.bottom = SCREEN_HEIGHT - 10
        self.speed_x = 0
        self.speed_y = 0

    def update(self):
        # Move the airplane based on user input
        self.rect.x += self.speed_x
        self.rect.y += self.speed_y

        # Keep the airplane inside the screen
        if self.rect.left < 0:
            self.rect.left = 0
        elif self.rect.right > SCREEN_WIDTH:
            self.rect.right = SCREEN_WIDTH
        if self.rect.top < 0:
            self.rect.top = 0
        elif self.rect.bottom > SCREEN_HEIGHT:
            self.rect.bottom = SCREEN_HEIGHT

    def shoot(self):
        # Create a bullet sprite and add it to the appropriate groups
        bullet = Bullet(self.rect.centerx, self.rect.top)
        all_sprites.add(bullet)
        bullets.add(bullet)

        if output == "Left":
            airplane.speed_x = -50
            airplane.speed_y = 0
        elif output == "Right":
            airplane.speed_x = 50
            airplane.speed_y = 0
        elif output == "Up":
            airplane.speed_y = -50
            airplane.speed_x = 0
        elif output == "Down":
            airplane.speed_x = 0
            airplane.speed_y = 50
```

THE END