A red logo on a white background

Description automatically generated**NORTHEASTERN UNIVERSITY INFORMATION SYSTEMS - Fall 2023**

***Conversation Emotion Classifier using Deep Learning Methods and Parallel Computing with PyTorch***

**Parallel Machine Learning & AI**

CSYE7105

**Guided By:**

**Prof. Handan Liu**

**Team-17**

**Sukruth Mothakapally and Nisarga Venkatesh**

# **INTRODUCTION**

Our project is an innovative venture in conversation emotion classification. It uses a CNN model and Mel spectrograms to classify emotions in audio conversations.

To handle the computational demands of deep learning, we employed parallel computing techniques such as various pool methods like map, starmap, and apply, multiprocessing method and even Pytorch methods like data loader, Data Parallel and Distributed Data Parallel. This allows us to efficiently manage large datasets and complex models, significantly speeding up the training process.

Below are the parallel methods we would be utilizing in our project :

• The Pool methods like map, starmap, apply etc., are part of the multiprocessing module in Python. These methods allow you to run a function in parallel across multiple inputs, leveraging multiple processors on a machine.

• The multiprocessing method is a module in Python that supports the creation of separate processes. This allows for parallel execution of code, which can lead to significant speedups for computationally intensive tasks.

• Data Parallel and Distributed Data Parallel methods are provided by PyTorch. They implement data parallelism at the module level and can run across multiple machines2. These methods are used to synchronize gradients and buffers, and are particularly useful when your model is too large to fit on a single GPU

# **BACKGROUND**

In the rapidly evolving field of artificial intelligence, one area that has garnered significant attention is emotion classification in textual conversations. This task involves discerning the emotional context of each utterance in a conversation, a challenge that is compounded by the reliance on textual context. This project employs a Convolutional Neural Network (CNN) model, Mel spectrograms calculated using PyTorch, and parallel computing methods, specifically Data Parallel and Distributed Data Parallel methods.

Deep learning, a subset of machine learning, has shown immense promise in emotion classification tasks. CNNs, in particular, have been extensively used for emotion classification. In our project, we utilize a CNN model to classify emotions. The advantage of CNNs lies in their ability to automatically and adaptively learn spatial hierarchies of features, making them well-suited for emotion classification tasks.

To further enhance the performance of the CNN model, we calculate Mel spectrograms using PyTorch. Mel spectrograms provide a visual representation of the spectrum of frequencies of sound as they vary with time. They are particularly useful in emotion classification as they capture the tonal characteristics of an utterance.

The computational intensity and data requirements of deep learning pose significant challenges to conventional computing platforms. This is where parallel computing comes into play. Parallel computing involves dividing a problem into subproblems that can be solved simultaneously. This approach is highly beneficial in deep learning, where large datasets and complex models are the norm.

In our project, we utilize two types of parallel computing methods provided by PyTorch: Data Parallel and Distributed Data Parallel methods. Data Parallelism involves distributing the data across multiple GPUs, where each GPU operates on a different partition of the data. On the other hand, Distributed Data Parallelism implements data parallelism at the module level, allowing it to run across multiple machines.

Parallelism is crucial in our project for several reasons. Firstly, it allows for the efficient handling of large datasets and complex models, significantly accelerating the training process. Secondly, it enables the model to fit into memory. If a model is too large to fit on a single GPU, it can be split across multiple GPUs using model parallelism. Lastly, parallelism allows for scalability. As the size of the dataset or the complexity of the model increases, additional resources can be added to maintain performance.

In conclusion, the background is rooted in leveraging the power of deep learning, Mel spectrograms, and parallel computing to effectively classify emotions in textual conversations. The use of parallel computing, in particular, plays a pivotal role in managing the computational intensity of deep learning and ensuring the scalability.

# **MOTIVATION**

the scale of data and the complexity of models have grown exponentially. This growth has led to an increase in the computational requirements of machine learning tasks, particularly those involving deep learning models such as Convolutional Neural Networks (CNNs). The motivation behind our project, “Conversation Emotion Classifier using Deep Learning Methods and Parallel Computing with PyTorch”, lies in addressing these computational challenges through the use of parallel computing methods.

1. **The Need for Speed**

One of the primary motivations for this project is the need for speed. Training deep learning models is a time-consuming process, especially when dealing with large datasets. This is where parallel computing comes into play. By distributing the data across multiple GPUs or even across multiple machines, we can significantly accelerate the training process. This is particularly important in a real-world setting where time is often a critical factor.

1. **Overcoming Memory Limitations**

Another motivation for this project is to overcome the memory limitations of individual GPUs. Deep learning models, particularly CNNs, can be quite large and may not fit into the memory of a single GPU. By utilizing parallel computing methods, we can distribute the model across multiple GPUs, effectively increasing the available memory and enabling us to train larger, more complex models.

1. **Scalability**

Scalability is a key motivation for this project. As the size of the dataset or the complexity of the model increases, we can simply add more resources to maintain performance. This is particularly important in the field of emotion classification, where the amount of available data is growing rapidly. By utilizing parallel computing methods, we ensure that our project can scale to meet these increasing demands.

1. **Utilizing PyTorch’s Capabilities**

Our motivation also lies in fully utilizing the capabilities of PyTorch, a popular deep learning framework. PyTorch provides several methods for parallel computing, including Data Parallel and Distributed Data Parallel (DDP) methods. By leveraging these methods, we can effectively distribute the computational load across multiple GPUs or machines, accelerating the training process and enabling the handling of larger models.

By utilizing parallel computing methods, we aim to accelerate the training process, overcome memory limitations, ensure scalability, and fully utilize the capabilities of PyTorch. This focus on parallelism sets our project apart and positions it at the forefront of innovation in the field of emotion classification.

# **GOALS**

The primary objective of our project is to harness the power of parallel computing to address the computational challenges associated with training deep learning models. Our goals are centered around three key aspects: speedup, improved efficiency, and reduced execution time.

1. **Speedup**

Our first goal is to achieve a significant speedup in the training process of our Convolutional Neural Network (CNN) model. Training deep learning models is a computationally intensive task that can take a considerable amount of time, especially when dealing with large datasets. By utilizing parallel computing methods, we aim to distribute the computational load across multiple GPUs or even across multiple machines, thereby significantly accelerating the training process.

1. **Improved Efficiency**

Our second goal is to improve the efficiency of our model. Efficiency in this context refers to the effective utilization of computational resources. By employing parallel computing methods, we aim to ensure that each GPU is fully utilized and that the computational load is evenly distributed. This not only improves the efficiency of our model but also ensures that we are making the most of our available resources.

1. **Reduced Execution Time**

Our third goal is to reduce the execution time of our model. Execution time is a critical factor in the real-world deployment of machine learning models. By leveraging parallel computing methods, we aim to minimize the time it takes for our model to process an input and produce an output. This is particularly important in the field of emotion classification, where timely results can be crucial.

1. **Utilizing PyTorch’s Parallel Computing Methods**

To achieve these goals, we plan to utilize the parallel computing methods provided by PyTorch, namely Data Parallel and Distributed Data Parallel (DDP) methods. Data Parallelism involves distributing the data across multiple GPUs, where each GPU operates on a different partition of the data. On the other hand, Distributed Data Parallelism implements data parallelism at the module level, allowing it to run across multiple machines. By leveraging these methods, we aim to effectively distribute the computational load, thereby achieving our goals of speedup, improved efficiency, and reduced execution time.

By focusing on speedup, improved efficiency, and reduced execution time, we aim to push the boundaries of what is possible in the field of emotion classification

# **DATASET DESCRIPTION**

**Description of Dataset**

This dataset is a comprehensive collection of audio files from four different sources: RAVDESS, CREMA-D, SAVEE, and TESS. The recordings are sorted into seven categories based on the emotion expressed: Angry, Happy, Sad, Neutral, Fearful, Disgusted, and Surprised.

**Here’s a brief breakdown:**

• Angry: Contains 2167 records, making up 16.7% of the dataset.

• Happy: Contains 2167 records, making up 16.46% of the dataset.

• Sad: Contains 2167 records, making up 16.35% of the dataset.

• Neutral: Contains 1795 records, making up 14.26% of the dataset.

• Fearful: Contains 2047 records, making up 16.46% of the dataset.

• Disgusted: Contains 1863 records, making up 15.03% of the dataset.

• Surprised: Contains 592 records, making up 4.74% of the dataset.

A graph of different emotions

Description automatically generated with medium confidence

**In terms of the source of these files:**

• CREMA-D (Crowd-sourced Emotional Multimodal Actors Dataset) contributes the most with 7442 files, which is about 58.15% of the total data.

• TESS (Transiting Exoplanet Survey Satellite) provides 2800 files, approximately 21.88% of the total data.

• RAVDESS (Ryerson Audio-Visual Database of Emotional Speech and Song) adds another 2076 files, around 16.22% of the total data.

• SAVEE (Surrey Audio-Visual Expressed Emotion) contributes the least with 480 files, making up about 3.75% of the total data.

A pie chart with numbers and a percentage of files with Crust in the background

Description automatically generated

**Data Source**

<https://www.kaggle.com/datasets/uldisvalainis/audio-emotions>

# **METHODOLOGY**

**Step 1: Exploratory Data Analysis**

Exploratory Data Analysis (EDA) is a critical first step in our methodology. It allows us to understand the structure, patterns, and characteristics of our data. By analyzing the distribution of emotions, types of audio files, and average duration of the audio files, we gain insights into the balance of our classes, the variety of our data, and the length of our audio samples.

Visualizations such as waveform, power spectrums, spectrograms, and MFCC visualizations provide us with a deeper understanding of the audio data. They reveal patterns and features that are not immediately apparent in the raw data. For instance, pitch analysis can uncover tonal characteristics associated with each emotion, which can be crucial for emotion classification.

Below are few of the snapshots of EDA tasks performed :

1. Waveform Visualization of Sad emotion for TESS dataset -

A blue graph with white text

Description automatically generated

1. Spectrogram visualization of happy emotion for RAVDESS dataset –

A close-up of a purple and pink spectrogram

Description automatically generated

1. Pitch Analysis of Neutral emotion –

A graph with red lines

Description automatically generated

Introducing **Parallelism** in EDA task –

We noticed that the calculation of average audio duration task was taking a lot of time as it was going through the entire dataset, hence we introduced parallelism here

We utilized the Pool methods like map and starmap to expedite the duration calculation task.

The Pool.map function is a built-in method that applies a function to every item of an iterable, such as a list or array. When used in conjunction with multiprocessing.Pool, it allows us to perform these operations in parallel, utilizing multiple CPUs to process the data concurrently. This can lead to a substantial reduction in the overall computation time, making our exploratory data analysis more efficient.

The Pool.starmap function is a built-in method that applies a function to every item of an iterable, similar to Pool.map. However, unlike Pool.map, Pool.starmap is designed to handle functions that take multiple arguments. It unpacks each item in the iterable as arguments to the function. When used in conjunction with multiprocessing.Pool, Pool.starmap allows us to perform these operations in parallel, utilizing multiple CPUs to process the data concurrently. This can lead to a substantial reduction in the overall computation time, making our exploratory data analysis more efficient.

## We also tried **Pool.apply** for comparision

* The Pool.apply function is a built-in method that applies a function to arguments. Unlike Pool.map and Pool.starmap, Pool.apply does not divide the iterable into chunks for parallel processing. Instead, it blocks until the result is ready. This means that Pool.apply runs a single function call in a separate process, waiting for the result before moving on to the next. This can be useful when the function calls are dependent on the results of previous calls.

**Comparing Execution times, Speedup, and Efficiency**

A graph of execution time comparison

Description automatically generated

A graph with a green line

Description automatically generated

A graph with a line

Description automatically generated

**Observation** –

* Both the map and starmap methods of parallel processing significantly reduced the execution time from around 250 seconds (for the non-parallel version) to around 30 seconds. This demonstrates the power of parallel processing when dealing with large datasets.
* The map method had an execution time of 30 seconds, while starmap was slightly higher with 32 seconds. The difference between them is negligible, indicating that both methods are equally efficient for this task.
* However, when we tried to use apply for parallel processing, it was taking longer than the non-parallel version. Given that it was taking more time, we decided to stop the execution midway as it was clear that apply was not an efficient choice for this particular task.
* Efficiency, which is the speedup per processor, was 0.3 for both map and starmap methods, indicating suboptimal use of processors. Despite minor differences in performance, the choice between map and starmap should be guided by the specific requirements of the function.
* In conclusion, both **map and starmap** are excellent choices for this task due to their ability to handle multiple tasks concurrently, leading to a substantial reduction in execution time. The choice between map and starmap would depend on the specific requirements of the function and how we prefer to pass arguments to it. In scenarios where the function is designed to take a single tuple as an argument, map would be a suitable choice. On the other hand, if the function is designed to take multiple arguments separately, starmap would be more appropriate.

**Step 2: Data Preprocessing**

Data Preprocessing is the next crucial step in our methodology. It involves preparing the data for the training process.

**Data Augmentation**

One of the key tasks in this step is to check and remove bias in the dataset. Bias in the dataset can lead to a model that is biased towards certain classes, which can negatively impact the model’s performance on unseen data.

In our case, we noticed that the ‘surprised’ emotion had less data compared to other emotions. To address this, we augmented the data for the ‘surprised’ emotion. Data augmentation is a technique that can help to balance the classes in our dataset by artificially increasing the number of samples in under-represented classes. This ensures that our model is trained on a balanced dataset, leading to a more robust and fair model.

Emotion Distribution before and after removing bias –

A graph of different emotions

Description automatically generated with medium confidence A graph of blue bars

Description automatically generated with medium confidence

The execution of this task took a lot of time and hence we decided to **parallelize** it

We utilized **concurrent.futures and ThreadPoolExecutor** which are Python modules used to augment audio data in an imbalanced dataset, by leveraging multiple threads to improve efficiency and hence we saw significant improvement in execution times.

**Calculating Mel Spectrogram**

Another important task in this step is the calculation of Mel spectrograms. Mel spectrograms provide a visual representation of the spectrum of frequencies of sound as they vary with time. They are particularly useful in emotion classification tasks as they capture the tonal characteristics of an utterance, which often carry emotional content

We considered mel spectrogram instead of mfcc - because

• While both Mel Frequency Cepstral Coefficients (MFCCs) and Mel spectrograms are commonly used features in audio and speech processing, they each have their strengths and are suited to different tasks.

• MFCCs are a compact representation of the power spectrum of an audio signal, specifically designed to mimic the human auditory system. They are excellent for tasks where a compact and highly discriminative feature representation is required, such as speaker identification or speech recognition.

• On the other hand, Mel spectrograms provide a more detailed representation of the power spectrum over time. They capture more fine-grained spectral details compared to MFCCs. This makes them particularly suited to tasks where these details are important, such as emotion recognition from audio data.

And after calculation, we got a good execution time even without parallelism here, but we tried to experiment what would be the time if **parallelism** was introduced here.

Again we used **pool.map**, as it proved beneficial in the EDA stage to calculate average duration of the audio files.

Execution time without Parallel –

A screenshot of a computer

Description automatically generated

Execution time with Parallel –

A screenshot of a computer

Description automatically generated

**Comparing Execution times, Speedup and Efficiency –**

A blue rectangular bars with white text

Description automatically generated A graph with a green line

Description automatically generated A graph with a line

Description automatically generated

**Observation -**

Unfortunately, the parallel version of the code did not reduce the execution time compared to the non-parallel version. In fact, the total execution time for the parallel version was slightly higher. This could be due to the overhead associated with creating and managing multiple processes, which can sometimes outweigh the benefits of parallelism, especially for tasks that are not highly computationally intensive or when the data transfer between processes is high.

Given these results, we planned to experiment with the **apply\_async** method from Python’s multiprocessing module. The apply\_async function allows for asynchronous processing, which means it doesn’t block the execution of subsequent tasks while waiting for the current task to complete. This could potentially lead to a better utilization of CPU resources and improve the overall execution time.

**Results –**

In our case, parallelization with pool.map and pool.apply\_async took longer than non-parallel processing. This is likely due to the overhead of setting up parallel processes and inter-process communication. The operations performed weren’t computationally intensive enough to benefit from parallelization. Hence, **serialization** was more efficient for calculating the Mel spectrogram of the audio files in this specific scenario

Given these results, we conclude that for our specific task of calculating the Mel spectrogram of audio files, a **serialized** approach is more efficient and hence **parallelization is a failure here**

**3. RESULT AND ANALYSIS**

**Step 3: Model Development**

**Model 1 – Custom CNN model without parallelism**

**Model Architecture**

For our task of conversation emotion classification, we will be using a Convolutional Neural Network (CNN), a type of deep learning model, implemented in PyTorch The model architecture consists of a convolutional neural network (CNN) with one convolutional layer, followed by max-pooling, a fully connected layer, and an output layer. The CNN processes input audio spectrograms, and the output layer produces predictions for the emotion classes using the softmax activation function.

The preprocessed data for each emotion is combined and resized. These are then converted into tensors and split into training and testing sets. A Convolutional Neural Network (CNN) model is defined in PyTorch with one convolutional layer and two fully connected layers. The model is trained using the training set, with early stopping implemented to prevent overfitting. After each training epoch, the model is tested on the testing set and the accuracy is calculated. If the accuracy doesn't improve for a certain number of epochs, the training stops early.

A screenshot of a computer screen

Description automatically generated

A blue squares with white text

Description automatically generated

Result – with just 10 epochs and batch size of 32, we could achieve around 50% accuracy

**Model 2 - Resnet50 model**

we opted to use the ResNet-50 model, a variant of the ResNet architecture. ResNet-50 is a deep convolutional neural network known for its ability to effectively handle complex visual recognition tasks.

We chose ResNet-50 for its pre-trained weights on large image datasets, allowing us to leverage its feature extraction capabilities for our audio emotion classification task. The model's architecture includes residual blocks, enabling it to efficiently capture hierarchical features, making it suitable for our multi-class audio emotion classification problem.

We fine-tuned the last fully connected layer to adapt the model to our specific classification task, resulting in a powerful and effective solution for audio emotion recognition

A screenshot of a computer code

Description automatically generated

**Observation** –

* This model gave similar accuracy for intial epochs. But as the number of epochs increased, the accuracy didn't see any significant jump.
* Moreover, this model took a lot of time to train compared to our custom CNN model
* Since resnet50 model took a lot of time to train, we thought of introducing **parallelism** here.
* We utilized **Data parallel** method with 1 gpu and all the available cpus to parallelize the model training.

A screenshot of a computer

Description automatically generated

A blue rectangular object with red line

Description automatically generated

**Observation –**

* We can observe a significant improvement in training time
* Data parallel method helped in reducing the training time from 3800 seconds to around 200 seconds

**Model Evaluation**

We chose our **custom CNN model** as Restnet50 took too more time to train even with parallelization without any improvement in accuracy

Custom CNN model without parallel –

A screenshot of a computer

Description automatically generated

Resnet50 model with parallelization –

A screenshot of a computer

Description automatically generated

# **Finding the ideal number of CPUs by parallelizing our selected model with multiprocessing**

* Multiprocessing is a Python module that supports the execution of parallel processes, allowing for the concurrent execution of tasks. In our use case, we chose multiprocessing to parallelize the training and testing processes of our selected model with varying numbers of CPUs.
* This enables us to efficiently explore the impact of different CPU configurations on the overall performance and training time of the model, helping us identify the optimal number of CPUs for our specific task.

A graph of a graph

Description automatically generated with medium confidence

**Observation** –

* The training time slightly decreased with an increase in the number of CPUs, reaching the lowest with 4 CPUs. However, the reduction was not substantial.
* Also as we increased the number of CPUs, the efficiency decreased, indicating diminishing returns in resource utilization. Speedup was notably better with 4 CPUs, offering a good balance between improved performance and acceptable efficiency. While 8 CPUs provided some speedup, it didn't significantly enhance efficiency.
* Therefore, in our case, **4 CPUs** appear to be the ideal choice for a well-balanced trade-off between training time, speedup, and efficiency.
* We will utilize 4 CPUs in the next steps of parallelization

# **CNN Model Using PyTorch with Data Parallel Method Having 1 GPU -**

**Data Preparation**

Here, we prepare our data for training. We create PyTorch datasets and data loaders for our training and testing data. The data loaders are responsible for loading the data in batches and shuffling it for training. We vary the number of workers in the data loader (num\_workers) based on the number of CPUs available to us. This allows us to leverage multiple CPUs to load the data in parallel, thereby reducing the data loading time.

**Model Parallelism**

Once our data is ready, we create an instance of our model and check if multiple GPUs are available. If they are, we wrap our model using PyTorch’s DataParallel method. This method replicates our model across multiple GPUs, allowing each GPU to process a different portion of the data. This is the first part of our methodology - leveraging Data Parallel to distribute the computational load across multiple GPUs, thereby accelerating the training process.

**Training and Testing**

With our model ready and our data prepared, we proceed to train our model. For each epoch, we iterate over our training data, perform a forward pass to compute the output of our model, calculate the loss, perform a backward pass to compute gradients, and update our model’s parameters. After each epoch, we test our model on our testing data and calculate the accuracy of our model.

Throughout the training process, we keep track of the execution time for each number of CPUs. This allows us to evaluate the effectiveness of our parallel computing methods and understand the relationship between the number of CPUs and the execution time.

By leveraging these parallel computing methods, we aim to accelerate the training process, improve the efficiency of our model, and reduce the execution time.

We use PyTorch's Data Parallel method, which is a simple yet effective way to parallelize training. It involves splitting the input data across the available GPUs, computing the forward pass (and subsequently, the backward pass) on each GPU, and then aggregating the results on one device. This method allows for faster processing times and efficient utilization of multiple GPUs.

A screenshot of a computer code

Description automatically generated

**Observation**

* We can see that with Data Parallel method utilizing 1 GPU and 4 CPU, the training time significantly reduced
* Hence proving that parallelization was a success in training this model

## **CNN Model Using Pytorch With Distributed Data Parallel Method Having 1 GPU -**

In the second part of our methodology, we utilized Distributed Data Parallel (DDP). We initialize the distributed environment and create an instance of our model for each process. Each model is moved to a specific GPU and wrapped using PyTorch’s DDP method. This method allows each GPU to process a different portion of the data, further distributing the computational load and accelerating the training process.

DDP differs from the Data Parallel method in how it handles the distribution of data and the aggregation of gradients. In DDP, each process operates on a full model replica and a distinct subset of the data. The gradients are then synchronized across the processes during the backward pass. This results in all replicas having the same parameters.

A screenshot of a computer code

Description automatically generated

Observation – DDP has a higher training time compared to DP method

# **Ideal Parallel Method and Number of GPUs**

* The project has demonstrated the transformative impact of parallel computing on the efficiency of deep learning models.
* Our exploration of both Data Parallel and Distributed Data Parallel (DDP) methods has shown that these techniques can significantly reduce execution time, thereby accelerating the training process. This is a critical advantage in the field of machine learning, where training complex models on large datasets can be computationally intensive and time-consuming.

A graph of different models

Description automatically generated

A blue rectangular shapes with green line

Description automatically generated

**Observation** –

As observed in the results, **Data Parallel (DP) with one GPU** exhibited a notable speedup compared to the non-parallel model, leading to enhanced training efficiency. The speedup and efficiency metrics, as depicted in the accompanying graphs, underscore the substantial performance gains achieved through parallel computing strategies, validating our choice of DP for its efficacy in our specific configuration

# **CONCLUSION**

The project has demonstrated the transformative impact of parallel computing on the efficiency of deep learning models.

Our exploration of both Data Parallel and Distributed Data Parallel (DDP) methods has shown that these techniques can significantly reduce execution time, thereby accelerating the training process. This is a critical advantage in the field of machine learning, where training complex models on large datasets can be computationally intensive and time-consuming.

In our specific application, **Data Parallel with a single GPU and 4 CPUs** emerged as the most efficient configuration, striking a balance between computational speed and resource utilization. However, the choice of parallel computing method and configuration is highly dependent on the specific requirements of the task, including the size and nature of the dataset, the complexity of the model, and the computational resources available.

While our exploration of parallel computing in this project was focused on improving computational efficiency, it’s important to note that these methods do not inherently improve the accuracy of the model. The task of optimizing model performance involves additional considerations, including model architecture, feature selection, and hyperparameter tuning.

In conclusion, parallel computing, particularly methods like Data Parallel and DDP, holds great promise for enhancing the efficiency of deep learning models. As we continue to push the boundaries of what’s possible in machine learning, these techniques will undoubtedly play a pivotal role in enabling us to effectively handle increasingly complex tasks and larger datasets. However, the implementation of these methods requires careful consideration and optimization to ensure they are used effectively and judiciously. As such, the journey of mastering parallel computing in deep learning is an ongoing process, filled with both challenges and opportunities.

**GPU Configuration –**

A screenshot of a computer program

Description automatically generated

# **5. REFERENCES**

[1] Audio Emotions Dataset: <https://www.kaggle.com/datasets/uldisvalainis/audio-emotions>

[2] Exploratory Data Analysis: <https://towardsdatascience.com/speech-emotion-recognition-with-convolution-neural-network-1e6bb7130ce3>

[3] Calculating Mel Spectrogram: <https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0>

[4] Convulated Neural Network Model development: <https://github.com/jeffprosise/Deep-Learning/blob/master/Audio%20Classification%20(CNN).ipynb>

[5] Pytorch fundamentals: <https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html>

[6] Pytorch for audio classification: <https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html>

[7] Doubts and Clarifications: <https://chat.openai.com/>