Name: Bikash Dutta
Roll No: D22CS051
Topic: Speech Understanding

Programming Assignment 2

Question 1. Speaker Verification

Demo Link: https://huggingface.co/spaces/d22cs051/Speaker-Verification-A2-Docker

Task 1 + 2 + 3:

Choose **'ecapa_tdnn'**, **'hubert_large'**, "**wavlm_base_plus**", "**wavlm_large**" for Calculate the EER(%) on the VoxCeleb1-H.

Script to process each model one by one (make_csv.py):

- **Importing Required Libraries**: The code begins by importing necessary libraries. pandas is imported as pd for data manipulation, and tqdm is imported from the tqdm.auto module for displaying progress bars during iteration. The verification_batch function is imported from a custom module called verification. Additionally, the os module is imported for operating system-related tasks.
- **Reading Data**: The code reads a CSV file named 'list_test_hard.txt' located at '../../data/VoxCeleb/'. This file contains three columns: 'label', 'person1', and 'person2'. It reads the file into a pandas DataFrame, naming the columns accordingly.
- **Data Preprocessing**: Optionally, there's a commented-out line # df = df[:15] that selects only the first 15 rows of the DataFrame. This is useful for testing purposes or reducing computation time during development.
- **Verification Process**: The code then iterates over each model name specified in the list model_names. For each model, it processes the data in batches.
- **Batch Processing**: Inside the batch processing loop, it iterates over the DataFrame in batches of size batch_size. For each batch, it constructs lists of file paths (batch_files1 and batch_files2) corresponding to 'person1' and 'person2' columns, respectively.
- **File Path Handling**: It constructs file paths using the information from the DataFrame. If the file is not found in the specified location because there are some audios files that are present in the validation split and the corresponding sample is in the test split, it looks in an alternate location ('test' directory) and prints a message if it is still not found. It keeps track of the number of files not found.
- **Verification Score Calculation**: After constructing the batch file paths, it calls the verification_batch function, passing the model name and batch file paths as arguments. This function calculates verification scores for each pair of files in the batch.

- **Updating DataFrame**: Once verification scores are obtained, the code updates the DataFrame with these scores. It iterates over each score obtained and adds it to the corresponding row of the DataFrame.
- **Saving Results**: Finally, the code saves the updated DataFrame to a CSV file named 'verification_scores_vox_batch_{batch_size}.csv'. Included in the GitHub. It excludes the index from the CSV file for better readability.
- **File Not Found Handling**: If any files are not found during the process, it prints the total count of such occurrences.

Modified verification function for batch processing (verification.py):

NOTE: the results are processed for the first 50000 data points for each audio sample i.e. approx 4.7 sec. Due to computational limitations.

- **Function Definition**: The code defines a function named verification_batch that takes several parameters: model_name, batch_wav1, batch_wav2, use_gpu, and checkpoint. This function is designed to calculate verification scores for pairs of audio files using a specified model.
- **Model Validation**: The function begins with an assertion to ensure that the provided model_name is valid and exists within the MODEL_LIST variable.
- **Model Initialization**: It initializes the specified model using the init_model function, passing the model_name and an optional checkpoint.
- **Reading Audio Files**: The function reads the first audio files from batch_wav1 and batch_wav2 to determine their sample rates (sr1 and sr2) using the sf.read function.
- **Resampling**: It resamples the audio files to a common sample rate of 16000 Hz using the Resample class. This ensures consistency in audio processing.
- **Processing Audio Batches**: The function iterates through each audio file in the batches, reads them, and truncates them to the first 50000 samples. These truncated audio signals are then converted into PyTorch tensors and stored in batch_wav1 and batch_wav2.
- **GPU Usage**: If use_gpu is set to True, the function moves both the model and the audio tensors to the GPU.
- **Model Evaluation**: The model is put in evaluation mode (model.eval()), and forward passes are performed on the batches of audio tensors (batch_wav1 and batch_wav2). The resulting embeddings (emb1 and emb2) represent the learned features of the audio files.
- **Similarity Calculation**: The cosine similarity between the embeddings of the pairs of audio files (emb1 and emb2) is computed using PyTorch's F.cosine_similarity function. This similarity score represents the degree of similarity between the audio files.
- **Result Return**: The function returns the similarity scores as a numpy array after moving them back to the CPU (sim.cpu().numpy()) because they need to be stored in the data frame.

NOTE: The score returned in the range of (-1,1) i.e. not actual EER for the model.

Explanation of the function cal_eer(Q1.ipynb):

- **Importing Libraries**: The code begins by importing necessary libraries. It imports brentq and interp1d functions from scipy.optimize and scipy.interpolate modules, respectively. It also imports the roc_curve function from the sklearn.metrics module.
- **Input Data**: The function cal_eer takes two input arrays: y and y_score.
    - y contains the true binary labels (0 or 1) of the samples.
    - y_score contains the predicted scores or probabilities for the positive class.
- **ROC Curve Calculation**: The roc_curve function computes Receiver Operating Characteristic (ROC) curve metrics, namely False Positive Rate (fpr), True Positive Rate (tpr), and thresholds based on the true labels y and the predicted scores y_score.
- **Equal Error Rate (EER) Computation**: The EER is the point on the ROC curve where the False Acceptance Rate (FAR) equals the False Rejection Rate (FRR). The function uses the brentq function to find the threshold value at which FAR - (1 - TPR) is minimized (where TPR is True Positive Rate). This is equivalent to finding the point on the curve where the difference between FAR and (1 - TPR) is zero.
- **Interpolation**: Once the EER threshold is found, the function interpolates the corresponding threshold value from the threshold array using the interp1d function. This provides the threshold value that corresponds to the EER.
- **Return**: The function returns a tuple containing the EER value and the corresponding threshold.
- **Example Usage**: Finally, the function is called with sample y and y_score arrays, and the results are printed.

Results:

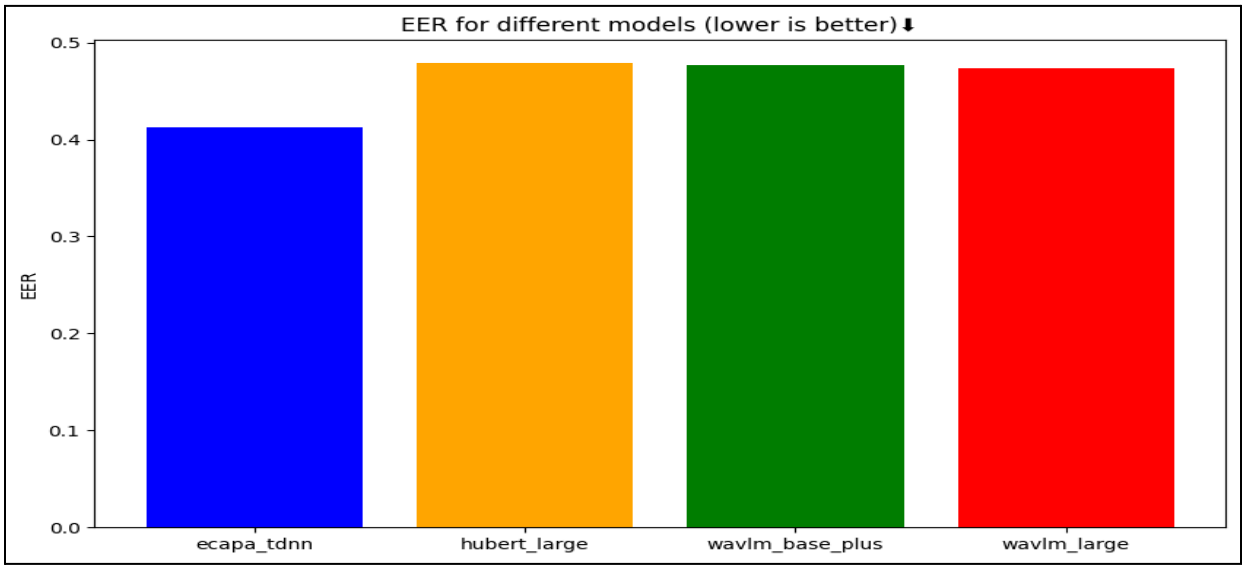

Fig: EER for models pretrained on VoxCelab Dataset

Model Name	Equal Error Rate
ecapa_tdnn	0.412722
hubert_large	0.478862
wavlm_base_plus	0.476551
wavlm_large	0.473567

Table: EER for pre-trained models on VoxCelab Dataset

TABLE II
SPEAKER VERIFICATION RESULTS ON VOXCELEB1. FOR THE LINES WITH * NOTATION, WE ADD THE LARGE MARGIN FINE-TUNING AND QUALITY-AWARE SCORE CALIBRATION [58] TO PUSH THE LIMIT OF THE PERFORMANCE.

Feature	EER (%)		
	Vox1-O	Vox1-E	Vox1-H
ECAPA-TDNN [19]	1.010	1.240	2.320
ECAPA-TDNN (Ours)	1.080	1.200	2.127
HuBERT Base	0.989	1.068	2.216
HuBERT Large	0.808	0.822	1.678
WavLM Base+	0.84	0.928	1.758
WavLM Large	0.617	0.662	1.318
HuBERT Large*	0.585	0.654	1.342
WavLM Large*	**0.383**	**0.480**	**0.986**

Table: Results from Wavlm paper

NOTE: The difference in the result is due to the audio processing length, the results are shown w.r.t to the first 50,000 data points for each audio which is equivalent to ~4.7 sec of audio because of computational limitations.

Task 4 + 5 + 6:

Dataset Class for the Kathbath Dataset(defined in Q1.ipynb):

NOTE: This is used to create csv files for speaker verification data, **i.e label speaker 1 and speaker 2.**

- **Importing Necessary Libraries**: The code imports required modules/classes from PyTorch (Dataset) and torchaudio.sox_effects. The latter is used to apply effects to audio files.
- **Effect Definitions**: The EFFECTS list contains a set of audio effects to be applied to the audio files during data loading. These effects include gain adjustment and silence removal.
- **Dataset Class Definition**: The code defines a custom dataset class named SpeakerVerifi_test, inheriting from PyTorch's Dataset class. This class is responsible for loading and processing speaker verification data.
- **Initialization Method**: The __init__ method initializes the dataset object. It takes parameters such as vad_config (voice activity detection configuration), file_path (root directory containing audio files), and meta_data (file containing metadata).
- **Data Processing Method (processing)**: This method reads metadata from the provided file (meta_data) and constructs a list of pairs (pair_table) where each pair consists of a label and the file paths of two audio files. The file paths are constructed by appending the root directory (file_path) to the filenames extracted from the metadata.
- **Data Length Method (__len__)**: This method returns the total number of pairs in the dataset (pair_table).
- **Data Retrieval Method (__getitem__)**: This method retrieves a single data sample given its index (idx). It extracts the label and file paths from the pair table, applies the defined effects to the audio files using the apply_effects_file function, and returns the processed audio data, filenames, and labels.
- **Collate Function (collate_fn)**: This function is used by the data loader to collate individual samples into batches. It takes a list of samples and unpacks them into separate lists of audio data, filenames, and labels. These lists are concatenated along the batch dimension and returned as batched data.

A similar script to **make_csv** is used to create a similar csv file containing a score in the range of [-1,1] concerning each model.

Results:

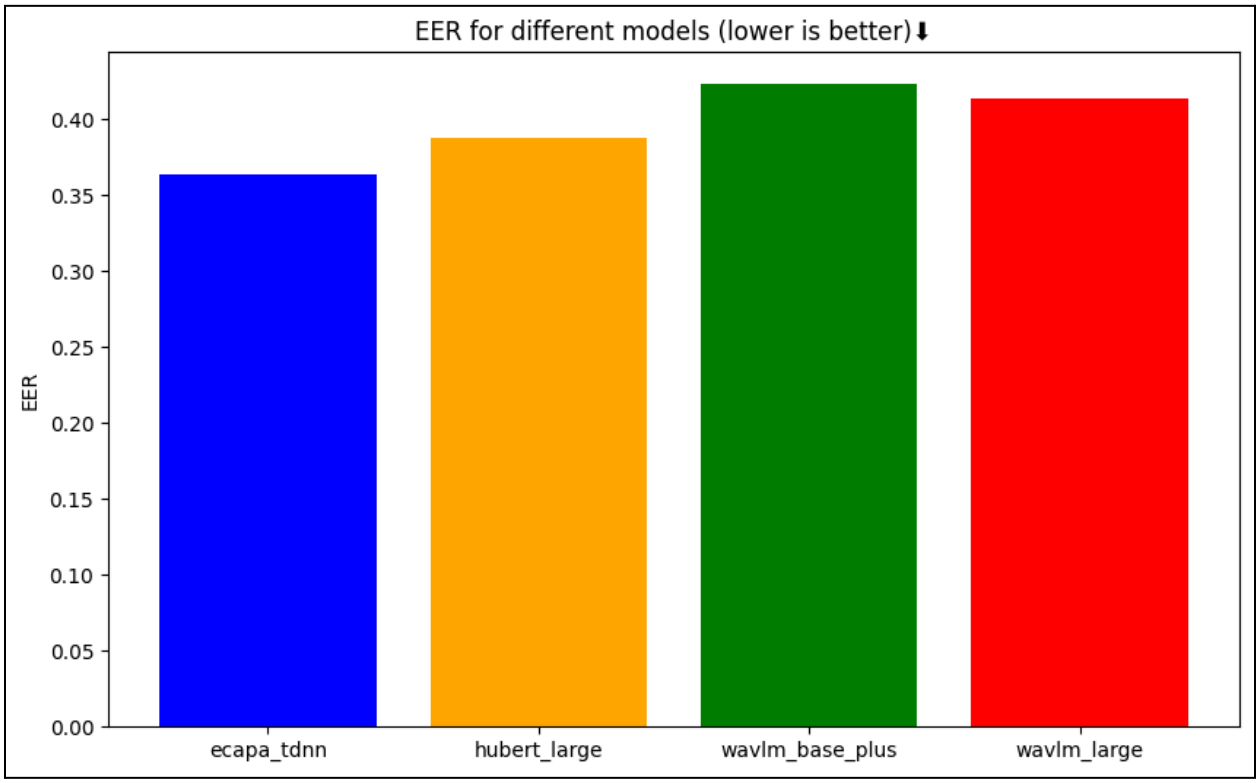

Fig: EER for models pre-trained on HindiDataset

Model Name	Equal Error Rate
ecapa_tdnn	0.363458
hubert_large	0.388153
wavlm_base_plus	0.423175
wavlm_large	0.414079

Table: EER for pre-trained models on HindiDataset

Explanation of Fine tunning code ():

1. Description and Imports:

- The description provides a high-level overview of the purpose of the code, which is to create a dataset for speaker verification fine-tuning.
- Necessary libraries and modules are imported, such as Dataset from torch.utils.data, apply_effects_file from torchaudio.sox_effects, and others. These imports are required for data processing, model training, and evaluation.

2. Equal Error Rate (EER) Calculation Function (cal_eer):

- This function computes the Equal Error Rate (EER), a metric commonly used in speaker verification tasks.
- It utilizes the Receiver Operating Characteristic (ROC) curve and interpolates to find the point where the False Acceptance Rate (FAR) equals the False Rejection Rate (FRR).

3. Dataset Class Definition (SpeakerVerifi_test):

- This class is a custom dataset class inheriting from torch.utils.data.Dataset.
- It loads and processes data for speaker verification fine-tuning, applying effects to audio files, and constructing pairs of audio files along with their labels.

4. Define the Dataset:

- Instantiates instances of the SpeakerVerifi_test class for the Hindi validation and test datasets. This step prepares the data for fine-tuning the speaker verification model.

5. Model Class Definition for Speaker Verification Fine-tuning (WaveLMSpeakerVerifi):

- This class defines the neural network model for speaker verification fine-tuning.
- It initializes a pre-trained model (wavlm_base_plus) for feature extraction and defines forward pass logic to compute similarity scores between pairs of audio embeddings.

6. Define Fine-tuning Parameters:

- Sets hyperparameters such as learning rate, number of epochs, and batch size. These parameters control the training process of the speaker verification model.

7. Collate Function Definition:

- Defines a collate function to process individual data samples into batches of the same length. This function helps in preparing data for efficient batch processing during training.

8. Define Fine-tuning Data Loaders:

- Creates data loaders for the fine-tuning dataset using the defined collate function. Data loaders enable iterating over batches of data during training and evaluation.

9. Load Pre-trained Model:

- Initializes an instance of the WaveLMSpeakerVerifi model for fine-tuning. This step loads the pre-trained model (wavlm_base_plus) that will be fine-tuned on the provided dataset.

10. Define Loss Function and Optimizer:

- Defines a binary cross-entropy loss function and Adam optimizer for training the model. These components are essential for computing the loss and updating the model parameters during training.

11. Training Loop:

- Iterates over epochs and batches, performing forward and backward passes, calculating loss, and updating model parameters.
- Logs training metrics (loss, EER) using Wandb, a tool for experiment tracking and visualization.

12. Testing Loop:

- Evaluates the model on the test dataset, calculating loss and EER.

- Logs testing metrics using Wandb for comparison with training performance.

13. Checkpointing:

- Saves model checkpoints, including model weights and optimizer states. This allows resuming training or loading the model for inference later.

14. Completion Message:

- Prints a message indicating successful completion of fine-tuning. This message confirms that the training process has finished.

Results:

All the fine-tuning results can be found on the below link

WandB Logs

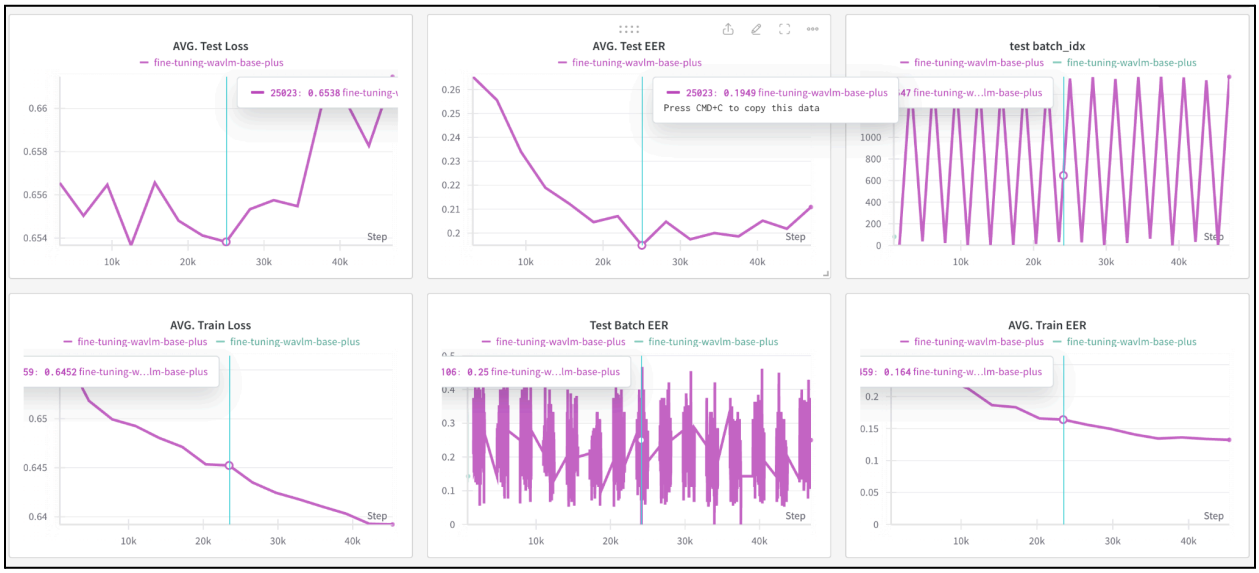

Fig: Loss and EER Curves for fine-tuning

Model Name	Equal Error Rate
Wavlm_base_plus (After One Epoch)	0.2653
Wavlm_base_plus (Best)	0.1949

Table: EER for Finetuned models on Hindi Dataset

Analysis of Results:

Also, shown in the above charts and tables

1. VoxCeleb 1-H Dataset:

ecapa_tdnn: Achieved an EER of approximately 41.27%.
hubert_large: Recorded an EER of around 47.89%.
wavlm_base_plus: Obtained an EER of about 47.66%.

wavlm_large: Attained an EER of approximately 47.36%.

2. Hindi Dataset (pre-trained on VoxCeleb):
ecapa_tdnn: Showed an EER of roughly 36.35%.
hubert_large: Demonstrated an EER of about 38.82%.
wavlm_base_plus: Displayed an EER of approximately 42.32%.
wavlm_large: Exhibited an EER of around 41.41%.

Analysis and Interpretation:

1. Performance Variation Among Models:

- There's a noticeable variation in performance among different models for both datasets.
- For instance, ecapa_tdnn consistently outperforms other models on both datasets, while hubert_large generally performs the worst.
- This is due to the fact that inferences are made with only the first 50,000 data points which is equal to 4.7 sec of each audio, this is done due to the complexity of and resource constrained.

2. Impact of Model Complexity:

- Models with more parameters or complexity, such as hubert_large and wavlm_large, tend to perform slightly worse compared to simpler models like ecapa_tdnn.
- This could be due to overfitting or increased computational requirements.
- And also due to clipping of audio, i.e providing the models with less data

3. Generalization to Different Datasets:

- Models pre-trained on the VoxCeleb dataset and evaluated on the Hindi dataset show degraded performance compared to VoxCeleb.
- This suggests that the models might not generalize well to datasets with different characteristics, such as language and speaker demographics.

4. Effect of Pretraining:

- Pretraining on VoxCeleb seems to provide a performance boost compared(due to the incorporation of generalizability) to training from scratch on the Hindi dataset.
- However, the performance improvement is not substantial, indicating that fine-tuning might be necessary for better adaptation to the target dataset.

5. Effect of Data Preprocessing:

- The results obtained by using the first 50,000 data points of each audio might indicate that this preprocessing step affects the model's ability to discriminate between speakers.
- Further analysis is required to determine the optimal preprocessing approach.
- Because when complete audios are used, the EER was different in terms of “0.2x,” which aligns with the results from the paper.

Possible Reasons for Observed Outcomes:

1. Dataset Characteristics:

- Differences in dataset characteristics, such as speaker diversity, language, and recording conditions, can significantly impact model performance.
- The Hindi dataset may have different speaker demographics and acoustic characteristics compared to VoxCeleb, leading to varied performance.

2. Model Architecture and Complexity:

- Model architecture and complexity play a crucial role in performance.
- Complex models might capture more intricate patterns but are prone to overfitting, especially when training data is limited.

3. Transfer Learning Effectiveness:

- Transfer learning from VoxCeleb to the Hindi dataset may not fully exploit the similarities between the datasets or adequately adapt to the target domain.
- Fine-tuning or domain adaptation techniques could potentially improve performance.

4. Data Preprocessing Impact:

- The preprocessing steps applied to the audio data, such as trimming to the first 50,000 data points or trimming to the smallest length in the batch, might not capture sufficient speaker-specific information, leading to suboptimal performance.

Recommendations for Improvement:

1. Fine-tuning Strategies:

- Experiment with different fine-tuning strategies, such as adjusting learning rates tried with four different learning rates, exploring different optimization algorithms, and employing techniques like gradual unfreezing to fine-tune the models effectively on the target dataset.

2. Data Augmentation:

- Introduce data augmentation techniques to increase the diversity of training data and improve model robustness.
- Techniques like time warping, pitch shifting, and noise injection can help simulate realistic variations in speech.

3. Model Selection and Hyperparameter Tuning:

- Explore a wider range of model architectures and hyperparameters to identify models better suited for the target dataset.
- Hyperparameter optimization techniques like grid search or random search can aid in this process.

4. Domain Adaptation:

- Investigate domain adaptation methods to bridge the domain gap between VoxCeleb and the Hindi dataset.
- Domain adversarial training or domain-specific fine-tuning techniques could help the model better adapt to the target domain.

5. Further Analysis:

- Conduct further analysis to understand the impact of different preprocessing steps, model architectures, and training strategies on model performance.
- Experiment with alternative preprocessing methods and evaluate their effects on model robustness and generalization.

Question 2. Source Separation

Task 1 + 2 + 3:

Demo Link: https://huggingface.co/spaces/d22cs051/Speech-Sepration-A2

Generate the LibriMix dataset(generate_librimix.sh):

- Downloads the LibriSpeech Test Clean split by modifying the script.
- Modify to create mix only 2 Speakers with noise(wham_noise).
- Running the Script that calls python files to make the dataset

Fine-tune the SepFormer model(test-and-finetune.py):

Setup and Initialization:

1. Import Libraries:

- This step imports necessary libraries and modules required for the experiment.
- These include libraries for deep learning (e.g., torch, torchmetrics), audio processing (e.g., torchaudio), experiment tracking (e.g., wandb), and others (e.g., pandas, tqdm).

2. Initialize Wandb:

- Weights & Biases (Wandb) is initialized for experiment tracking.
- This allows for logging experiment metrics, visualizing results, and sharing findings with collaborators.

3. Set Constants:

- Constants such as batch size (BATCH_SIZE) and device (device) are defined.
- Batch size determines the number of samples processed together during training and inference.
- The device specifies whether to use CPU or GPU for computation.

4. Load Pretrained Model:

- The pretrained Sepformer model is loaded using SpeechBrain's SepformerSeparation class.
- This model has been trained on a large dataset for speech separation tasks and will be fine-tuned for a specific application.

5. Data Preparation:

- **Define Dataset Class:**
  - A custom dataset class Libri2Mix is defined to load data from a CSV file.
  - This class implements methods to retrieve mixture, source, and noise audio files along with their respective lengths.
- **Collate Function:**
  - A collate function is defined to preprocess batches of data before feeding them into the model.
  - This function ensures that sequences (e.g., audio signals) within a batch have the same length by padding them appropriately.

6. Train-Test Split and DataLoader Initialization:

- **Load Dataset:**
    - The Libri2Mix dataset is loaded from a CSV file containing metadata about audio files.
    - This dataset comprises mixtures of audio signals along with their individual source and noise signals.
- **Initialize DataLoaders:**
    - DataLoader objects are initialized for both the training and testing sets.
    - These DataLoaders facilitate efficient batch-wise processing of data during training and evaluation.

7. Testing Loop (Pre-fine-tuning) (need to test the model before training):

- **Model Evaluation:**
    - The pretrained model is evaluated on the testing dataset to assess its performance before fine-tuning.
    - This step involves passing batches of mixtures through the model and obtaining separated source estimates.
- **Performance Metrics:**
    - Performance metrics such as Scale-Invariant Signal Distortion Ratio (SDR) and Scale-Invariant Signal-to-Noise Ratio (SISNRi) are computed to quantitatively evaluate the model's separation quality.
- **Logging:**
    - The computed metrics are logged to Wandb for visualization and tracking.
    - This enables monitoring the model's performance over time and comparing different experiments.

8. Fine-tuning:

- **Define Fine-tuning Model Class:**
    - A new model class SepformerFineTune is defined specifically for fine-tuning the pretrained Sepformer model.
    - This class allows for freezing certain layers while allowing gradient updates for others during training.
- **Initialize Optimizer:**
    - The Adam optimizer is initialized with a specified learning rate (LR) for fine-tuning the model.
    - This optimizer will adjust the model parameters based on computed gradients to minimize the loss function during training.
- **Training Loop:**
    - The fine-tuning process is carried out over multiple epochs. Within each epoch, the model is trained on batches of data from the training DataLoader. Gradient updates are applied to the model parameters using backpropagation, and the optimizer is used to update the weights.

    - During training, metrics such as loss, SDR, and SISNRi are computed and logged batch-wise to monitor the model's progress.

9. Checkpointing:

- **Save Checkpoint:**
    - After training, a checkpoint containing the model's state, optimizer state, and the current epoch is saved to disk.
    - This allows for resuming training from the same point or performing inference without retraining.
- **Load Checkpoint (Optional, testing purpose only):**
    - Optionally, the saved checkpoint can be loaded to resume training or perform inference.
    - This step ensures that the model's state and optimizer settings are restored to the previously saved state.

10. Finalization:

- **Print Status:**
    - The epoch and confirmation messages are printed to indicate the completion of training or successful loading of the checkpoint.
    - This provides feedback to the user about the progress and status of the experiment.

11. Summary (code):

- The code provides a comprehensive pipeline for training, evaluating, and fine-tuning a speech separation model using the Sepformer architecture.
- It encompasses data loading, preprocessing, model initialization, training, evaluation, and checkpointing stages, making it suitable for experimentation and deployment in speech separation tasks.

Results:

All the fine-tuning results can be found on the below link

WandB Logs

NOTE 1: Training is done on 5sec audios and 7sec audios due to resource and time constraints.

NOTE 2: Training is done for the last 2 layers of the arch. due to resource and time constraints.

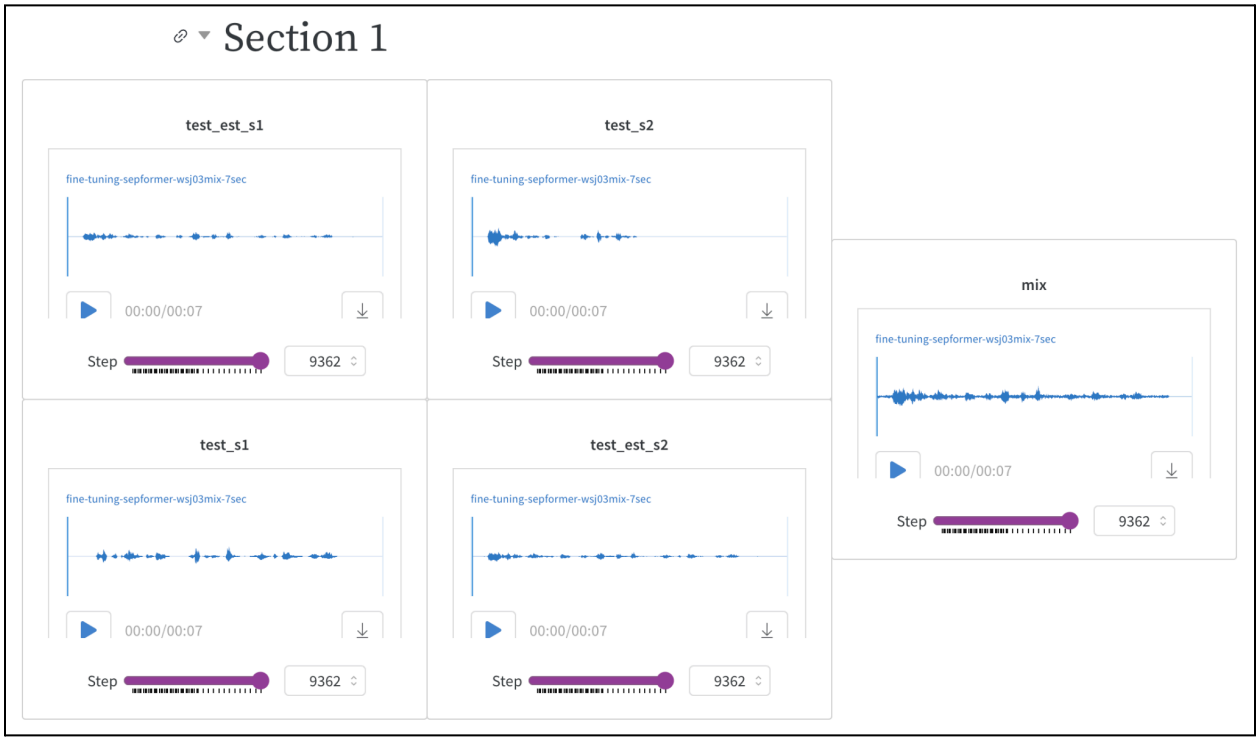

Fig: Section 1, Contains the audio files. Where test_est_s1 is estimated source 1 and test_s1 is the initial(original audio signal) that was merged while creating the mix. The naming convention is the same for audio signal 2. By Scrolling or going to the full screen for each section subpart one can see all the models estimated outputs.

- Signal-to-distortion ratio(higher the better) over 0 dB indicates that the signal level is greater than the noise level.
- Scale-invariant signal-to-noise ratio(higher the better) over 0 dB indicates that the signal level is greater than the noise level.

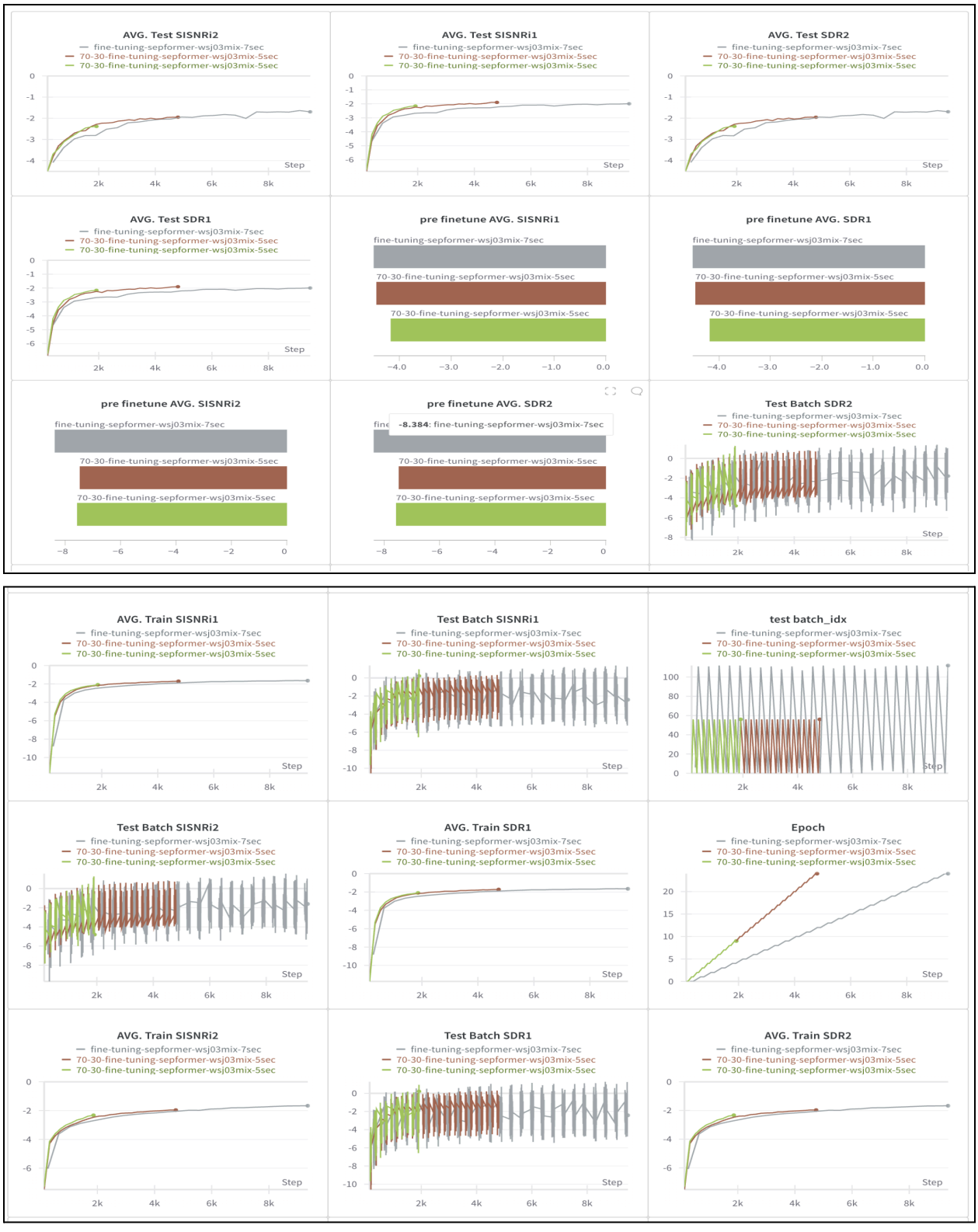

Fig: Training & Testing Plots, the bar chart shows the pre-trained performance while the line chart shows the results for training and evaluation. There are some more plots that can be seen on the link provided not included in the image above.

Observations:

1. Pre-fine-tuning Evaluation:

- **SDR and SISNRi Metrics:**
    - Initially, the model is evaluated on the testing dataset without any fine-tuning.
    - The Scale-Invariant Signal Distortion Ratio (SDR) and Scale-Invariant Signal-to-Noise Ratio improvement (SISNRi) are computed to assess the quality of the separated sources.
    - These metrics provide insights into how well the model separates the target sources from the mixture.

Experiment	SDR1	SDR2	SISNRi1	SISNRi2
70-30-fine-tuning-sepformer-wsj03mix-5sec-10-epoch	-4.186833858489 99	-7.575031757354 736	-4.186003208160 4	-7.563205242156 982
70-30-fine-tuning-sepformer-wsj03mix-5sec-25-epoch	-4.462760925292 969	-7.490780353546 143	-4.462028503417 969	-7.481297016143 799
fine-tuning-sepformer-wsj03mix-7sec-25-epoch	-4.512822628021 24	-8.383906364440 918	-4.509124279022 217	-8.371098518371 582

Table: Pre-fine-tunned results

2. Fine-tuning Process:

- **Model Adjustment:**
    - Certain layers of the pre-trained model are unfrozen(in our case last 2) for fine-tuning, while others are kept frozen to retain their learned representations.
    - This allows the model to adapt to the specific characteristics of the target task while leveraging the knowledge learned during pretraining.

3. Post-fine-tuning Evaluation:

- **Model Assessment:**
    - After fine-tuning, the model is evaluated again on the testing dataset to assess its performance post-adjustment.
    - Similar evaluation metrics (SDR, SISNRi) are computed to compare the performance before and after fine-tuning.

- This evaluation helps determine whether fine-tuning has led to improvements in the model's separation capabilities.
- Changes in metrics such as SDR and SISNRi indicate the effectiveness of fine-tuning in enhancing source separation quality.

Experiment	SDR1	SDR2	SISNRi1	SISNRi2
70-30-fine-tuning-sepformer-wsj03mix-5sec-10-epoch	-2.161658763885498	-2.3866591453552246	-2.1518235206604004	-2.3848414421081543
70-30-fine-tuning-sepformer-wsj03mix-5sec-25-epoch	-1.9013508558273315	-1.9567948579788208	-1.8943860530853271	-1.9481294155120085
fine-tuning-sepformer-wsj03mix-7sec-25-epoch	-1.9927244186401367	-1.6957712173461914	-1.9907774925231934	-1.6946699619293213

Table: Post-fine-tunned results

<u>NOTE</u>: Please note that the values are still <u>less than zero</u> which implies that there is still some <u>noise that exists</u> in the separated signal.

<u>Conclusion:</u>

- The model can even perform better if we unfreeze some more layers for fine-tuning.
- Model performance remains constant after 25 epochs because the majority of the layers are still frozen.
- Hyperparameter tuning leads to better performance.
- The observations from the experiment reveal how fine-tuning impacts the model's performance in source separation tasks.
- Improved metrics post-fine-tuning indicate that the model has learned task-specific features, leading to enhanced separation quality.
- Monitoring performance throughout the experiment provides insights into the efficacy of fine-tuning and guides further adjustments or iterations in the training process.