Table of Contents

1. Speaker Details

Initially, speaker details were extracted from the dataset and stored in a CSV file.

For Shaip
https://drive.google.com/file/d/15T0U1_aTHkVl0BD_TBlG7z0UfQhzQD7O/view?usp=drive_link

For Megdap
https://drive.google.com/file/d/1FkwB_3G0qFcq0lXOBGh-fW5FxlimiZtP/view?usp=drive_link

2. Train, Test and Val split

I split the dataset into three parts: training, testing, and validation, making sure each part had a balanced mix of gender, language, and district. Each part has its own folder with the respective details stored inside.

For Shaip
https://drive.google.com/drive/folders/1Tb2xaecnLJ9jwwAH-oF_F1rMwiahd09W?usp=drive_link

For megdap
https://drive.google.com/drive/folders/1vX3yTrm0sxqcO5zPlB6cSsN5sBTeclZS?usp=drive_link

3. Model pyannote/wespeaker-voxceleb-resnet34-LM

3.1. Pretrained

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ Resnet34	0.03255	2.4949	0.00699	0.10874	0.1206

3.2. Fine Tuning

3.2.1. Experiment 1

No. of speaker

	Train	Test	Validation
Shaip	4733	1578	1578
Megdap	4006	1334	1334

Each Speaker is having 5 audio files

Results on Shaip Test set

I fine-tuned the model by modifying only the last layer to accommodate the desired number of classes. All other parameters were kept consistent, adhering to the configuration outlined in a YAML file employed during training. To address CUDA out of memory issues, the batch size was reduced from 64 to 16. The fine-tuning procedure spanned 15 epochs and utilized the shaip training set.

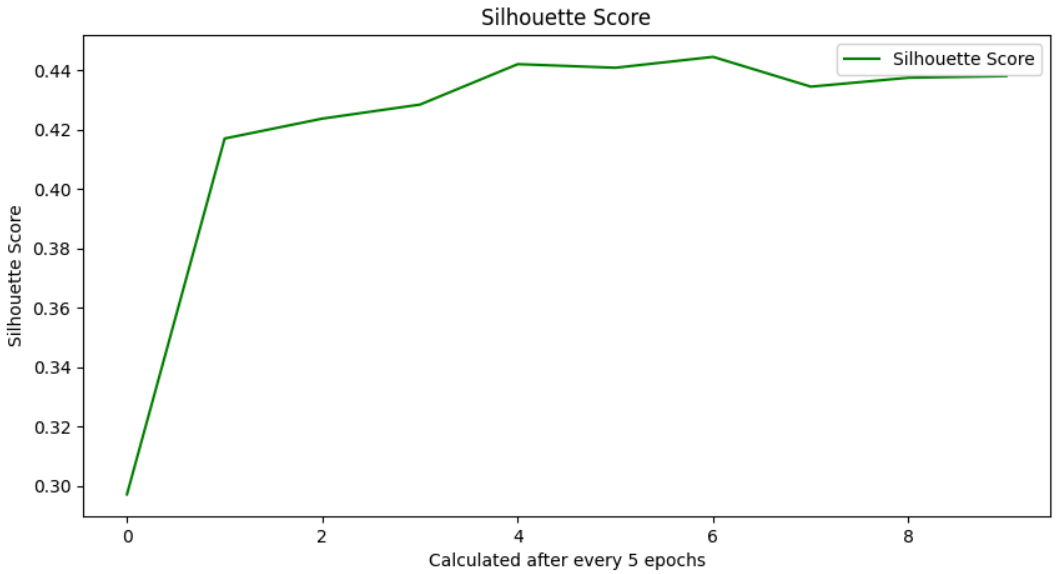
EER%(Equal error rate)		
	Pretrained	**Fine Tuned**
Shaip Test set	0.03255	0.100408
Voxceleb Test set	2.4949	6.4896
Shaip Validation set	0.00699	0.0584

Analysis

The fine-tuned model isn't exhibiting notable improvement despite training it for 15 epochs. Validation loss appears to plateau after the third epoch. Subsequent evaluation reveals that the fine-tuned model performs less effectively compared to the pretrained model, as assessed by the Equal Error Rate (EER). However, it's noteworthy that the inference speed is 5x quicker for the fine-tuned model.

3.2.2. Experiment 2

In this experiment, I'm replicating the setup from Experiment 1. However, I've modified the evaluation process. Previously, I evaluated and saved the model based on the validation loss. This approach might not be ideal because the speakers in the training and validation sets are distinct. Therefore, for this experiment, I'm evaluating the model's ability to cluster together acoustic features from the same speaker, as opposed to features from different speakers. This will provide a more accurate assessment of the model's performance in real-world scenarios where it encounters unseen speakers. Batch size = 64.



EER%(Equal error rate)			
	Pretrained	**Fine Tuned Experiment 1**	**Fine Tuned Experiment 2**
Shaip Test set	0.03255	0.100408	0.3399
Voxceleb Test set	2.4949	6.4896	5.901
Shaip Validation set	0.00699	0.0584	0.3569

3.2.3. Experiment 3

Changed the loss to triplet loss from cross entropy loss and again I am evaluating and saving the model based on validation loss.

```python
def generate_triplets(embeddings, labels):
    triplets = []
    for i in range(len(labels)):
        anchor_embedding = embeddings[i]
        anchor_label = labels[i]
        positive_indices = torch.where(labels == anchor_label)[0]
        negative_indices = torch.where(labels != anchor_label)[0]
        if len(positive_indices) == 0 or len(negative_indices) == 0:
            continue
        positive_embedding = embeddings[positive_indices[torch.randint(0, len(positive_indices), (1,))]]
        negative_embedding = embeddings[negative_indices[torch.randint(0, len(negative_indices), (1,))]]
        triplets.append((anchor_embedding, positive_embedding, negative_embedding))
    return triplets

def triplet_loss(anchor_embeddings, positive_embeddings, negative_embeddings, margin=1.0):
    distance_positive = nn.functional.pairwise_distance(anchor_embeddings, positive_embeddings)
    distance_negative = nn.functional.pairwise_distance(anchor_embeddings, negative_embeddings)
    loss = nn.functional.relu(distance_positive - distance_negative + margin)
    return loss.mean()
```

Results

The EER% on the SHAIP test is 24.68, indicating a significant drop in performance compared to previous experiments. Despite trying different margins such as 1 and 0.1, the results remain consistent.

3.2.4. Experiment 4

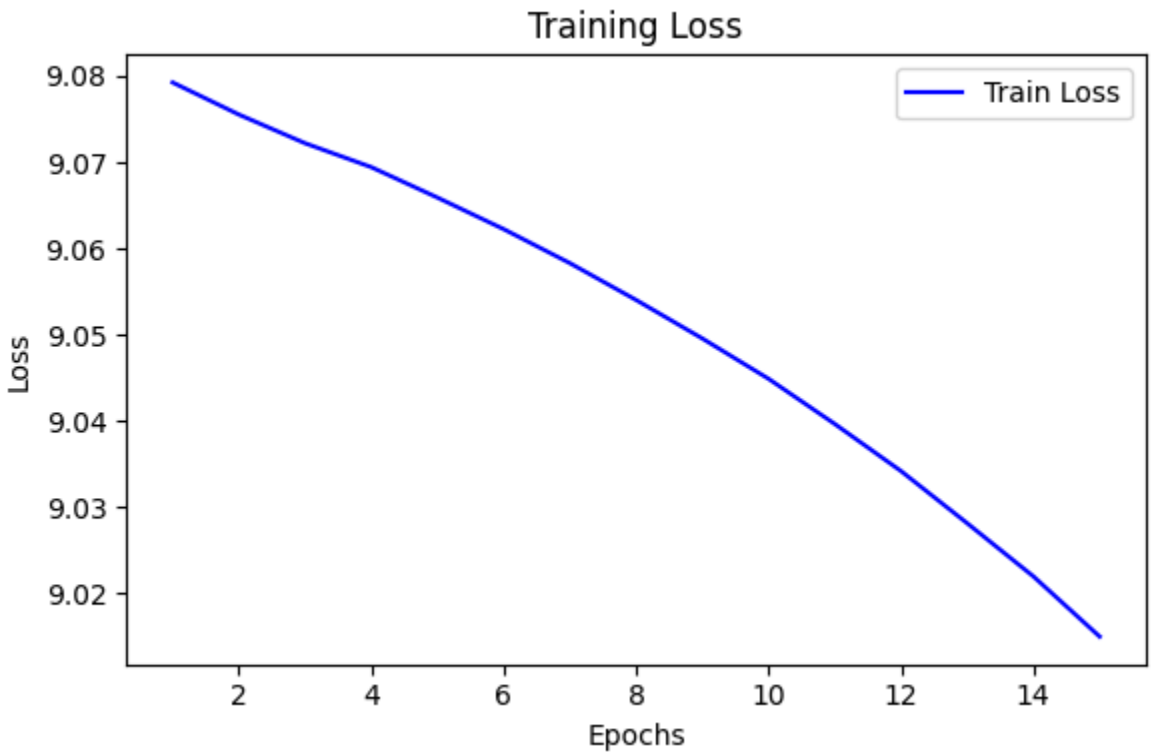
In this I am using both the Megdap and Shaip data for training.
I fine-tuned the model by adding the last layer to accommodate the desired number of classes. All other parameters were kept consistent, adhering to the configuration outlined in a YAML file employed during training. To address CUDA out of memory issues, the batch size was reduced from 64 to 32. The fine-tuning procedure spanned 15 epochs.

No. of speaker

	Train	Test	Validation
Shaip	4733	1578	1578
Megdap	4006	1334	1334

Each Speaker is having 5 audio files

Plots



EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ Resnet34	**0.03255**	**2.4949**	**0.00699**	0.10874	0.1206
Fine Tuned	0.07203	3.0028	0.04749	**0.03758**	**0.03973**

3.2.5. Experiment 5

Same as experiment 4 the difference is batch size = 64 and using adapters.

```
class Adapter(nn.Module):
    def __init__(self, input_dim, bottleneck_dim):
        super(Adapter, self).__init__()
        self.down_proj = nn.Linear(input_dim, bottleneck_dim)
        self.activation = nn.ReLU()
        self.up_proj = nn.Linear(bottleneck_dim, input_dim)
        self.layer_norm = nn.LayerNorm(input_dim)

    def forward(self, x):
        residual = x
        x = self.down_proj(x)
        x = self.activation(x)
        x = self.up_proj(x)
        x = self.layer_norm(x + residual)
        return x
```

```
# Freeze the parameters of the pretrained model
for param in model.parameters():
    param.requires_grad = False

# Define the adapter layer
adapter = Adapter(input_dim=2498, bottleneck_dim=128)
model.resnet.layer1.add_module('adapter', adapter)

adapter = Adapter(input_dim=1249, bottleneck_dim=64)
model.resnet.layer2.add_module('adapter', adapter)

adapter = Adapter(input_dim=625, bottleneck_dim=32)
model.resnet.layer3.add_module('adapter', adapter)

# Add the extra layer
extra_layer = nn.Linear(in_features=256, out_features=num_classes)
model.resnet.seg_1 = nn.Sequential(model.resnet.seg_1, extra_layer)
```

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ Resnet34	**0.03255**	**2.4949**	**0.00699**	0.10874	0.1206
Fine Tuned	5.9284	34.71	5.749	6.498	6.072

3.2.6. Experiment 6

In this I am using both the Megdap and Shaip data for training.
I fine-tuned the model by adding the last layer to accommodate the desired number of classes. All other parameters were kept consistent, adhering to the configuration outlined in a YAML file employed during training. To address CUDA out of memory issues, the batch size was reduced from 64 to 32. The fine-tuning procedure spanned 15 epochs.(same as experiment 4)

The only change is that I am using data augmentation for this experiment.

```
def __len__(self):
    # Each original audio is seen once and each augmented audio is seen once
    #print("length_train" ,len(self.file_list) * 2 )
    return len(self.file_list) * 2

def __getitem__(self, idx):
    original_idx = idx // 2
    is_augmented = idx % 2 == 1

    root_dir, file_name = self.file_list[original_idx]
    file_path = os.path.join(root_dir, file_name)

    # Load original audio file
    waveform, sample_rate = torchaudio.load(file_path)

    # Resample the audio to 16 kHz if the sampling rate is different
    if sample_rate != 16000:
        resampler = Resample(orig_freq=sample_rate, new_freq=16000)
        waveform = resampler(waveform)

    # Ensure the audio is mono (if not already)
    if waveform.shape[0] > 1:
        waveform = waveform.mean(dim=0, keepdim=True)

    # Apply augmentation by mixing with a noisy audio file if required
    if is_augmented:
        snr_range = [0, 15]

        noisy_file = random.choice(self.noisy_files)
        noisy_path = os.path.join(self.noisy_dir, noisy_file)
        noisy_waveform, noisy_sample_rate = torchaudio.load(noisy_path)

        # Resample the noisy audio to 16 kHz if the sampling rate is different
        if noisy_sample_rate != 16000:
            resampler = Resample(orig_freq=noisy_sample_rate, new_freq=16000)
            noisy_waveform = resampler(noisy_waveform)

        # Ensure the noisy audio is mono (if not already)
        if noisy_waveform.shape[0] > 1:
            noisy_waveform = noisy_waveform.mean(dim=0, keepdim=True)

        target_snr = random.uniform(snr_range[0], snr_range[1])
        target_power = torch.mean(waveform ** 2) / (10 ** (target_snr / 10))
        noisy_waveform = adjust_power_to_target(noisy_waveform, target_power)
```

```
        # # Trim or pad noisy waveform to match the length of the original waveform
        noisy_waveform = self._adjust_waveform_length(noisy_waveform, target_length=waveform.shape[1])

        # Mix original and noisy audio
        waveform = waveform + noisy_waveform

    # Apply preprocessing transformations if specified
    if self.transform:
        waveform = self.transform(waveform)

    # Pad or trim waveform to the maximum length
    waveform = self._adjust_waveform_length(waveform, target_length=self.max_length)

    # Get speaker ID from filename and convert to label
    speaker_id = file_name.split('_')[2]
    label = self.speaker_to_label[speaker_id]

    return waveform, label
```

```
def adjust_power_to_target(signal, target_power):
    current_power = torch.mean(signal ** 2)
    scaling_factor = torch.sqrt(target_power / current_power)
    adjusted_signal = signal * scaling_factor
    return adjusted_signal
```

This collection includes various types of environmental and mechanical noises, human sounds, and musical elements. Environmental noises feature city traffic, tree cutting, and dam overflow sounds. Mechanical noises encompass the hum of vacuum cleaners, car engine sounds from Ford and BMW, and construction sounds like jackhammers and hammering. Human sounds include various coughs, hiccups, giggles, burps, and screams, as well as distinct actions like heavy breathing, door creaks, and footfalls. Additionally, there are musical and media-related elements, such as soundtracks from trailers, live performances, and songs by artists like Simply Red.

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ Resnet34	0.03255	2.4949	**0.00699**	0.10874	0.1206
Fine Tuned	**0.03172**	**1.9765**	0.02362	**0.0563**	**0.03746**

3.2.7. Experiment 7

Same as experiment 6 but added speed perturbation.

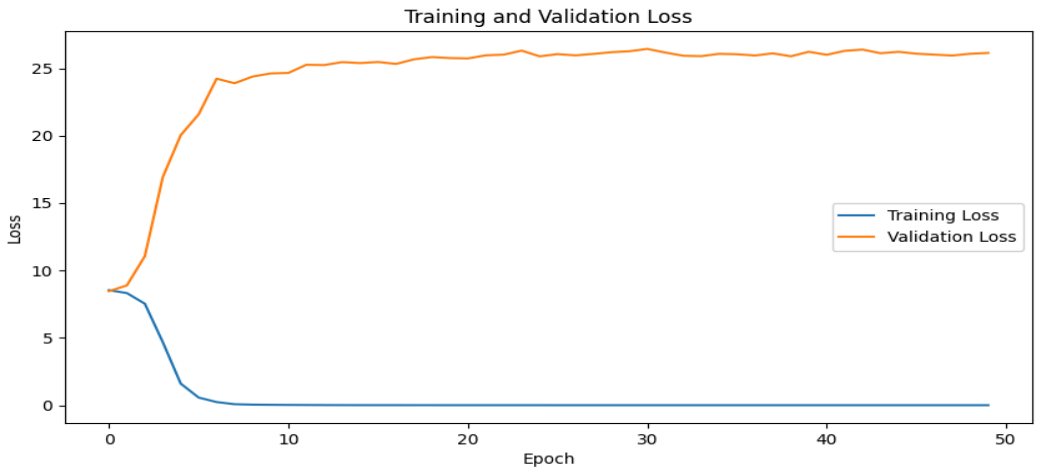
```
def __len__(self):
    #print("length",len(self.file_list) * 3  )
    return len(self.file_list) * 3  # Each original audio, speed perturbed audio,

def __getitem__(self, idx):
    original_idx = idx // 3
    perturb_type = idx % 3

    root_dir, file_name = self.file_list[original_idx]
    file_path = os.path.join(root_dir, file_name)

    # Load original audio file
    waveform, sample_rate = torchaudio.load(file_path)

    # Resample the audio to 16 kHz if the sampling rate is different
    if sample_rate != 16000:
        resampler = Resample(orig_freq=sample_rate, new_freq=16000)
        waveform = resampler(waveform)

    # Ensure the audio is mono (if not already)
    if waveform.shape[0] > 1:
        waveform = waveform.mean(dim=0, keepdim=True)

    if perturb_type == 1:  # Apply speed perturbation
        speed_idx = random.randint(0, 1)
        speed_factor = speeds[speed_idx]
        wav, _ = torchaudio.sox_effects.apply_effects_tensor(
            waveform, sample_rate,
            [['speed', str(speed_factor)], ['rate', str(sample_rate)]])
        waveform = wav

    # Apply augmentation by mixing with a noisy audio file if required
    elif perturb_type == 2:
        snr_range = [0, 15]

        noisy_file = random.choice(self.noisy_files)
        noisy_path = os.path.join(self.noisy_dir, noisy_file)
        noisy_waveform, noisy_sample_rate = torchaudio.load(noisy_path)
```

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ Resnet34	0.03255	**2.4949**	**0.00699**	0.10874	0.1206
Fine Tuned	**0.02951**	2.7462	0.01623	**0.02677**	**0.02681**

3.3. Train from Scratch

3.3.1. Experiment 1

- The acoustic features consist of 80-dimensional log Mel-filter banks (Fbank), computed with a frameshift of 10ms and a frame window of 25ms. During training, all data is segmented into chunks of 200 frames, and Cepstral Mean Normalization (CMN) is applied without Concurrent Voice Normalization (CVN).
- Using a batch size of 64 and a sampling rate of 16000Hz, the audio data is converted to mono. The model architecture remains unchanged, except for the final layers, which are adjusted for classification purposes. The initial learning rate is set to 0.0001, and a ReduceLROnPlateau scheduler is employed with a reduction factor of 0.2, a patience of 5 epochs, and verbosity enabled for monitoring.

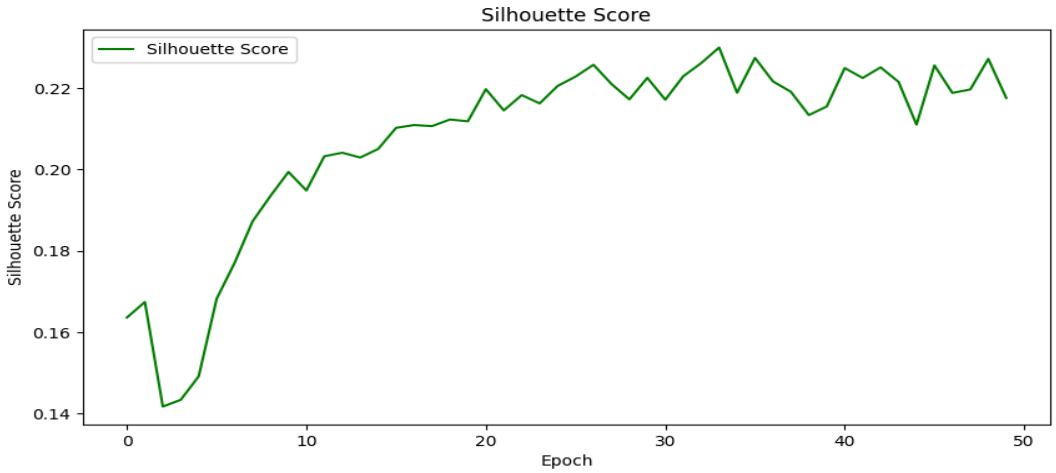


EER%(Equal error rate)			
	Pretrained	**Fine Tuned**	**Train from Scratch**
Shaip Test set	0.03255	0.100408	5.838
Voxceleb Test set	2.4949	6.4896	16.87
Shaip Validation set	0.00699	0.0584	5.5435

3.3.2. Experiment 2

In this experiment, I'm replicating the setup from Experiment 1. However, I've modified the evaluation process. Previously, I evaluated and saved the model based on the validation loss. This approach might not be ideal because the speakers in the training and validation sets are distinct. Therefore, for this experiment, I'm evaluating the model's ability to cluster together acoustic features from the same speaker, as opposed to features from different speakers. This will provide a more accurate assessment of the model's performance in real-world scenarios where it encounters unseen speakers.

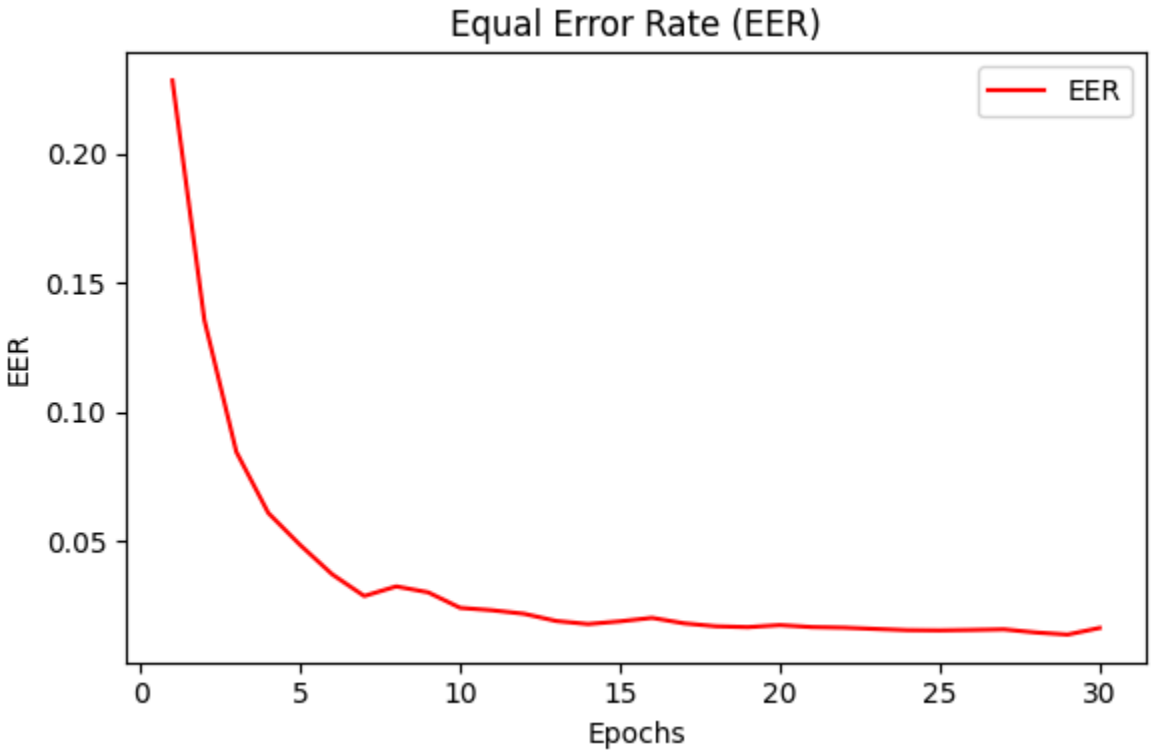
```python
# Define function to evaluate model using silhouette score
def evaluate_model_silhouette(model, validation_loader):
    model.eval()
    all_embeddings = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in validation_loader:
            embeddings = model(inputs)
            all_embeddings.append(embeddings.cpu().numpy())
            all_labels.append(labels.cpu().numpy())
    all_embeddings = np.concatenate(all_embeddings, axis=0)
    all_labels = np.concatenate(all_labels, axis=0)

    # Perform clustering
    num_clusters = len(np.unique(all_labels))  # Number of clusters equals the number of unique speakers
    kmeans = KMeans(n_clusters=num_clusters)
    cluster_labels = kmeans.fit_predict(all_embeddings)

    # Calculate silhouette score
    silhouette = silhouette_score(all_embeddings, cluster_labels)

    return silhouette
```



EER%(Equal error rate)			
	Pretrained	**Fine Tuned**	**Train from Scratch**
Shaip Test set	0.03255	0.100408	2.37
Voxceleb Test set	2.4949	6.4896	9.82
Shaip Validation set	0.00699	0.0584	2.28

3.3.3. Experiment 3

The procedure remains the same as in experiment 2, with the only changes being the evaluation metric, which is now EER (Equal Error Rate), and the inclusion of both the Megdap and Shaip datasets for evaluation.



EER%(Equal error rate)						
		Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Train Scratch	**Exp_3**	2.2955	9.6970	2.1136	1.4375	1.3559

4. Summary

EER%(Equal error rate)						
		Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_Resnet34		0.03255	2.4949	**0.00699**	0.10874	0.1206
Pretrained_xvector		0.02105	2.776	0.01120	0.005138	0.02806
Fine Tuned	**Exp_1**	0.100408	6.4896	0.0584		
	Exp_2	0.3399	5.901	0.3569		
	Exp_3	24.68				
	Exp_4	0.07203	3.0028	0.04749	0.03758	0.03973
	Exp_5	5.9284	34.71	5.749	6.498	6.072
	Exp_6	0.03172	**1.9765**	0.02362	0.0563	0.03746
	Exp_7	**0.02951**	2.7462	0.01623	**0.02677**	**0.02681**
Train Scratch	**Exp_1**	5.838	16.87	5.5435		
	Exp_2	2.37	9.82	2.28		
	Exp_3	2.2955	9.6970	2.1136	1.4375	c

Fine tuned

- Experiment 2 changed the evaluation of validation set to silhouette score.
- Experiment 3 changed the loss from cross entropy to triplet loss.
- Experiment 4 used both the dataset megdap and shaip for training.
- Experiment 5 same as 4 but here using adapters
- Experiment 6 I am using data augmentation adding different types of background noise.

- Experiment 6 I am using data augmentation adding different types of background noise and speed perturbation

Train Scratch

- Experiment 2 changed the evaluation of validation set to silhouette.
- Experiment 3 changed the evaluation of validation set to EER and used both the dataset megdap and shaip for training.

Analysis

In summary, the pretrained Resnet34 model showed strong initial performance with low EERs. Fine-tuned models had mixed results: Experiment 6 (with data augmentation) had the best EERs, while Experiments 3 (triplet loss) and 5 (with adapters) performed poorly. Using both datasets (Experiment 4) and data augmentation (Experiment 6) significantly improved performance. Models trained from scratch generally had higher EERs, with some improvement seen when both datasets were used and evaluation methods were adjusted. The best results overall were from the Fine-Tuned Experiment 6.

I have experimented with both training from scratch and fine-tuning the model. I tested two different loss functions: cross entropy loss and triplet loss. To identify the best model, I explored various evaluation techniques, including the silhouette score for assessing cluster efficiency based on speaker embeddings, validation loss, and the equal error rate (EER) metric. Additionally, I fine-tuned the model using adapters and applied data augmentation by adding different background noises to the audio files.

In my latest experiment with data augmentation, I achieved better performance than the pretrained model, including superior results on the Voxceleb test set.

Plan

To further enhance performance, I will experiment with data augmentation techniques like speed perturbation and reverberation. Additionally, I will explore various fine-tuning methods, such as gradual unfreezing and selectively freezing certain layers, to assess their impact. I will also replicate these experiments using the pyannote x-vector model and try deeper versions of the wespeaker models for comparison.

5. pyannote/embedding(xvector)

5.1. Pretrained

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ xvector	0.02105	2.776	0.01120	0.005138	0.02806

5.2. Finetune

5.2.1. Experiment 1

In this I am using both the Megdap and Shaip data for training.
I fine-tuned the model by adding the last layer to accommodate the desired number of classes. All other parameters were kept consistent, adhering to the configuration outlined in a YAML file employed during training. The batch size was increased from 64 to 200. The fine-tuning procedure spanned 15 epochs. Adam optimizer.
I am using data augmentation for this experiment. Both adding backgroud noise and speed pertubation.

```
def __len__(self):
    #print("length",len(self.file_list) * 3  )
    return len(self.file_list) * 3  # Each original audio, speed perturbed audio,

def __getitem__(self, idx):
    original_idx = idx // 3
    perturb_type = idx % 3

    root_dir, file_name = self.file_list[original_idx]
    file_path = os.path.join(root_dir, file_name)

    # Load original audio file
    waveform, sample_rate = torchaudio.load(file_path)

    # Resample the audio to 16 kHz if the sampling rate is different
    if sample_rate != 16000:
        resampler = Resample(orig_freq=sample_rate, new_freq=16000)
        waveform = resampler(waveform)

    # Ensure the audio is mono (if not already)
    if waveform.shape[0] > 1:
        waveform = waveform.mean(dim=0, keepdim=True)

    if perturb_type == 1:  # Apply speed perturbation
        speed_idx = random.randint(0, 1)
        speed_factor = speeds[speed_idx]
        wav, _ = torchaudio.sox_effects.apply_effects_tensor(
            waveform, sample_rate,
            [['speed', str(speed_factor)], ['rate', str(sample_rate)]])
        waveform = wav

    # Apply augmentation by mixing with a noisy audio file if required
    elif perturb_type == 2:
        snr_range = [0, 15]

        noisy_file = random.choice(self.noisy_files)
        noisy_path = os.path.join(self.noisy_dir, noisy_file)
        noisy_waveform, noisy_sample_rate = torchaudio.load(noisy_path)
```

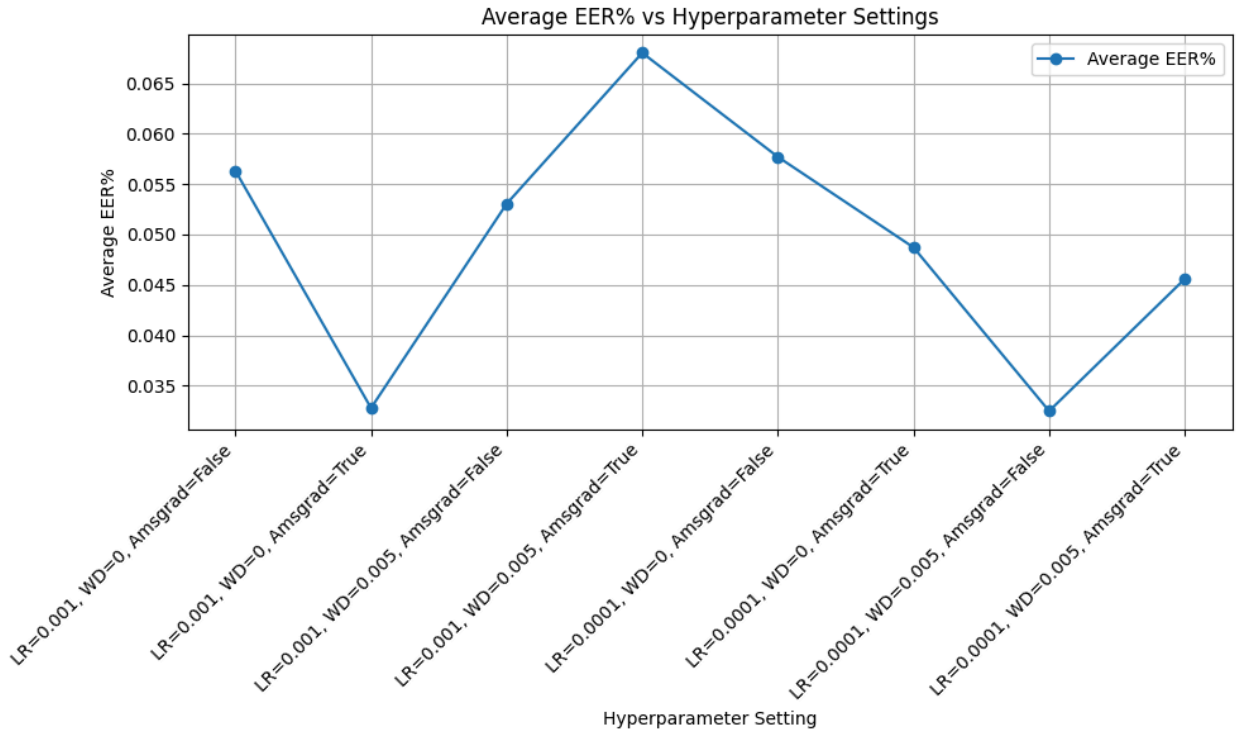
```python
def adjust_power_to_target(signal, target_power):
    current_power = torch.mean(signal ** 2)
    scaling_factor = torch.sqrt(target_power / current_power)
    adjusted_signal = signal * scaling_factor
    return adjusted_signal
```

This collection includes various types of environmental and mechanical noises, human sounds, and musical elements. Environmental noises feature city traffic, tree cutting, and dam overflow sounds. Mechanical noises encompass the hum of vacuum cleaners, car engine sounds from Ford and BMW, and construction sounds like jackhammers and hammering. Human sounds include various coughs, hiccups, giggles, burps, and screams, as well as distinct actions like heavy breathing, door creaks, and footfalls. Additionally, there are musical and media-related elements, such as soundtracks from trailers, live performances, and songs by artists like Simply Red.

EER%(Equal error rate)					
	Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_ xvector	0.02105	2.776	0.01120	0.005138	0.02806

Fine Tuned		**Exp1**					
Learning Rate	**Weight Decay**	**Amsgrad**	**Shaip Test set**	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
0.001	**0**	**False**	0.0521	6.834	0.0514	0.0535	0.0683
0.001	**0**	**True**	**0.0377**	5.595	0.0383	0.0268	**0.0284**
0.001	**0.005**	**False**	0.0701	5.712	0.0635	0.0433	0.0353
0.001	**0.005**	**True**	0.0709	5.859	0.0632	0.06623	0.07174
0.0001	**0**	**False**	0.0559	4.505	0.0741	0.0485	0.0523
0.0001	**0**	**True**	0.0642	4.550	0.0624	0.0295	0.0388

0.0001	**0.005**	**False**	0.0393	4.126	**0.0346**	**0.0244**	0.0317
0.0001	**0.005**	**True**	0.0451	**4.061**	0.0603	0.0342	0.0428



5.3. Train Scratch

6. Summary

EER%(Equal error rate)							
		Shaip Test set	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
Pretrained_Resnet34		0.03255	2.4949	**0.00699**	0.10874	0.1206
Pretrained_xvector		0.02105	2.776	0.01120	0.005138	0.02806
Resnet34						
Fine Tuned	**Exp_1**	0.100408	6.4896	0.0584		
	Exp_2	0.3399	5.901	0.3569		
	Exp_3	24.68				
	Exp_4	0.07203	3.0028	0.04749	0.03758	0.03973
	Exp_5	5.9284	34.71	5.749	6.498	6.072
	Exp_6	0.03172	**1.9765**	0.02362	0.0563	0.03746
	Exp_7	**0.02951**	2.7462	0.01623	0.02677	**0.02681**
xvector						
Fine Tuned		**Exp1**				
Learning Rate	**Weight Decay**	**Amsgrad**	**Shaip Test set**	**Voxceleb Test set**	**Shaip Val set**	**Megdap Test**	**Megdap Val**
0.001	**0**	**False**	0.0521	6.834	0.0514	0.0535	0.0683
0.001	**0**	**True**	0.0377	5.595	0.0383	0.0268	0.0284

0.001	**0.005**	**False**	0.0701	5.712	0.0635	0.0433	0.0353
0.001	**0.005**	**True**	0.0709	5.859	0.0632	0.06623	0.07174
0.0001	**0**	**False**	0.0559	4.505	0.0741	0.0485	0.0523
0.0001	**0**	**True**	0.0642	4.550	0.0624	0.0295	0.0388
0.0001	**0.005**	**False**	0.0393	4.126	0.0346	**0.0244**	0.0317
0.0001	**0.005**	**True**	0.0451	4.061	0.0603	0.0342	0.0428
Resnet34							
Train Scratch		**Exp_1**	5.838	16.87	5.5435		
		Exp_2	2.37	9.82	2.28		
		Exp_3	2.2955	9.6970	2.1136	1.4375	

Xvector

Fine tuned

- Experiment 1 I am using data augmentation adding different types of background noise and speed perturbation and experimenting with different hyperparameter.

Resnet34

Fine tuned

- Experiment 2 changed the evaluation of validation set to silhouette score.
- Experiment 3 changed the loss from cross entropy to triplet loss.
- Experiment 4 used both the dataset megdap and shaip for training.x
- Experiment 5 same as 4 but here using adapters
- Experiment 6 I am using data augmentation adding different types of background noise.
- Experiment 7 I am using data augmentation adding different types of background noise and speed perturbation

Train Scratch

- Experiment 2 changed the evaluation of validation set to silhouette.
- Experiment 3 changed the evaluation of validation set to EER and used both the dataset megdap and shaip for training.

Path to code -

My gcp instance - 35.200.234.154

Dataset Used for fine tuning
/data/Root_content/Vaani/Speaker_ID/Dataset

Experiments on pyannote_xvector model
/data/Root_content/Vaani/Speaker_ID/pyannote_xvector

Experiments on Wespeaker_resnet34 model
/data/Root_content/Vaani/Speaker_ID/Wespeaker