Step 1: Preparing the LRS2 Dataset

The first thing I did was extract the LRS2 dataset from its original `.rar` archive. Since I wanted to work with a manageable portion for training and later for validation, I flattened part of the structure (30 GB of data used for training and an additional 12 GB for validation). This involved renaming files and folders to create **unique identifiers**, avoiding naming collisions later in the pipeline — especially during caching and alignment.

This organizational step was essential to streamline the rest of the preprocessing process.

Step 2: Audio Extraction

After organizing the dataset, I needed to extract audio from the `.mp4` video clips so I could generate phoneme-level alignments later.

To do this, I wrote a Python script using `ffmpeg` to convert every video file into a `.wav` file. I made sure the audio was extracted in **mono** and resampled to **16 kHz**, which is the standard requirement for most alignment tools like Montreal Forced Aligner (MFA).

For every `.mp4`, this gave me a corresponding `.wav` file, and I matched each one with its associated `.txt` transcript from the dataset — making sure the filenames aligned correctly across modalities.

This gave me a full set of audio–transcript pairs, which served as input to the alignment step.

Step 3: Phoneme Alignment Using Montreal Forced Aligner (MFA)

Once I had all the `.wav` audio files and their corresponding `.txt` transcripts, I moved on to generating precise phoneme-level alignments using **Montreal Forced Aligner (MFA)**.

I first installed MFA in a separate conda environment to avoid any conflicts with my main lipreading project. I used one of MFA's pretrained acoustic models — in this case, the **English MFA model** — along with an English **pronunciation dictionary** (`english.dict`). This dictionary is crucial because it maps each word in the transcript to its corresponding phoneme sequence (e.g., `"cat"` → `["K", "AE", "T"]`).

Here's how MFA works in practice:

- It takes the `.wav` file (spoken audio),
- The corresponding `.txt` transcript (what was said),
- And uses the dictionary to break the sentence into phonemes,

- Then aligns those phonemes **in time** with the audio using acoustic modeling (based on HMMs and GMMs under the hood).

After running MFA in batch mode, I obtained `.TextGrid` files — one per clip. Each `.TextGrid` file contains precise **timestamps for every phoneme** in the audio. This gives me a rich, time-aligned phoneme transcription for each video, down to the millisecond.

These `.TextGrid` files were the core of the dataset's linguistic annotation.

Step 4: Extracting Phoneme Sequences

After generating the `.TextGrid` files with MFA, I needed to extract the phoneme sequences into a simpler and more accessible format for later use in training.

To do this, I wrote a script that parses each `.TextGrid` file and extracts only the **phoneme tiers**, ignoring word-level or other metadata. For each phoneme, I extracted its **label**, **start time**, and **end time**, and saved this sequence into a new `.phn` file.

So, each clip now had a `.phn` file that looked like this:

SIL 0.00 0.12

B 0.12 0.19

AE 0.19 0.31

T 0.31 0.45

SIL 0.45 0.62

This step allowed me to treat phonemes as a **time-aligned label sequence** that I could later associate directly with video frames — a key part of building a phoneme-level lipreading model.

Step 5: Video Frame Extraction

Once I had the phoneme timelines, I needed to process the visual data — specifically, extracting frames from each `.mp4` video.

I wrote a script using OpenCV's `cv2.VideoCapture()` to extract video frames at a fixed rate — typically **25 frames per second**. This means that for every second of video, I obtained 25 evenly spaced frames. These were saved as `.jpg` images, named sequentially (e.g., `frame_000.jpg`, `frame_001.jpg`, etc.).

Each set of frames was stored in a separate folder corresponding to the video clip it came from. I made sure the naming matched the phoneme data so that everything remained aligned.

This step gave me the raw visual input sequences needed for training, with **precise temporal alignment** to the phoneme data from MFA.

Step 6: Cropping the Mouth Region (Mouth ROI)

With all the video frames extracted, I needed to isolate the region of interest — the **mouth area** — since it's the most informative part for phoneme-level lipreading.

To do this, I used **Dlib's facial landmark detector**, which returns 68 key facial points. I focused on points **48 to 67**, which correspond specifically to the mouth region (both outer and inner lips).

For each frame:

- I detected the face,
- Located the mouth landmarks,
- Computed the bounding box around the mouth,
- Then cropped the image to just that region.

To keep things uniform, I resized each mouth crop to a fixed resolution — usually **112×112** — ensuring consistent input dimensions across the dataset.

This process significantly reduced noise in the input data, focusing the model's attention on the area that actually contains lip motion relevant to speech.

Step 7: Matching Frames to Phonemes

Now that I had both the visual frames and the phoneme time alignments, I needed to **associate each video frame with the phoneme being spoken at that exact moment**.

Each frame corresponds to a timestamp, which I calculated based on the frame index and the known frame rate (25 FPS → frame 10 = 0.40 seconds). I then used the `.phn` data to determine which phoneme was active at that timestamp.

For example:

- If a frame timestamp was 0.28 seconds, and the `.phn` file said that `AE` spanned from 0.19 to 0.31 seconds, then this frame was labeled with the phoneme `AE`.

This matching process was repeated for all frames in each video, and I stored the aligned data into a `.json` file containing:

json

CopierModifier

```
{
  "frames": ["frame_000.jpg", "frame_001.jpg", ...],
  "phonemes": ["SIL", "B", "AE", "T", "SIL"]
}
```

These `.json` files served as the **final form of my aligned and annotated dataset**, ready to be loaded and cached for model training.

Step 8: Caching Aligned Sequences

Once I had the aligned `.json` files linking each sequence of video frames to its corresponding phoneme labels, I realized that loading and processing this data on-the-fly during training was too slow — especially with frame loading, tensor conversion, and phoneme mapping happening for each sample.

To solve this, I implemented a **caching mechanism**. I wrote a script that loops through each `.json` file, loads the image frames into tensors, maps phonemes to their index representations using the vocabulary, computes input/output lengths, and saves everything as a single `.pt` file using `torch.save`.

Each cached file contains:

- A tensor of frames: `(T, C, H, W)`
- A tensor of phoneme indices: `(L,)`
- The input and target lengths (for CTC loss)

This step reduced training time significantly, especially when using a CPU, and made the evaluation process much faster and more consistent.

I applied the same caching method for both my training and validation sets, allowing both models to work efficiently on the exact same structured data.