We explore three approaches for keyword spotting leveraging OpenAI’s Whisper model as the core architecture:

1. **Full Fine-Tuning of Whisper-Small on ATC Dataset:**

In this approach, we fine-tune all parameters of the Whisper-small model using a domain-specific speech dataset composed of air traffic control (ATC) communications. The goal is to specialize the model for transcription tasks in high-noise environments with aviation-specific terminology and speaker variability. The training process involves preprocessing the audio to a 16 kHz sample rate, converting it into log-Mel spectrogram features, and updating the full model weights through supervised learning on transcribed ATC conversations. This allows the model to learn domain-adapted acoustic and linguistic patterns, improving both transcription accuracy and downstream keyword spotting performance. This approach can be similarly expanded to other domain specific datasets catering to military applications.

2. **Fine-Tuning with LoRA for Hindi Dataset (Parameter-Efficient Training)**:

To address resource constraints and support low-resource languages, we adopt a parameter-efficient tuning technique using Low-Rank Adaptation (LoRA). In this method, we freeze the majority of Whisper’s weights and introduce a small number of trainable rank-decomposition matrices into selected attention layers (e.g., query and value projections). Only these additional parameters—amounting to roughly 1% of the full model—are updated during training on a Hindi-language dataset. This significantly reduces memory usage and training time while allowing the model to adapt effectively to the linguistic characteristics of Hindi. The resulting model remains lightweight yet capable of accurate transcription and keyword detection in Hindi speech.

3. **Whisper Baseline (Transcription → Translation → Keyword Spotting)**:

In this approach, we use the pre-trained Whisper model directly, without any task-specific fine-tuning. The model is first used to transcribe audio input in its native language. If the audio is in a non-English language (e.g., Hindi), we translate the transcription into English. Keyword spotting is then performed on the resulting English text using simple string-matching techniques. This pipeline showcases ability to handle multilingual transcription and translation in a plug-and-play fashion, making it especially useful for rapid prototyping or low-resource scenarios.

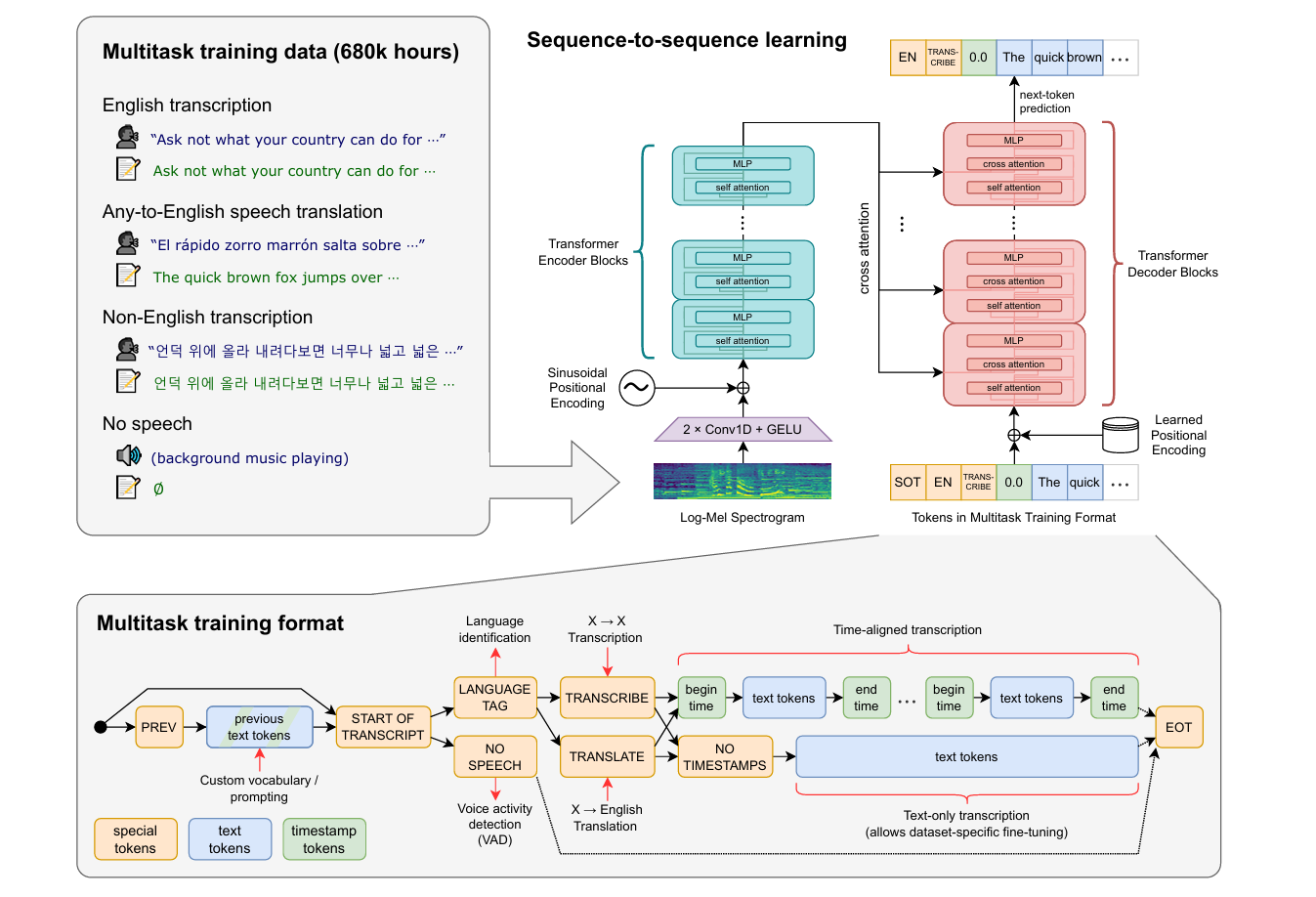
**WHISPER:**

Whisper is a powerful general-purpose speech recognition model developed by OpenAI and released in 2022. Whisper was trained on a large and diverse set of multilingual and multitask data collected from the web. Moreover, Whisper supports speech-to-text tasks in nearly 100 languages and can also perform speech translation into English, making it one of the most versatile open-source ASR systems available.

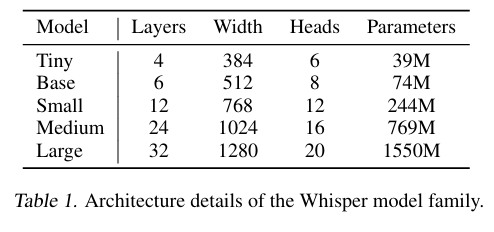
Whisper is based on a sequence-to-sequence Transformer architecture. The model comprises two main components: an encoder and a decoder. The encoder is responsible for processing the audio input, which is first converted into a sequence of log-Mel spectrogram features. These features are passed through multiple layers of Transformer blocks consisting of multi-head self-attention and feedforward layers. The encoder outputs a high-dimensional representation of the audio input, capturing both local and global acoustic patterns. The decoder, on the other hand, is autoregressive and generates the transcription (or translation) token by token. It uses causal (left-to-right) self-attention to condition on previously generated tokens and cross-attention to incorporate information from the encoder outputs. The decoder operates over a fixed vocabulary of around 50,000 tokens that include characters, timestamps, language identifiers, and special markers like "start of transcript".

Before being fed into the model, audio undergoes a carefully designed preprocessing pipeline. The audio files are resampled to 16 kHz and transformed into an 80-channel log-Mel spectrogram on overlapping windows (25 ms in length with a 10 ms stride). Whisper processes audio in 30-second chunks, which allows the model to handle long-form audio recordings in a computationally efficient way.

Whisper was trained using 680,000 hours of supervised audio-transcription pairs, making it one of the largest ASR training sets ever constructed. Of this, approximately 117,000 hours are in English, while the remaining 563,000 hours consist of audio in other languages paired with their English translations. This not only teaches the model to transcribe but also to translate speech to English. The model was trained using a cross-entropy loss objective to maximize the likelihood of correct token sequences.

****

**Figure 1: Overview of the Whisper model architecture**

****

**Table 1: Whisper model sizes**

***Approach 1: Fine-Tuning Whisper for ATCO Speech Recognition***

# 1. Introduction

This report outlines the process of fine-tuning OpenAI's Whisper small model on an ATCO (Air Traffic Control Operators) speech dataset to improve transcription performance. The Whisper model, originally trained on a large multilingual and multitask corpus, is adapted for a domain-specific English transcription task.

# 2. Dataset

Name: ATCO Speech Corpus  
Format: Parquet

Link: [jacktol/atc-dataset · Datasets at Hugging Face](https://huggingface.co/datasets/jacktol/atc-dataset)  
Accessed from local directories using Hugging Face’s load\_dataset.

Each sample contains:  
- audio: waveform array and sampling rate  
- text: transcription

# 3. Methodology

**3.1 Preprocessing**

To ensure consistency and compatibility with the Whisper model, all input audio is downsampled to a fixed sampling rate of 16 kHz. This is achieved using the resample function from the scipy.signal library, which interpolates the waveform to the appropriate number of samples based on the original and target sampling rates. Once resampled, the audio is passed through the WhisperFeatureExtractor from Hugging Face, which converts the waveform into log-Mel spectrogram features. These features serve as the input to the Whisper model and align with the format used during its original training.

The corresponding textual transcriptions are tokenized using the WhisperTokenizer. The tokenizer encodes the text into a sequence of token IDs, suitable for Whisper’s decoder input. Padding and truncation are applied to maintain a consistent maximum length of 60 tokens. Additionally, attention masks are used to ignore padded positions during loss computation, and non-attended tokens are masked using the value -100 to ensure they are not considered in the loss calculation.

**3.2 Dataset Handling**

To facilitate efficient training, a custom PyTorch Dataset class is implemented. This class wraps around the Hugging Face dataset and performs preprocessing on-the-fly, including audio resampling, feature extraction, and tokenization of labels. It returns a dictionary with two elements: input\_features, which are the model-ready spectrograms, and labels, which are the masked token IDs representing the ground truth transcription. A PyTorch DataLoader is used to create mini-batches of size 8 and shuffle them between epochs to ensure robustness during training.

**3.3 Model Training**

The base model used for fine-tuning is the openai/whisper-small variant, which is loaded from the Hugging Face model hub. Training is conducted on a CUDA-enabled GPU for performance optimization. The optimizer used is AdamW with a learning rate of 1e-5, which is suitable for transformer-based models and prevents overfitting through weight decay. Training is conducted over five epochs, with each epoch comprising a forward pass through the model to compute predictions and the associated loss, followed by a backward pass to compute gradients and an optimization step to update model weights. Gradients are reset after each update using optimizer.zero\_grad().

The model uses teacher forcing during training, where the ground truth token sequence is provided to the decoder. The decoder is initialized using the model’s decoder\_start\_token\_id, which corresponds to the beginning-of-sequence token. The final output of the model includes a loss value that is backpropagated to update model

- Base Model: openai/whisper-small  
- Optimizer: AdamW with learning rate 1e-5  
- Epochs: 5  
- Batch Size: 8  
- Loss: CTC loss (inherent in Whisper)  
- Metric: Word Error Rate (WER)

**2.4 Evaluation**

Model performance is evaluated using the Word Error Rate (WER) metric, both before and after fine-tuning. For evaluation, the model is switched to inference mode to disable dropout and other training-specific behaviors. Input audio is preprocessed in the same manner as during training, and predictions are generated using the model’s generate method. The output token IDs are then decoded into text using the tokenizer. WER is calculated by comparing the predicted transcription with the ground truth using the jiwer and evaluate libraries.

Evaluation is conducted on a set of 200 validation samples before training to establish a baseline, and then repeated after each training epoch to monitor progress. The trend in WER is visualized using matplotlib, providing insights into the learning trajectory of the model. A final evaluation is performed on the same validation set after training is complete, revealing significant improvements in transcription accuracy.

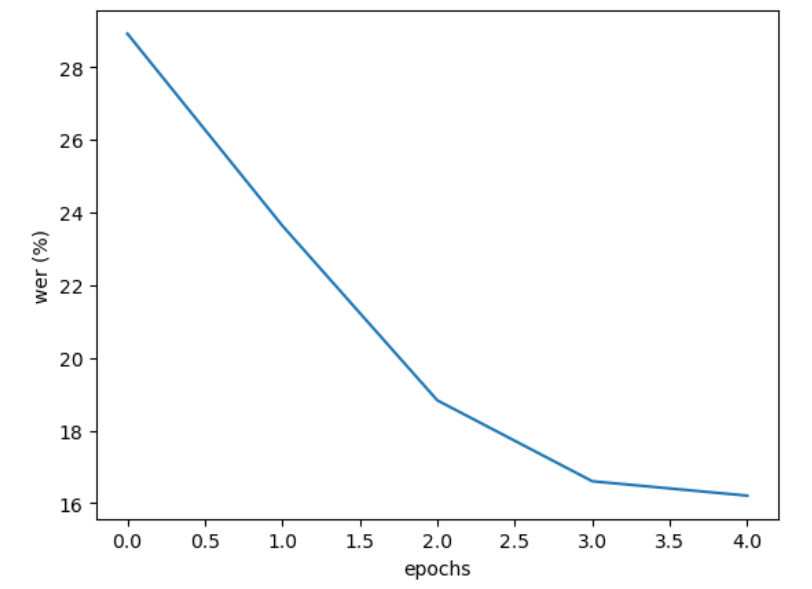
# 4. Results

## 4.1 Before Fine-Tuning

Average WER on 200 Validation Samples: 1.2165

## 4.2 During Fine-Tuning

WER reduced steadily across 5 epochs, plotted using matplotlib.



## 4.3 After Fine-Tuning

Final Average WER on 200 Validation Samples: 0.2837

**WER Comparison Table:**

|  |  |
| --- | --- |
| Evaluation Metric | WER |
| Before Fine-Tuning | 1.2165 |
| After Fine-Tuning | 0.2837 |

# 5. Conclusion

Fine-tuning Whisper on the ATCO speech dataset significantly reduced the Word Error Rate. This highlights Whisper’s ability to adapt to domain-specific language and acoustic environments, making it a practical tool for real-world air traffic communication transcription tasks.

***Approach 2: Fine-Tuning with LoRA for Hindi Dataset (Parameter-Efficient Training)***

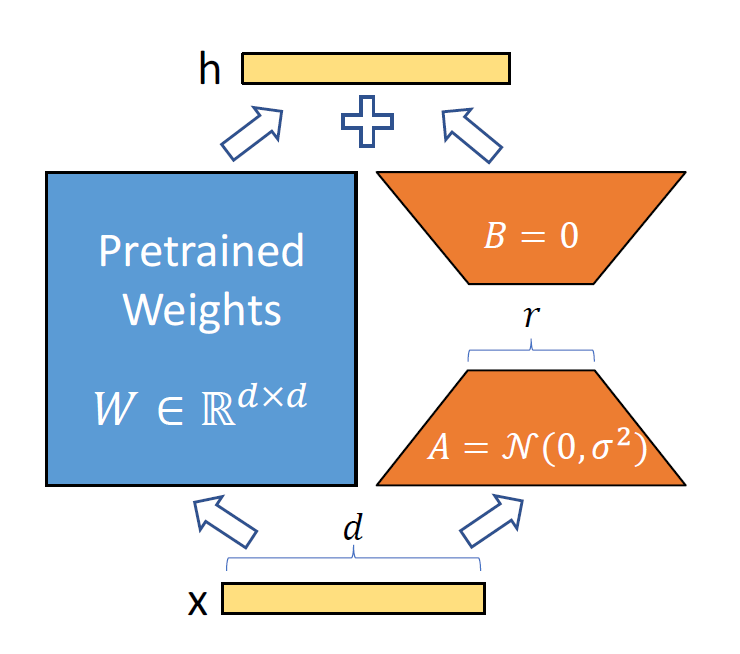
**Background:**

**What is PEFT?**

[1] Full fine-tuning of transformer-based Pretrained Language Models (PLMs) involves training the entire model, including all layers and parameters, on a specific downstream task using task-specific data. Initially, PLMs are trained on large-scale datasets with unsupervised learning objectives like language modeling or masked language modeling, to learn general language representations. However, these PLMs may not perform optimally when applied to specific tasks due to a lack of appropriate domain knowledge or lesser application specific training. Full fine-tuning provides an effective solution to address this limitation. During full fine-tuning, the PLM is initialized with pretrained weights and subsequently trained on task-specific data using techniques like backpropagation and gradient descent. All model parameters, including pretrained weights, are updated to minimize a task-specific loss that quantifies the disparity between predicted outputs and ground truth. In this way, full fine-tuning enables the model to learn task specific patterns and nuances from the labelled data, facilitating predictions or outputs tailored to the target tasks. Notably, full fine-tuning necessitates substantial computational resources and labelled data, as the model is trained from scratch for the specific target task. Moreover, as PLMs grow in size and with the advent of LLMs containing billions of parameters, full fine-tuning places even greater demands on computational resources. In contrast, PEFT methods aim to alleviate these requirements by selectively updating or modifying specific parts of the PLMs while still achieving performance comparable to full fine-tuning. Furthermore, full fine-tuning may give rise to overfitting when the task-specific dataset is small or when the PLMs are already well-suited to the target task.

Parameter-Efficient Fine-Tuning (PEFT) [1] helps mitigate the problem of catastrophic forgetting by preserving the pre-trained model’s original knowledge while enabling task-specific learning. In traditional fine-tuning, updating all the model's parameters for a new task often leads to overwriting the weights important for previous tasks, causing the model to forget earlier knowledge—a phenomenon known as catastrophic forgetting. PEFT addresses this by freezing the base model and introducing a small number of trainable parameters, such as low-rank adapters or prompt embeddings, specific to each task. This separation ensures that learning new tasks does not interfere with previously learned ones, making PEFT particularly effective in fine-tuning tasks. Thus, PEFT is a class of techniques designed to adapt large pre-trained models to downstream tasks without updating all of their parameters. Also, traditional fine-tuning methods require significant computational resources and memory, as they involve adjusting the entire model. In contrast, PEFT methods introduce a small number of additional parameters—such as low-rank adapters, prompts, or side modules—while keeping the vast majority of the model frozen. This approach greatly reduces training costs, enables fast adaptation to new tasks, and allows for efficient deployment, especially in low-resource settings. Techniques like LoRA (Low-Rank Adaptation), Prefix Tuning, and Adapters exemplify the PEFT paradigm and have demonstrated competitive or superior performance compared to full fine-tuning in many NLP applications.

**LoRA (Low Rank Adaptation):**



***Figure : Low-Rank Adaptation (LoRA) of Pretrained Weights via Randomized Parameter Injection [2]***

Low-Rank Adaptation (LoRA) [2] is an efficient fine-tuning technique designed to adapt large pretrained models using significantly fewer parameters. Instead of updating all the weights of a model, LoRA introduces low-rank trainable matrices into existing layers while keeping the original pretrained weights frozen. This is achieved by decomposing the weight update into two smaller matrices — one initialized from a normal distribution and the other often set to zero — and applying their product as a low-rank update to the original weights. By doing so, LoRA dramatically reduces memory and computational requirements during training, making it highly suitable for fine-tuning large models like Whisper on domain-specific tasks without needing to retrain or store the full model.

A neural network contains many dense layers which perform matrix multiplication. The weight matrices in these layers typically have full-rank. When adapting to a specific task, the pre-trained language models have a low “intrinsic dimension” and can still learn efficiently despite a random projection to a smaller subspace. Inspired by this, it is hypothesized that the updates to the weights also have a low “intrinsic rank” during adaptation.

Instead of updating the large pre-trained weights W ∈ ℝ^{d×d}, LoRA **freezes W** and introduces **two low-rank trainable matrices**, A and B, such that:

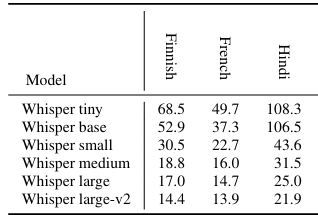
ΔW=BA

So, during training, only A and B are updated while W remains fixed.

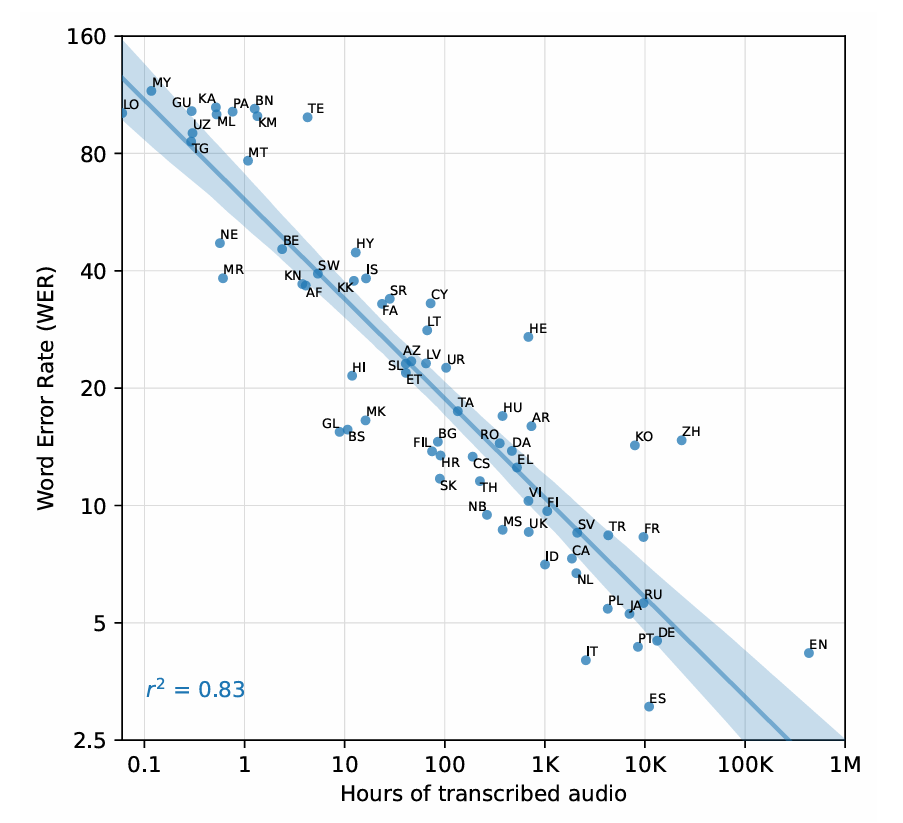
* **Pretrained Weights:**
  + W ∈ ℝ^{d×d} is the original weight matrix of a large model
  + It is not trained during LoRA fine-tuning — it is frozen.
* **Input and Output:**
  + x: Input vector to the layer of size d.
  + h: Output after processing via LoRA (added on top of the normal output).
* **Trainable Low-Rank Matrices:**
  + A ∈ ℝ^{r×d}: Randomly initialized (normal distribution), trainable matrix.
  + B ∈ ℝ^{d×r}: Initialized as zero, also trainable.
  + These two represent a low-rank decomposition with rank r (r << d).
* **Computation Path:**
  + Input x is passed through A → produces a lower-dimensional representation (r).
  + That is then passed through B → maps it back to the original dimension (d).
  + This output BAx is added to the output from the frozen W (i.e., Wx) as a residual connection:

h=Wx+BAx

Thus, LoRA (Low-Rank Adaptation) is particularly well-suited for fine-tuning large-scale models like Whisper-large (with 1.55 billion parameters) to improve performance on application-specific tasks such as transcription in a particular language or dialect. While Whisper-large is trained on a diverse, multilingual dataset, its generalized training may not fully capture the nuances, pronunciation patterns, or linguistic variations of a specific language—especially if it is underrepresented in the pretraining corpus. By fine-tuning with LoRA, the model can learn these language-specific patterns more effectively. LoRA introduces additional trainable low-rank matrices into select layers while keeping the pretrained weights fixed, allowing the model to adapt to new linguistic contexts without forgetting its foundational knowledge. This targeted adaptation not only enhances transcription accuracy in the target language but also preserves the robustness and fluency inherited from the large-scale pretraining. Thus, LoRA serves as a powerful mechanism to specialize Whisper-large for high-performance, language-specific speech recognition.



**Table 2: WER on common voice 9 dataset**

****



**Figure 2: Whisper originally trained on 12 hours of Hindi audio**

**Experimental setup:**

We are using common voice 21 (Hindi dataset) for finetuning using LoRA.

Train set: 9.38 hours

Test set: 4.73 hours

Train samples: 7563

Test samples: 3337

**WER before finetuning:**

WER on Train Set: 68.92%

WER on Test Set: 71.67%

**Steps: 100**

**Trainable parameters:**

trainable params: 15,728,640 || all params: 1,559,033,600 || trainable%: 1.0089

* Trainable params:15,728,640  
  These are the parameters being updated during training—introduced by LoRA (e.g., low-rank adapters in attention layers).
* All params: 1,559,033,600  
  This is the total number of parameters in the model, including both the frozen base model and any fine-tuned layers.
* Trainable %: 1.0089%  
  This shows that only ~1% of the model's total parameters are being fine-tuned, thanks to LoRA. The rest remain frozen.

**LORA Config:**

r=32: sets the **rank** of the LoRA adapter matrices A and B.

lora\_alpha=64: scaling factor for LoRA output.

target\_modules=["q\_proj", "v\_proj"]: injecting LoRA **only into specific submodules** of the model (likely self-attention components).  
bias="none": don't adapt bias terms.

lora\_dropout=0.05: adds dropout during adapter training.

**Methodology (explanation of code):**

**1. Model Setup for Hindi Speech-to-Text with Whisper**

* Loads OpenAI Whisper large model for Hindi ASR.
* Initializes tokenizer, feature extractor, and model.
* Sets up the GPU (cuda) for inference.

**2. Whisper Processor Initialization**

* Combines tokenizer and feature extractor via WhisperProcessor.

**3. Dataset Processing (train+dev, test)**

* Merges train and dev sets.
* Converts to DatasetDict.
* Keeps only audio and sentence columns.

**4. Audio Resampling**

* Ensures all audio samples are 16kHz using torchaudio.

**5. Baseline Evaluation**

* Computes WER on train and test sets before fine-tuning.

**6. Transcription Examples**

* Shows actual vs predicted transcriptions for 10 test samples.

**7. Feature Extraction and Tokenization**

* Applies log-mel spectrograms + transcription tokenization.

**8. LoRA Fine-Tuning (Parameter-Efficient)**

|  |
| --- |
| from peft import LoraConfig, get\_peft\_model  config = LoraConfig(r=32, lora\_alpha=64, target\_modules=["q\_proj", "v\_proj"], lora\_dropout=0.05, bias="none")  model = get\_peft\_model(model, config)  model.print\_trainable\_parameters() |

**9. Training**

* Saves LoRA adapter weights separately (lightweight).

**10. Evaluation and Inference**

* Loads base Whisper model + LoRA adapters.
* Uses 8-bit mode for efficiency.
* Caches enabled for fast inference.
* Saves combined model and processor.

**11. Final Evaluation**

* Saves full final model.
* Computes post-training WER again.

**Results:**

| **Step** | **Training Loss** | **Validation Loss** |
| --- | --- | --- |
| 100 | 0.460100 | 0.316768 |

Train WER: 43.76%

Test WER: 47.22%

**References:**

1. Xu, Lingling, et al. "Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment." *arXiv preprint arXiv:2312.12148* (2023).
2. Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *ICLR* 1.2 (2022): 3.
3. Radford, Alec, et al. "Robust speech recognition via large-scale weak supervision." *International conference on machine learning*. PMLR, 2023.

***Approach 3: Whisper Baseline (Transcription → Translation → Keyword Spotting):***

***Introduction***

This approach aims to perform automatic keyword spotting in Hindi speech using OpenAI's Whisper ASR (Automatic Speech Recognition) model. The system detects the presence of specific English keywords from audio clips containing Hindi speech. Since Whisper outputs transcriptions in native script (Devanagari), the pipeline includes transliteration to Latin script followed by hybrid keyword matching using both fuzzy string similarity and phonetic encoding.

**Methodology**

## Dataset

The dataset consists of `.wav` audio files located in:  
C:/Users/WORKSTATIONS/Desktop/BijoyashreeDas/storm\_vad-20250602T113932Z-1-001/storm\_vad  
Each file is assumed to be a spoken utterance in Hindi, sampled at 16 kHz, containing operational or conversational terms.

## Whisper Model Setup

Model Used: openai/whisper-large-v3  
- The Whisper model is loaded with WhisperProcessor and WhisperForConditionalGeneration.  
- Configured for Hindi transcription using forced decoder IDs.  
- Runs on GPU if available.

## Transcription

Audio is loaded and checked for correct sampling rate (16 kHz). Whisper transcribes the Hindi speech into text in Devanagari script.

## Transliteration

A custom transliteration step uses the indicate.transliterate library to convert Devanagari words into Latin script (English phonetic equivalents). Only Devanagari words are transliterated.

## Hybrid Keyword Matching

A predefined keyword set includes English operational terms such as:  
{"report", "calling", "over", "guide", "army", "commandant", "wilko", "point", "checking", "vehicle", "namaste"}  
  
Matching is done via:  
- Fuzzy String Similarity (RapidFuzz): Matches if score ≥ 85%  
- Phonetic Encoding (Double Metaphone): Matches if phonetic representations match  
  
The matcher outputs: keyword ~ matched\_word (score%, match\_type)

# **Results**

Output File: hindi\_transcriptions.txt  
Each entry in the output file includes:  
- Audio file name  
- Hindi transcription  
- Latin transliteration  
- Matched keywords (with similarity scores and type of match)  
- Errors if any occur during processing  
  
Example Output Entry:  
🎧 File: clip001.wav  
📝 Hindi: कॉलिंग प्वाइंट पर गाड़ी रिपोर्ट कर रही है  
🔤 Latin: calling point par gaadi report kar rahi hai  
🔎 Keywords found (≥85% similarity or phonetic match): calling ~ calling (100%, string), point ~ point (100%, string), report ~ report (100%, string)

# 5. Conclusion

This pipeline showcases an effective methodology for detecting English keywords in Hindi speech by integrating Whisper transcription, script transliteration, and hybrid matching techniques.