TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

A Major Project Mid Term Report
On
A Dual Transformer Pipeline with Conformer-Based Speech Recognition and Decoder-Only Language Modeling

Submitted By:

Sangam Parajuli	(THA078BCT039)
Saurav Katwal	(THA078BCT040)
Shaswot Poudyal	(THA078BCT041)
Shreejay Shakya	(THA078BCT042)

Submitted To:
Department of Electronics and Computer Engineering
Institute Of Engineering, Thapathali Campus
Kathmandu, Nepal

February 2026

TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

A Major Project Mid Term Report
On
A Dual Transformer Pipeline with Conformer-Based Speech Recognition and Decoder-Only Language Modeling

Submitted By:

Sangam Parajuli	(THA078BCT039)
Saurav Katwal	(THA078BCT040)
Shaswot Poudyal	(THA078BCT041)
Shreejay Shakya	(THA078BCT042)

Submitted To:
Department of Electronics and Computer Engineering
Thapathali Campus
Kathmandu, Nepal

In partial fulfillment for the award of the Bachelor's Degree of Engineering in Electronics, Communication and Information.

Under the Supervision of
Er. Ganesh Kumal

February 2026

ACKNOWLEDGMENT

We extend our heartfelt appreciation to the **Department of Electronics and Computer Engineering at Thapathali Campus, Institute of Engineering** for providing us with the invaluable opportunity to engage in learning, research, and the realization of our insights through a major project venture.

With great pleasure, we would like to express our sincere gratitude to our supervisor **Er. Ganesh Kumal** for providing us with unwavering support, invaluable guidance and insightful suggestions. Furthermore, we would like to also extend our appreciation to every individual for their valuable guidance, support, contributions and insightful feedback.

We are also grateful to the authors of various papers that we have referenced for our project. We respect all the researchers' invaluable time being used in the research and writing of such articles.

Additionally, we would like to thank all of our teachers, friends and other direct and indirect contributors for their invaluable help, support and encouragement during our study of this project.

Sangam Parajuli (THA078BCT039)
Saurav Katwal (THA078BCT040)
Shaswot Poudyal (THA078BCT041)
Shreejay Shakya (THA078BCT042)

ABSTRACT

Modern speech LLM systems heavily rely on the integration of Automatic Speech Recognition (ASR) and Large Language Models (LLMs). However, most existing architectures adopt tightly coupled ASR-LLM pipelines which limits their flexibility, scalability, and modularity. This project proposes a modular approach that decouples ASR and LLM components allowing independent development, fine-tuning, and optimization. The ASR system is designed using a Conformer-based Transformer architecture enhanced with Residual Multi-Head Self-Attention (ResMHA) and progressive temporal downsampling for accurate and efficient speech transcription. The transcribed output is then processed by a lightweight, decoder-only Transformer LLM, which generates coherent and contextually relevant responses. Despite certain trade-offs like increased latency and lack of real-time interaction, the proposed architecture demonstrates high adaptability and performance in various applications such as AI assistants and domain-specific information systems. The project plans to demonstrate the potential of a decoupled architecture in advancing scalable and maintainable AI systems.

Keywords: Automatic Speech Recognition (ASR), Large Language Models (LLM), Residual Multi Head Attention (ResMHA), Conformer-based Transformer, Progressive downsampling

TABLE OF CONTENTS

List of Figures

List of Tables

LIST OF ABBREVIATIONS

ASR	Automatic Speech Recognition
BERT	Bidirectional Encoder Representations from Transformers
BPE	Byte-Pair Encoding
CER	Character Error Rate
CNN	Convolutional Neural Network
CTC	Connectionist Temporal Classification
DNN	Deep Neural Network
FFN	Feed Forward Network
FFT	Fast Fourier Transform
GPT	Generative Pre-trained Transformer
LLM	Large Language Model
LSTM	Long Short-Term Memory
ML	Machine Learning
NLP	Natural Language Processing
ResMHA	Residual Multi-Head Attention
RNN	Recurrent Neural Network
TTS	Text-To-Speech
WER	Word Error Rate

1 INTRODUCTION

1.1 Background

The application of voice-based interfaces range from transcription to customer service bots. Accurate speech-to-text conversion and understanding language in context aware manner are essential for implementation of these systems. However, building integrated ASR-LLM systems is complex, making it difficult to scale individual components for specific tasks.

Automatic Speech Recognition (ASR) and Large Language Models (LLMs) are the basic foundational technologies for modern AI used for conversational AI. While the whole system offers integrated performance, they often lack flexibility and are difficult to maintain. This project implements a different modular approach, decoupling ASR and LLM components to enhance scalability and independent optimization. The ASR system is designed using a Conformer-based Transformer architecture with Residual Multi-Head Attention (ResMHA) to improve the accuracy of transcripts and modeling of long-range contexts. The output is processed by a lightweight, custom LLM that makes sure of dialogue generation that is efficient and coherent. This design of the pipeline enables easier upgrades while also maintaining strong overall system performance.

1.2 Motivation

Most modern conversational AI systems are built with Automatic Speech Recognition (ASR) and Large Language Models (LLMs) as their core parts. Even so, most existing frameworks combine these parts in a close or full connection which makes them less flexible, scalable and harder to interpret. Errors in different sections of such models tend to spread without being caught and optimizing one section usually means retraining all parts of the model. This brings a lot of difficulties in saving time, handling new domains and ensuring the software is simple to manage. This makes it important for new and existing pipelines to be decoupled and composed of components that can be individually developed, refined and changed without affecting all the other components.

The main goal is to develop a more functional and expandable system for speech-to-dialogue by having each component run alone but still connected together. Because we separate these modules, we can improve just one area of the system such as transcription or speech, without the need for general retraining. Besides, such architecture allows specific changes to be added easily, so the system can work well in customer service, personal assistant and real-time captioning roles.

Our model uses progressive temporal downsampling which decreases the amount of effort and delay needed in real-time tasks and does not harm the quality of transcriptions. After that, the improved ASR system's output goes into a lightweight custom LLM meant for managing tokens and creating smooth dialogue. While Style-Talker is good at saving voice qualities and producing responsive answers, it gives up being modular and adaptable. It covers the trade-offs, however, it is most concerned with increasing flexibility and making the system work better rather than focusing on small improvements in how the AI speaks. Overall, the purpose of the pipeline is to show that a carefully planned, separated architecture is able to perform competitively and provide more control, clarity and expandability fitting well for continuous development in real situations.

1.3 Problem Definition

Modern Automatic Speech Recognition (ASR) and Large Language Model (LLM) systems face several practical challenges that limit their adaptability and accessibility. These systems often tightly couple the speech recognition and language generation components, making it difficult to optimize or replace individual modules independently. This lack of modularity hinders experimentation and innovation. This can be improved by a modular architecture consisting of independent ASR and Language model.

Most pre-trained models are built for general-purpose tasks and are not easily adaptable to domain-specific datasets. Fine-tuning them often involves complex retraining pipelines or full-system adjustments, which are time-consuming and demand deep technical expertise. Even minor changes frequently require retraining the entire system, increasing maintenance effort and slowing down iteration cycles. Having a modular architecture allows for easier fine tuning and customized models that are easier to train and implement.

1.4 Objectives

The objectives of our project are:

- To design a context aware conformer based ASR model enhanced with Residual Multi-Head Self Attention and progressive downsampling for accurate and efficient speech transcription.
- To build a lightweight decoder-only transformer language model that processes ASR generated text to produce coherent response in a modular pipeline.

1.5 Scope and Application

1.5.1 Applications

- **AI Assistant With Conversational Ability:** The pipeline supports virtual assistants that accurately understand speech and generate natural language responses, enabling personalized and scalable user interactions.
- **Domain-Specific Information Systems:** The modular design allows for easy fine-tuning and adaptation to specialized fields such as healthcare, legal, and finance, where precise speech understanding is critical.
- **Language Learning and Pronunciation Tools:** With accurate transcription and feedback capabilities, the system can be integrated into educational platforms to help learners improve speech and acquire new languages.

1.5.2 Limitations

- **Latency:** The modular nature of the system introduces cumulative latency across ASR and LLM stages.
- **Prosodic Inconsistency:** The decoupled ASR-to-LLM interface may result in unnatural prosody and less expressive responses, particularly when used in conjunction with downstream TTS modules.
- **Error Propagation:** Transcription error from the ASR stage directly affect the LLM's response quality due to lack of feed back loop or joint optimisation.
- **Resource Utilization:** While modular, the independent training of ASR and Language model component may lead to increased utlization of resources.
- **Not real-time:** Due to sequential pipeline and lack of tight integration between ASR and Language model, the system is not capable of real time interaction.

2 LITERATURE REVIEW

Spoken language processing has become a fundamental midpoint in that it has become the place where two potentially evolutionary technologies converge: high-accuracy Automatic Speech Recognition (ASR) systems and reasoning-powered Large Language Models (LLMs). This is the recent stage of a long history of development whereby primitive systems of speech transcription have evolved into systems able to not only understand but also participate in a conversational environment. On one hand, an increasing body of current research supports the idea of closely integrated, end-to-end Speech-LLMs [1], including, but not limited to the AudioGPT [2] and Style-Talker models [3]. These systems aim at deep multi-modal fusion, creating LLMs capable of being fed and processed with raw or latent acoustic representations. Its goal is to realize a unified cognitive agent existing along the spectrum between acoustic waveform and intelligent response and is therefore capable of preserving paralinguistic information, such as prosody, tone and emotion, which are encoded in the speech signal but are lost when the speech is used in text transcription. Its supporters believe that this level of integration can be optimized in common against a single goal, with the wide range of linguistic and world knowledge of the LLM able to inform and, possibly, correct the process of creating acoustic perception online in the synergistic sense. However, this paradigm also comes with a high cost to system design (architectural rigidity that hampers independent component upgrades, a vast need of matched audio-text dialogue pairs, data that is relatively rare when compared to mono-modal datasets), and high-level of computational overhead that makes deployment and scaling difficult, as the entire monolithic model needs to be trained, fine-tuned, and simply served as an inseparable unit.

The modular, decoupled pipeline, however, reflects a classical concept of systems engineering: separation of concerns. The methodological basis of the current work is this approach that forms spoken language systems as the sequential modular assembly. A special, high-performance ASR engine is a transducer device used specifically, which converts the acoustic speech signal to a discrete textual transcript. This transcript is then an interoperable, standardized interface to a separate, perhaps generic, LLM module that performs comprehension, reasoning, and response generation. The strength of this architecture is its built-in modularity and independence, which allows the ASR and LLM components to be researched, developed, trained on domain-specific data, optimized to run on specific hardware, and deployed at scale without the need to depend on each other. This encourages such a profound specialization; the ASR can be optimized exhaustively towards acoustic robustness, noise invariance, and speaker adaptation using large volumes of transcribed speech, whereas the LLM can be carefully

fine-tuned to domain-specific dialogue, safety, and style generation using the larger universe of text-based conversations and documents. As a result, the system architecture will achieve exceptional flexibility, maintainability, and scalability since the components can be changed, upgraded or even expanded across the horizontal depending on demand without requiring an entire retraining of the system. Despite the trade-offs inherent in this decoupled implementation, including a rise in latency of the pipeline, the possibility of error propagation where ASR transcription errors become unchangeable LLM input, and the sheer loss of non-textual acoustic information, this decoupled implementation strategy has many practical advantages. In voice-based assistants in limited-scope areas, audio and video stream content moderation, meeting transcription and summarization, and domain-specific query systems the advantages of robustness, simple programming, and scalability of operations tend to vastly outweigh the disadvantages.

The modern discussion of integration-related strategies is based on the groundbreaking neural architecture breakthroughs, whose primary force is the Transformer. Applying the initial Transformer architecture to ASR immediately faced a rather fundamental bottleneck: the computational complexity of the self-attention mechanism scales quadratically $O(n^2)$ with sequence length). Order of magnitude longer than text sequences are speech signals, which can be represented as a sequence of frames (such as 100 Hz); this makes the vanilla training and inference of Transformer prohibitively costly. This difficulty prompted more speech-specific variants to be developed in an efficient manner. The most effective of them is the Conformer (Convolution-Augmented Transformer) by Gulati et al. (2020) [4], which has become the default encoder in the state-of-the-art ASR. The cleverness of The Conformer is that the hybrid design is specifically touched upon in the dual nature of speech. A typical Conformer block is a series of feed forward, multi-headed self-attention improved from "Multi-Head Attention: Collaborate Instead of Concatenate" [5], [6], convolution, and feed forward modules. The self-attention element represents long-range, global dependencies and contextual relationships throughout an utterance, which is critical to solving semantic ambiguity, and, in parallel, the convolution module finds it easy to model narrow, high time-density correlations in the acoustic signal, viz phonetic patterns, spectral transitions. A combination of these complementary capabilities allows the Conformer to recognize speech as a signal with high local stationarity, best represented by convolutional neural networks, embedded in a hierarchical linguistic structure, best represented by attention mechanisms. This new invention in architecture has given the strong, efficient and very accurate speech encoding mandatory to present systems.

Parallel with this, the Transformer architecture similarly brought a parallel revolution

in natural language processing. Transformers trained in decoder only on a cause-and-effect language-modeling task (predicting the next token) were found to be able to scale in a way never seen before. Empirical validation of this came with GPT series (Generative Pre-trained Transformer), in which Radford et al. (2018, 2019) [7] scaled up such models, in both parameters and data, and found them to have impressive emergent capabilities in a variety of reasoning, instruction-following and in-context learning, with no task-specific creation of architectures. The quantitative roadmap provided by the empirical study of scaling laws by Kaplan et al. (2020) ([8]) provided an essential quantitative understanding of predictable power-law enhancesments in performance with model size, dataset size, and computing budget. This scaling thesis immediately gave rise to the age of LLMs like GPT-3, PaLM, and LLaMA [9], whose parametric knowledge and generative fluency are the basis of modern conversational AI. The presence of these two transformer-based powerful pillars, high-accuracy Conformer ASR and GPT-like LLM with its knowledge, naturally resulted in the current area of research interest in integrating them.

Before the development of transformer-based architectures, the first generation of deep learning used on automatic speech recognition (ASR) was not so much a game-changer as it was an improvement. Instead of a complete replacement of the statistical framework, deep learning presented a very successful component replacement into the already successful paradigm. This shift was formalised by the breakthrough study by Hinton et al. (2012) which showed that a hybrid deep neural network-hidden Markov model (DNN-HMM) architecture was effective. In this scheme a neural network, implementing the deep neural network, replaced the Gaussian mixture model (GMM) as the model of estimations of the posterior probabilities of HMM states (senones). Although the HMM still addressed temporal dynamics, and decoded sequence, the deep, hierarchical non-linear layers of the DNN network had had highly enhanced abilities to learn discriminative feature presentation directly off acoustic frames, and provided substantial gains in the word error rate. At the same time, the search of architectural simplification initiated an interest in the real end-to-end systems. Graves et al. (2006) proposed the Connectionist Temporal Classification (CTC) [10] algorithm which solved the fundamental temporal alignment problem between sequences of variable length of input and output sequences by introducing a blank word, and by applying a dynamic-programming path-collapsing algorithm. This allowed single neural networks usually recurrent networks like long short term memory (LSTM) networks to directly learn the acoustic features sequence-to-character/wordpiece sequence mapping, which simplifies the training pipeline and lets the field be more aligned to end-to-end learning principles.

The developments of these deep-learning replaced the statistical paradigm of long dom-

inance in ASR, which is the GMM-HMM paradigm. Having been theorized by Bahl, Jelinek, and Mercer (1983)[11], this paradigm redefined the speech recognition as a Bayesian inference problem. Bayes theorem was used to divide the task into an acoustic model, which is, P(A 0 W) and a language model, P(W). A GMM-HMM was used to implement the acoustic model with the hidden Markov models modeling sequence of context-dependent sub-word units (triphones), and the probability density of acoustic feature vectors (e.g. MFCCs) at each state being modeled by Gaussian mixture models. Typically, an n-gram model, the language model, offered prior knowledge of likelihoods of word-sequences. This statistical model, despite being an impressive engineering accomplishment and allowing its commercial use overall, did not achieve performance improvements indefinitely because the representational capacity of GMMs was limited, and HMMs relied on the assumption of conditional-independence.

The very idea of statistical era was the direct reaction to the inadequacy of the original rule-based approach. The acoustic-phonetic paradigm dominated in the 1960s and 1970s. It was influenced by linguistic and perceptual theories to build recognisers, and devised explicit rules and detectors that would scale invariant acoustic landmarks, usually formant frequencies and trajectories, to discrete phonemic categories. This bottom-up approach was not possible in the context of continuous speech due to the ubiquitous, context-dependent phenomenon of coarticulation. The articulatory gestures of successive sounds are similar leading to the acoustic realisation of a phoneme to differ dramatically with phonetic neighbours of that phoneme. This variability nullified the hypothesis of acoustic invariance and showed that the mapping of sound to symbol is highly contextualized and non-linear and so placing the field on probabilistic rather than deterministic methodologies.

Overall, the historical trend of ASR is one of failure, success, and improvement: at first, the unsuccessful results in handcrafted, rule-based detection, then, the successful results in engineered work of statistical decomposition, then, the component-level success obtained with the use of deep neural networks, and, lastly, the efficiency and architectural simplification of specialised transformers. Nowadays, the field of work is concerned with the strategic combination of these capabilities with the reasoning capabilities of the large language models. This undertaking falls in a decoupled, modular integration paradigm. We conjecture that in a wide range of practical applications that demand robustness, maintainability and scaleable implementation the principled separation between acoustic transduction and linguistic reasoning is a practical and better solution. We will present the actual application of a more advanced, production-scalable ASR module and a customized large language model, and thus embody the tangible performance and system-level benefits of this modular architectural ideology.

3 REQUIREMENT ANALYSIS

3.1 Project Requirements

3.1.1 Hardware requirements

Google Colab

It is a lightweight and inexpensive option to train our models and has access to high-performance GPUs including the NVIDIA T4 or P100, multi-core CPU and up to 12GB of RAM on the free plan. The precise hardware setup also depended on the availability of resources but the real-time management of hardware as well as the smooth connection with Google Drive made Colab a good option to take into consideration of our computational requirements. Colab allows us to access the free resources to do lightweight tasks that featured premium GPUs, longer runtimes, and larger memory allocations depending on the requirement. It also allows to suspend or cancel sessions when they were idle in order to optimize cost so that the available compute hours are not wasted.

3.1.2 Software requirements

PyTorch

PyTorch is a widely used machine learning library which was developed by Facebook's AI research lab (FAIR). During the building and testing of deep learning models, it shines due to its ease of use. It works well with python and can run effectively with both CPUs and GPUs. It is also equipped with powerful tools like TorchVision, TorchText, and TorchAudio, making it easier to work in different areas of Artificial Intelligence. We used PyTorch's dynamic computation graph and intuitive interface to rapidly prototype the ASR module's key components, including the Residual Multi-head Attention (ResMHA) mechanisms , Connectionist Temporal Classification (CTC) loss function. For the language model, PyTorch's native Transformer modules and attention masking capabilities can be used to build the decoder only model and attention masking capabilities.

Numpy

NumPy serves a foundational package for numerical computation in python, supplying different types of powerful data structures such as multi-dimensional arrays and matrices, along with a large number of mathematical methods to operate on the arrays effectively. It has helped the building process of various models by supporting the computational backbone for all low-level tensor operation and data preprocessing tasks. We have heavily relied on NumPy's optimized array operation for extracting features in our ASR system. It is heavily used when converting raw audio waveform into Mel Spectrograms through FFT and filter bank applications. During data augmentation, NumPy

enabled fast spectrogram augmentation like time warping, time masking and frequency masking using its slicing and broadcasting features.

Matplotlib

Matplotlib is a comprehensive plotting library in Python used for creating static, animated, and interactive visualizations. It provides a flexible and easy-to-use interface for generating a wide variety of plots, including line charts, bar graphs, histograms, scatter plots, and more. Matplotlib plays a crucial role in data exploration, model evaluation, and result interpretation by allowing users to visually analyze trends, patterns, and anomalies within their data. Matplotlib is used to visualize the spectrogram and its types and verifying and validating the spectrogram augmentation during the initial phase of dataset preparation. It integrates seamlessly with other Python libraries such as NumPy, Pandas, and SciPy, enabling efficient visualization of numerical data and statistical results. With its object-oriented API, Matplotlib supports both simple and highly customized visual outputs suitable for publication-quality figures or quick exploratory analysis ideal for project of our nature.

NeMo

NeMo is an open source toolkit created by NVIDIA to assemble, train, and fine tune state of the art conversational AI models. NeMo Fine-tuning a FastConformer model takes advantage of its modular ASR pipeline and, therefore, simplifies the steps in dataset preparation and configuration. It involves loading a pre-trained FastConformer checkpoint, changing the architecture or dataset parameters of a YAML configuration file, and running the training script. NeMo supports distributed training and mixed-precision arithmetic and allows popularising the model on new acoustic data with efficient optimisation.

3.2 Feasibility Analysis

3.2.1 Technical Feasibility

The project being investigated includes an Automatic Speech Recognition (ASR) module along with a small language model. It requires huge libraries and two sets of datasets and architectural paradigms to implement. The technical requirements are quite high, but the availability of modules and libraries already built into the PyTorch ecosystem, along with the free availability of datasets, makes the project technically achievable. In early experiments in which the ASR component was trained directly, the training and validation loss curves showed a consistent decreasing trend and ostensibly a convergence, which is considered internal optimization. The accuracy of validation also

showed steady increase, although to a small degree. However, Word Error Rate (WER) was consistently higher than 95% an error rate that is significantly poorer than the 5-10% WERs that might be seen with state-of-the-art models on the LibriSpeech corpus. This difference may indicate that despite minimizing the loss, the model is not generalizing to the correct transcription work. Therefore, it was considered to be unfeasible to continue with full-scale training of the ASR module considering the existing limitations of the computational capabilities and development schedule. We therefore decided to optimize an already trained ASR model, thus using the already developed linguistic and acoustic features to achieve a better performance by requiring significantly lower data and computational requirements.

Figure 3-1: Validation WER Graph

3.2.2 Economic Feasibility

The major cost includes paid online computing service such as Colab pro for training our custom conformer model and SLM. Other costs include printing and for other softwares and tools. Given the scale of the model and our dataset availability, the project is economic feasible in accordance to our proposed budget.

3.2.3 Schedule Feasibility

This project can be chunked down to phases starting from audio data pre-processing, training our custom conformer model to training our SLM. Due to our system being a modular independent pipelines, the training can be performed independently and inte-

grated together allowing us with sufficient time to divide tasks. This allows us to ensure that milestones are achievable given the academic calendar schedule.

4 SYSTEM ARCHITECTURE AND METHODOLOGY

4.1 Proposed System Architecture

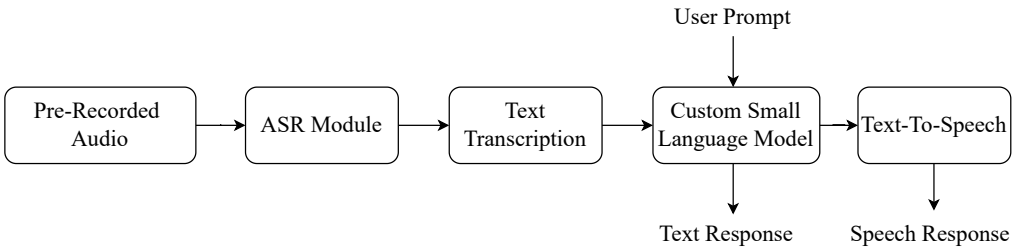

Figure 4-1: Proposed System Architecture

Our system pipeline begins with audio input which is user's speech. The input should be in English language. The audio is then passed to ASR module, which performs feature extraction (Mel Spectrograms), then it used our neural network ie Conformer to convert the audio into text transcription. Once the spoken input is transcribed completely, then the text is passed to a Small Language Model (SLM). The SLM is fine tuned to customized dataset optimised for local inference and response generation. The SLM interprets the meaning, context and intent behind transcribed input and then generates coherent, text response. Finally, the response is returned to user using Text-To-Speech (TTS) module in accordance to user input prompt. Text response is also directly available as a output form.

4.2 Automatic Speech Recognition

4.2.1 Proposed Architecture of ASR

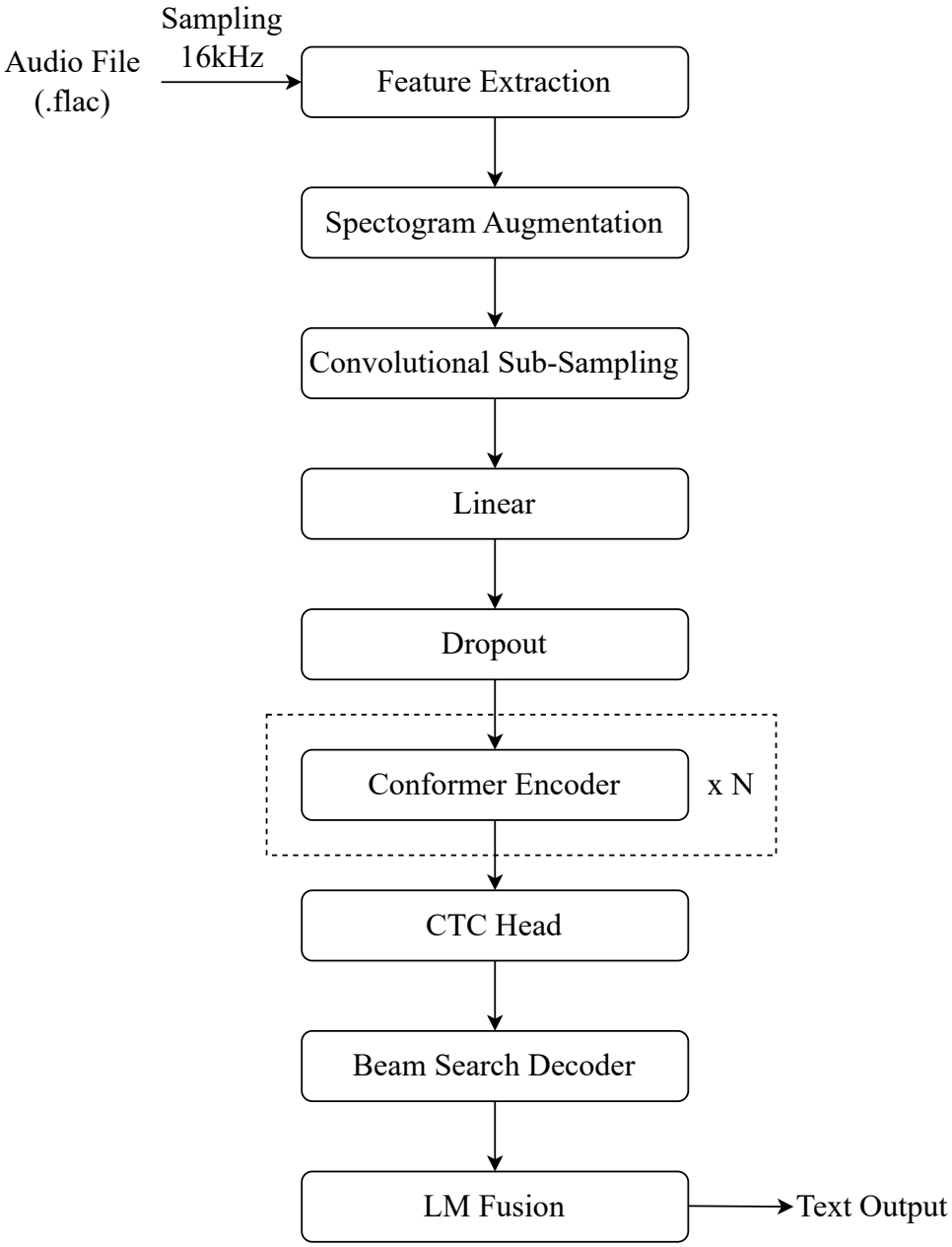

Figure 4-2: Proposed Architecture of ASR

4.2.2 Feature Extraction

Sampling Rate

This sample rate is consistent across the LibriSpeech datasets, which ensures consis-

tency across all inputs (for training as well as inference). It is also a common practice in speech processing. Human speech typically contains most of its important information in the frequency range up to 8 kHz. According to the Nyquist theorem, to accurately capture frequencies up to 8 kHz, we need at least $2 \times 8{,}000 = 16{,}000$ samples per second

Windowing

- Window Size (25ms)
  This defines the time of every frame of time used to calculate spectral features. A 25 ms window provides enough temporal information for speech without at the same time being too local (i.e. less than 25 ms).

- Stride
  The hop length of 10 ms allows for overlapping frames, which ensures smooth transitions between consecutive feature vectors, capturing fine temporal details.

Mel-Frequency Features (80 Bin)

- Extracting 80 mel-frequency features allows for a small size yet expressive representation of the audio signal focusing on perceptually relevant frequencies for human speech.

- The number of bins balances between resolution and model complexity.

Logarithmic Scaling

- Applying a logarithmic transformation to the mel-spectrogram simulates the human auditory system's detection of sound intensity, which makes the features more meaningful for ASR models.

- This has the additional advantage of reducing problems with dynamic range in the spectrogram.

Dithering

- Adding small random noise helps to avoid numerical instability in computations - especially if one is working with very small values or zero-padding in the spectrogram.

4.2.3 Spectrogram Augmentation

The model's overall performance is improved by adding spectrogram augmentation as a way to modify the data. Here, the features of the input spectrogram are modified by means of time masking , frequency masking , and time warping . To apply time masking, consecutive periods on the spectrogram are set to zero, yet for frequency masking, it is the continuous frequency lines that are masked. Also, bending time adds non-adjustable changes to the time axis in the spectrogram by moving it along a smooth curve, much like changes in a person's speaking rhythm. With these changes, the model works on features that are robust to acoustic variables, for example, faster speech, altered background noise, or sudden pitch changes. Augmentation with spectrograms in training prevents the Conformer from becoming attached to individual sound patterns and makes it more ready for handling a wide range of noises.

4.2.4 Convolutional Subsampling

In the stage of convolutional subsampling, Conformer is particularly useful for ASR tasks because input speech audio sequences can be very long and processing them is heavy computationally. Subsampling mainly seeks to lower the time frame of the input data without losing important details. ASR often works with Mel-spectrograms that are created at a rate of 10 milliseconds each frame. But working on such sequences straight away is inefficient because several of them have redundant neighboring frames. So, the sequence was subsampled through convolutional networks, decreasing the time resolution of each frame from 10ms to 40ms. Using strided convolution or pooling, the compression makes the model work more efficiently and achieve the same results. For example, the first stride of 2 in the convolution halves the size of the sequence to 6 samples and the resolution becomes 20ms. A second stride of 2 gives a sequence of 3 samples and 40ms resolution. By using these two steps, the sequence is compressed properly while keeping the main elements of the audio quality.

While frames separated by 10ms include a lot of details, this level of detail is usually not important for further use in the speech processing. The model pays attention to important patterns for recognition when data is subsampled to 40ms. Moreover, computational costs are greatly reduced because each layer of the Conformer gets smaller input vectors by working on trimmed parts of the sequences.

4.2.5 Linear

Conformer uses a linear layer in the model after the first process of convolutional subsampling to map the newly subsampled features into a larger space. This step is required because it brings the feature dimensions to the size of the hidden layers used by Trans-

former and convolutional modules. To explain this, the linear block picks the results from the downsampling, which lowers the time resolution and extracts local details, and completes the process by adding an extra layer to increase its dimensions. Doing this makes sure the achieved representation is able to hold detailed acoustic details and blends well with the self-attention and feed-forward modules from the Conformer design. This kind of design gives a balance between performance and calculations, helping the Conformer perform well in speech recognition.

4.2.6 Dropout

The dropout layer is put in place after the linear transformation that occurs after convolutional subsampling. This is very significant as it helps the model avoid complications during training. It reports that, by setting a percentage of the input tensor to zero randomly, dropout helps the learning process become a bit random. For this reason, the model must use all kinds of inputs, forming robust and generalized acoustic representations. This way of controlling overfitting matters, because speech data has many dimensions and the Conformer model includes both convolution and Transformers. Applying dropout in training guarantees steady and effective results for big speech recognition systems.

In the inference part, dropout is not utilized, so the model can use all its data to make predictions. Combining dropout with the Conformer's design approach lets the architecture handle both performance and generalization to reach excellent results in automatic speech recognition

4.2.7 Conformer Encoder

Conformer Architecture

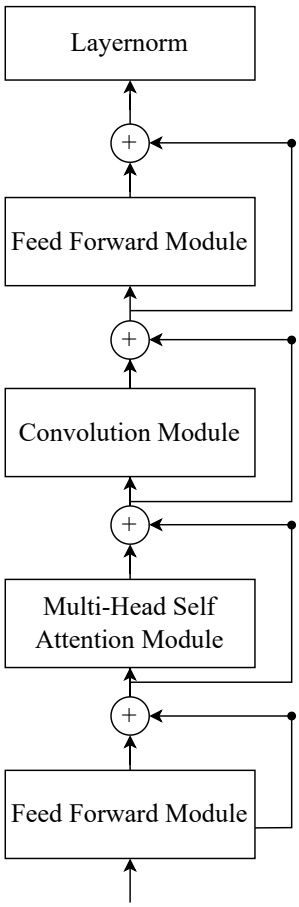

Figure 4-3: Conformer Architecture

Feed Forward Module

The Conformer architecture requires the Feed Forward Module to deal with features at single-token level. It includes two FFNs that work in the same way, one before and one after the focus on attention and convolutional techniques. They function as processes before and after training to boost the model's ability to learn features and still use few resources. All FFNs include certain layers in their structure: layer normalization, transformations, activation, and residual connections.

All FFNs start off with layer normalization , ensuring that the mean and variance of the input are stable. In doing this step, the inputs to the next layers become more controlled,

making the whole training process more stable. Once the module is normalized, it goes on to use a linear transformation followed by a non-linear GeLU or ReLU function. This way, the model can represent situations that are not simple, but more complex. After the activation, the results are passed through a linear layer that reshapes them to the original number of features. To sum up, residual connections enable gradients to continue, making sure issues like vanishing gradients are not a problem during training.

Multi-Headed Self Attention Module

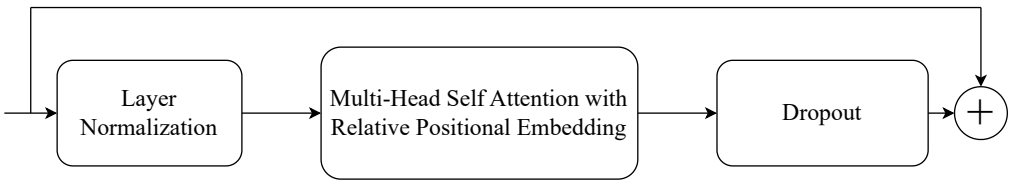

Figure 4-4: Block Diagram of Multi-Headed Self Attention Module Architecture

Multi-Head Self Attention with Relative Positional Embedding

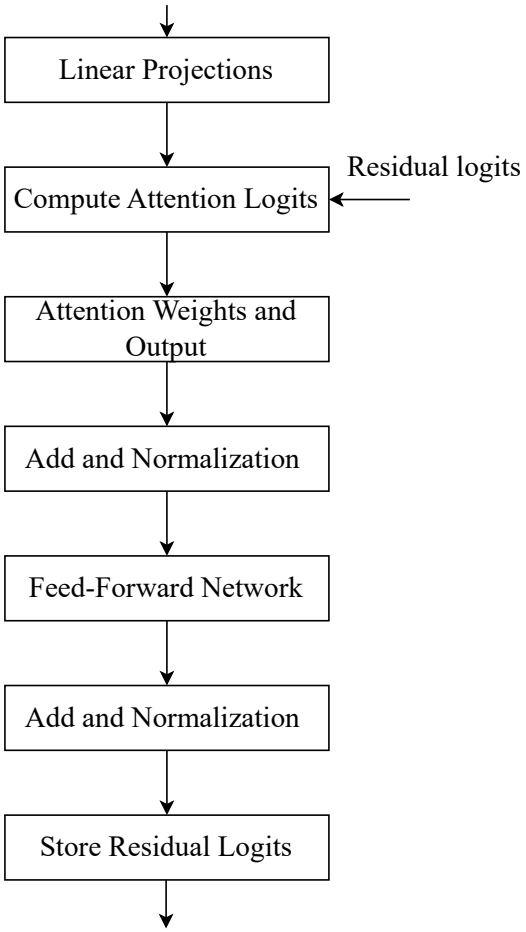

Figure 4-5: Block Diagram of Multi-Head Self Attention with Relative Positional Embedding

Convolution Module

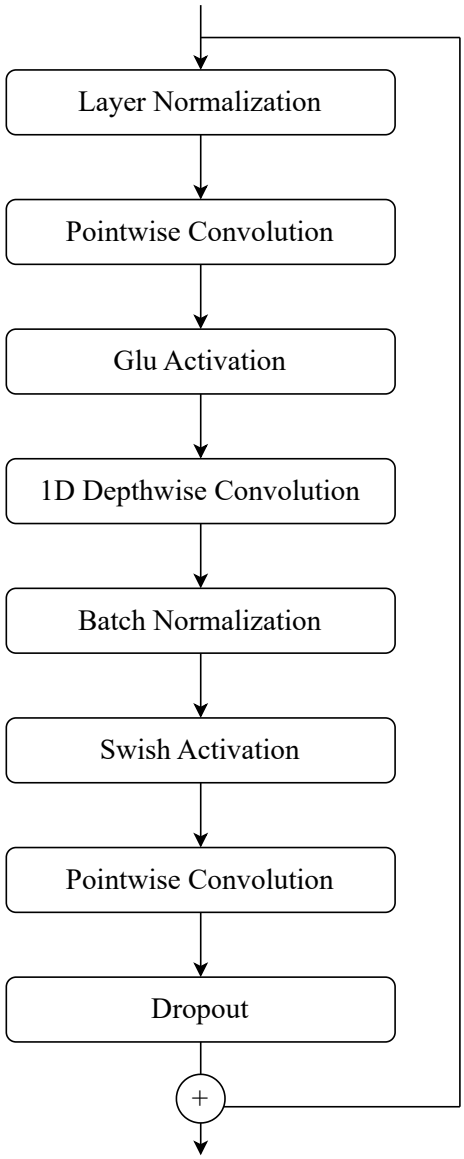

Figure 4-6: Block Diagram of Convolution Module

i. **Pointwise Convolution**

Part of the convolutional modules makes use of pointwise convolution to reorganize feature representations efficiently. A pointwise convolution also known as 1x1 convolution works by running a convolutional filter over all input channels with a kernel size of 1. In Conformer, time is what is being transformed in the temporal dimension of the speech features. One of the main uses of pointwise

convolution is to keep the input's spatial or temporal features while transforming it to a different dimension.

With pointwise convolutions, the Conformer saves computations and improves model performance, since it skips costly spatial computations.

ii. **GLU Activation**

The GLU activation function is added to bring non-linearity and make the features better represented. The input is separated into two pieces, and the "gate" part gets processed by a sigmoid function, then those values are multiplied with the remaining data. Because of gating, the model can control information being exchanged and give more attention to significant elements as needed. By using GLU on the feed-forward and convolutional layers in the Conformer increases the ability to represent speech data while maintaining a good balance between simplicity and complexity.

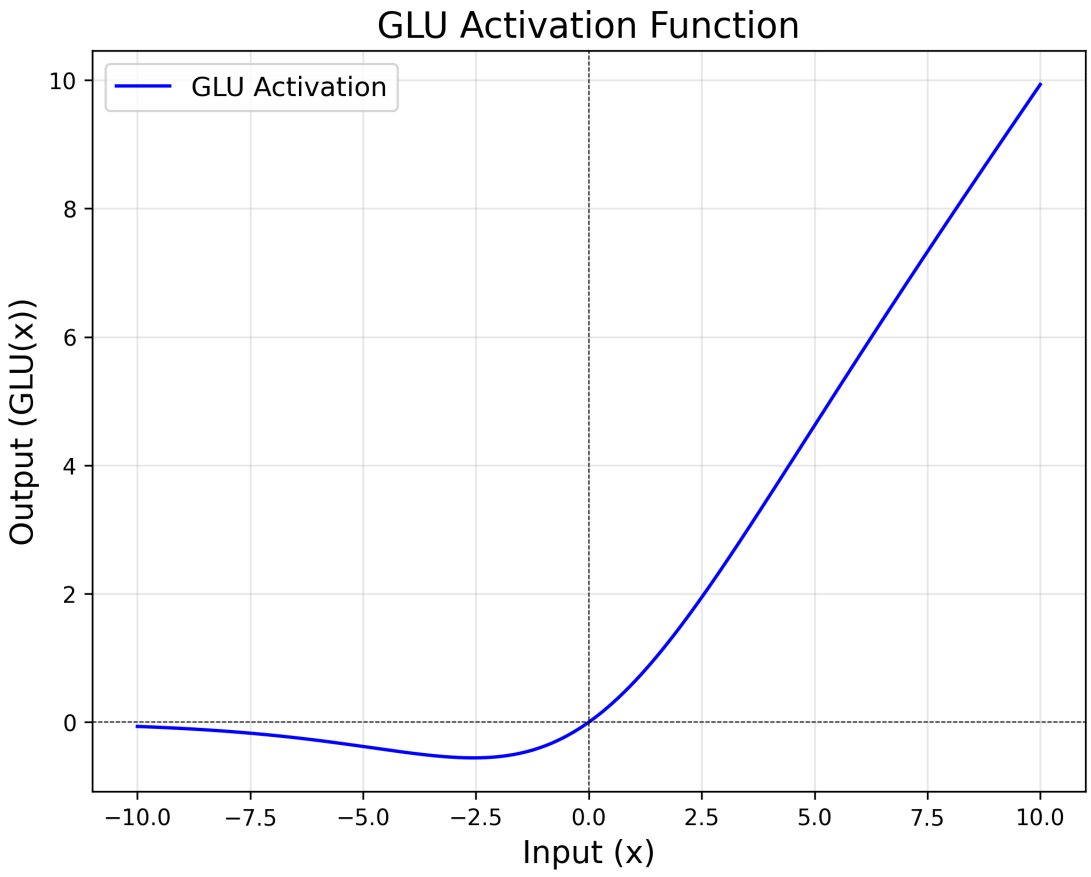

Figure 4-7: Graph of GLU activation

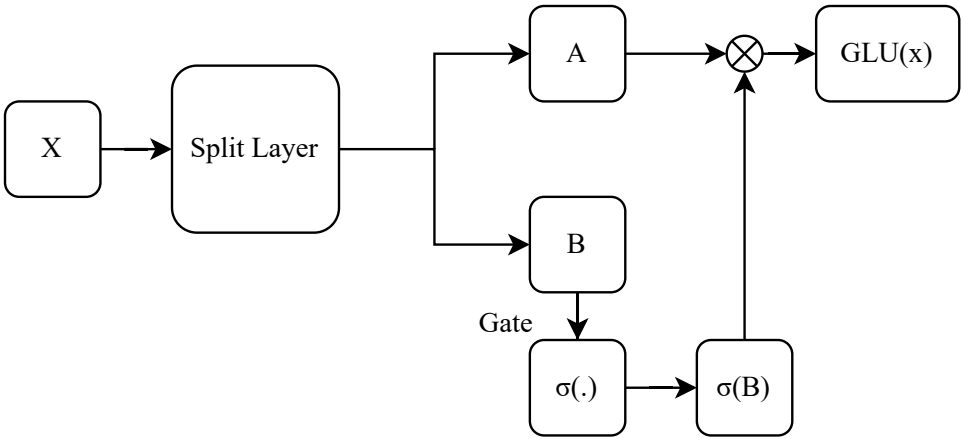

Figure 4-8: Block Diagram of GLU activation

iii. **1D depthwise convolution**

The Conformer designs 1D depthwise convolution to catch the connections between parts of speech by filtering each channel in sequence along the time axis. Consequently, resources are saved without losing the following local features. It combines pointwise convolution to help channels connect, thus it works well for modeling features in successive speech.

iv. **Swish Activation**

The Swish activation function is well known in deep learning since it performs better than simpler activation functions like ReLU (Rectified Linear Unit). Swish is useful in the Conformer architecture because it boosts both the strength and speed of training for automatic speech recognition tasks. Swish is defined as:

$$\text{Swish}(x) = x \cdot \sigma(x) \tag{4-1}$$

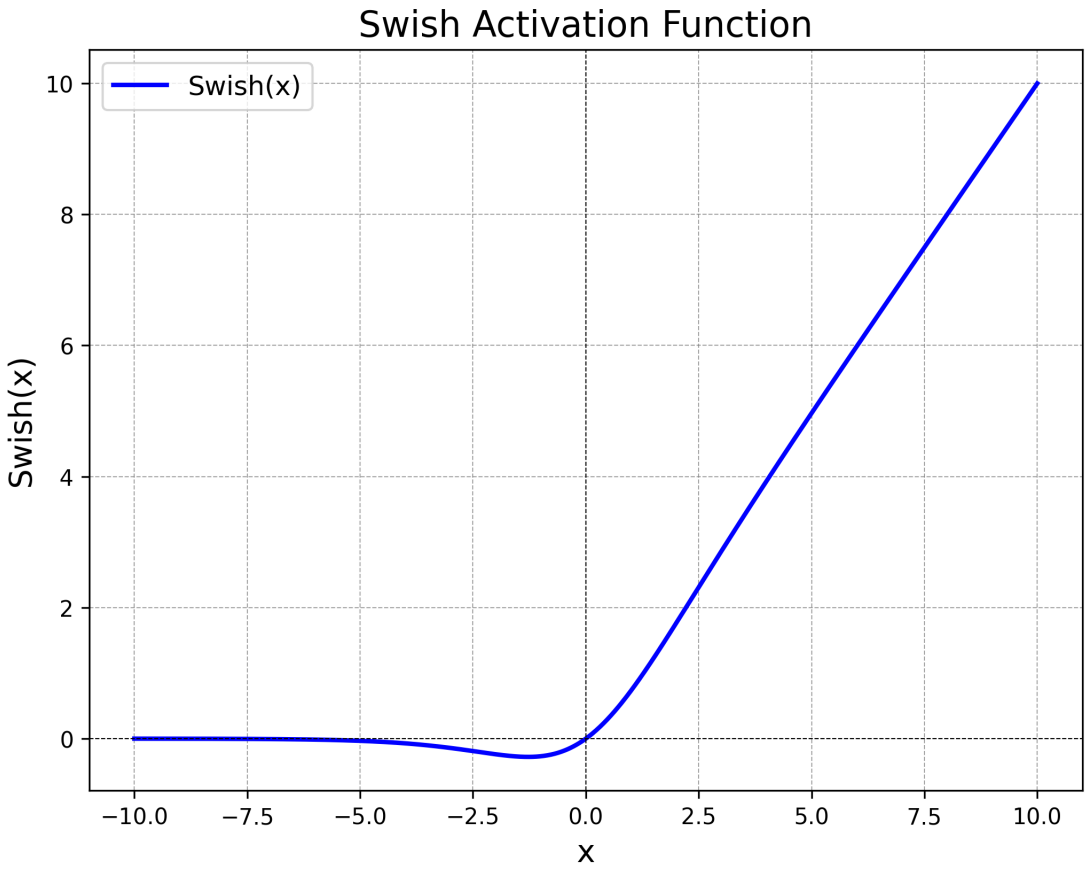

Figure 4-9: Graph of Swish Activation

Layer Normalization

The purpose of Layer Normalization is to keep the training stable and speedy by normalizing the activations in all layers. To be clear, it standardizes each input value in every feature dimension (the hidden dimension) for each step. This lets the mean and variance of the activations not change, so problems with vanishing or exploding gradients are avoided. It is applied in many places in the network to make sure the gradients do not experience fluctuations. As internal covariate shift is limited, the model performs better at obtaining strong features from speech data, seen mainly in deep networks with lots of layers. This method becomes crucial for reaching consistent and efficient convergence in automatic speech recognition.

Conformer Block Computation

Our proposal for Conformer includes putting the Multi-Headed Self-Attention and the Convolution modules between two Feed Forward modules. Macaron-Net [12] was a motivation behind this sandwich structure, and it suggests removing the regular feed-forward layer from a normal Transformer block and placing two reduced-size feed-forward layers, one before and another after the attention layer. Like in Macron-Net,

we add half-step residual weights to the feed-forward (FFN) parts of our model. The second feed-forward module comes ahead of the last layernorm layer. In math terms, yi is the output produced by Conformer block i as it takes input xi as input.

Mathematically, this means that for input x_i to a Conformer block i, the output y_i of the block is:

$$\tilde{x}_i = x_i + \frac{1}{2}\text{FFN}(x_i) \tag{4-2}$$

$$x'_i = \tilde{x}_i + \text{MHSA}(\tilde{x}_i) \tag{4-3}$$

$$x''_i = x'_i + \text{Conv}(x'_i) \tag{4-4}$$

$$y_i = \text{LayerNorm}\left(x''_i + \frac{1}{2}\text{FFN}(x''_i)\right) \tag{4-5}$$

4.2.8 SentencePiece Tokenizer

Sentence Piece is a language-neutral subword tokenisation system that is actively used in end-to-end automatic speech recognition (ASR) systems to alleviate vocabulary sparsity and out-of-vocabulary (OOV) effects. SentencePiece is unlike tokenisers, which rely on whitespace, and works on an unsegmented stream of Unicode characters, where subword units are directly learned from the input. It is therefore especially beneficial when dealing with languages that do not have clear word delimiting boundaries and with infrequent or misspelt words in speech transcripts. The framework incorporates the use of either the Byte-Pair Encoding (BPE) as the base algorithms and thus enables one to trade-off the granularity of tokens and the general model performance. SentencePiece produces strong, compact representations, operating on a subset of corpora coded uniformly and reversibly into subword tokens, such as special symbols marking the start and end of an utterance, and these representations are compatible with the output of acoustic models in modern ASR pipelines.

Algorithm 1 Train SentencePiece Tokenizer (BPE)

Input: Raw training text, target vocabulary size, model type (BPE), special tokens

Output: Trained tokenizer model, subword vocabulary, encoding and decoding rules

1. Preprocess the training text by normalizing whitespace and optionally adding sentence boundary markers
2. Initialize a character-level vocabulary using all unique Unicode symbols present in the text
3. Add predefined special tokens (e.g., `<s>`, `</s>`, `<unk>`) to the vocabulary
4. If the model type is BPE:
    - Initialize all characters as individual tokens
    - Convert the full text into a sequence of tokens based on the current vocabulary
    - Repeat until the target vocabulary size is reached:
        - Count the frequency of all adjacent token pairs
        - Select the most frequent token pair
        - Merge the selected pair into a new subword token
        - Add the new token to the vocabulary
        - Replace all occurrences of the merged pair in the token sequence
5. Save the learned vocabulary, subword merge rules, and model parameters
6. Return the trained tokenizer capable of encoding and decoding arbitrary text sequences

4.2.9 Connectionist Temporal Classification (CTC) Head

Connectionist Temporal Classification (CTC) is commonly used in sequence-to-sequence tasks such as speech recognition and handwriting recognition, where the input and output sequences are of different lengths and are not pre-aligned. CTC can be used both as a loss function during training and as a decoding strategy during inference.

CTC introduces a special *blank symbol*, typically denoted as b, which allows the model to emit either an output symbol y_t or a blank at each time step of the input sequence.

Let the input sequence be

$$\mathbf{x} = (x_1, x_2, x_3, \ldots, x_T), \tag{4-6}$$

where T denotes the input sequence length, and let

$$\mathbf{y} = (y_1, y_2, \ldots, y_U) \tag{4-7}$$

be the target output sequence.

CTC defines a collapsing function $\mathcal{B}$ that removes repeated symbols and blank symbols. The set of all alignment paths that collapse to $\mathbf{y}$ is denoted by

$$\mathcal{B}^{-1}(\mathbf{y}). \tag{4-8}$$

CTC Objective

The conditional probability of the output sequence $\mathbf{y}$ given the input sequence $\mathbf{x}$ is defined as

$$P(\mathbf{y} \mid \mathbf{x}) = \sum_{\boldsymbol{\pi} \in \mathcal{B}^{-1}(\mathbf{y})} P(\boldsymbol{\pi} \mid \mathbf{x}), \tag{4-9}$$

where $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_T)$ represents an alignment path.

Each alignment probability is factorized over time steps as

$$P(\boldsymbol{\pi} \mid \mathbf{x}) = \prod_{t=1}^{T} P(\pi_t \mid \mathbf{x}), \tag{4-10}$$

with

$$\pi_t \in \mathcal{Y} \cup \{b\}, \tag{4-11}$$

where $\mathcal{Y}$ is the output vocabulary and b denotes the blank symbol.

The summation over all valid alignments is efficiently computed using dynamic programming techniques such as the forward–backward algorithm.

Inference

During inference, the output sequence is simplified by collapsing consecutive repeated symbols and removing blank symbols. For example,

$$\texttt{h\_l\_l\_o} \longrightarrow \texttt{hello}. \tag{4-12}$$

Comparison with Attention-Based Methods

Unlike attention-based models that perform global alignment between input and output sequences, CTC performs alignment locally at each time step. This local alignment strategy avoids scanning the entire sequence during decoding, leading to improved computational efficiency, especially for long input sequences.

4.2.10 Beam Search Decoder

In non-streaming ASR systems, once the features of a whole audio input are processed by the encoder-decoder model, the final step is to generate the most likely transcription. Instead of selecting the highest-probability word at each step(greedy decoding), this project utilizes the Beam Search decoding algorithm to improve the contextual accuracy of the output.

Beam search is an approximate search method that visits the multiple possible output sequences parallely. It is a way to try many possible word sequences at the same time. It keeps the top k best guesses at each step. Then it looks at all the possible next words for each guess and keeps only the best k sequences based on their total probability.

$$\hat{Y} = \arg\max_{Y \in \mathcal{B}} \log P(Y \mid X) \tag{4-13}$$

Where:

- $\hat{Y}$ is the final predicted output sequence.
- X is the input (e.g., audio features).
- Y is a candidate output sequence.
- $\mathcal{B}$ is the set of top k hypotheses maintained at each step (beam size).
- $P(Y \mid X)$ is the conditional probability of the sequence Y given input X.

4.2.11 AdamW Optimizer

AdamW is an improvement over the traditional Adam optimizer, as it decouples the weight decay term from the gradient-based parameter updates, thereby enabling more effective regularization. This property is particularly beneficial when training large-scale deep learning models such as FastConformer, a highly efficient variant of the Conformer architecture used in speech recognition.

In the standard Adam formulation, weight decay is applied through L2 regularization

of the loss, which can interfere with the adaptive learning-rate mechanism. AdamW addresses this limitation by applying weight decay directly to the model parameters after the Adam update step, as shown in the following equation:

$$\theta_{t+1} = \theta_t - \eta \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) - \eta \lambda \theta_t \tag{4-14}$$

where θ_t represents the model parameters at training step t, η is the learning rate, $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected first and second moment estimates, respectively, and λ denotes the weight decay coefficient.

In architectures such as FastConformer, which combine convolutional and self-attention layers, training stability and generalization are critical. By providing cleaner and more consistent regularization, AdamW typically leads to better convergence behavior and improved generalization compared to the standard Adam optimizer, especially when training transformer-based models with large batch sizes and long input sequences.

4.2.12 Noam Annealing

Noam annealing is a learning-rate scheduling strategy introduced in *Attention Is All You Need* by Vaswani et al. (2017) for training Transformer models. The schedule is specifically designed to complement adaptive optimizers such as Adam by avoiding the need for conventional learning-rate decay schemes.

The Noam schedule combines an initial linear warmup phase with an inverse square root decay, defined as:

$$\mathrm{lr}(t) = d_{\text{model}}^{-0.5} \cdot \min \left(t^{-0.5},\ t \cdot \text{warmup\_steps}^{-1.5} \right) \tag{4-15}$$

where t denotes the current training step, d_{model} is the model dimensionality (e.g., embedding size), and warmup_steps is a predefined hyperparameter.

During the warmup phase, the learning rate increases linearly with the training steps, which helps stabilize optimization when parameters are randomly initialized. After the warmup period, the learning rate decays proportionally to $t^{-0.5}$, allowing for smoother convergence and improved generalization.

Noam annealing provides stable optimization dynamics in architectures such as FastConformer that rely heavily on self-attention and convolutional blocks. When combined

with optimizers like Adam or AdamW, this scheduling strategy effectively balances rapid initial learning with gradual refinement, making it a widely adopted choice for transformer-based speech and language models.

4.2.13 LM Fusion

LM fusion works by adding a language model trained separately to improve how the machine transcribes what is spoken. An acoustic model is used in traditional ASR to make the audio into phonetic or word sequences and the language model adds extra knowledge, like grammar and word chances, to make the result clearer. LM fusion fuses different models, often applying shallow fusion, deep fusion or cold fusion. Shallow fusion works by combining the scores from the LM with the output from the ASR during the decoding stage, most often with log-linear techniques. In deep fusion, the LM's hidden states join with the features in the ASR encoder and cold fusion trains the ASR model alongside a pre-trained LM to make their representations match. The LM makes it possible to understand longer and complex sentences which decreases mistakes in confusing or poor-quality audio and gives a better overall result.

4.2.14 Softmax Function

Softmax function helps convert raw logits from the output layer into a distribution of probabilities for the target vocabulary. This means the model creates probabilities for every output token (such as phonemes, characters or words) and, during decoding, it can pick the most likely sequence.

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{4-16}$$

- z_i: The input score (logit) for class i
- z_j: The input score (logit) for class j
- K: The total number of classes

4.2.15 Word Error Rate (WER)

Word Error Rate (WER) is the most popular metric used to evaluate the performance of the Automatic Speech Recognition (ASR) systems. It is used to measure the accuracy of transcription by comparing the hypothesis of the system to a ground-truth reference transcript.

$$\text{WER} = /fracS + D + IN \tag{4-17}$$

where,

- S = Number of substitutions (wrong words)
- D = Number of deletions (missing words)
- I = Number of insertions (extra words)
- N = Total number of words in the reference

4.3 Small Language Model

4.3.1 Proposed Architecture Of Small Language Model

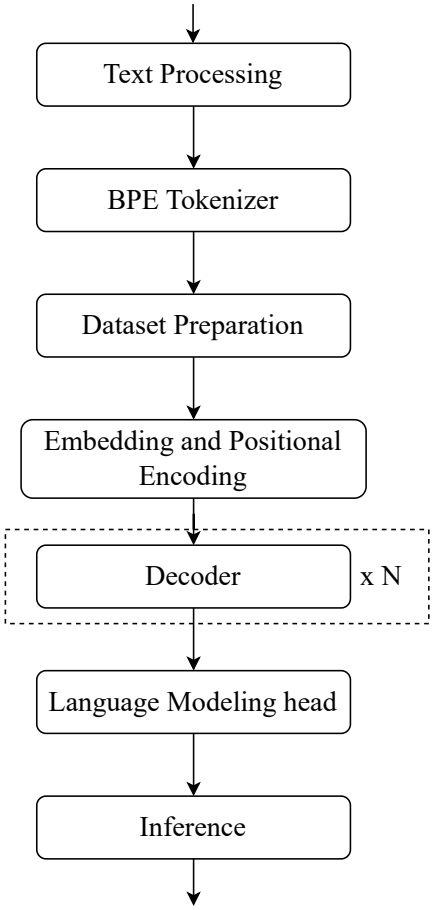

Figure 4-10: Proposed Architecture Of Small Language Model

4.3.2 Text Processing

This initial stage focuses on cleaning and normalizing raw text data to make it suitable for subsequent processing. The main objective is to reduce noise and inconsistencies. Whitespace normalization is performed where multiple spaces are combined into one and extra spaces at the beginning and the end of the string is removed. Special characters are also handled by converting them into tokens.

4.3.3 BPE Tokenizer

Byte Pair Encoding (BPE) is a subword tokenization algorithm that balances vocabulary size and sequence length by Splitting text into subword units, out-of-vocabulary words through meaningful subword combinations and reducing vocabulary size while maintaining linguistic coherence.

Algorithm 2 Train BPE Tokenizer

Input: Training text, desired vocabulary size, special tokens

Output: Vocabulary, BPE merge rules

1. Replace each space in the text with the symbol Ġ
2. Initialize the vocabulary with all unique characters in the text
3. Add any special tokens (e.g. $|endoftext|$) to the vocabulary
4. Convert the full text into a list of token IDs based on the current vocabulary
5. Repeat until vocabulary size reaches target:
   - Count all adjacent token pairs in the text
   - Find the most frequent pair
   - Create a new token by merging the pair
   - Add the new token to the vocabulary
   - Replace all occurrences of the pair with the new token
6. Return the final vocabulary and merge rules

4.3.4 Dataset Preparation

This step includes preparing the data so that it can be fed to a machine learning model, using batching, padding and making attention masks.

i. **Sequence Truncation/Padding**
   Making sure that the length of each sequence in a batch is the same. Too long or short sequences are adjusted to meet the set limit by adding or removing `<PAD>` tokens.

ii. **Batching**
   Combining several sequences into batches so that GPUs handle them more efficiently in parallel.

iii. **Attention Mask Creation**
   Creating a binary mask to distinguish between data tokens and padding tokens. It prevents the model from paying attention to padding tokens when performing

attention calculations.

iv. **Label/Target Generation**

If the model is taught to do a particular task (like predict the following word or translate), the required target data are created. When doing language modeling, the goal is frequently to predict the next token in the sequence.

4.3.5 Embedding and Positional Encoding

Here, each numerical token ID is converted into a dense vector and the location of each token within the sequence is included.

i. **Token Embeddings**

Every token ID is attached to a large, continuous, vector-like object (embedding). These embeddings are built up over training and pick up on similarities between words (e.g. the words ”king” and ”queen” might have similar embeddings).

ii. **Positional Encoding**

Since Transformers ignore word order and process all sequences at once, they add positional encodings to the embeddings. They are often fixed curves (like sine waves) or memorized vector sequences that explain the position of each token. Since Transformers ignore word order and process all sequences at once, they add positional encodings to the embeddings. They are often fixed curves (like sine waves) or memorized vector sequences that explain the position of each token.

The processed token and its positional encoding are combined to form the final representation that goes to the next layer.

4.3.6 Decoder

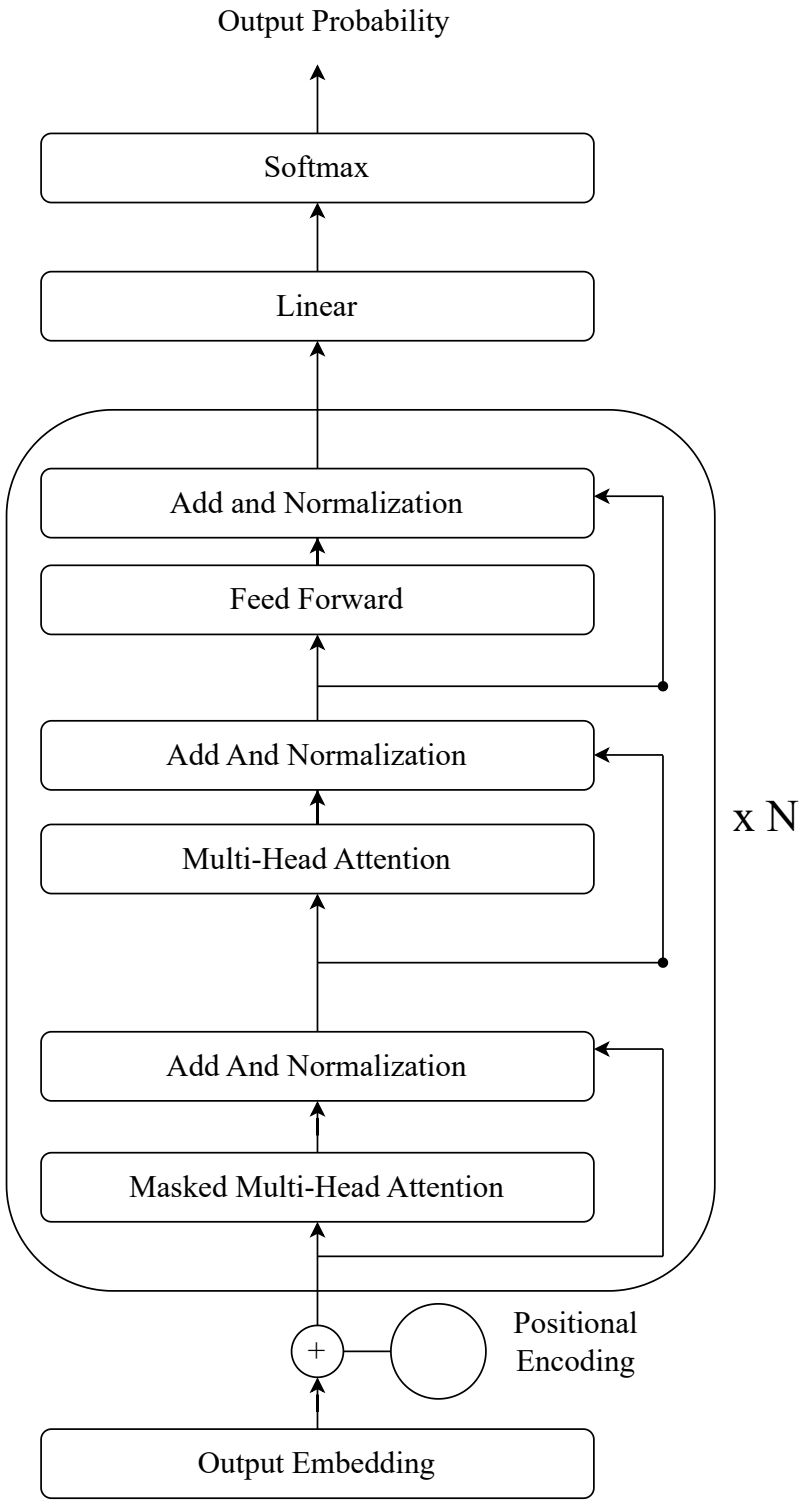

Figure 4-11: Block Diagram of Decoder Architecture

i. **Masked Multi-Head Self-Attention:**

It helps the decoder focus on all the previous tokens it has created. Because tokens are ”masked,” only tokens that appeared earlier in the sequence are considered, so later tokens do not influence the ones that appear afterward.

Scaled Dot-Product Attention

Query-Key-Value Projection

Input token embeddings $X \in \mathbb{R}^{n \times d_{\text{model}}}$ are projected into queries (Q), keys (K), and values (V) using learnable weight matrices for each head i:

$$Q_i = XW^{Q_i}, \quad K_i = XW^{K_i}, \quad V_i = XW^{V_i} \tag{4-18}$$

where: - Q_i, K_i, V_i are the query, key, and value projections for head i,
- $W^{Q_i}, W^{K_i}, W^{V_i} \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (or d_v for V) are learnable parameters.

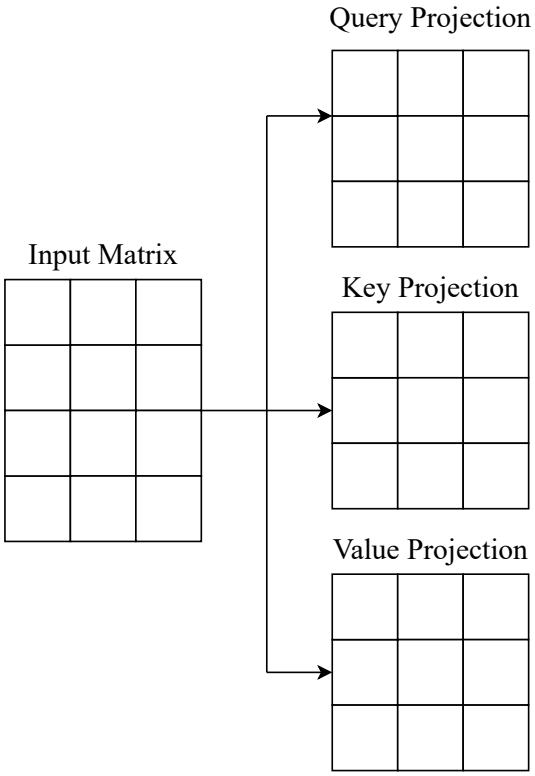

Figure 4-12: Query Key Value Projection

ii. **Attention Scores**

Compute logits via *scaled dot-product attention*. For layer l, residual logits $R^{(l-1)}$

from the previous layer are added:

$$A_i^l = \frac{Q_i (K_i)^\top}{\sqrt{d_k}} + R_i^{l-1} \tag{4-19}$$

Purpose of Scaling:

The dot-products $Q_i K_i^\top$ grow large in expectation as the dimension d_k increases, which can lead to small gradients when using activation functions like softmax. Dividing by $\sqrt{d_k}$ normalizes the variance of the dot-products across different values of d_k, stabilizing the gradients and preventing vanishing or exploding gradients during training.

Causal Masking

Future tokens are masked by setting their attention scores to $-\infty$ in the attention matrix before applying softmax. This ensures that, during training or inference, the model cannot attend to subsequent tokens, thereby enforcing autoregressive generation.

iii. **Cross-Attention:**

Enables the decoder to take the output (key and value) from the encoder into account using attention. With this mechanism, the decoder can pay attention to necessary parts of the input when producing a token.

iv. **Feed-Forward Networks:**

A position-wise fully connected network applies non-linear transformations to refine token representations. Each position uses its own FFN with ReLU activation:

$$\mathrm{FFN}(Y) = \max(0, YW_1 + b_1)W_2 + b_2 \tag{4-20}$$

v. **Residual Connections & Layer Normalization:**

To help train very deep networks, a residual connection and layer normalization come after each attention and feed-forward sub-layer.

vi. **Iterative Generation:**

During inference, the decoder generates tokens one at a time using previously generated tokens as inputs (e.g., autoregressive generation of "Winter is cold" from the input "Winter is"). Decoding strategies include:

- **Greedy Decoding:** Selects the highest-probability token at each step.
- **Beam Search:** Maintains the top k candidate sequences to improve contextual coherence

4.3.7 Language Modelling Head

This block uses the wrapped information from the decoder to form predictions about which token is going to appear next.

- **Linear Transformation:** A linear layer (fully connected neural network) takes the high-dimensional output from the decoder and changes it to a vector as wide as the vocabulary.
- **Output:** The final product of this layer is a set of raw scores (logits) for each token that can be used. The more positive the logit is, the higher the possibility that token is next in the sequence.

$$P(y_i|x) = \frac{e^{z_i}}{\sum_{j=1}^{V} e^{z_j}} \tag{4-21}$$

Where Z_i are the logits for token i, and V is the vocabulary size.

4.3.8 Loss Functions

The next token that the model predicts is compared to the true next token (the ground truth label) during training.

Cross-Entropy Loss: Cross-entropy loss is used in classification tasks that deal with language modeling. It finds the difference between the predicted and actual probability distributions. The model tries to lower the amount of loss.

$$L = -\sum_{i=1}^{V} y_i \log(\hat{y}_i) \tag{4-22}$$

Where y_i is true for the next token and 0 otherwise, and $\hat{y}_i$ is the predicted probability for token i.

4.3.9 Inference

At inference time (when the model generates text), the softmax scores help decide on the following token to include.

- **Greedy Decoding:** The token that is most likely to occur is always used first when making a decision.

- **Beam Search:** Keeps track of the top k sequences at each step which results in outputs that make more sense and cover a wider range of styles.
- **Sampling:** Usually, the next word is decided by choosing it randomly from the probability distribution and controlling how random it is by adjusting the temperature.

4.3.10 Perplexity

Perplexity is a metric that measures the uncertainty of a model's predictions. It quantifies how well the model predicts the next word in a sequence. When a model makes predictions,it assigns probabilities to possible next words.

$$\text{Perplexity}(P) = 2^{H(P)} \tag{4-23}$$

Entropy measures the level of uncertainty in the model's output. Lower entropy means the model is more certain about its predictions and therefore, the perplexity is lower.

Perplexity indicates the level of confidence the model has in its prediction—lower perplexity suggests higher confidence and better performance in predicting the next word, while higher perplexity signals more uncertainty and less reliability.

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i \mid w_{i-1}, w_{i-2}, \ldots, w_1)\right) \tag{4-24}$$

where,

- $P(w_i \mid w_{i-1}, w_{i-2}, \ldots, w_1)$ is the predicted probability of the i^th word
- N is the total number of words in the sequence

4.3.11 Masked Loss

Masked Cross-Entropy Loss computes the negative log-likelihood of the correct target tokens while excluding padded or invalid positions from contributing to the loss. It measures how well the model predicts the correct tokens, but it ignores padded or invalid tokens. The valid tokens are masked while ignoring the tokens marked with ignore index.
It ensures that loss computation is reflected for only meaningful tokens and prevents the

padding symbols from affecting the model evaluation.

$$\mathcal{L}_{\text{masked}} = -\frac{1}{N}\sum_{t=1}^{T}\mathbb{I}(y_t \neq \text{ignore}) \log p_\theta(y_t \mid x_t) \tag{4-25}$$

4.3.12 BERTScore F1

BERTScore-F1 measure the semantic similarity between the generated text and reference text. It does this by computing token-level cosine similarity between contextual embeddings produced by a pretrained BERT model. It checks how similar the generated text is to the reference text by comparing their word meaning with BERT embeddings. It evaluates semantic correctness even when surface-level word overlap is low.

$$\text{BERTScore}_{F1} = \frac{2PR}{P+R} \tag{4-26}$$

Precision

Precision measures, on average, how well each token in the candidate matches some token in the reference. High precision means the candidate words are "covered" by reference words.

$$P = \frac{1}{|X|}\sum_{x\in X}\max_{y\in Y}\cos(\mathbf{e}_x, \mathbf{e}_y) \tag{4-27}$$

where:

- X = tokens in the candidate/generated text.
- Y = tokens in the reference text.
- e_x, e_y = embeddings of tokens x and y.
- $\cos(e_x, e_y)$ = cosine similarity between embeddings of x and y.

Recall

Recall measures how much of the reference content is captured by the candidate. High recall means the candidate includes most of the reference's meaning.

$$R = \frac{1}{|Y|}\sum_{y\in Y}\max_{x\in X}\cos(\mathbf{e}_y, \mathbf{e}_x) \tag{4-28}$$

where:

- X = tokens in the candidate/generated text.
- Y = tokens in the reference text.
- e_x, e_y = embeddings of tokens x and y.
- $\cos(e_y, e_x)$ = cosine similarity between embeddings of y and x.

4.3.13 ROUGE

ROUGE-L evaluates the similarity between generated and reference text based on the length of their Longest Common Subsequence (LCS). It measures how similar the generated text is to reference text by looking at the longest sequence of words they share in the same order. It captures sentence-level structure and word ordering beyond simple n-gram overlap.

$$\text{ROUGE-L}_{F1} = \frac{2PR}{P+R} \tag{4-29}$$

LCS

Given two sequences, LCS measures their similarity by identifying the maximum-length subsequence common to both.
$\text{LCS}(X,Y)$ = length of the Longest Common Subsequence between candidate text X and reference text Y.

Precision

It is the measure where it captures the order by the candidate of all reference token.

$$P = \frac{\text{LCS}(X,Y)}{|Y|} \tag{4-30}$$

Here, $|Y|$ = number of tokens in the reference.

Recall

It is the measure of matching references of all the candidate tokens.

$$R = \frac{\text{LCS}(X,Y)}{|X|} \tag{4-31}$$

$|X|$ = number of tokens in the candidate.

4.3.14 BLEU

BLEU stands for Bilingual Evaluation Understudy. It was originally created for machine translation evaluation. It measures how many n-grams in the generated text appear in the reference text. BLEU is focused on precision. It counts the overlap of the n-grams but doesn't directly reward covering all the references i.e. recall isn't explicit. It works by breaking the candidate and reference into n-grams (usually 1-gram to 4-gram). It computes the precision for each n-gram and combines the n-gram precision geometrically.

Step 1: n-gram Precision

For n-grams of size n, the precision is defined as:

$$P_n = \frac{\text{Number of n-grams in candidate that appear in reference}}{\text{Total number of n-grams in candidate}} \tag{4-32}$$

where:

- P_n = precision for n-grams of size n

Step 2: Combine n-gram Precisions

For BLEU using 1 to N grams, we take the geometric mean of precisions:

$$\text{Precision}_{1\text{-}N} = \exp\left(\frac{1}{N}\sum_{n=1}^{N}\log P_n\right) \tag{4-33}$$

where:

- N = maximum n-gram size (usually 4)
- P_n = precision for n-grams of size n

Step 3: Brevity Penalty (BP)

To penalize short candidates:

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{4-34}$$

where:

- c = length of candidate sentence
- r = length of reference sentence

Step 4: Final BLEU Score

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log P_n\right) \tag{4-35}$$

where:

- w_n = weight of n-gram precision (usually $w_n = 1/N$ for uniform weighting)

5 IMPLEMENTATION DETAILS

5.1 ASR

5.1.1 Dataset Preparation

The LibriSpeech corpus is hierarchically organized into a directory structure to facilitate efficient storage and retrieval. In the LibriSpeech 100-hour split (`train-clean-100`), the root directory contains subdirectories named after speaker IDs (e.g., `19`, `26`). Each speaker directory further contains chapter-level subdirectories (e.g., `198`, `261`), which store short audio utterances in lossless `.flac` format. These utterances typically range from 1 to 10 seconds in duration, are sampled at 16 kHz with 16-bit depth, and use lossless compression. Each chapter directory is accompanied by a transcript file (`.txt`) that contains the transcriptions for all utterances in that chapter.

For ease of implementation during model development, the `torchaudio` library in PyTorch provides the `LIBRISPEECH` dataset class, which automatically parses this directory structure and returns audio waveforms together with the corresponding metadata.

In addition, several modern speech recognition toolkits such as NVIDIA NeMo, ESPnet, and Fairseq rely on manifest files to organize training data. A manifest is typically stored in the JSON Lines (JSONL) format, where each line corresponds to a single utterance and contains the following fields:

- `audio_filepath`: path to the `.flac` audio file,
- `duration`: duration of the utterance in seconds,
- `text`: the cleaned transcript of the utterance.

To construct a manifest file for the LibriSpeech `train-clean-100` split, the transcript (`.txt`) files associated with each chapter can be read and mapped to their corresponding `.flac` audio files using the utterance identifiers. The resulting entries are written sequentially to a manifest file (e.g., `librispeech_train_clean_100.json`). This representation enables faster randomized access, supports dynamic batching, and allows seamless integration with data loaders that are independent of PyTorch-specific dataset classes.

Listing 1: LibriSpeech Dataset Downloader

```
from torchaudio.datasets import LIBRISPEECH

dataset = LIBRISPEECH(root="data", url="train-clean-100",
 download=True)
```

```
waveform, sample_rate, transcript, speaker_id, chapter_id,
 utterance_id = dataset[0]
```

5.1.2 Computing Mel Spectrogram

In order to to feed Conformer, we need to pre-process raw audio data. We employed feature extraction pipeline which transforms raw audio waveforms in log Mel Spectrograms. This process is important in limiting the large dimensionality of input signal.

Sliding window approach is applied to cut the raw audio waveform into overlapping frames. The windowing process is specified by the following parameters:

Table 5-1: Audio Feature Extraction Parameters

Parameter	**Value**
Sample Rate ($sample\_rate$)	16000
Normalization ($normalize$)	per_feature
Window Size ($window\_size$)	0.025
Window Stride ($window\_stride$)	0.01
Window Function ($window$)	hann
Number of Features ($features$)	80
FFT Size (n_fft)	512
Log Scaling (log)	True
Frame Splicing ($frame\_splicing$)	1
Dither ($dither$)	1×10^{-5}
Padding To (pad_to)	0
Padding Value (pad_value)	0.0

The magnitude spectrogram is converted into a Mel-scale representation through a bank of triangular filters in order to highlight perceptually interesting frequencies and down sample. Mel Filters are used where each spectrogram is projected in 80 Mel filters creating a compact representation. Logarithmic scaling is performed so that the filter bank is changed to a scale closer to human auditory perception and present better numerical stability during training.

5.1.3 Spectrogram Augmentation

The graph 5-1 displays the raw audio waveform of the first sample from a random batch, it is visualized as a continuous line plot where the horizontal axis represents the time in terms of the audio frames and the vertical axis represents the corresponding amplitude. The waveform appears dense and oscillatory, with various fluctuations indicating complex sound signals such as speech. This type of waveform is obtained by loading the audio file using torchaudio.load(), which returns the waveform and sampling rate, the waveform tensor is then squeezed and converted to NumPy array for plotting using matplotlib.plot().

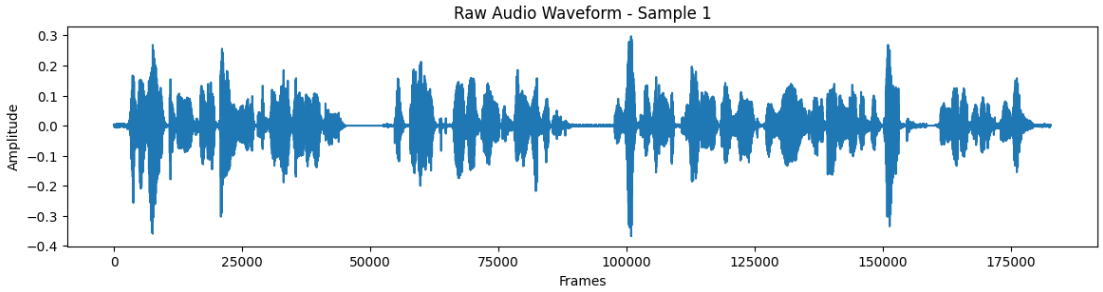

Figure 5-1: Raw Audio Waveform

The graph 5-2 shows a Mel spectrogram of the first audio sample, visualized using a color map. The spectrogram represents the frequency over time, with brighter colors indicating higher magnitudes and darker regions indicating lower magnitudes of energy. This Spectrogram is derived from the raw audio waveform plotted in the previous image, which was processed using short-time Fourier transform (STFT).

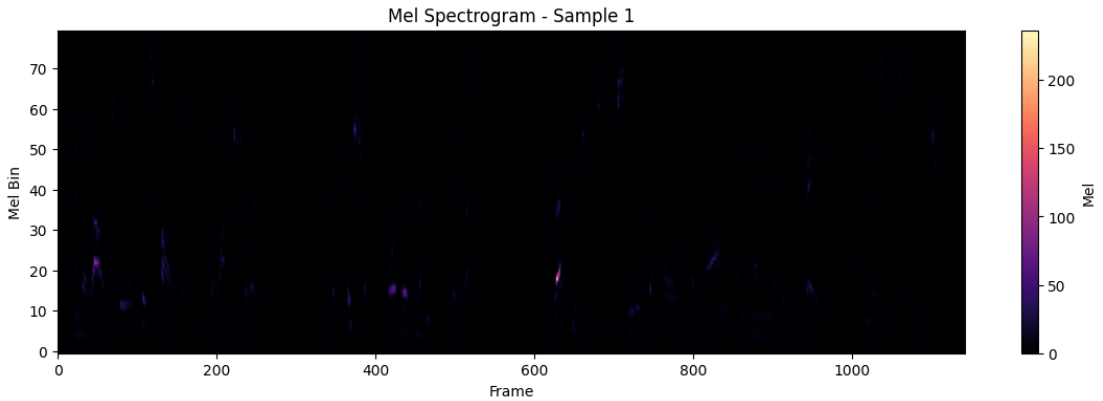

Figure 5-2: Mel Spectrogram Of Raw Audio

This is the Log Mel Spectrogram of an audio sample. Each value of energy magnitude at mel frequency bin for a particular time frame is calculated and log is applied.

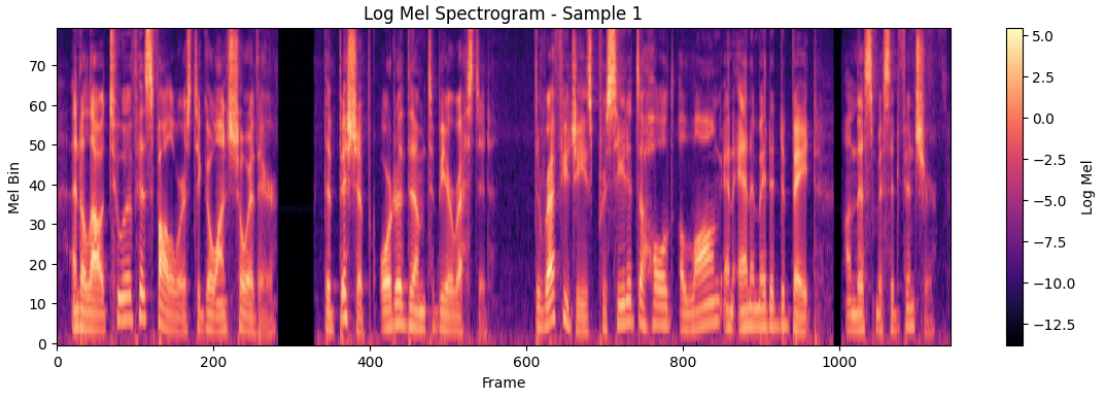

Figure 5-3: Log Mel Spectrogram

This graph 5-4 shows the frequency masking of a log mel spectrogram. The values of some frequency band is masked (hidden) at random. It simulates missing parts in audio. It forces the model to not rely too much on specific frequency bands. The model learns to focus on broader context and other frequency features, not just one narrow band.

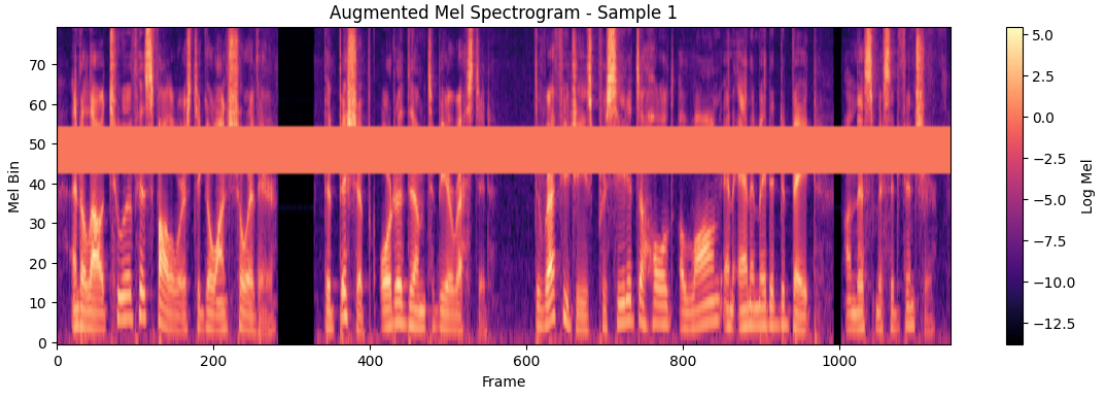

Figure 5-4: Spec Augmentation with Frequency Mask = 20

The graph 5-5 shows the time masking of a log mel spectrogram. The values of some continuous time frames are masked(hidden) at random. It helps the model to fill in gaps or use information from before and after the masked time segment. It makes the model not rely too much on specific time parts of the audio (like certain words or sounds).

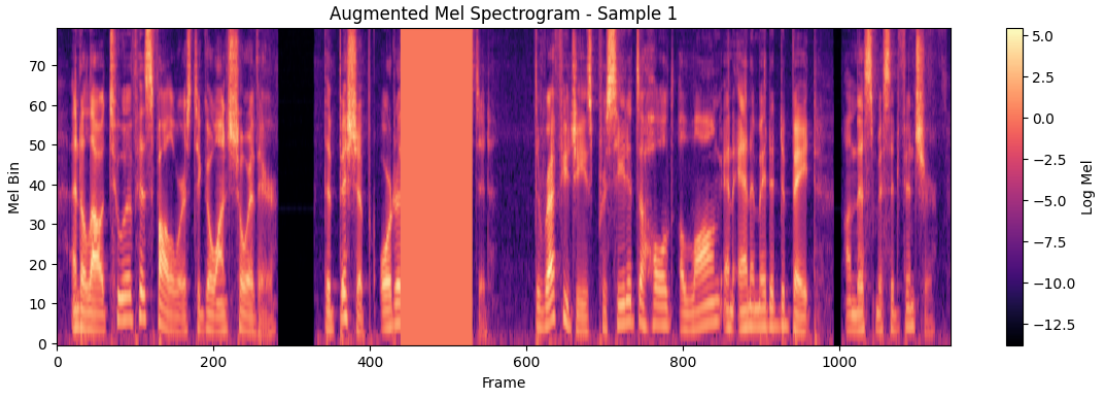

Figure 5-5: Spec Augmentation with Time Mask = 20

The graph 5-6 shows the time warping of a log mel spectrogram. Time Warping means stretching or compressing parts of the spectrogram along the time axis. It changes the timing of events in the audio without changing the contents. It helps us simulate the various speaking speeds. It improves the ability to generalize across different speakers and speech styles.

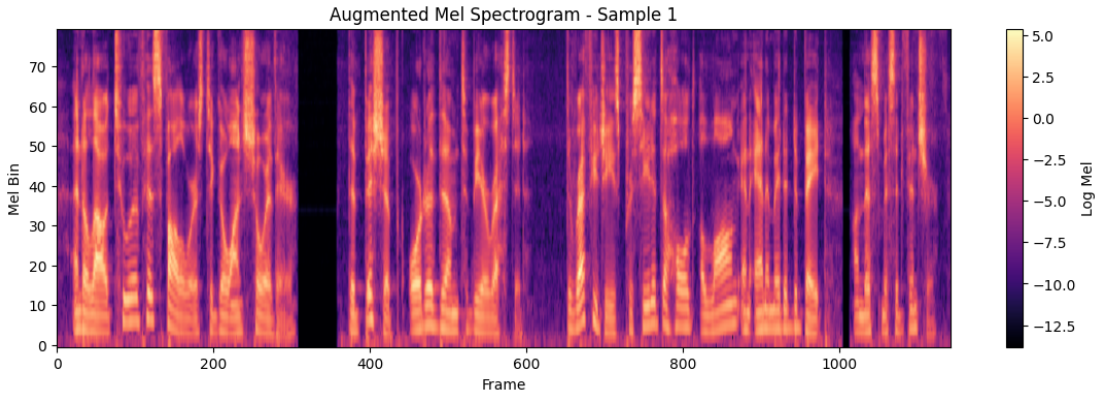

Figure 5-6: Spec Augmentation with Time Warp = 20

The graph 5-7 shows the combination of above 3 augmentation spectrograms. It simulates the different gaps in speech, the rate of speaking and helps the model to not rely on particular frequency bands.

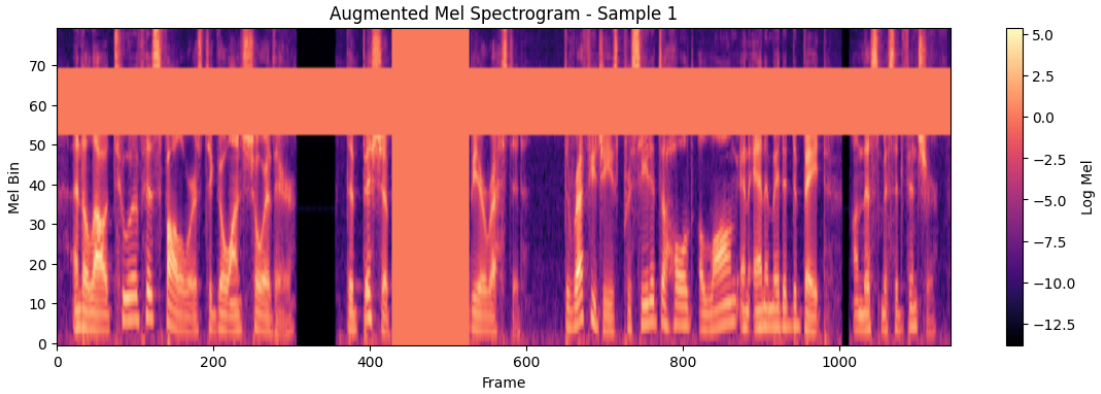

Figure 5-7: Spec Augmentation with Time Warp, Frequency and Time Masking = 20

Hyper Parameters of Spectrogram Augmentation

Table 5-2: Spectrogram Augmentation Parameters

Parameter	**Value**
Frequency Masks ($freq\_masks$)	2
Time Masks ($time\_masks$)	5
Frequency Mask Width ($freq\_width$)	27
Time Mask Width ($time\_width$)	0.05

5.1.4 Batching using Collate Function

We have a collate function which operates on batch of variable length sequential item. The collate function takes a batch of sequences and applies padding to each sequence, ensuring they all match the length of the longest sequence in the batch. Additionally, the collate function outputs three components: the padded sequence, the original length of each sequence before padding, and the corresponding label for each sequence. In deep learning packages such as PyTorch, the inputs are fed in batches by data loaders. Speech data and their transcripts are normally of unequal length. This operation makes sure that all sequences in a batch have been padded to the same length making them be easily processed by the model.

Listing 2: Collate Function

```
def collate_fn(batch):
    specs, transcripts,waveforms = zip(*batch)
    max_len = max(spec.shape[-1] for spec in specs)
    padded_specs = [
        F.pad(spec, (0, max_len - spec.shape[-1]))
```

```
        for spec in specs
    ]
    batch_specs = torch.stack(padded_specs)
    return batch_specs, transcripts,waveform
```

5.1.5 Encoder

i. **Input Features**

The input features are 80-dimensional and are provided to the encoder. These features are mel-spectrogram representations derived from raw audio signals and capture relevant spectral characteristics required for speech recognition.

ii. **Output Features**

The output feature dimension is set to -1, meaning it remains consistent with the internal model dimension (d_{model}), which is 176. As a result, the encoder outputs 176-dimensional feature representations.

iii. **Number of Layers**

The encoder consists of 16 stacked Conformer blocks. Each block integrates self-attention, convolutional modules, and feed-forward networks to effectively model both local and global sequential dependencies.

iv. **Model Dimension**

The internal representation dimension of the encoder is set to 176. This dimension defines the size of the hidden states used throughout the model.

v. **Subsampling Method**

Temporal resolution reduction is achieved through strided subsampling. This method skips time steps to reduce computational cost while preserving essential information in the sequence.

vi. **Subsampling Factor**

The input sequence length is reduced by a factor of 4. For instance, an input sequence of 100 time steps is subsampled to 25 time steps.

vii. **Subsampling Channels**

A convolutional subsampling layer with 176 channels is applied. This ensures that the feature dimension after subsampling matches the model's internal dimension.

viii. **Feed-Forward Expansion Factor**

Each Conformer block includes a feed-forward network that expands the input dimension by a factor of 4. This expansion allows the model to learn richer representations before projecting back to the original dimension.

ix. **Self-Attention Mechanism**

The encoder employs self-attention with relative positional encoding. This mechanism captures relative distances between tokens, enhancing the model's ability to handle long input sequences effectively.

x. **Number of Attention Heads**

The multi-head self-attention mechanism uses 4 attention heads. This allows the model to focus on multiple parts of the input sequence simultaneously.

xi. **Attention Context Size**

The attention mechanism considers the entire sequence, indicated by the unbounded context setting $[-1,-1]$. This enables global context modeling, which is crucial for automatic speech recognition tasks.

xii. **Attention Scaling**

Dot-product attention scores are scaled by dividing by the square root of the model dimension. This scaling improves numerical stability by preventing excessively large attention values.

xiii. **Untied Biases**

Biases used in relative positional encoding are untied, meaning they are learned separately for each attention head. This provides greater flexibility in learning position-specific representations.

xiv. **Convolutional Kernel Size**

Each Conformer block includes a depthwise convolutional layer with a kernel size of 31. This large kernel size is effective in capturing local temporal dependencies in speech signals.

xv. **Overall Dropout Rate**

A dropout rate of 0.1 is applied across the model to reduce overfitting and improve generalization during training.

xvi. **Attention Dropout Rate**

The self-attention mechanism uses a dropout rate of 0.1. This regularizes attention weights, enhances generalization, and supports effective modeling of long-range dependencies in speech sequences.

Table 5-3: Encoder Configuration Parameters

Parameter	**Value**
Input Feature Dimension ($feat\_in$)	80
Output Feature Dimension ($feat\_out$)	-1 (equals d_{model})
Number of Layers (n_layers)	16
Model Dimension (d_{model})	176
Subsampling Method	Striding
Subsampling Factor	4
Subsampling Convolution Channels	176
Feed-Forward Expansion Factor	4
Self-Attention Model	Relative Positional Encoding
Number of Attention Heads (n_heads)	4
Maximum Positional Embedding Length	5000
Convolutional Kernel Size	31
Dropout Rate	0.1

5.1.6 Decoder

The decoder component in this ASR system is implemented using NVIDIA NeMo's `ConvASRDecoder` class. It serves as the final classification layer that maps high-level acoustic representations to linguistic tokens. The decoder follows a Connectionist Temporal Classification (CTC) framework, which enables effective sequence-to-sequence alignment without requiring explicit frame-level supervision.

Input and Output Dimensions

- The decoder receives 176-dimensional input features, which correspond to the output of the encoder's final layer. These features represent compact and context-rich acoustic embeddings produced after convolutional and transformer-based processing.
- The number of output classes (`num_classes`) is set to 1024. This includes 1023 subword tokens and one additional blank token required by the CTC loss for alignment flexibility during decoding.

Vocabulary Design

The vocabulary consists of 1023 subword units generated using Byte-Pair Encoding (BPE). This design provides a balance between vocabulary compactness and linguistic expressiveness, enabling efficient handling of both frequent and rare words.

Token Composition

The vocabulary includes a combination of special tokens and multiple levels of subword granularity to support robust speech-to-text modeling.

- **Special Tokens**
  The `<unk>` token represents unknown or out-of-vocabulary words and is assigned index 0. A word-boundary marker represented by `\u2581` is used to denote a preceding whitespace, allowing the model to learn word segmentation explicitly.
- **Subword Types**
  Character-level tokens include individual letters from `a` to `z`. Morpheme-level tokens capture common suffixes such as `-ing`, `-ed`, `-tion`, and `-ment`. Word-piece tokens represent frequently occurring syllables or partial words. Whole-word tokens correspond to high-frequency content words that appear often in the training data.

5.1.7 CTC Implementation

Architecture Components

- Single linear projection layer
- LogSoftmax activation for probability normalization
- No hidden layers, which is standard for CTC-based output decoding

CTC Loss Integration

- **Blank Token**: Implicitly included as the 1024th output class
- **Alignment**: Supports variable-length sequence alignment using dynamic programming
- **Gradient Flow**: Enables direct backpropagation through all valid time-aligned paths

5.1.8 Optimizer Configurations

The model was trained by the AdamW optimizer [Loshchilov & Hutter, 2017] [13] and a cosine annealing learning rate schedule. The hyperparameters of the optimization were chosen very carefully with respect to the empirical observations and the best practices that are applicable in the domain of speech recognition.

Optimizer: AdamW

- **Learning Rate**: 5×10^{-4}, reduced from the pre-training learning rate of 1×10^{-3}. This reduction allows finer parameter updates during fine-tuning, preserving previously learned representations while adapting to the target domain.
- **Betas**: $[0.9, 0.98]$. These momentum parameters control the exponential decay rates of the first and second moment estimates. The chosen values follow Transformer conventions, with a slightly higher second-moment decay compared to the commonly used value of 0.999, making them better suited for speech data.

The model architecture is relatively compact, which reduces the risk of overfitting due to a smaller parameter count. Additional implicit regularization is achieved through extensive data augmentation using SpecAugment **park2019specaugment**. The reduced learning rate during fine-tuning further constrains parameter updates, while explicit dropout within the model provides additional regularization.

Learning Rate Schedule: Cosine Annealing with Warmup

- **Scheduler**: `NoamAnnealing`. The learning rate increases linearly during the warmup phase and subsequently decays proportionally to the inverse square root of the training step. This scheduling strategy is particularly effective for Transformer-based architectures.
- **Warmup Steps**: 10,000. During the initial warmup phase, the learning rate is gradually increased from near-zero to its peak value, which helps prevent optimization instability in the early stages of training.
- **Minimum Learning Rate**: 1×10^{-6}. As training progresses, the learning rate decays toward this lower bound, enabling stable convergence in later stages without disrupting previously learned representations.

Table 5-4: Optimizer and Learning Rate Scheduler Configuration

Parameter	**Value**
Optimizer Name	AdamW
Learning Rate (lr)	2.0
Betas (β_1, β_2)	[0.9, 0.98]
Weight Decay	0
Scheduler Name	NoamAnnealing
Model Dimension (d_{model})	176
Warmup Steps	10000
Warmup Ratio	None
Minimum Learning Rate (min_lr)	1×10^{-6}

5.2 SLM

5.2.1 Dataset

The model is trained on Alpaca dataset [14] which is an instruction following dataset which consists of about 52,000 examples of instructions with 4 fields.

- **Instruction:** A description of task to be performed
- **Output:** Optional contextual or additional information for the task
- **Text:** A formatted combination of all components using a standardized template

This dataset is then prepared for training by selecting instruction, input and output columns and changing the format into the dictionary which created the proper structure for training.

5.2.2 Data Preparation and Pipeline

The dataset is processed to support instruction fine-tuning. The pipeline is designed to handle variable-length sequences. We use a custom instruction Dataset class which pre-tokenizes the data by concatenating the instruction, input, and expected output into a single continuous format:

```
Instruction + Input + ###Response: output
```

A custom collate function is used to construct training batches where sequences are

padded to the maximum length in the batch using the $endoftext$ token with ID 50256. The target values for padded positions are set to an ignore index of -100 so the model does not learn from padding tokens and excludes them from loss calculation.

5.2.3 Loading weight and Training configuration

The model is initialized with pre-trained GPT-2 weights. We implement a custom loading mechanism to map external parameters to the specific PyTorch architecture, where weights are assigned to their corresponding layers. The token embeddings (wte) and positional embeddings (wpe) are directly mapped to the model embedding layers. The pretrained attention weights (c_{attn}) are split along the last dimension into three distinct components: Query, Key, and Value (Q, K, V). The weight matrices for the attention and feed-forward layers are transposed to match PyTorch linear layer expectations.

To train the model, the AdamW optimizer is used to properly handle weight decay, and a low learning rate is applied to fine-tune the model without destabilizing learned representations. The dataset is partitioned using an 85/10/5 split, where training is performed on 85% of the dataset, testing on 10%, and validation on 5%.

Table 5-5: Hyperparameters used for model training

Hyperparameter	**Value**
Batch Size	8
Learning Rate	5x10-5
Optimizer	AdamW
Number of Epochs	5
Weight Decay	0.1
Max Sequence Length	1024 tokens

The training loop evaluates the model on validation set every 5 steps to track convergence and prevent overfitting.

5.2.4 BPE Tokenizer

Character level vocabulary is built including all ASCII characters where a special whitespace character Ġ is used. The merging rule is used to combine symbol pairs into sub word tokens. During training, the tokenizer maps each character to its ID and finds the most adjacent pair and replaces occurrences of the pair with new token ID and records the merge until desired vocabulary size is reached. The existing encoder.json vocabulary and vocab.bpe are rules of merging list which assigns each merge a rank so that lower pairs are merged first. During encoding, each chunk is looked up in the vocab

and if the chunk is not found then it is passed to the BPE merger which is split into individual character level token IDs then the merging is done using merge rules from vocab.bpe with accordance to its rank. This process is repeated until no further merges can be performed. The decoding reverses the process converting Ġ back into real spaces and concatenating tokens into readable text.

Listing 3: Output of BPE Tokenizer on a sample text

```
Input Text : We study in IOE Thapathali #2@ located
near Maitighar Mandala.

Token id: [1135, 2050, 287, 24418, 36, 536, 499, 776, 7344,
1303, 17, 31, 5140, 1474, 285, 4548, 394, 283, 6855, 6081]

Tokenized text: We  study  in  IO E  Th ap ath ali  # 2 @
located near  m ait igh ar  mand ala
```

5.2.5 Data Embedding

Text Embedding

The token IDs are converted into embedding vectors. First the embedding weights are initialized with random values which get changed during the training process. Token embedding provides meaning to the ids. During training tokens dimensions will change so tokens having similar meaning are near to each other in vector space. Embeddings are used to find meanings among the words and not see words as unrelated, random weights. To create embedding vectors a matrix of size $vocab\_size * output\_dim$ is created. If $vocab\_size = 6$ and $output\_dim = 3$ then,

```
tensor([[-0.25091976,  0.90142861,  0.46398788],
        [ 0.19731697, -0.68796272, -0.68801096],
        [-0.88383278,  0.73235229,  0.20223002],
        [ 0.41614516, -0.95883101,  0.9398197 ],
        [ 0.66488528, -0.57532178, -0.63635007],
        [-0.63319098, -0.39151551,  0.04951286]], requires_grad=True)
```

Positional Embedding

Self attention mechanism does not have a notion of position or order of tokens within a sequence. Absolute positional embedding is used which is optimized during the training process. For each token ids a vector is created which is than added to previous token embedding and passed to the model.

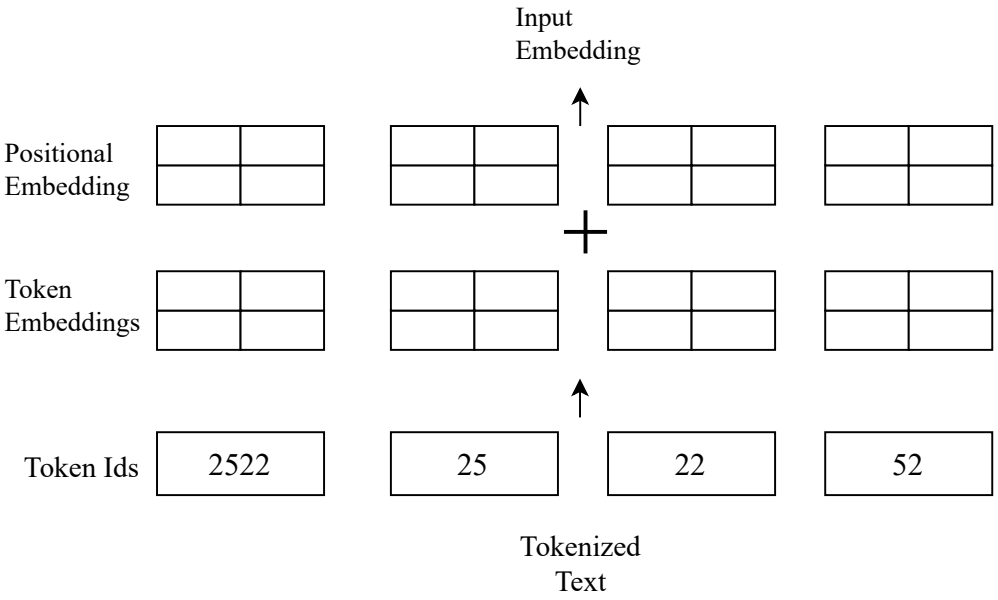

Figure 5-8: Token and Positional Embedding

5.2.6 Masked Multi-Head Attention Mechanism

The Multi-Head Attention module uses a casual scaled dot product attention mechanism with multiple heads to create a context vector.

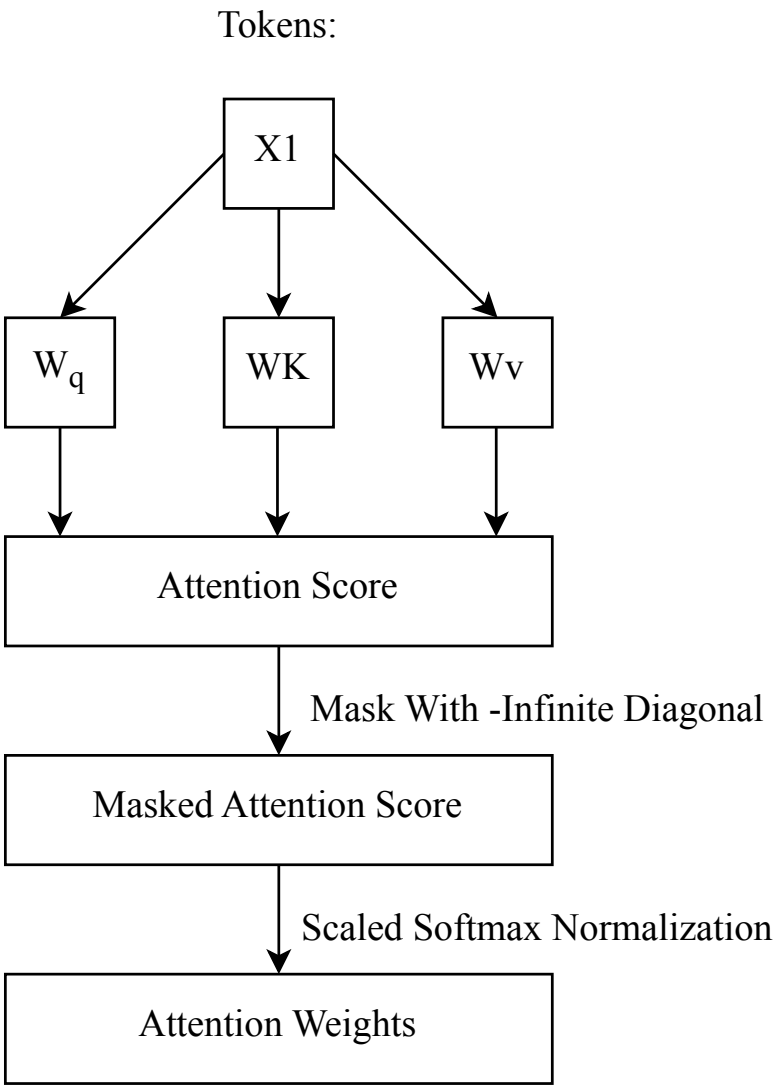

Figure 5-9: Attention Mechanism

- **Initialization and projection:** The input tensor has shape (B, T, d_{in}) where B is the batch size and T is the sequence length. The output dimension d_{out} should be divisible by the number of heads h, i.e., $d_k = \frac{d_{out}}{h}$ The first three linear layers $W_{\text{query}}, W_{\text{key}}, W_{\text{value}}$ are initialized to project input embeddings into query, key, and value vectors. A causal mask using an upper triangular matrix is applied, ensuring that during training the model attends only to previous positions, preventing

information leakage.

$$Q = XW_Q \quad K = XW_K \quad V = XW_V$$

The data is projected once and reshaping of resulting tensors is done instead of using separate layers for each head.

- **Scaled dot-product attention:** First the raw attention scores are calculated by performing matrix multiplication between queries and the transposed keys. To prevent information leakage a casual mask M an upper triangular matrix of negative infinity is used with scores. Then the attention weights are computed by scaling the scores by $\sqrt{d_k}$ and applying a Softmax function:

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^T}{d_k} + M\right)V \tag{5-1}$$

The scaling factor $\sqrt{d_k}$ is critical here to counteract the effect of large dot products pushing the Softmax function into regions with extremely small gradients.
After applying the attention weights to the value vectors V, the resulting context vectors from all h heads are transposed and concatenated to restore the original sequence shape (B,T,d_{out}). Finally, a linear output projection W is applied to mix the concatenated representations across heads, yielding the final output of the module.

5.2.7 Model Creation

Layer Normalization and GELU activation

For layer normalization which normalizes activations across the embedding dimension for each token independently to compute the mean and variance along the last dimension.

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y = \gamma\hat{x} + \beta \tag{5-2}$$

This feed network uses the Gaussian Error Linear Unit (GELU) activation. It introduces non-linearity while allowing small negative values to pass smoothly.

$$\text{GELU}(x) = 0.5x\left[1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right] \tag{5-3}$$

Table 5-6: Model Hyperparameters

Parameter	**Value**
Vocabulary Size ($vocab\_size$)	50257
Context Length ($context\_length$)	1024
Dropout Rate ($drop\_rate$)	0.0
QKV Bias (qkv_bias)	True
Embedding Dimension (emb_dim)	768
Number of Layers (n_layers)	12
Number of Heads (n_heads)	12

Transformer block and model

The first input goes through layer normalization. The feed forward part consists of two linear layers. The first expands the embedding dimension to 4 times its size, applies GELU and projects the original embedding dimension towards the second linear layer. Which is followed by multi-head attention sub-layer. A dropout is applied to the attention output and a residual connection adds a shortcut. This process is repeated multiple times to form a core transformer stack. The model begins with a token embedding layer and positional embedding of the same dimensions. Which then flows to the stack of n_layers of the transformer block. A last layer normalization is applied and linear output head is used that projects the embeddings to a vocabulary size. The result logits represents unnormalized probabilities over next token for each position in the sequence.

6 RESULT AND ANALYSIS

6.1 ASR

6.1.1 Word Error Rate (WER) Comparison and Analysis

The fine-tuned model has only 13.2 million parameters, which is about 4x to 8x times fewer than smaller architectures (e.g., Conformer-M, 50 million parameters) and larger ones (e.g., DeepSpeech 2, 110 million parameters). This compactness makes the architecture very compatible with edge devices and low-latency applications. The encoder is 98.6% of the total number of parameters, meaning that it washes up all the model capacity, the decoder is insignificant, consisting of only 181K parameters. This distribution is consistent with CTC based structures that avoid the use of complicated auto-regressive decoders, which limit both memory size and inference complexity.

The achieved WER of 6.14% per cent is higher than the performance of higher-ranking models, most of which use three to eight times the number of parameters and large amounts of data augmentation, nevertheless, it is a strong performance-per-parameter trade-off. In the case of a situation where constraints in model size, power consumption, or real-time response are involved, the existing model provides a realistically compromised state. The use of a CTC-BPE hybrid output scheme avoids the use of outside language models during decoding, which simplifying the scheme, rather than the many transducer or attention-based models, retains a reasonable degree of sub-word level generalization.

Model	WER	Parameters
Deep Speech 2	5.33	110M
Gated ConvNets	4.8	100M
Seq-to-Seq Attention (LSTM Based)	3.82	50M–100M
Convolution Speech Recognition	3.26	30M–70M
wav2vec-wav2letter	2.7	30M / 50M
Transformer	2.6	60M–100M
Squeezeformer (L)	2.47	50M
LSTM Transducer	2.23	40M–80M
Conformer (M)	2.0	50M
Stateformer	1.76	40M–60M
Our Model	6.14	13.2M

Table 6-1: Word Error Rate (WER) and parameter count of models trained on the LibriSpeech dataset

6.1.2 Qualitative Analysis

To illustrate examples of model predictions on held-out utterances, below there are three representative samples, each paired with its ground-truth reference, the model's hypothesis, and the corresponding Word Error Rate (WER):

Reference	Hypothesis	WER
ON THE SIXTH OF APRIL EIGHTEEN THIRTY THE CHURCH OF JESUS CHRIST OF LATTER DAY SAINTS WAS FORMALLY ORGANIZED AND THUS TOOK ON A LEGAL EXISTENCE	on the sixth of april eighteen thirty the church of jesus christ of later saints was formerly organized and thus took on a legal existence	0.11
ITS ORIGIN WAS SMALL A GERM AN INSIGNIFICANT SEED HARDLY TO BE THOUGHT OF AS LIKELY TO AROUSE OPPOSITION	its origin was small a germ an insignificant seed hardly to be thought of as likely to arouse opposition	0.0
INSTEAD OF BUT SIX REGULARLY AFFILIATED MEMBERS AND AT MOST TWO SCORE OF ADHERENTS THE ORGANIZATION NUMBERS TODAY MANY HUNDRED THOUSAND SOULS	instead of but six regularly affiliated members and at most two score of adherents the organization numbers to day many hundred thousand souls	0.09

These examples demonstrate that the model accurately captures most of the semantic content. Errors are primarily lexical (e.g., “latter” vs. “latter day”, “to day” vs. “today”) or morphological, which are common in CTC-based systems without external language models. Notably, the model achieves perfect transcription on complex formal language in the second example, highlighting its robustness on well-articulated speech

6.1.3 Training Dynamics

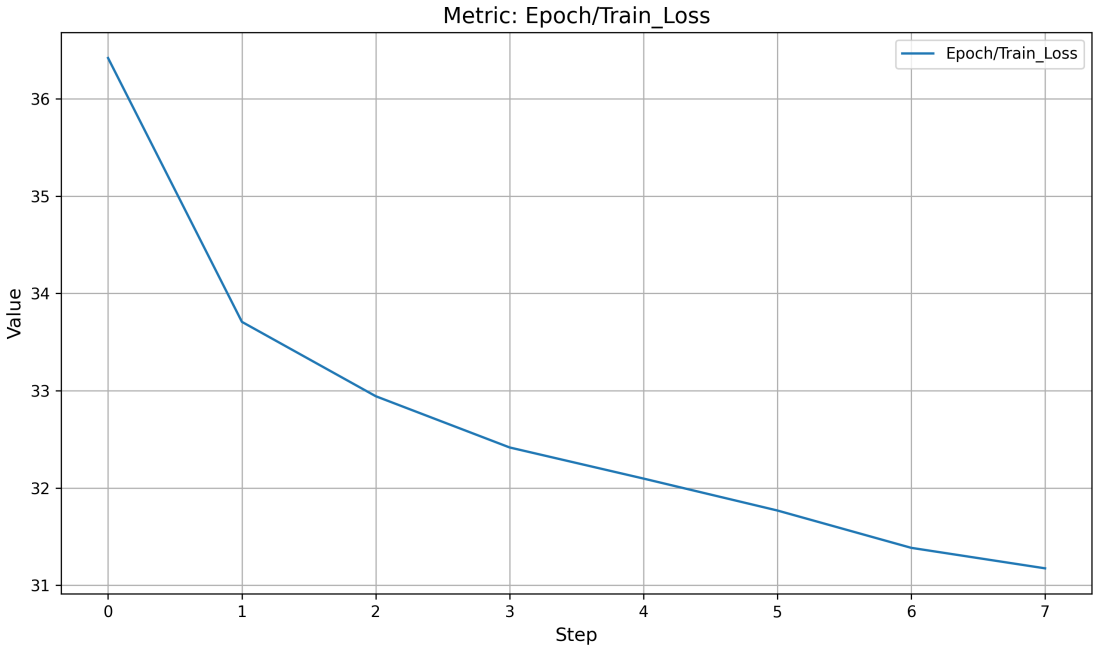

Figure 6-1: Epoch-wise Training Loss

Training Loss (Figure 6-1) decreases steadily from 36.2 to 31.2, indicating consistent learning.

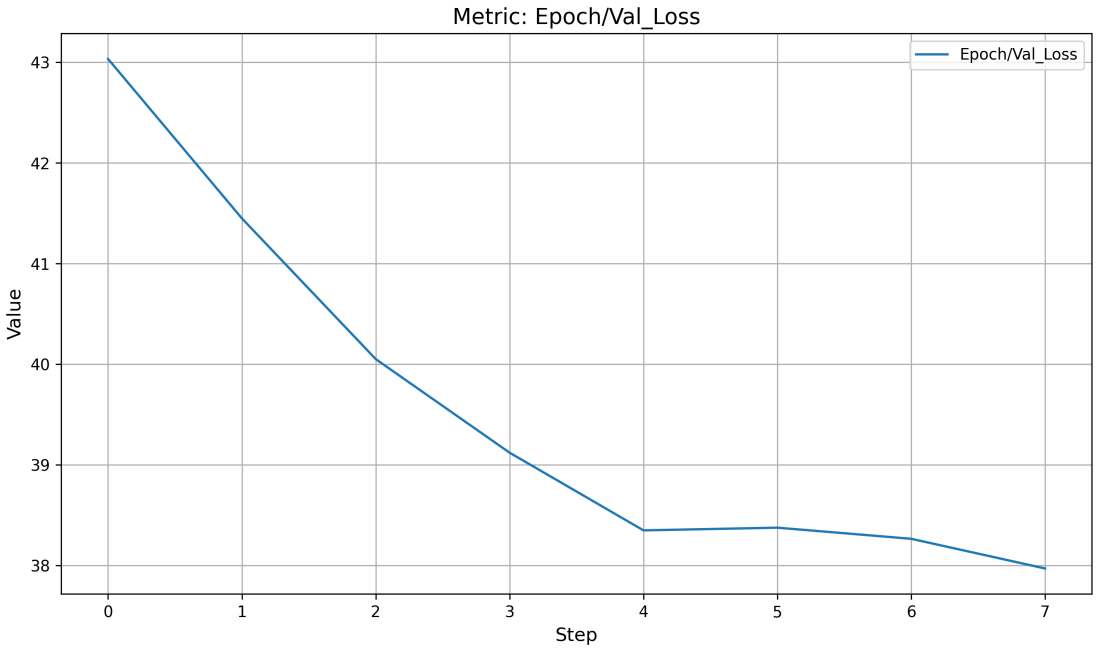

Figure 6-2: Epoch Wise Validation Loss

Validation Loss (Figure 6-2) shows a similar downward trend, reaching 37.9 at epoch

7, with only minor fluctuations (e.g., a small uptick at epoch 5), suggesting good generalization without severe overfitting.

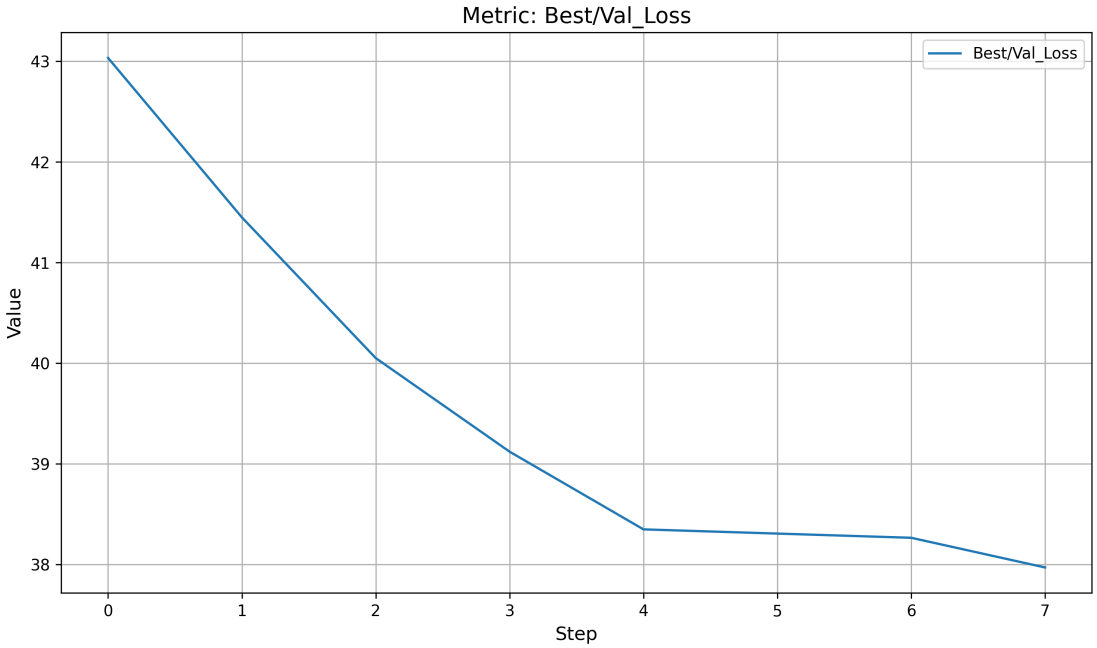

Figure 6-3: Best Validation Loss (Cumulative Minimum)

The Best Validation Loss (Figure 6-3) is monotonically non-increasing, confirming that the best checkpoint improves over time.

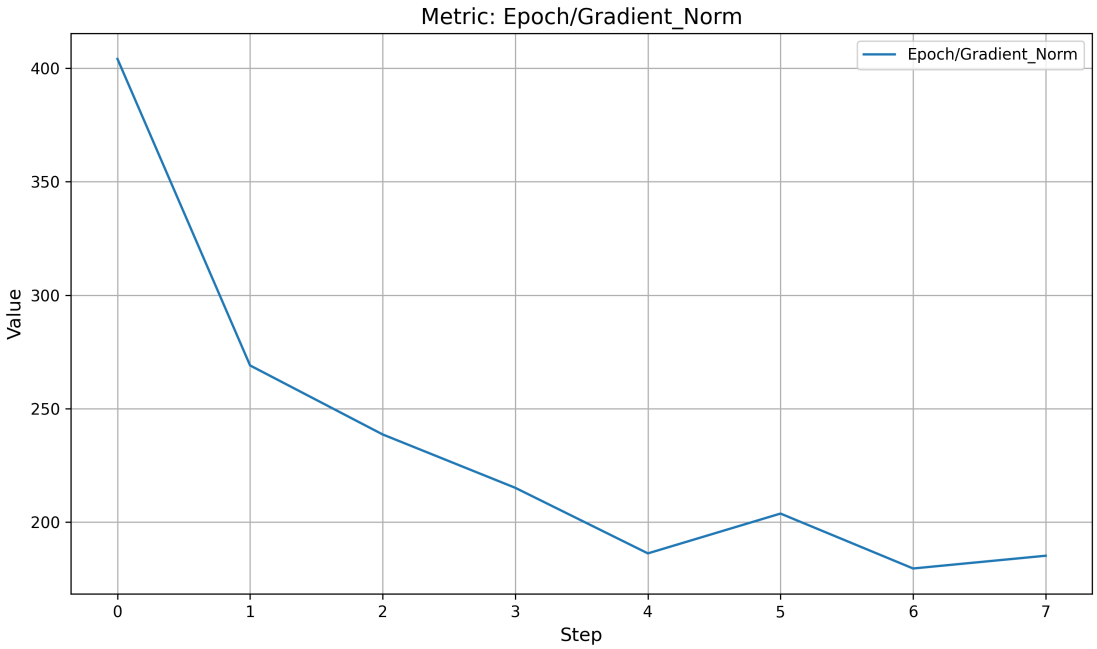

Figure 6-4: Gradient Norm per Epoch

The Gradient Norm (Figure 6-4) drops rapidly in early epochs and stabilizes around 180–185, signaling convergence and stable optimization. Collectively, these curves confirm a stable, well-behaved training process with no signs of divergence or instability.

6.2 LLM

6.2.1 Inference

Before Training

- **Input text:** What type of cloud is typically associated with thunderstorms?
- **Output:** What type of cloud is typically associated with thunderstorms? imaginaryonetolution Mortgage TT remember gard ACTIONSussedOND repeEdge Storage severerelease

After loading weights

- **Input text:** What type of cloud is typically associated with thunderstorms?
- **Output:** What type of cloud is typically associated with thunderstorms? Why did lightning get it going down in Oklahoma?" What sort of cloud was seen on September 19 that led to reports

After instruction finetuning

```
Below is an instruction that describes a task. Write a response that
appropriately completes the request.
Rewrite the sentence using a simile.
### Input:
The car is very fast.
Correct response:
>> The car is as fast as lightning.
Model response:
>> The car is as fast as a bullet.
---------------------------------------
Below is an instruction that describes a task. Write a response
that appropriately completes the request.
### Instruction:
What type of cloud is typically associated with thunderstorms?
Correct response:
```

```
Model response:
>> The type of cloud associated with thunderstorms is a cumulus cloud.
-------------------------------------
Below is an instruction that describes a task. Write a response
that appropriately completes the request.
### Instruction:
Name the author of 'Pride and Prejudice'.
Correct response:
>> Jane Austen.
Model response:
>> The author of 'Pride and Prejudice' is Jane Austen.
```

6.2.2 Evaluation Metrics

BLEU Score Analysis

BLEU is a common metric to evaluate machine-generated text with one or more reference texts. A single BLEU score calculates precision where zero being the worst and one being perfect. Typically for NLP tasks, a BLEU4 score of at least 0.4 is considered good, while a score of 0.6 and higher is exceptional. The comparative analysis of the performance regarding our model shows a relatively low score of 0.3061 for BLEU-4 and 0.45 for BLEU-1 score.

The BLEU-1 score checks for only single word overlaps while the BLEU-2 and higher grams look for matches of 2,3 and 4 word sequences respectively. These are obviously harder to match as generated text doesn't use the same phrasing as the reference texts. This is the reason we see a drop from .45 to .3061 from BLEU-1 to BLEU-4. The descending slope in the graph confirms that matching long sequences is tougher.

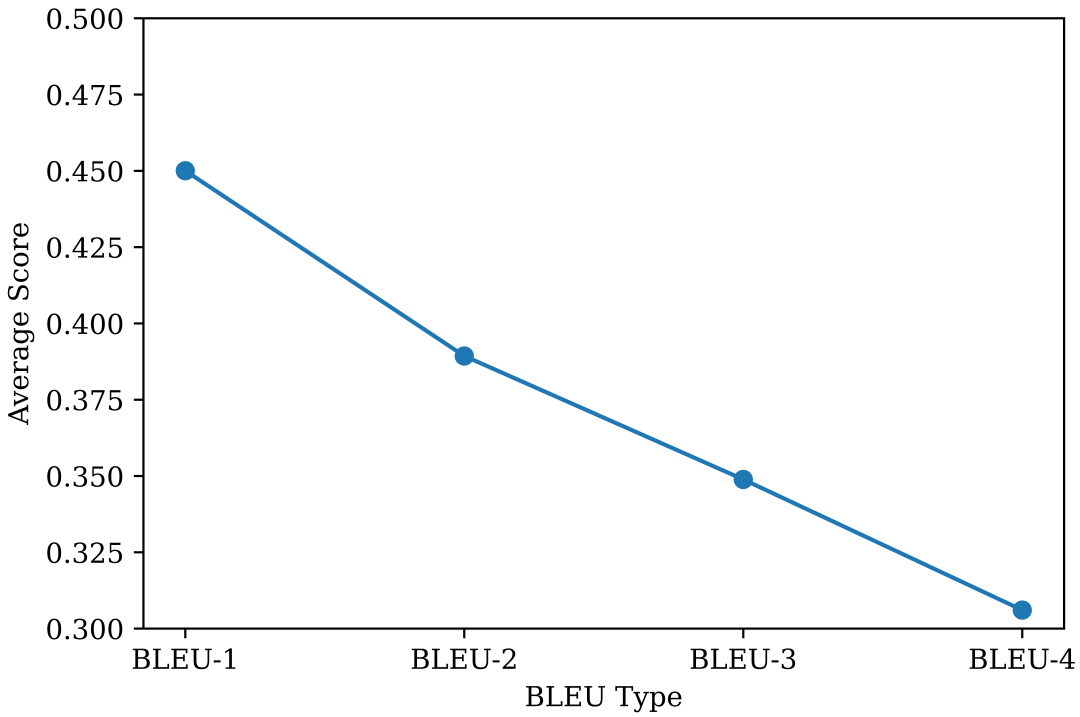

Figure 6-5: BLEU Score

ROUGE Score Analysis

ROUGE score is another evaluation metric used for NLP and text-generation related tasks. There are various ROGUE metrics like ROUGE-N, ROUGE-L etc. ROUGE-L evaluates the similarity between generated and reference text based on the length of their Longest Common Subsequence (LCS).

- **Precision (0.5965)**
  - About 59.65% of the words that are generated appeared also within the reference in correct order.
  - High precision indicates the text has correct information but could miss some parts of the reference.
- **Recall (0.6256)**
  - About 62.56% of the reference text is covered by the generated text
  - Higher recall indicates the generated text covers most of the reference text
- **F1 (0.5862)**
  - The F1 score balances the precision and recall
  - Here the 58.62% means that the generated text is moderately good in terms

of the coverage and correctness when it is compared with reference text

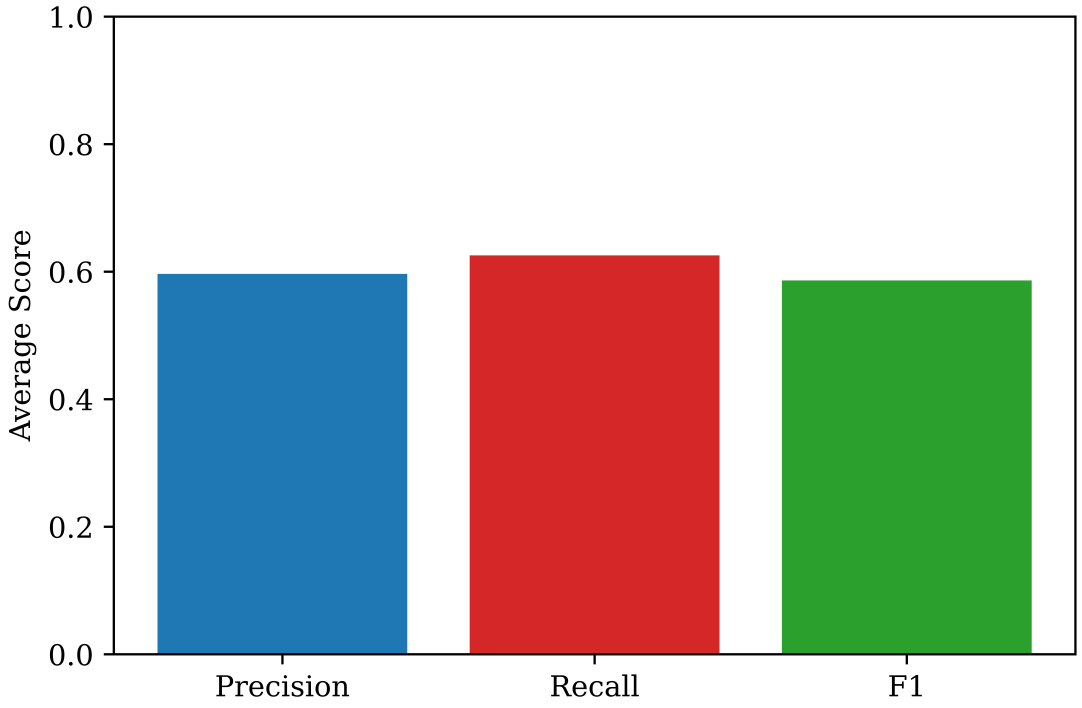

Figure 6-6: BLEU Score

Evaluation Metrics Table

Table 6-2: Evaluation Metrics

Metric	Value
Masked Loss	0.6722
Perplexity	1.96
BERTScore F1	0.9303
ROUGE-L	0.5862

6.2.3 BLEU vs ROUGE Score

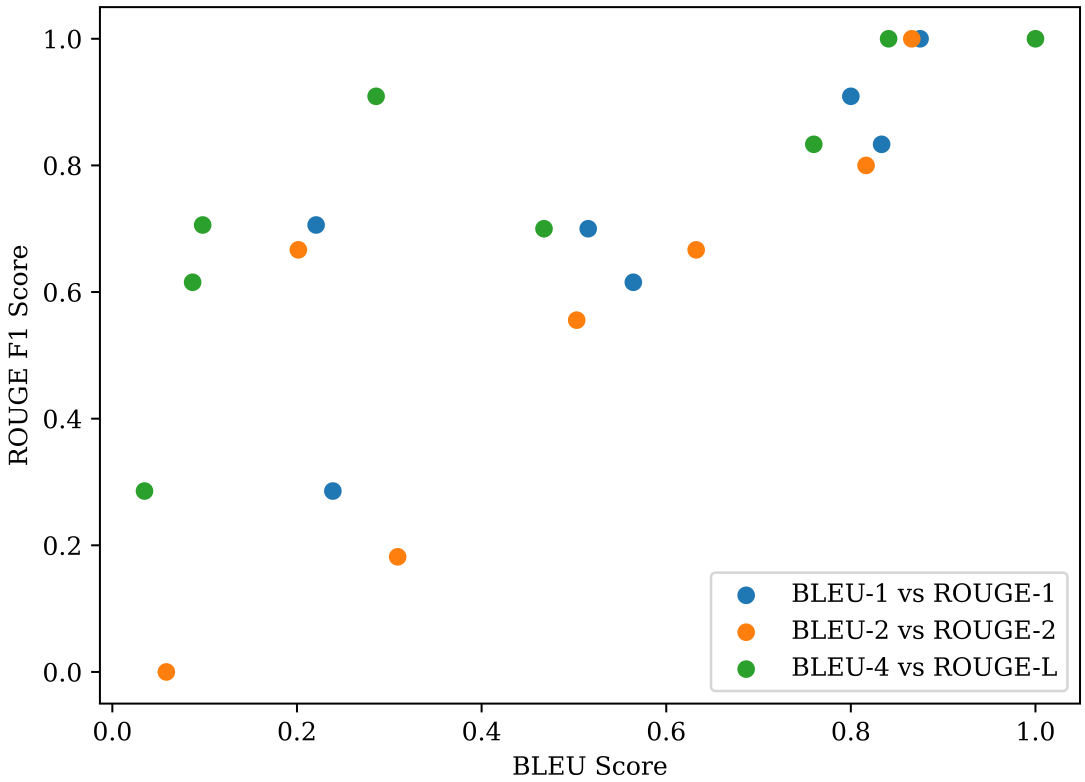

Figure 6-7: BLEU vs ROUGE scores for different n-grams

The scatter plot compares BLEU and ROUGE-F1 scores across three metrics: BLEU-1 vs. ROUGE-1 (red), BLEU-2 vs. ROUGE-2 (blue), and BLEU-4 vs. ROUGE-L (green). Overall, there is a positive correlation, higher BLEU scores means higher ROUGE scores.

Upward Sloping trend

Each cluster shows an upward trend, which shows the model performing well in BLEU(n-gram precision) also tends to perform well in ROUGE. Points near the diagonal typically show a system where both metrics are balanced. Points above the diagonal indicate a higher ROUGE Score than a BLEU score meaning the model prioritizes concise but incomplete outputs. Similarly, points below the diagonal indicate a higher BLEU score than ROUGE SCORE indicating the model focuses more on recall rather than precision.

Differences Among the Clusters

The orange cluster (BLEU-1 vs. ROUGE-1) is tightly aligned, as both focus on unigram overlap. The blue cluster (BLEU-2 vs. ROUGE-2) shows more variation, reflecting

sensitivity to bigram accuracy. The green cluster (BLEU-4 vs. ROUGE-L) is the most dispersed because BLEU-4 requires strict 4-gram matches, whereas ROUGE-L checks the longest common subsequence which might not align perfectly with exact four-gram overlap.

7 REMAINING TASKS

7.1 ASR Module

Application of Beam Search Decoder and Language-Model Fusion to Enhance WER

- The existing ASR solution makes use of a small model with limited contextualization capability, which can lead to less accurate transcriptions.
- To address this limitation, a beam search decoder is employed, enabling exploration of multiple possible output sequences rather than selecting only the most probable token at each decoding step.
- In addition, language-model (LM) fusion is incorporated during decoding to integrate external linguistic knowledge, helping to correct frequent errors such as homophones, missing words, and grammatical inconsistencies.
- During beam search, an n-gram language model trained on domain-relevant text data is integrated with the ASR outputs.
- This approach is expected to significantly reduce the Word Error Rate (WER) without increasing the size or computational complexity of the base ASR model.

Refinement on More Diverse Datasets

- The model will be fine-tuned on a broader and more diverse range of speech data to improve robustness and generalization.
- The datasets will include variations in accents, background noise, speech rate, and domain-specific vocabularies, such as conversational speech, technical terminology, and local dialects.
- Additional training data will be sourced from publicly available corpora such as Common Voice, Switchboard, and similar datasets.
- Careful fine-tuning will be performed using validation monitoring and early stopping to prevent overfitting, given the limited capacity of the model.
- The overall goal is to enhance the stability of the ASR system in real-world scenarios while preserving its lightweight design.

7.2 Small Language Model

Model Scaling, Parameter Reduction and Ablation studies

The model needs to be downscaled from its base model with a parameter count of 124M to create more lightweight variants by removing each layer to create a smaller model. To understand the contribution of specific architectural components the removing or altering of individual parts of the model and observing the loss and generation quality of the model These includes Attention head pruning to remove attention heads to determine the minimum number of heads which gives satisfactory result.Feed forward Network reduction and Layer wise removal of model.

Comparative Training

Every down-scaled model needs to undergo training to compare against the baseline by training the model architecture in the original dataset, monitoring training dynamics and see how evaluation metrics change with different configurations. The goal is to find the relationship between parameter count and model performance to visualize.

Impact Analysis and Metric Evaluation

The analysis of trade-off between model size and performance capability needs to be done. Find the relationship between parameter count and model performance to visualize the impact. Sensitivity analysis to quantify which components have the highest impact on model accuracy and performance shows which components are negotiable or not and the importance of individual components. Select the smallest possible architecture that shows satisfactory results.

8 APPENDICES

Appendix A : Gantt Chart

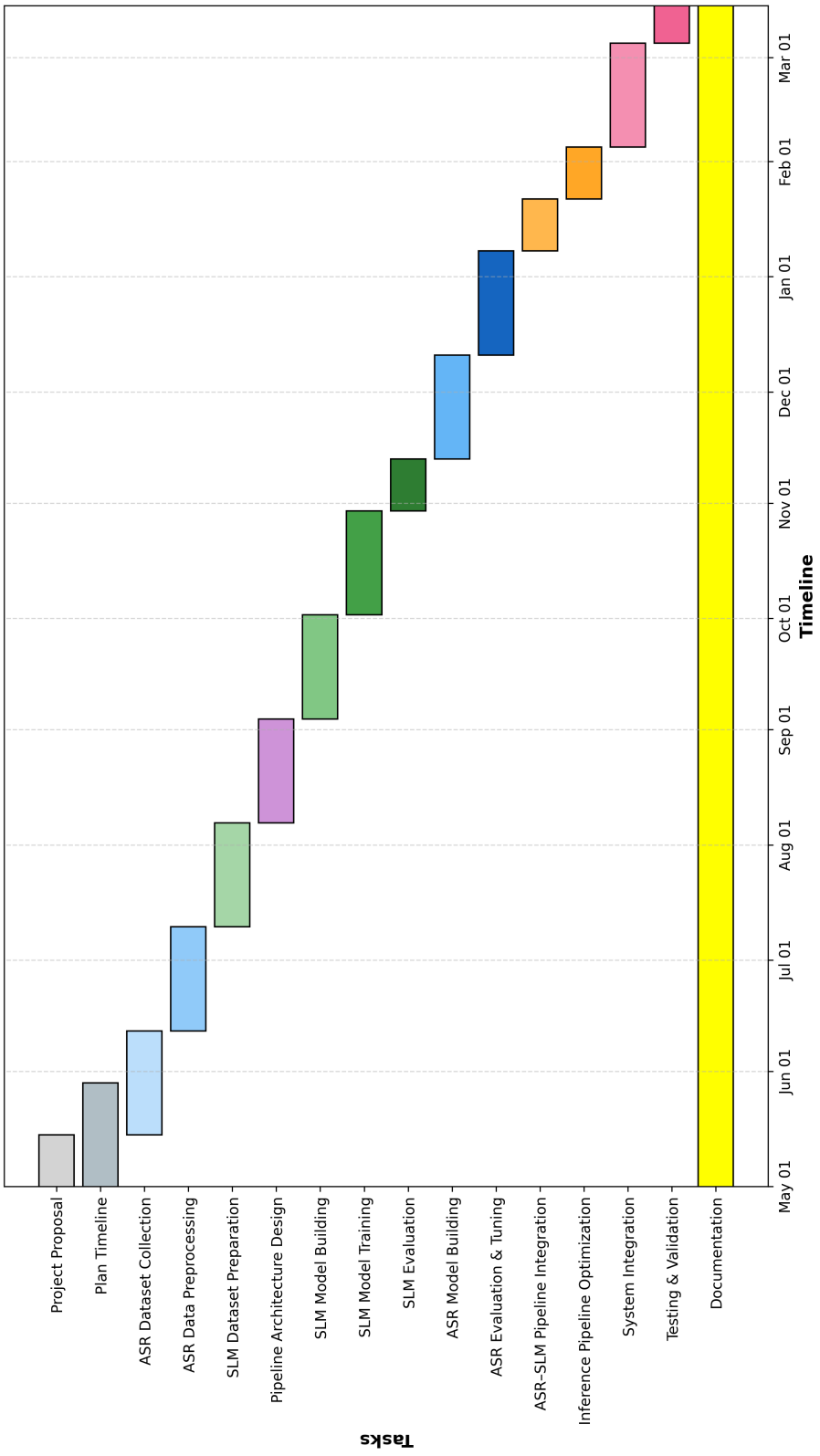

Figure 8-1: Gantt Chart

Appendix B : Project Budget

Table 8-1: Project Budget

SN.	**Particular**	**Estimated Cost (Rs)**
1	Printing and Documentation	5000
2	Cloud Computing	15000
3	Miscellaneous	2000
	Total	22000

Appendix C : Multi Head Attention Mechanism

Mutli Head Attention Mechanism Implementation

```
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout,
        num_heads, qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible
            by n_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("mask", torch.triu(torch.ones(
            context_length, context_length), diagonal=1))

        def forward(self, x):
        b, num_tokens, d_in = x.shape

        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        keys = keys.view(b, num_tokens, self.num_heads, self.
            head_dim)
        values = values.view(b, num_tokens, self.num_heads,
            self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads,
            self.head_dim)

        # Transpose: (b, num_tokens, num_heads, head_dim) to (b
            , num_heads, num_tokens, head_dim)
        keys = keys.transpose(1, 2)
```

```
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        attn_scores = queries @ keys.transpose(2, 3)

        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(attn_scores / keys.shape
            [-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = (attn_weights @ values).transpose(1, 2)
        context_vec = context_vec.reshape(b, num_tokens, self.
            d_out)
        context_vec = self.out_proj(context_vec)  # optional
            projection

        return context_vec
```

Appendix D : Model Architecture

Model Architecture

```
class TransformerBlock(nn.Module):
  def __init__(self, cfg):
    super().__init__()
    self.att = MultiHeadAttention(
    d_in=cfg["emb_dim"],
    d_out=cfg["emb_dim"],
    context_length=cfg["context_length"],
    num_heads=cfg["n_heads"],
    dropout=cfg["drop_rate"],
    qkv_bias=cfg["qkv_bias"])
    self.ff = FeedForward(cfg)
    self.norm1 = LayerNorm(cfg["emb_dim"])
    self.norm2 = LayerNorm(cfg["emb_dim"])
    self.drop_resid = nn.Dropout(cfg["drop_rate"])

  def forward(self, x):
    # Shortcut connection for attention block
    shortcut = x
    x = self.norm1(x)
    x = self.att(x)   # Shape [batch_size, num_tokens, emb_size
       ]
    x = self.drop_resid(x)
    x = x + shortcut  # Add the original input back

    # Shortcut connection for feed-forward block
    shortcut = x
    x = self.norm2(x)
    x = self.ff(x)
    x = self.drop_resid(x)
    x = x + shortcut  # Add the original input back

    return x

class GPTModel(nn.Module):
  def __init__(self, cfg):
    super().__init__()
    self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim
       "])
```

```
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["
            emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
        *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"], cfg["vocab_size"
            ], bias=False)

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=
            in_idx.device))
        x = tok_embeds + pos_embeds  # Shape [batch_size,
            num_tokens, emb_size]
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Appendix E : Feed Forward ,Normalization and Activation

Feed Forward ,Normalization and Activation

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)

        return self.scale * norm_x + self.shift

class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
        torch.sqrt(torch.tensor(2.0 / torch.pi)) *
        (x + 0.044715 * torch.pow(x, 3))
        ))

    class FeedForward(nn.Module):
        def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
        nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
        GELU(),
        nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

    def forward(self, x):
        return self.layers(x)
```

References

[1] J. Peng, Y. Wang, Y. Fang, *et al.*, *A survey on speech large language models*, 2025. arXiv: 2410.18908 [eess.AS]. [Online]. Available: https://arxiv.org/abs/2410.18908.

[2] X. Huang *et al.*, "Audiogpt: Understanding and generating speech, music, sound, and talking head," *arXiv preprint arXiv:2304.12995*, 2023.

[3] Y. A. e. a. Li, "Style-talker: Finetuning audio language model and style-based text-to-speech model for fast spoken dialogue generation," *arXiv preprint arXiv:2408.11849*, 2024.

[4] A. Gulati, J. Qin, C.-C. Chiu, *et al.*, *Conformer: Convolution-augmented transformer for speech recognition*, 2020. arXiv: 2005.08100 [eess.AS]. [Online]. Available: https://arxiv.org/abs/2005.08100.

[5] K. Wei, P. Guo, and N. Jiang, *Improving transformer-based conversational asr by inter-sentential attention mechanism*, 2022. arXiv: 2207.00883 [cs.SD]. [Online]. Available: https://arxiv.org/abs/2207.00883.

[6] J.-B. Cordonnier, A. Loukas, and M. Jaggi, *Multi-head attention: Collaborate instead of concatenate*, 2021. arXiv: 2006.16362 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2006.16362.

[7] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *OpenAI Blog*, 2018.

[8] J. Kaplan, S. McCandlish, T. Henighan, *et al.*, *Scaling laws for neural language models*, 2020. arXiv: 2001.08361 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2001.08361.

[9] H. Touvron, T. Lavril, G. Izacard, *et al.*, *Llama: Open and efficient foundation language models*, 2023. arXiv: 2302.13971 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2302.13971.

[10] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," *ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning*, vol. 2006, pp. 369–376, Jan. 2006. DOI: 10.1145/1143844.1143891.

[11] L. Bahl, F. Jelinek, and R. Mercer, "A maximum likelihood approach to continuous speech recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. PAMI-5, pp. 179–190, Apr. 1983. DOI: 10.1109/TPAMI.1983.4767370.

[12] Y. Lu, Z. Li, D. He, *et al.*, *Understanding and improving transformer from a multi-particle dynamic system point of view*, 2019. arXiv: 1906.02762 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1906.02762.

[13] I. Loshchilov and F. Hutter, *Decoupled weight decay regularization*, 2019. arXiv: 1711.05101 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1711.05101.

[14] R. Taori, I. Gulrajani, T. Zhang, *et al.*, *Stanford alpaca: An instruction-following llama model*, https://github.com/tatsu-lab/stanford_alpaca, 2023.

[15] L. Dong and B. Xu, *Cif: Continuous integrate-and-fire for end-to-end speech recognition*, 2020. arXiv: 1905.11235 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1905.11235.

[16] N. Moritz, T. Hori, and J. L. Roux, *Streaming automatic speech recognition with the transformer model*, 2020. arXiv: 2001.02674 [cs.SD]. [Online]. Available: https://arxiv.org/abs/2001.02674.

[17] E. Tsunoo, Y. Kashiwagi, and S. Watanabe, *Streaming transformer asr with blockwise synchronous beam search*, 2020. arXiv: 2006.14941 [eess.AS]. [Online]. Available: https://arxiv.org/abs/2006.14941.

[18] S. Raschka, *Build A Large Language Model (From Scratch)*. Manning, 2024, ISBN: 978-1633437166. [Online]. Available: https://www.manning.com/books/build-a-large-language-model-from-scratch.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[20] A. Baevski, Y. Zhou, A. Mohamed, and M. Auli, "Wav2vec 2.0: A framework for self-supervised learning of speech representations," *NeurIPS*, 2020.

[21] S. Watanabe, T. Hori, S. Karita, T. Hayashi, *et al.*, "Espnet: End-to-end speech processing toolkit," in *Proc. Interspeech*, 2018.

[22] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, *Attention is all you need*, 2023. arXiv: 1706.03762 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1706.03762.

[23] G. Ling, Z. Wang, Y. Yan, and Q. Liu, *Slimgpt: Layer-wise structured pruning for large language models*, Dec. 2024. DOI: 10.48550/arXiv.2412.18110.

[24] Y. Yang, Z. Cao, and H. Zhao, *Laco: Large language model pruning via layer collapse*, 2024. arXiv: 2402.11187 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2402.11187.

[25] S. Sheikholeslami, H. Ghasemirahni, A. H. Payberah, T. Wang, J. Dowling, and V. Vlassov, "Utilizing large language models for ablation studies in machine learning and deep learning," Mar. 2025, pp. 230–237. DOI: 10.1145/3721146.3721957.

[26] W. Ikeda, K. Yano, R. Takahashi, J. Lee, KeigoShibata, and J. Suzuki, “Layerwise importance analysis of feed-forward networks in transformer-based language models,” in *Second Conference on Language Modeling*, 2025. [Online]. Available: https://openreview.net/forum?id=oP3b5YBFoP.