BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

LARGE LANGUAGE MODELS IN SPEECH RECOGNITION

VELKÉ PŘEDTRÉNOVANÉ JAZYKOVÉ MODELY V ROZPOZNÁVÁNÍ ŘEČI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

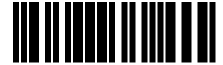
AUTHOR **MARTIN TOMAŠOVIČ**
AUTOR PRÁCE

SUPERVISOR **Ing. KAREL BENEŠ**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment



153477

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Tomašovič Martin**
Programme:	Information Technology
Title:	**Large Language Models in Speech Recognition**
Category:	Speech and Natural Language Processing
Academic year:	2023/24

Assignment:

1. Get acquainted with statistical language modeling and its application to speech recognition
2. Get acquainted with large pretrained langauge models (LLM)
3. Using a suitable dataset evaluate, how do LLMs improve transcriptions from publicly available ASR systems
4. Analyse under what conditions are these improvements achieved and how important is the size of the LLM

Literature:

- Zeping Min, Jinbo Wang: Exploring the Integration of Large Language Models into Automatic Speech Recognition Systems: An Empirical Study, 2023, https://arxiv.org/abs/2307.06530

Requirements for the semestral defence:
1, 2, progress on 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Beneš Karel, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	3.5.2024

// Abstract
This thesis explores the conditions under which a Large Language Model (LLM) improves Automatic Speech Recognition (ASR) transcription.

Specifically, the thesis focuses on n-best rescoring with masked and autoregressive language models. The n-best hypotheses are scored using LLM and then this score is interpolated with the scores from ASR. This approach is tested across different ASR settings and datasets.

Results demonstrate that rescoring hypotheses from Wav2Vec 2.0 and Jasper ASR systems reduces the error rate. LLM fine-tuning proves to be very beneficial. Smaller fine-tuned models can surpass larger non-fine-tuned ones.

The findings of this thesis broaden the knowledge of the conditions for LLM (autoregressive, masked) utilization in ASR rescoring. The thesis observes the influence of fine-tuning, normalization and separating scores from a CTC decoder on the decrease of word error rate.

Abstrakt
Táto práca má za cieľ preskúmať, v akých podmienkach veľké jazykové modely vylepšujú prepisy automatického rozpoznávania reči.

Konkrétne sa zameriava na preskórovanie n-najlepších hypotéz pomocou maskovaných aj autoregresívnych jazykových modelov. Pomocou nich sa každej hypotéze priradí skóre, ktoré sa následne interpoluje so skórami získanými zo systému prepisu reči. Tento postup som testoval naprieč datasetmi a rôznymi systémami pre prepis reči s rôznym nastavením.

Výsledky vykazujú, že preskórovanie znižuje mieru chybovosti hypotéz získaných z modelov Wav2Vec 2.0 a Jasper. Dotrénovanie modelov sa overilo byť veľmi prospešné pri danej úlohe. Menšie dotrénované modely prekonali väčšie nedotrénované pri preskórovaní.

Výsledky tejto práce prispievajú k lepšiemu porozumeniu, v akých podmienkach použiť jazykový model (autoregresívny, maskovaný) pri preskórovaní prepisov reči. Táto práca skúma vplyv dotrénovania, normalizácie a rozdelenia skóre z CTC dekodéra, na zníženie miery chybovosti slov.

Keywords
automatic speech recognition, n-best rescoring, large language model, masked language modeling, autoregressive language modeling

Kľúčové slová
automatické rozpoznávanie reči, preskórovnaie n-najlepších hypotéz, veľký jazykový model, maskované modelovanie jazyka, autoregresívne modelovanie jazyka

Reference
TOMAŠOVIČ, Martin. *Large language models in speech recognition.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Karel Beneš

Rozšírený abstrakt

Veľké predtrénované jazykové modely sa dnes používajú hlavne v rôznych úlohách spracovania prirodzeného jazyka. Otázne je, či sa dá použitím veľkých jazykových modelov v automatickom prepise reči dosiahnuť zlepšenie kvality prepisov.

Veľké jazykové modely sú natrénované na obrovskom množstve textu. Sú naučené, ako by mal korektný text vyzerať. Systémy automatického prepisu reči posielajú na výstup hypotézy s určitou pravdepodobnosťou. Predpoklad je, že veľké jazykové modely môžu vybrať lepšiu hypotézu, než tú, s najväčším skóre od systému pre prepis reči. Výskumy v danej oblasti sú nejednoznačné a preto sa oplatí dané viac preskúmať.

V práci skúmam prístup preskórovania n-najlepších hypotéz. Vyskúšal som rôzne jazykové modely, v kombinácii s tromi rôznymi systémami prepisov reči a tromi datasetmi. Medzi vybranými sú zástupcovia maskovaných a autoregresívnych jazykových modelov podobných veľkostí, aby sa ich dalo porovnať. Okrem iného, som testoval vplyv dotrénovania jazykový modelov na výsledok preskórovania.

Výsledné zistenia poskytujú informácie o efektivite konkrétnych nastavení jazykových modelov a systémov pre prepis reči. Tieto poznatky môžu pomôcť k implementácii preskórovania pomocou veľkých jazykových modelov v iných systémoch.

Proces prepisu nahrávky zvuku je nasledovný. Vstupom do systému je súbor so zvukom (typicky vo formáte WAV). Ten je vzorkovaný na 16 kHz. Zvukové nahrávky v experimentoch sú prevzaté z anglických datasetov, menovite: LibriSpeech, GigaSpeech a TED-LIUM. Tieto zahŕňajú čítanú aj spontánnu reč z rôznych tém.

Pre prepis reči z datasetov na hypotézy som použil nasledovné End-to-end systémy prepisu reči: Wav2Vec 2.0 Base 960h, Whisper Medium a STT En Jasper10x5dr. Všetky využívajú beam search pre vytvorenie viacerých hypotéz. Wav2vec a Jasper fungujú s CTC dekodérmi a malými n-gram jazykovými modelmi. Ja som počas dekódovania použil 4-gram KenLM model natrénovaný na LibriSpeech.

Upravil som CTC dekodér pre Wav2Vec, aby na výstup posielal skóre akustického modelu, skóre KenLM, a počet slov oddelene. Ďalej som upravil dekodér pre Whisper, aby na výstup posielal viacero hypotéz. Model Jasper a jeho dekodér som použil bez zmien v zdrojovom kóde programu.

Hypotézy zo systému prepisu reči som oskóroval pomocou maskovaných a autoregresívnych jazykových modelov. Skórovanie bolo uskutočnené súčtom pravdepodobností pre každý token v danej hypotéze. Maskované modely pri jednom vstupe predikujú jediný token pre danú hypotézu. Predikovaný token je token pod takzvanou maskou označený `[MASK]`. Aby sa oskóroval text, musí sa viackrát spustiť predikcia tokenu pod maskou, konkrétne x krát, kde x je počet tokenov v texte. Autoregresívne modely spracúvajú text zľava doprava. Text stačí prejsť raz. Ich nevýhoda je, že nemajú prístup k nasledovným tokenom.

Boli vykonané experimenty s Wav2Vec 2.0 s KenLM a lexikónom, iba s lexikónom a bez KenLM a lexikónu. Pri experimentoch s modelom Jasper bol jeho výstup dekódovaný s využitím KenLM. Najlepšie namerané zlepšenie WER je 4% absolútne.

V experimentoch som použil tieto maskované jazykové modely: BERT uncased vo veľkosti Base 110M a Large 340M, RoBERTa vo veľkostiach Base 125M a Large 355M. A tieto autoregresívne modely: GPT-2 s veľkosťami Base 137M a Medium 380M, TinyLlama 1.1B, Falcon 7B, Mistral 7B, MPT 7B, Llama2 o veľkostiach 7B a 13B.

Pre ďalšie ešte väčšie vylepšenie preskórovania, som dotrénoval modely BERT Base uncased a GPT-2 na texte z LibriSpeech datasetu. Modely sa po dotrénovaní zlepšili najmä pri preskórovaní LibriSpeech datasetu. Okrem týchto som dotrénoval model Llama2 7B s využitím LoRA na texte z datasetu GigaSpeech XL. Vyskúšal som tri rôzne nastavenia:

$r = 8$, $r = 32$ a $r = 128$. Tieto tri dotrénované Llama modely vykazujú konzistentné zlepšenie naprieč datasetmi.

Preskórovanie prebieha v poslednej časti spracovania zvuku. Pre skóre získané z akustického modelu, veľkého jazykového modelu, počet slov a v niektorých prípadoch skóre n-gramu, sa musia nájsť vhodné váhy. Po prenásobení jednotlivých skóre príslušnými váhami sa výsledky sčítajú. Daný súčet nazývam nové skóre textu a vyberá sa hypotéza s najlepším novým skóre.

V popísaných experimentoch som pre hyperparametre každého dekodéra a pre všetky váhy našiel najlepšie hodnoty. Váhy som hľadal najskôr prehľadávaním mriežky a následným polením intervalu okolo najlepšieho výsledku z prvej časti hľadania.

Zistil som, že daný postup bol v dvoch z troch (pre modely Whisper nie) systémov úspešný. Bola nájdená korelácia medzi veľkosťou jazykového modelu a zlepšením v prepisoch reči s drobnými výnimkami – GPT-2 a GPT-2 Medium. Okrem toho som našiel aj výnimky v prípade relatívne malého modelu TinyLlama, ktorý dosahoval podobné zlepšenie ako násobne väčšie modely.

Large language models in speech recognition

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Karel Beneš. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Martin Tomašovič
May 9, 2024

Acknowledgements

I would like to sincerely thank my supervisor Ing. Karel Beneš for his mentoring and help.

Computational resources were provided by the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Contents

1 Introduction **3**

2 Automatic speech recognition **4**
2.1 End-to-End systems 4
2.2 ASR evaluation metrics 4
2.3 Feature extraction 5
2.4 Connectionist temporal classification 5
2.5 Beam search 6
2.6 Patience 6

3 Language modeling **10**
3.1 Statistical language model 10
3.2 Metric 11
3.3 Text standardization 11
3.4 Tokenization and indexing 12
3.5 Encoding and embedding 12
3.6 Large Language Models 13
3.7 Transformer architecture 13
3.8 Positional encoding 15
3.9 Attention 16
3.10 Activation function 20
3.11 Residuals and Layer Normalization 21
3.12 Low-Rank Adaptation 22
3.13 Dropout 23
3.14 Language models in automatic speech recognition 23
3.15 Sentence scoring 24

4 Pre-trained models **26**
4.1 Byte Pair Encoding tokenizer 26
4.2 Wav2Vec 2.0 27
4.3 Whisper 28
4.4 Jasper 30
4.5 Autoregressive Large Language models 32
4.6 Masked Language Models 35

5 Experiments **37**
5.1 Datasets 38
5.2 Fine-tuning 40

5.3 CTC decoder edit . . . 42
5.4 Wav2Vec 2.0 experiments . . . 42
5.5 Jasper experiments . . . 45
5.6 Whisper experiments . . . 47
5.7 Experiments summary . . . 48

6 Conclusion **50**

Bibliography **51**

A Wav2Vec measurements **56**

B Jasper measurements **62**

C Whisper measurements **65**

D Difference in insertions, deletions and substitutions **72**

Chapter 1

Introduction

Automatic speech recognition (ASR) finds applications across various domains and industries. From simple internet browsing by saying “Hey Google” or “Hey Siri”, through writing court reports to emergency hot-line robots. The improvement is very welcome. One way to improve it is using language models that know what correct text should look like.

Generative artificial intelligence is one of the most groundbreaking innovations in the field of artificial intelligence in the last few years. It encompasses large language models. Which achieves state-of-the-art in multiple natural language processing tasks. These great results inspired research in applying large language models in other areas. One of them is automatic speech recognition.

The studies about improving speech transcriptions using large language models show varying results [19, 52, 29]. In this thesis, I joined this effort to improve automatic speech recognition systems using large language models. More precisely, by setting ASR systems to produce more than one possible transcription of a speech from an audio file and then using LLM to help ASR systems choose the best speech transcription.

This thesis aims to test how large language models influence ASR transcriptions. Find how important is the size of the large language model. Compare the performance of masked and autoregressive models in this task. Observe factors influencing the result of rescoring.

In chapter 2, ASR systems are described. The chapter 3 aims to explain how language models work and how they can be used with ASR systems from the previous chapter. Transformer architecture is also described here. Following chapter 4 is more practical, all ASR models and large language models used in experiments are described here. The experiments and datasets description is in chapter 5. It specifies experiments and describes the results. There is a summary of findings, and further steps are proposed, in the last chapter 6.

Chapter 2

Automatic speech recognition

Automatic speech recognition (ASR) is the process of transcribing speech from audio to text form using computer software. In the past, statistical models were used for ASR, in recent years they have been replaced with deep learning-based end-to-end systems [33].

2.1 End-to-End systems

End-to-end (E2E) models are composed of these parts: encoder, alignment and decoder. The encoder maps the input acoustic frame sequence into a higher-level representation. To encode the input recurrent neural networks (uni-directional or bi-directional), convolutional networks, or transformers can be utilized.

Captured signal usually contains noise. Noises may cause waveforms of two different words to be indistinguishable, therefore it would be difficult in further steps to correctly transcribe these words. The pre-processing aim is to reduce noise in the signal. Many different methods are used to do it. Some of them are normalization, pre-emphasis and other [27].

To find alignment between the acoustic frame sequence and the corresponding label sequence, in E2E there are multiple approaches: explicit alignment, implicit alignment, and attention-based E2E with alignment modeling. Connectionist temporal classification belongs to the explicit alignment finding approaches and is described further in this chapter.

All ASR models I used have an autoregressive decoder utilizing beam search. Autoregressive means, it predicts an output token based on the previous prediction.

To validate how well an ASR model is, there exist multiple metrics I focused on word error rate described in the following section.

2.2 ASR evaluation metrics

In ASR word error rate (WER) and sometimes character error rate (CER) are used to measure how good the speech transcriptions are. Mandarin or Japanese transcriptions are typically evaluated using CER.

$$WER = \frac{I + D + S}{N} \times 100 \tag{2.1}$$

WER computation is in Equation 2.1, where I is the number of insertions, D deletions, S substitutions and N is the number of words. The numerator in the equation is Levenshtein

distance. Levenshtein distance between two sequences is the minimum number of single-word edits – insertions, deletions or substitutions – required to change one word. CER has a similar definition, but Levenshtein distance is counted over characters and N is the number of characters [13].

2.3 Feature extraction

Some ASR systems need to preprocess audio into mel frequency. It works like the human ear's perception of sound. The lower frequencies have better resolution than higher frequencies.

In general, the frequency content of a speech signal over time is described by the power spectrum of the signal. Usually, the peaks in a spectrum relate to the formant frequencies. Formants are frequency peaks in the spectrum which have a high degree of energy. They are especially prominent in vowels. Each formant corresponds to a resonance in the vocal tract – the spectrum has a formant approximately every 1000 Hz. [2, 6].

Mel-Frequency Cepstral Coefficients (MFCC) extraction is depicted in Figure 2.1 inspired by [1]. It is done in the phase of feature extraction, which is a process of extracting hidden information from a raw data signal. It minimises discontinuities of the signal. In Equation 2.2 for MFCC, f denotes frequency and $mel(f)$ is the mel frequency [6].

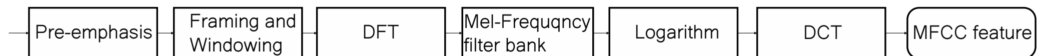


Figure 2.1: MFCC extraction process. The result is Mel-Frequency Cepstral Coefficients (MFCC). MFCC extraction is depicted in (Figure 2.1) taken from [1]. Hamming window is applied, to the input signal. Then Discrete Fourier Transform (DFT) is performed over the signal. In this step, the power spectrum is computed. The next step is mel frequency warping, where the numbers of coefficients are obtained. After that, the logarithm of the power spectrum is taken. In the end, the cepstral coefficients are then computed by transforming the log spectrum to the cepstral domain using an inverse Discrete Fourier Transform (IDFT).

$$mel(f) = 2595 \cdot log_{10}\left(\frac{1+f}{100}\right) \tag{2.2}$$

2.4 Connectionist temporal classification

Finding alignment between input and output letters or words is a very time-consuming task. To address this, a neural network sequence-to-sequence model with Connectionist temporal classification (CTC) [15] is utilized. The model finds alignment between input and output sequences of variable lengths.

The basic idea is that based on an input sequence, the model's outputs are interpreted as a probability distribution over label sequences. An objective function can be derived from this distribution. The objective function directly maximises the probabilities of the correct labellings. The network can be trained with backpropagation through time because the objective function is differentiable [15].

CTC works over frames, a single character may be in multiple frames, which causes character repetition in the output. This could be solved by merging characters that repeat.

However, words where characters repeat for example in words “hello” or “running”. To handle repeating characters in the output, there is a special token called a blank token. Tokens between two blank tokens are being merged [16].

2.5 Beam search

Search is the last step in ASR. For an observed source sequence, the search algorithm generates the most likely target sentence of unknown length. To get words and sentences from a sequence of probabilities obtained from previous steps, the decoder is needed. Common decoder choices are greedy search and beam search.

Beam search [33, 12] serves to select the best subset of hypotheses $\tilde{C}$ from all possible combinations of output sequences.

The inputs for beam search are symbols (characters and a blank symbol) and their corresponding log probabilities. The output from the algorithm is a list of transcriptions and their scores. The basic beam search works as follows. The best hypotheses set is constructed by processing input symbols from left to right and at each step retaining a fixed number (beam) of the candidates with the highest log probability. How many candidates to keep in each step is determined by the beam size parameter. When the end-of-sequence symbol occurs within the highest-scoring candidates and the transcription is added to the final transcription list, the beam size is reduced. The search stops when the beam size is equal to zero. Then the best transcription is chosen according to the highest normalized log probability.

The greedy search does not consider alternative hypotheses. This imposes degradation in decoding with longer target sequences. But the degradation is not big, in well-trained models it is minimal.

2.6 Patience

Widely used implementations of beam search – fairseq, Hugging Face’s Transformers – follow the first come, first served (FCFS) heuristic.

FCFS heuristic returns the best candidate when k finished candidates are found, where k is the beam size. In that moment it also discards all current, unfinished sequences. So breadth and depth of the beam search depend on beam size k. To separate the breadth and depth control the patience factor can be used. The patience factor modifies the stopping criterion and thereby changes the depth of the search. It controls how many finished candidates have to be found before terminating the decoding [22].

Algorithm 1 FCFS beam decoding with controlled patience factor p [22]. The common implementation can be considered in cases where p = 1. Line 17 shows where is the patience factor used in the algorithm. F_t: already completed sequences; B_t: beam of continuing sequences. H_t: expanded hypotheses before the top-k operation. The input sequence to *score* is omitted.

k: beam size, M: maximum length, V: Vocabulary, $score(\cdot)$: scoring function, p: patience factor

```
1:  B_0 ← {⟨0, BOS⟩}, F_0 ← ∅
2:  for t ∈ {1, . . . , M − 1} do
3:      H ← ∅, F_t ← F_{t−1}
4:      for ⟨s, y⟩ ∈ B_{t−1} do                              ▷ Expansion.
5:          for y ∈ V do
6:              s ← score(y ◦ y), H.add(⟨s, y ◦ y⟩)
7:          end for
8:      end for
9:      B_t ← ∅
10:     while |B_t| < k do                          ▷ Find top k w/o EOS from H.
11:         ⟨s, y⟩ ← H.max()
12:         if y.last() = EOS then
13:             F_t.add(⟨s, y⟩)                             ▷ Finished hypotheses.
14:         else
15:             B_t.add(⟨s, y⟩)
16:         end if
17:         if |F_t| ≥ k · p then                           ▷ Originally, p = 1.
18:             return F_t.max()
19:         end if
20:         H.remove(⟨s, y⟩)
21:     end while
22: end for
23: return F_t.max()
```

Research has shown that adjusting patience significantly improves the generation performance on text summarization, with an insignificant slowdown in generation [22]. My hypothesis is that adjusting patience could be beneficial in generating transcription hypotheses and thus improve the final transcription after rescoring the hypotheses with a large language model. Before rescoring experiments, I did measurements on 100 randomly selected samples from the GigaSpeech dev subset with various beam sizes and patience values.

In Figure 2.2, it can be seen that patience greater than one influences how many final hypotheses the algorithm outputs. The greater the patience, the more the final hypotheses. The dependency between the patience factor greater than one and the number of outputted hypotheses is linear. Also the bigger the beam size, the steeper the increase in hypotheses count. This effect is caused by patience's influence on the stopping criterion. The patience factor which is greater than one, moves the stopping condition, so the search finds more final candidates.

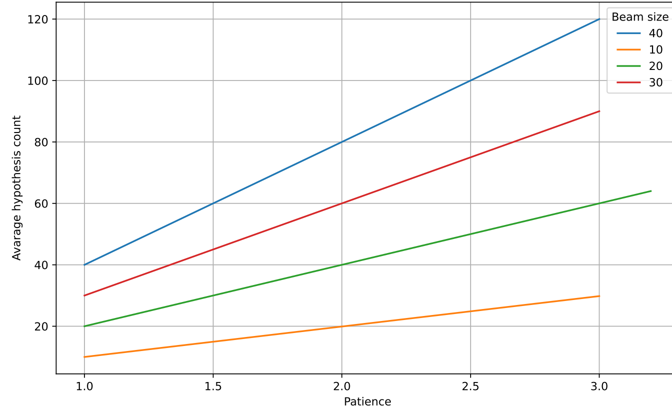
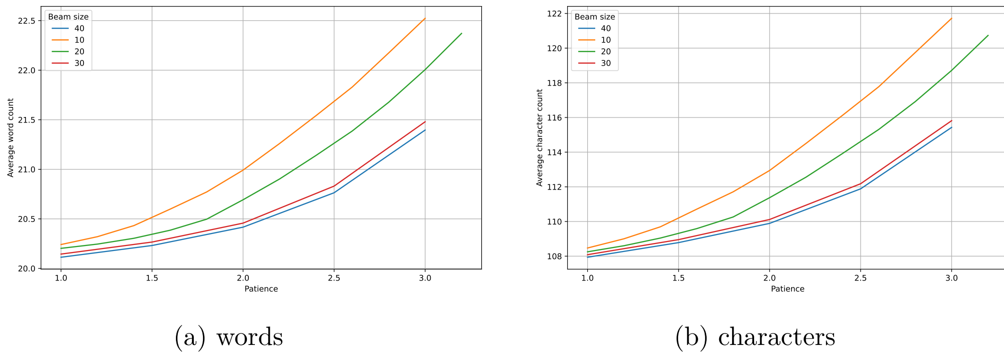


Figure 2.2: Graph showing average output hypotheses count depending on patience factor and beam size, using adjusted Whisper medium beam search decoder outputting multiple hypotheses.

Furthermore, with a higher patience value, generated hypotheses have more words and characters on average (Figure 5.1).



(a) words (b) characters

Figure 2.3: Graphs showing the average number of words (left) and characters (right) in a hypothesis depending on patience factor and beam size, using adjusted Whisper medium beam search decoder outputting multiple hypotheses.

The patience factor also influences WER. Figure 2.4 shows that the best possible WER is decreasing with a greater patience factor. The best possible WER is obtained by selecting the transcription with the lowest WER for each audio recording.

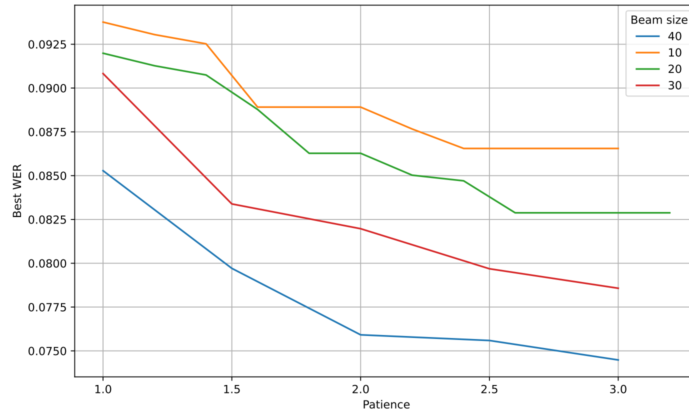


Figure 2.4: Graph showing best WER dependence on patience factor and beam size, using adjusted Whisper medium beam search decoder outputting multiple hypotheses.

The average time of transcription in seconds is shown in Figure 2.5. The transcription time is the time of inference and decoding, using the Whisper medium model. There is a slight increase in average transcription time with an increasing patience factor. Also, beam size increases the transcription time and this increase is more significant than the one caused by patience.

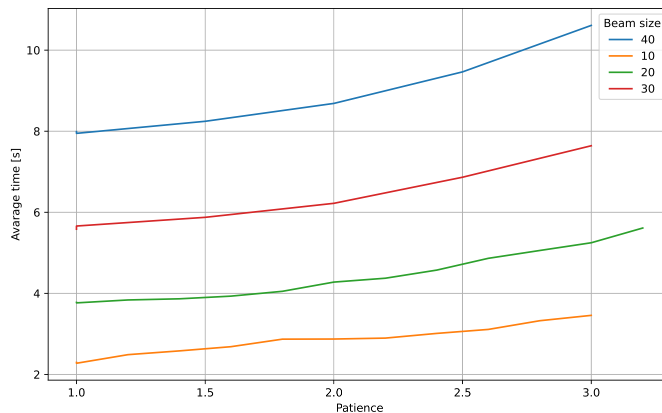


Figure 2.5: Graph showing average time of transcription (inference + decoding) in seconds. The transcription time depends on the patience factor and beam size. Measured using adjusted Whisper medium beam search decoder outputting multiple hypotheses.

Based on previous measurements, I assume setting a big enough patience factor in n-best rescoring can improve WER and save time. Increasing the patience factor may compensate for a small beam size.

Chapter 3

Language modeling

Language modeling captures regularities of natural language to improve the performance of various natural language applications. Language modeling amounts to estimating the probability distribution of various linguistic units, such as words, sentences, and whole documents [39].

Language modeling is crucial for a large variety of language technology applications. These include speech recognition, machine translation, document classification, optical character recognition, email spam detection, information retrieval, handwriting recognition, spelling correction, and many more. Speech recognition is where language modeling started.

Language modeling employs statistical estimation techniques using text as language training data. Because of the categorical nature of language and the large vocabularies commonly used in speech, statistical techniques must estimate a large number of parameters and consequently depend critically on the availability of large amounts of training data.

After large amounts of text became available online, it offered large training data, which led to a dramatic increase in the quality of language models.

3.1 Statistical language model

A statistical language model is a probability distribution $P(s)$ over all possible sentences s or spoken utterances, documents, or any other linguistic unit [39].

Statistical language models are usually used in the context of a Bayes' classifier, where they can play the role of either the prior or the likelihood function. For example, in ASR, given an acoustic signal a, the goal is to find the sentence that is most likely to have been spoken. The solution is in Equation 3.1, where the language model $P(s)$ plays the role of the prior.

$$s^* = \arg\max_s P(s|a) = \arg\max_s P(a|s) \cdot P(s) \tag{3.1}$$

N-grams [39] are statistical language models, that are widely used in ASR.

$$P(w_i|h_i)P(w_i|w_{i-n+1}, \ldots, w_{i-1}) \tag{3.2}$$

The n-gram language model can be described by an Equation 3.2, where w_i represents the word at the ith position in a given input and $h = w_1, w_2, \ldots, w_{i-1}$ represents history. They can model probability distribution over characters or words. In a word n-gram model, the probability of the next word depends on the $n - 1$ previous words. For example in a

sentence “I will do it tomorrow”, is the word “tomorrow” conditioned on “do it”, represented as $P(tomorrow|do\,it)$.

3.2 Metric

Perplexity is the most often used metric for evaluating language model performance [39, 26].

$$PPL(W) = \exp\left(-\frac{1}{t}\sum_{i}^{t}\log p_\theta(w_i|w_{<i})\right) \tag{3.3}$$

Perplexity can be interpreted as the geometric average branching factor of the language according to the model. It is a function of the language and the model. When considered a function of the model, it measures how good the model is. The better the model, the lower the perplexity. When considered a function of the language, perplexity estimates the entropy, or complexity, of that language. In the end, the quality of a language model must be measured by its effect on the specific application for which it was designed. Perplexity is calculated as shown in Equation 3.3, where $W = (w_0, w_1, \dots, w_t)$ represents a tokenized sequence comprising of t tokens. In the equation, $\log p_\theta(w_i|w_{<i})$ is the log-likelihood of a token w_i conditioned on the preceding tokens.

3.3 Text standardization

Text standardization [9] is a basic form of feature engineering that aims to erase encoding differences that the text processing model does not need. The standardization techniques improve model generalization, and the model will require less training data. For example, the model won’t need abundant examples of the word „Sunset“ when it knows „sunset“. The disadvantage of text standardization is the loss of information. The choice of text standardization schema must not conflict with the specific task that the model is meant for, for example, standardization schema removing the symbol „?“ used with the model extracting questions from the text.

A: Sunset came. I stared at the México sky, eating jalapeño. Isn't nature splendid?
B: **s**unset came **i** stared at the **m**éxico sky eating jalapeño **i**snt nature splendid
C: sunset came i stared at the m**e**xico sky eating jalape**n**o isnt nature splendid
D: sunset came i **[stare]** at the mexico sky **[eat]** jalapeno isnt nature splendid

Figure 3.1: Standardization schemes. One popular text standardization scheme is to convert text to lowercase and remove punctuation characters. Consider text A, applying the standardization scheme, the standardized text would be text B. Another common standardization scheme is to convert special characters to a standard form. That means to replace character „é“ with e, „ñ“ with „n“, „æ“ with „ae“, etc. When the two standardization schemes are applied to the text, the result is text C. An example of an advanced standardization pattern that is more rarely used is stemming. Stemming means converting variations of a term into a single shared representation. Examples of stemming are turning „caught“ and „been catching“ into „[catch]“, „was staring“ and „stared“ into „[stare]“ or „cats“ into „[cat]“. The resulting sentence after applying all three standardization schemes looks like text D.

There are many text standardization schemes. Three popular ones are depicted in Figure 3.1. In the context of ASR, text standardization is influenced by the ASR model and in the case of my experiments by the chosen dataset format.

3.4 Tokenization and indexing

Tokenization [9] means splitting text. To tokenize standardized text, three methods can be applied:

- Word-level tokenization – text is divided into words or smaller subword units for example „going“ can be split into „go“ + „ing“
- N-gram tokenization – tokens create groups of N consecutive words e.g. sentence „He is going to the train station“ would be tokenized into 2-gram tokens: „He is“, „is going“, „going to“, „to the“, „the train“, „train station“.
- Character-level tokenization – a token is a character e.g. „going“ is tokenized as „g“ „o“ „i“ „n“ „g“

Word-level tokenization is used in sequence models. Sequence models are text-processing models that take into consideration the order of words.

When a token is not included in the model vocabulary, it is called an out-of-vocabulary (OOV) token. Many tokenizers have a special marking for OOV tokens.

3.5 Encoding and embedding

Encoding [9] is the conversion of an index into a vector that can be processed by a neural network. A commonly used encoding is one-hot encoding. This assigns a vector of all zeros and a single one at the index number position. For example, 3 is encoded to 0000 1000. The disadvantage of this approach is that it creates sparse vectors. The created vectors

are orthogonal to each other – there is no relationship encoded in vectors representing the encoded words. The created vectors are high-dimensional, so they take too much memory. This is the reason why vocabularies mentioned in Section 3.4 are restricted to 20k or 30k most common words in the training dataset. This means 20k or 30k dimensions – the same amount of dimensions as the number of encoded words.

Embedding [9, 53] refers to a vector representation of words that encodes information about semantic relationships. Sequence of length l is converted to a matrix $\mathbf{W} \in \mathbb{R}^{l \times d}$, where d is the embedding dimension.

Two vectors representing two words should be close to each other when they have the same meaning, for example *happy* and *delighted*. On the other hand, vectors representing words with different meanings should lie far away from each other. An example of such real-world transformation is a plural vector which can be added to other vectors. For example, a „potato" vector, resulting in „potatoes". The advantage of the embedding vectors is that they are dense. They have typically 256, 512 or 1024 dimensions.

3.6 Large Language Models

Large Language Model (LLM) [31] is a deep learning model pre-trained on a vast amount of data. The LLMs in the experiments are built upon the Transformer architecture.

The historical evolution of LLMs can be traced back to statistical language models. Neural language modeling evolved from statistical language modeling. Initially, supervised task-specific language models were trained. Later self-supervised language models were developed to learn a generic representation for various NLP tasks. These models are called pre-trained language models. Subsequently, LLMs emerged from pre-trained language models, by increasing the number of model parameters.

3.7 Transformer architecture

Transformer architecture [54] proposes an innovative approach to sequence transduction tasks. The advantage of transformer architecture is that it does not use recurrence. The core of transformer architecture is attention (Section 3.9).

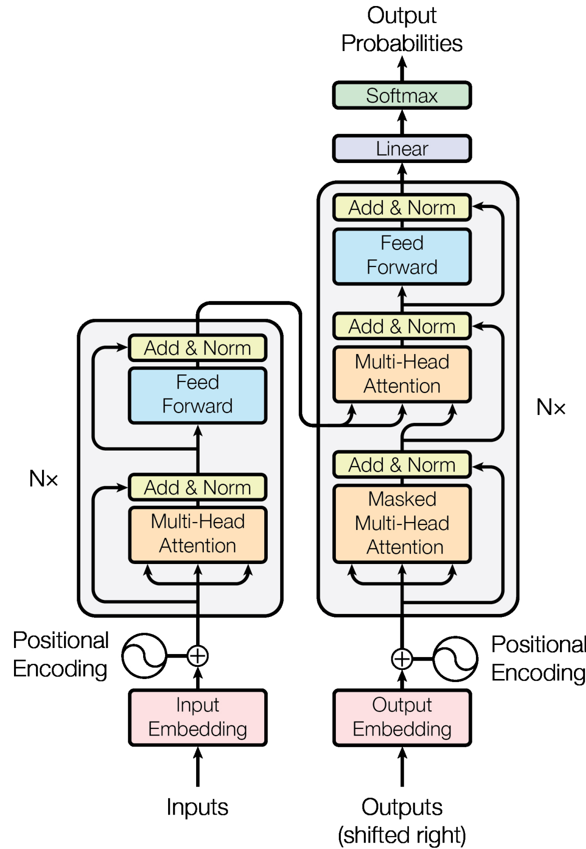


Figure 3.2: Transformer schema. Inputs to the transformer are positionally encoded. These positionally encoded inputs are processed by stacks of N encoder blocks (on the left) and N decoder blocks (on the right). Each block consists of multiple layers of self-attention and point-wise, fully connected layers.

The architecture is depicted in Figure 3.2 from [54]. At first, the input sequences are tokenized (Section 3.4) and converted to embeddings (Section 3.5) using the embedding layer. In the next step, the embeddings are positionally encoded. Positional encoding (Section 3.8) is used to incorporate information about the absolute position of the tokens in the sequence.

Matrices of word embeddings W and positional encodings P are summed to generate the input representation $X = W + P \in \mathbb{R}^{l \times d}$.

The transformer consists of 2 main blocks: an encoder and a decoder. The input for transformer is sequence $x_1, x_2 \ldots x_n$, encoder transforms the sequence to $z_1, z_2 \ldots z_n$ and decoder transforms sequence of zs to $y_1, y_2 \ldots y_n$, while previously generated output y tokens are fed to the model.

The transformer blocks are composed of multiple layers. Each of the layers in the encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. For each position, linear transformation defined in Equation 3.4 is performed.

$$FFN(\mathbf{x}) = max(0, \mathbf{x}\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2 \tag{3.4}$$

The transformations are consistent at different positions, but they differ at the layer level [54, 53].

The main advantage of attention layers against recurrent is that they reduce path lengths, and reduce the amount of computational steps that information has to go through from one point of the network to the other. Another advantage is that more computation can be parallelized.

3.8 Positional encoding

Positional encoding carries information about word order [53, 9]. Requirements for positional encodings are:

- Unique value at each time-step.
- Consistent distance between two time-steps across sentences of various lengths.
- Encoding results are generalized independent of the length of the sentence.
- The encoding is deterministic.

Absolute positional encoding

Absolute positional encoding passes information about the absolute position to be used during the attention weight calculation. It is used in the original Transformer [54], where sine and cosine functions of different frequencies are used to encode position. Positional encoding is represented by $\mathbf{P} \in \mathbb{R}^{L \times d}$ where L is the maximum sequence length and d is the embedding dimension. Each row $\mathbf{p}_i$ of $\mathbf{P}$ contains the positional encoding for the token at position i.

$$p_{i,2j} = \sin\left(i/1000^{2j/d}\right) \tag{3.5}$$

$$p_{i,2j+1} = \cos\left(i/1000^{2j/d}\right) \tag{3.6}$$

In Equations 3.5 and 3.6, i is the position and j is the dimension.

Relative positional encoding

Relative positional encoding [42, 53] leverages distance between two positions in a sequence. The relative distance between query $\mathbf{q}_i$ and key $\mathbf{k}_j$ vectors is incorporated in attention weight calculation, biasing the attention mechanism to consider the distance $i - j$ as the important quantity.

Rotary positional encoding

Rotary positional encoding (RoPE) [47] differs from previous approaches by not adding a positional vector. Instead, it encodes position using rotation.

$$\mathbf{q}_m = f_q(\mathbf{x}_m, m) \tag{3.7}$$

$$\mathbf{k}_n = f_k(\mathbf{x}_n, n) \tag{3.8}$$

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} W^{(11)}_{\{q,k\}} & W^{(12)}_{\{q,k\}} \\ W^{(21)}_{\{q,k\}} & W^{(22)}_{\{q,k\}} \end{pmatrix} \begin{pmatrix} x^{(1)}_m \\ x^{(2)}_m \end{pmatrix} \tag{3.9}$$

Positions m and n are incorporated into values of query $\mathbf{q}_m$ and key $\mathbf{k}_n$ through f_q and f_k as depicted in Equations 3.7 and 3.8, where $\mathbf{x}_m$ and $\mathbf{x}_n$ are word embeddings. Then, the query and key values are used to compute the attention weight (Section 3.9). A matrix for 2D rotation is depicted in Equation 3.9, where $(x_m^{(1)}, x_m^{(2)})$ is x_m expressed in the 2D coordinates.

3.9 Attention

The attention mechanism [53] can be viewed as a memory comprising of keys, values and a layer that produces output when queried. The output is generated from the value whose keys map the input. The query serves as the input for the attention layer. The values represent a body of knowledge from which information is extracted. Each value is assigned a key, which is in a format that can be easily compared to a query.

Formally, the memory unit consists of n key-value pairs $(\mathbf{k}_1, \mathbf{v}_1), ..., (\mathbf{k}_n, \mathbf{v}_n)$ with $k \in \mathbb{R}^{d_k}, v \in \mathbb{R}^{d_v}$. The attention layer receives input as query $\mathbf{q}$ and returns an output $\mathbf{o}$ with the same shape as the value $\mathbf{v}$: $\mathbf{q} \in \mathbb{R}^{d_q}, \mathbf{o} \in \mathbb{R}^{d_v}$.

The attention layer measures the similarity between the query and the key using a score function α (Equation 3.10). The α returns scores $a_1, ..., a_n$ for keys $\mathbf{k}_1, ..., \mathbf{k}_n$. Attention weights are computed by applying softmax to the values from α (Equation 3.11). Each element of $\mathbf{b}$ is computed as displayed by Equation 3.12. Equation 3.13 shows the calculation of the output $\mathbf{o}$.

$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i) \tag{3.10}$$

$$\mathbf{b} = softmax(\mathbf{a}) \tag{3.11}$$

$$b_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)} \tag{3.12}$$

$$\mathbf{o} = \sum_{i=1}^{n} b_i \mathbf{v}_i \tag{3.13}$$

According to the score function α, there are different types of score-based attention. Dot product attention is multiplicative and simple. It has no parameters to tune. Equation 3.14 displays the score function.

$$\alpha(\mathbf{q}, \mathbf{k}) = \mathbf{q} \cdot \mathbf{k} \tag{3.14}$$

Scaled dot product attention is multiplicative, too. Equation 3.15 shows the scoring function, that divides the dot product by $\sqrt{d_k}$ to remove the influence of dimension d_k. As the dimension increases, the dot products grow larger, which pushes the softmax function into regions with extreme gradients.

$$\alpha(\mathbf{q}, \mathbf{k}) = \frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}} \tag{3.15}$$

Linear attention is additive. The query and keys are projected to a hidden layer of dimension h. The weights $(\mathbf{W}_k, \mathbf{W}_q)$ are combined with values using a sigmoid function as given by Equation 3.16.

$$\alpha(\mathbf{q}, \mathbf{k}) = \mathbf{v}^T \tanh \mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q} \tag{3.16}$$

Additive scoring mechanisms are slower and less memory efficient than dot product or scaled dot product scoring.

Self-attention

The basic idea of self-attention is to convert input vectors, $\mathbf{x}_i$, to output vectors $\mathbf{z}_i$, using an attention matrix generated from the input vector. The attention matrix is used to produce a weighted average of the input feature. To generate the attention matrix, the similarity between two locations is measured by the dot product between the features at those two locations and then a softmax function is applied to handle the normalisation. This approach entangles information about the similarity between locations in the sequence with the content of the sequence itself [51].

The self-attention used in transformer architecture uses trainable matrices to produce query, key and value. Each input vector $\mathbf{x}_i$ generates three different vectors: query, key and value $(\mathbf{q}_i, \mathbf{k}_i, \mathbf{v}_i)$. These vectors are obtained by projecting the input vector, $\mathbf{x}_i$, at time i on the learnable weight matrices $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$ to get $\mathbf{q}_i$, $\mathbf{k}_i$ and $\mathbf{v}_i$, respectively. The weights in the weight matrices are trainable parameters of the model.

The query, key and value vectors are not combined by vector computation for each token i. The input matrix $\mathbf{X} \in \mathbb{R}^{l \times d}$ where l is the maximum length of the sequence and d is the dimension of the inputs, combines with each of the query, key and value matrices as a single computation (Equation 3.17).

$$attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{3.17}$$

Query, key and value vectors have the following roles in self-attention. The role of the query vector of token i, $\mathbf{q}_i$, is to combine with every other key vectors $\sum_{j=0}^{l} \mathbf{q}_i {\mathbf{k}_j}^T$ to influence the weights for its own output, $\mathbf{z}_i$. The role of the value vector of token i, $\mathbf{v}_i$, is extracting information by combining with the output of the query-key scores to get the output vector $\mathbf{z}_i$.

Multi-head attention

There are h self-attention heads that are parallel in the multi-head attention (MHA) [54]. The purpose of using multi-head attention is that more heads learn different types of dependencies in the input sequence by providing different subspace representations instead of just a single representation for the input. For example in the sentence „Einstein was a genius, he revolutionized physics.“, one attention head may not be enough to capture all connections to the word "revolutionized", such as "Einstein", "he", "physics". Using more heads increases the chances of finding all related words to "revolutionized".

In MHA each set of query, key and value weight matrices produces different query, key and value matrices for the inputs, eventually generating output matrices $\mathbf{z}_i$. The output matrices are concatenated. The final matrix, $\mathbf{Z}$, with vectors $\mathbf{z}_i$ as output for each $\mathbf{x}_i$ is obtained by multiplying the concatenated output matrix with an additional weight matrix, $\mathbf{W}_O$ (Equation 3.19). The $head_i$ from Equation 3.19 is defined in Equation 3.18.

$$head_i = attention(\mathbf{W}_q^i\mathbf{Q}, \mathbf{W}_k^i\mathbf{K}, \mathbf{W}_v^i\mathbf{V}) \tag{3.18}$$

$$MHA(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = concat(head_1, ..., head_h)\mathbf{W}_O \tag{3.19}$$

Masked attention

Masked attention [53] is an attention in which some tokens are masked. This is utilized by the decoder.

The decoder is autoregressive, it predicts the next token based on the sequence that has been seen by the model. To prevent it from seeing the future tokens, they are masked.

The decoder block of the transformer network uses the masked multi-head attention. Only previous tokens are considered, and future input tokens are masked. The masking is implemented using a masking weight matrix $\mathbf{M}$ that has $-\infty$ for future tokens and 0 for the previous tokens. The matrix $\mathbf{M}$ is added to the multiplication of $\mathbf{Q}$ and $\mathbf{K}^T$ (Equation 3.20). Then the softmax results in the actual scaled values for previous tokens and the value 0 for the future tokens.

$$maskedAttention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T + \mathbf{M}}{\sqrt{d_k}}\right)\mathbf{V} \tag{3.20}$$

Multi-Query attention

Multi-Query Attention (MQA) [43] is a variation of MHA. It consists of multiple attention heads. The difference against the MHA is that the heads share a single set of keys and values. MQA does not change the number of query heads.

Autoregressive decoder inference is a bottleneck for LLM models because of memory consumption at every decoding step. The main advantage of MQA addresses this problem, it reduces memory bandwidth from loading keys and values, and it lowers the ratio of memory access to arithmetic operations. However, it has the disadvantage that it can cause the training to be unstable and model quality to degrade [43, 3].

Grouped-Query Attention

In Grouped-Query Attention (GQA) [3], query heads are divided into G groups. A single key and value head is shared by each group of query heads. MHA and MQA can be considered as the edge cases of GQA. MHA is the GQA with the same number of groups as the number of query heads. On the other side, MQA is equal to GQA with one group $G = 1$ of query heads. The comparison of MHA, GQA and MQA is displayed in Figure 3.3 taken from research paper [3].

GQA is not used in the encoder, only in the decoder self-attention layers. The reason for this is that memory bandwidth is not the main bottleneck in the encoder because encoder representations are computed in parallel.

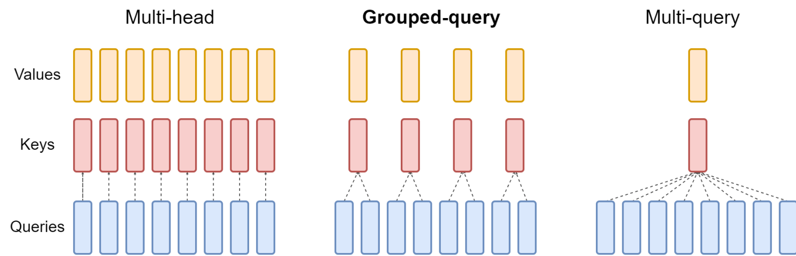


Figure 3.3: Comparison of Multi-head Attention MHA, Grouped-Query Attention GQA and Multi-Query Attention MQA. MHA has H query, key, and value heads. Single key and value heads are shared across all query heads in MQA. GQA is interpolating between multi-head and multi-query attention, single key and value heads for each group G of query heads are shared in GQA.

Sliding-Window Attention

Let W be the size of an attention window – the number of tokens each token can attend to. Sliding-Window Attention uses the stacked layers of a transformer model to extend its attention beyond the fixed window size W.

For a model with k layers, consider h_i to be the hidden state in position i of the layer k. The hidden stated h_i attends to all hidden states from the previous layer with positions between $i - W$ and i, maximally at a distance of $W \times k$ tokens. A theoretical attention span of approximately 131K tokens can be achieved using a window size of $W = 4096$, as in Mistral (Section 4.5). An example demonstrating Sliding-Window Attention is depicted in Figure 3.4, taken from [20].

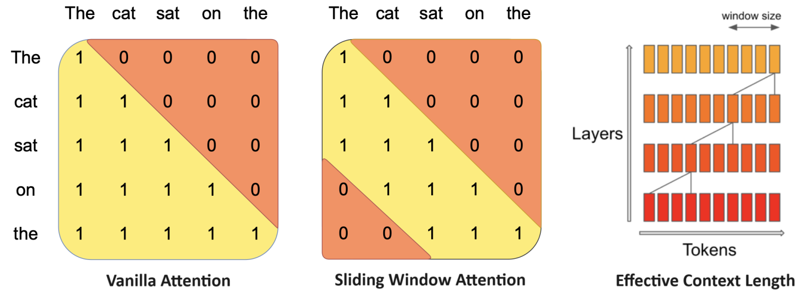


Figure 3.4: Sliding-Window Attention. In vanilla attention, the memory increases linearly with the number of tokens. During inference, this causes higher latency and smaller throughput because of reduced cache availability. Sliding window attention reduces this issue. Each token can attend to at most W tokens from the previous layer (in the picture, $W = 3$). The tokens outside the sliding window still influence next-word prediction. Information can move forward by W tokens at each attention layer. That means information can move forward maximally by $k \times W$ tokens, after k attention layers.

Attention with Linear Biases

Attention with Linear Biases (AliBi) [34] can accelerate training. Models with AliBi can extrapolate, they achieve smaller perplexity when a larger sequence is fed. AliBi tests have shown PPL = 19.73 when processing 512 token sequences, and PPL = 18.4 when fed sequences of 3072.

$$softmax\left(\mathbf{q}_i\mathbf{K}^T + m \cdot [-(i-1), \ldots, -2, -1, 0]\right) \tag{3.21}$$

In AliBi, positional embeddings are not incorporated in any position within the network. Instead a static, non-learned bias is added after the query-key dot product, as defined in Equation 3.21, where m is a scalar, a head-specific slope fixed before training. The set of slopes is a geometric sequence, which starts at $2^{\frac{-8}{n}}$ and it has a ratio of $2^{\frac{-8}{n}}$, where n is the number of heads. The values are static and don't need to be adjusted across models. To demonstrate for 16 heads, the slopes are: $\frac{1}{2^{0.5}}, \frac{1}{2^{1}}, \frac{1}{2^{1.5}}, \frac{1}{2^{2}}, \frac{1}{2^{2.5}}, \frac{1}{2^{3}}, \ldots, \frac{1}{2^{8}}$.

3.10 Activation function

Many LLM models (Section 4.5) differ in the activation function they use. The activation function is used to compute the output of a neuron in a neural network based on the weighted sum of the neuron inputs.

Softmax

Introduced to neural networks in [7]. Defined by Equation[1] 3.22.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K \tag{3.22}$$

Softmax is an element-wise function typically applied to the output of a neural network. It moves output elements to the range $(0, 1)$ and the sum of the scaled output elements is 1. The output of the softmax activation function represents probability distribution over classes given by input.

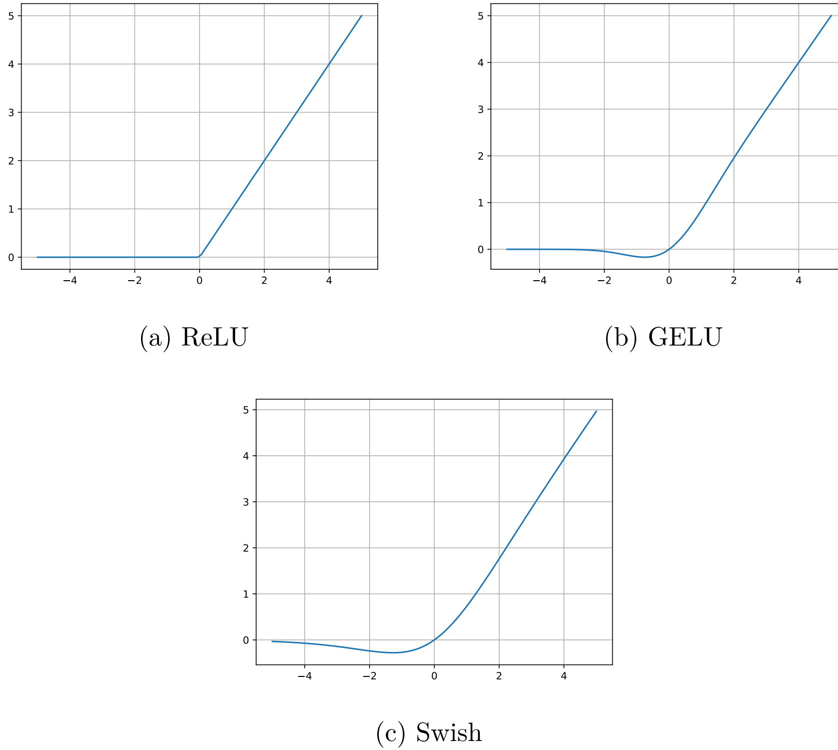
Rectified Linear Unit

Rectified Linear Unit function (ReLU) was introduced in neural networks in [30].

$$y_i = \begin{cases} x_i\ if x_i \geq 0 \\ 0\ \ if x_i < 0 \end{cases} \tag{3.23}$$

The function is defined by Equation 3.23.

[1] https://blmoistawinde.github.io/ml_equations_latex/#softmax



(a) ReLU

(b) GELU

(c) Swish

Figure 3.5: Graphs of ReLU, GELU and Swish functions.

ReLU introduces non-linearity to the neural network, which enables the network to produce richer hypotheses. It is displayed in Figure 5.5a.

SwiGLU

$$SwiGLU(x, W, V, b, c, \beta) = Swish\beta(xW + b) \otimes (xV + c) \tag{3.24}$$

SwiGLU [44] is a combination of Swish [38] (Figure 5.5c) and Gated Linear Unit (GLU) [10]. SwiGLU and Gaussian Error Linear Unit (GELU) are the best performing GLU variants for text-to-text transformers [44].

3.11 Residuals and Layer Normalization

The inputs, $\mathbf{X}$, are short circuited to the output, $\mathbf{Z}$, and both are added and passed through layer normalization $Norm(\mathbf{X} + \mathbf{Z})$ [17]. Layer normalization ensures each layer has 0 mean and a unit variance. For each hidden unit, h_i, can be computed

$$h_i = \frac{g}{\sigma}(h_i - \mu) \tag{3.25}$$

where g is the gain variable, μ is the mean given by $\frac{1}{H}\sum_{i-1}^{H} h_i$ and σ is the standard deviation given by $\sqrt{\frac{1}{H}(h_i - \mu)^2}$. The purpose of layer normalization is to improve the stability of the network during training and to better generalize.

Residual connection

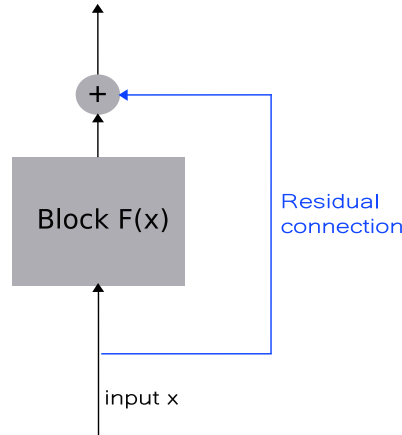


Figure 3.6: Residual connection. The input and output of the neural network block are summed.

When information goes through a function, the function adds noise to it. In neural networks, data goes through multiple functions. The noise accumulates and it can eventually overwhelm gradient information. It is called gradient vanishing. Residual connection [17] addresses this issue.

$$y = F(x) + x \tag{3.26}$$

Residual connection works the way that the original input of a neural network processing block is added to the output of the block (Equation 3.26).

3.12 Low-Rank Adaptation

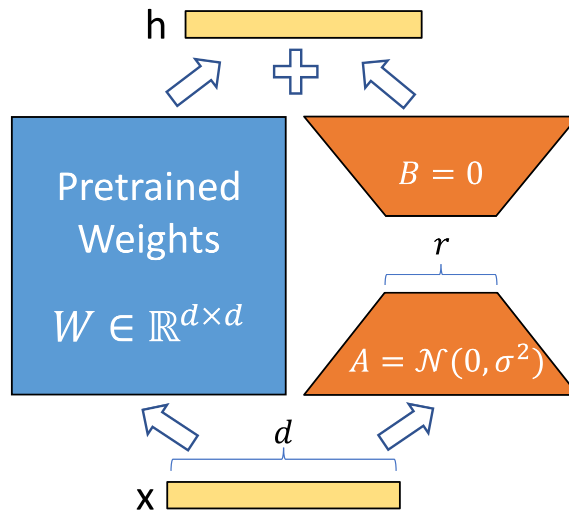


Figure 3.7: LoRA reparametrization. Only A and B are being trained. The figure is taken from [18]

It is very resource-intensive to fine-tune LLM. However, workarounds exist. One such is Low-Rank Adaptation (LoRA) [18]. In this approach, the pre-trained model weights are frozen and trainable rank decomposition matrices are injected into each layer of the Transformer architecture. It dramatically reduces the number of trainable parameters, thus it makes fine-tuning less resource-consuming. The reparametrization is depicted in Figure 3.7.

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{3.27}$$

This method modifies forward pass, where $W_0 \in \mathbb{R}^{d \times k}$ is the pre-trained weight matrix. The $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$ contain trainable parameters, rank $r \ll d, k$. At the beginning of training $\Delta W = BA = 0$, then it scaled by $\frac{\alpha}{r}$, where α is a constant in r, usually it is set to $\alpha = 2r$. So there are two main parameters be to set before training: rank and α.

3.13 Dropout

Dropout [9] was developed by students of Geoff Hinton, it is a regularization technique. When dropout is applied to a layer, some features are randomly dropped out during training. In practice, some values of the layer are set to zero during training. For example (0.2, 0.5, 0.7, 0.4, 0.3) after applying dropout would be (0.2, 0, 0.7, 0.4, 0). The rate of dropout represents the fraction of features that will be dropped out. In general dropout rate is usually between 0.2 and 0.5.

3.14 Language models in automatic speech recognition

While language models are used in ASR systems, the impact of incorporating LLM is not sufficiently explored. There are more approaches to incorporate a language model in ASR: shallow fusion, n–best and lattice rescoring, prompting, etc.

$$y^* = \arg\max_y [\log p(y|x) + \delta \log p_{LM}(y)] \tag{3.28}$$

Shallow fusion

Shallow fusion incorporates the language model into the decoding phase. At each decoding step, the E2E model score is combined with the language model score Equation 3.28, where x is the input speech sequence, $p(y|x)$ is the posterior probability from E2E model, $p_{LM}(y)$ is the language model score and δ is scaling factor of the language model's score. Smaller language models like n-grams are typically utilized in shallow fusion, however, it is possible to use larger language models for example 1.9B-parameter GLaM (during inference used only 145M parameters), as demonstrated in experiments [19].

N-best rescoring

Calling several billion parameter language models at each decoding step, as in shallow fusion, would be very resource-extensive. Alternative to the shallow fusion is *n-best* rescoring. In the first step, ASR produces n hypotheses (transcriptions of speech in the audio) with their scores. Each hypothesis is scored by LLM which is described in Section 3.15. The scores are together. The hypothesis with the best new score is chosen.

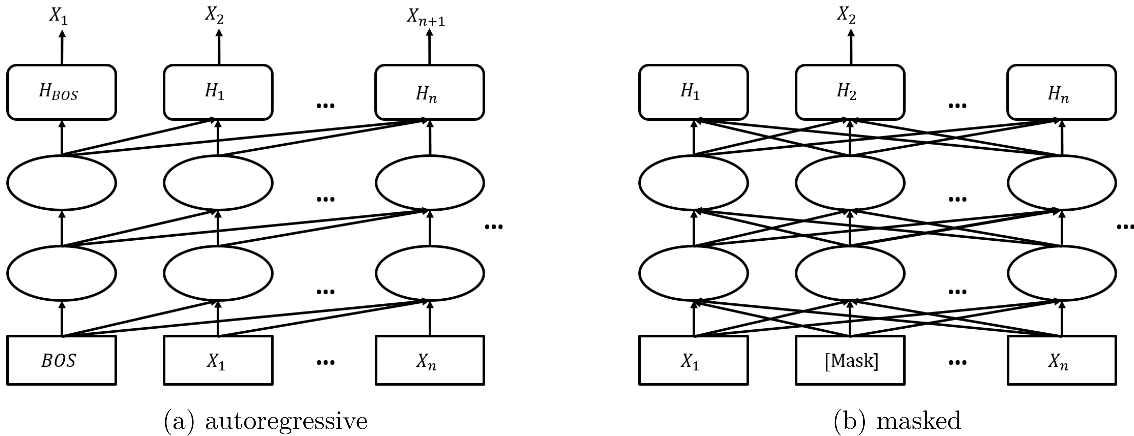
Rescoring experiments Jasper hypotheses by Transformer-XL found a correlation between the quality of the LLM and WER [23]. Another research showing LLM reduce WER in ASR rescoring is [52].

Promting

LLMs fine-tuned for question-answering tasks can be prompted to improve ASR transcripts. In this approach, WER reduction was not observed [29].

3.15 Sentence scoring

In the experiments, there are two types of transformer models: masked language models (MLM) and autoregressive language models. They process input in different ways, however, the processing has common steps: conversion of a sequence of characters into a sequence of tokens, at the end of processing the output is one number which represents the score of the input sequence.



(a) autoregressive (b) masked

Figure 3.8: Comparison of autoregressive (left) and masked (right) language models structures. Figure taken from [24].

In an autoregressive model, the n-th token can only attend to the tokens at positions $0, \ldots, n-1$. On the other side, a masked language model can capture the contextual information from tokens on positions $0, \ldots, n-1$ and also $n+1, \ldots, m$, where m is the position of the last token [24].

Masked language model scoring

Scoring a text using masked language modeling is depicted in Algorithm 2, which is inspired by article [45]. At first, x copies of a sentence are needed, where x is the number of tokens in the tokenized sentence. In each sentence, one token is masked. The masked language model predicts the probability matrix. Then the probability of the token that is in the original hypothesis (the token under the mask) is found. This probability is added to the sum of probabilities in the sequence, the final sum is the score of this sequence.

To make the process faster, the hypotheses are processed in batches and the algorithm is changed. I solved the problem of various length sentences in the batch by multiplying

the matrix with obtained scores for each token by the *attentionMask* of the original non-masked batch.

Algorithm 2 Masked language model scoring

Input: String *sentence* **Output:** Float *score*

1: $score = 0$
2: **for** $maskPosition$=1,2,. . . ,tokenCnt($sentence$) **do**
3: $sentenceCopy$ = copy($sentence$)
4: $tokenizedSentence$ = tokenize($sentenceCopy$, at=$maskPosition$)
5: $maskedSentence$ = mask($tokenizedSentence$, at=$maskPosition$)
6: $logitsMatrix$ = model($maskedSentence$)
7: $softmaxedLogitsMatrix$ = softmaxLayer($logitsMatrix$)
8: $tokenProbability$ = getValue($softmaxedLogitsMatrix$,
9: originalToken=$tokenizedSentence$[$maskPosition$])
10: $score$ += $tokenProbability$
11: **end for**
12: return $score$

Autoregresive language model scoring

The autoregressive scoring algorithm is simpler. Input sentences are tokenized and fed to the language model. Autoregressive models return the probability of all tokens in the given sentence at once. Then softmax is applied to the language model's output. Finally, the probabilities of all tokens in a sentence are summed together to form the sentence score. Scoring a text by an autoregressive model is depicted in Algorithm 3. The implementation of the autoregressive scoring algorithm was inspired by discussion.[2]

Algorithm 3 Autoregressive language model scoring

Inputs: String *sentences*, Integer N **Output:** Float *score*

1: $sentencesTokens$ = tokenize($sentences$, at=$maskPosition$)
2: $logitsMatrix$ = model($sentencesTokens$)
3: $softmaxedLogitsMatrix$ = softmaxLayer($logitsMatrix$)
4: $attentionMask$ = $sentences$.getAttentionMask()
5: $maskedTensor = attentionMask \cdot softmaxedLogitsMatrix$
6: $sentenceScores$ = sum($maskedTensor$) **return** $sentenceScores$

[2] https://discuss.huggingface.co/t/announcement-generation-get-probabilities-for-generated-output/30075/17

Chapter 4

Pre-trained models

In this chapter, I describe acoustic and language models, as well as the tools I employed in the ASR pipeline. For acoustic models, I used Wav2Vec 2.0 with a CTC decoder from the flashlight library, Jasper with a CTC decoder from the NeMo library[1] and the Whisper model. The Masked LLMs utilized in experiments are BERT and RoBERTa. On the other side, autoregressive LLMs include GPT-2, Falcon, Mistral, MPT, TinyLlama and Llama2.

When the speech transcription is passed from ASR to LLM, the text undergoes tokenization. Many LLM tokenizers employ Byte Pair Encoding, which is explained in the following section.

4.1 Byte Pair Encoding tokenizer

Byte Pair Encoding (BPE) [41] is a compression algorithm used for word segmentation. BPE allows the representation of an open vocabulary through a fixed-size vocabulary of variable-length character sequences.

The BPE algorithm has one hyperparameter, which is the number of merge operations. The algorithm can work over characters or bytes. There is a vocabulary, consisting of words represented as a sequence of characters with the end-of-word symbol at the end '·'. Then the algorithm iteratively counts all symbol pairs and replaces the most frequent pair with a new symbol, for example 'A', 'B' pair is replaced by a single symbol 'AB'. Eventually, each sequence is merged into a word.

An example of how it would work on vocabulary, with word occurrence frequency indicated after a colon

```
s p a c e · :10, k e y · : 7, a c c e l e r a t e d · : 4,  h o c k e y · :6
```

with the number of merges set to three:

```
Step 1) pair: (a, c)
vocabulary:  s p ac e · : 10, k e y · : 7, ac c e l e r a t e d · : 4,
h o c k e y · : 6

Step 2) pair: (k, e)
vocabulary:  s p ac e · : 10, ke y · : 7, ac c e l e r a t e d · : 4,
h o c ke y · : 6
```

[1] https://github.com/NVIDIA/NeMo

```
Step 3) pair: (ke, y)
vocabulary:  s p ac e · : 10, key · : 7, ac c e l e r a t e d · : 4,
h o c key · : 6
```

The BPE tokenizer, as used in GPT-2 for example, was originally built to operate over byte sequences. A byte-level BPE tokenizer can represent any Unicode string using a vocabulary of only 256 bytes, but it is not employed by GPT-2 and most models due to byte-level models showed poor performance on word-level [37]. This issue was addressed with the introduction of the ByT5 [56] model.

There is another challenge with applying BPE to byte sequences, it results in the inclusion of multiple versions of common words in the vocabulary. For instance, the word 'dog' occurs as 'dog.', 'dog!' and 'dog?' which wastes limited vocabulary slots. To mitigate this problem, BPE was forbidden to merge across character categories for any byte sequence, with an exception for spaces. This modification improves the compression efficiency and adds minimal fragmentation of words across multiple vocabulary tokens.

GPT-2 and similar models utilize a BPE tokenizer, which works with a vocabulary consisting of subwords. This allows to represent any word by multiple subword tokens. GPT-2 vocabulary size is 50k tokens.

4.2 Wav2Vec 2.0

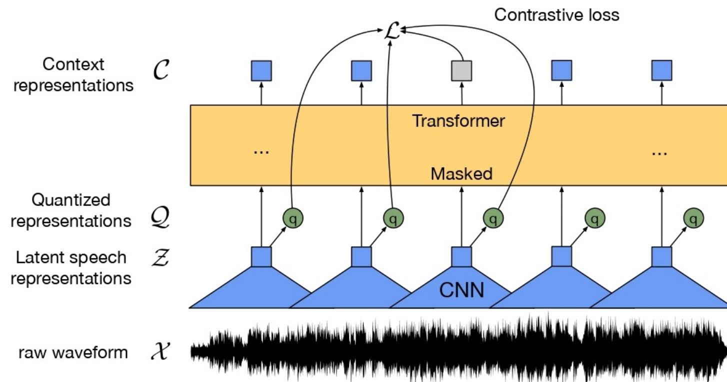


Figure 4.1: Wav2Vec 2.0 framework. This framework jointly learns contextualized speech representations and an inventory of discretized speech units. The Figure is taken from [5].

Wav2Vec 2.0 [5] is a self-supervised speech encoder, which needs to be fine-tuned with CTC-loss to be used in speech recognition. Wav2Vec 2.0 uses a multi-layer convolutional feature encoder to convert the input speech audio $\mathbf{X}$ into a sequence of latent speech representations, $(\mathbf{z}_1, ..., \mathbf{z}_T)$. Then a Transformer builds context representations $\mathbf{c}_1, \ldots, \mathbf{c}_T$ from $\mathbf{z}$s. The context representations $\mathbf{c}$s capture information from the entire sequence. The length of the sequence, T, is the number of time-steps in the audio [53].

The purpose of the feature encoder is to reduce the dimensionality of the audio data. The feature encoder consists of blocks containing temporal convolution followed by layer normalization (Section 3.11) and a GELU activation function (Section 3.10). The raw

waveform input is normalized to zero mean and unit variance. The total stride of the encoder determines the number of time-steps T.

The output of the feature encoder is fed to a context network of the Transformer architecture. A convolutional layer that acts as relative positional embedding, is used instead of fixed positional embeddings which encode absolute positional information. The output of the convolution is followed by a GELU. This is added to the inputs. Then layer normalization is applied.

Using product quantization [21], the output of the feature encoder $\mathbf{z}$ is discretized to a finite set of speech representations. Product quantization is equivalent to choosing quantized representations from multiple codebooks and concatenating them. Consider G codebooks – groups, with V entries $e \in \mathbb{R}^{V \times \frac{d}{G}}$. One entry is chosen from each codebook. Then resulting vectors $e_1, \ldots, e_G$ are concatenated. Then a linear transformation $\mathbb{R}^d \to \mathbb{R}^f$ is applied to obtain $\mathbf{q} \in \mathbb{R}^f$.

The training process combines unsupervised and self-supervised pre-training. During the pre-training, the model learns diverse features of natural language. After pre-training, the model needs relatively little labelled data for fine-tuning.

I used the `WAV2VEC2_ASR_BASE_960H`[2] model. The model has the `BASE` architecture (12 Transformer blocks) with an extra linear module. The model is pre-trained on 960 hours of unlabeled audio from the LibriSpeech dataset described in Section 5.1 specifically on the combination of `train-clean-100`, `train-clean-360`, and `train-other-500`. After the pre-training, the model was fine-tuned for ASR on the same audio data with the corresponding transcriptions.

4.3 Whisper

The Whisper model [36] is based on encoder-decoder transformer architecture (Section 3.7). The encoder learns representations of speech and the decoder generates the transcription. The Whisper models were trained on data from the internet. The architecture is displayed in Figure 4.2.

Audio input to the model has to be resampled to 16 kHz. An 80-channel log-magnitude Mel spectrogram representation is computed on 25-millisecond windows with a stride of 10 milliseconds [36]. The input is scaled to be between -1 and 1. Across the pre-training dataset, the input has approximately zero mean.

[2] https://pytorch.org/audio/stable/generated/torchaudio.pipelines.WAV2VEC2_ASR_BASE_960H.html

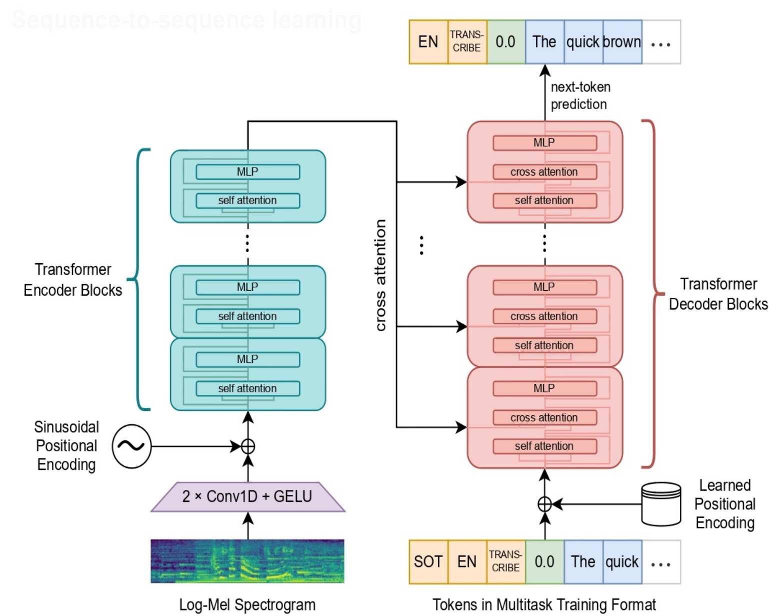


Figure 4.2: Whisper. The audio is processed by a stem of two 1D convolution layers and the GELU activation function. Then it is processed by the Transformer Encoder block. Finally, speech transcription is produced by the Transformer Decoder Block. The figure is taken from [36].

In the first part of the encoder, there is a small stem, that processes the input representation. The stem consists of two 1D convolution layers with a filter width of 3 and the GELU activation function (Section 3.10) where the second convolution layer has a stride of two. This means that the second convolutional layer moves by two items in the sequence. The Conv1D relies on the 1D window, the window slides across the input. Whisper uses sinusoidal position embeddings, which are added to the output of the stem.

Then the encoder Transformer blocks are applied. The transformer uses pre-activation residual blocks [8], and a final layer normalization is applied to the encoder output.

The decoder has the same number of transformer blocks and the same width. The decoder uses learned position embeddings and tied input-output token representations [35].

Whisper uses a byte-level BPE text tokenizer (Section 4.1). It is the same tokenizer that is used in GPT-2 Section 4.5 for English-only models. For multilingual models, the vocabulary is changed, while the size of the tokenizer stays the same.

There are five Whisper models of different sizes. All of them are in Table 4.1. Except for `Large`, all model sizes have English and multilingual variants. The `Large` is only multilingual.

Table 4.1: Architecture details of the Whisper model family from paper [36] and GPU inference requirements from site[3]

Model	Layers	Width	Heads	Parameters (M)	GPU VRAM [GB]
Tiny	4	384	6	39	1
Base	6	512	8	74	1
Small	12	768	12	244	2
Medium	24	1024	16	769	5
Large	32	1280	20	1550	10

Except for predicting spoken words from audio, Whisper can perform other tasks. The tasks are multilingual speech recognition, speech translation, spoken language identification, and voice activity detection. A sequence of input tokens to the decoder is used to specify the tasks. The beginning of the prediction is indicated with a <|startoftranscript|> token. At the beginning of audio prediction, the language of the speech in the audio is predicted. Each language is indicated by a unique token. In Whisper's training dataset, there are 99 unique language tokens. <|transcribe|> or <|translate|> tokens specify the task – transcription or translation respectively. In the audio segments without speech, the model predicts <|nospeech|> token.

It is recommended to normalize the output from the model to evaluate WER and other metrics. To follow all the rules from the authors, I decided to use tool *whisper-normalizer 0.0.8*.[4] Here are a few examples of rules that are applied during normalization: remove phrases between matching brackets and parenthesis, and remove commas between digits and periods not followed by numbers.

In the experiments, I used the multilingual `Medium` Whisper model. The decoder produces just one hypothesis, even though it supports not only greedy search but even beam search. I adjusted the decoder[5] to return all final hypotheses. The number of output hypotheses depends on the beam size and patience.

4.4 Jasper

Jasper [23] ("Just Another Speech Recognizer") is a family of end-to-end neural ASR models. Opposed to previous ASR models, Jasper doesn't contain a stack of attention blocks, it is a deep time-delay neural network (TDNN).

Jasper uses mel-filterbank (Section 2.3) features calculated from 20 ms windows, with 10 ms overlap. Jasper models are trained with a CTC decoder to generate an audio transcription.

[4] `https://pypi.org/project/whisper-normalizer/`
[5] `https://github.com/Martin-Toma/Whisper-Multi-Hyp`

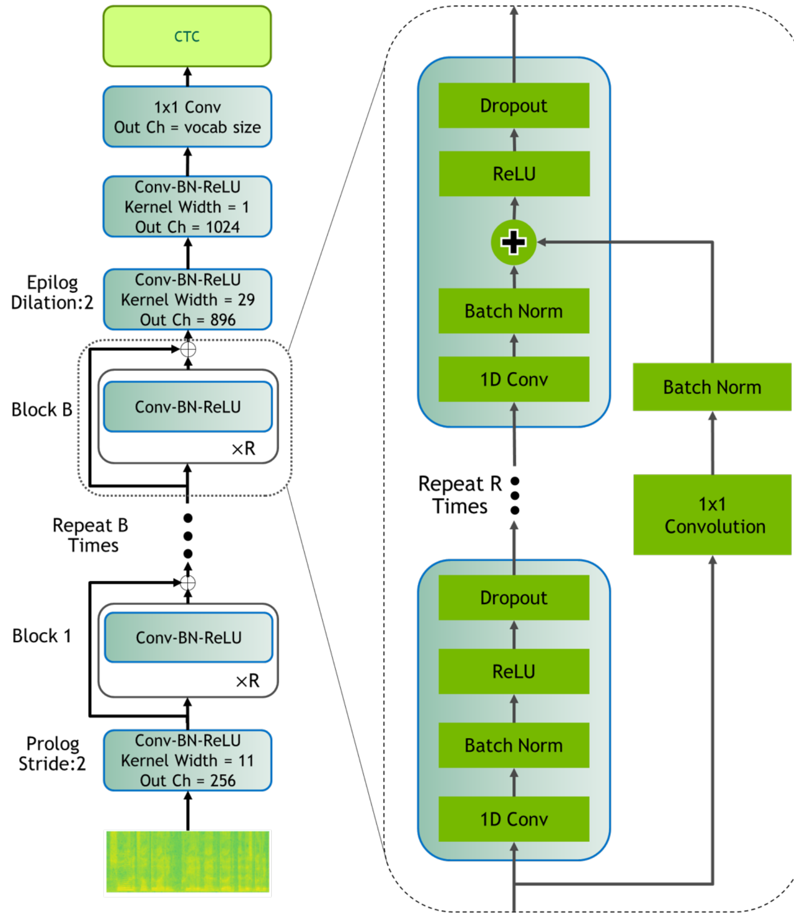


Figure 4.3: Jasper architecture. Block input is directly connected to the last sub-block. This connection is first projected through a 1×1 convolution. This residual projection handles different numbers of input and output channels. Then the connection is projected through a batch norm layer. The output of this batch norm layer is added to the batch norm layer output in the last sub-block. The block output is created by passing the sum through the activation function and dropout. There are four additional convolutional blocks in the Jasper models. One pre-processing block with stride 2 and three post-processing blocks. The figure is taken from [23].

The Jasper model has a block architecture (Figure 4.4). Blocks are composed of 1D-convolutional layers. All models are named by schema Jasper BxR model, where B is the number of blocks and R is the number of sub-blocks within each block. Each sub-block performs the following operations: a 1D-convolution, batch normalization, ReLU, and dropout. The sub-blocks in a block have the same number of output channels.

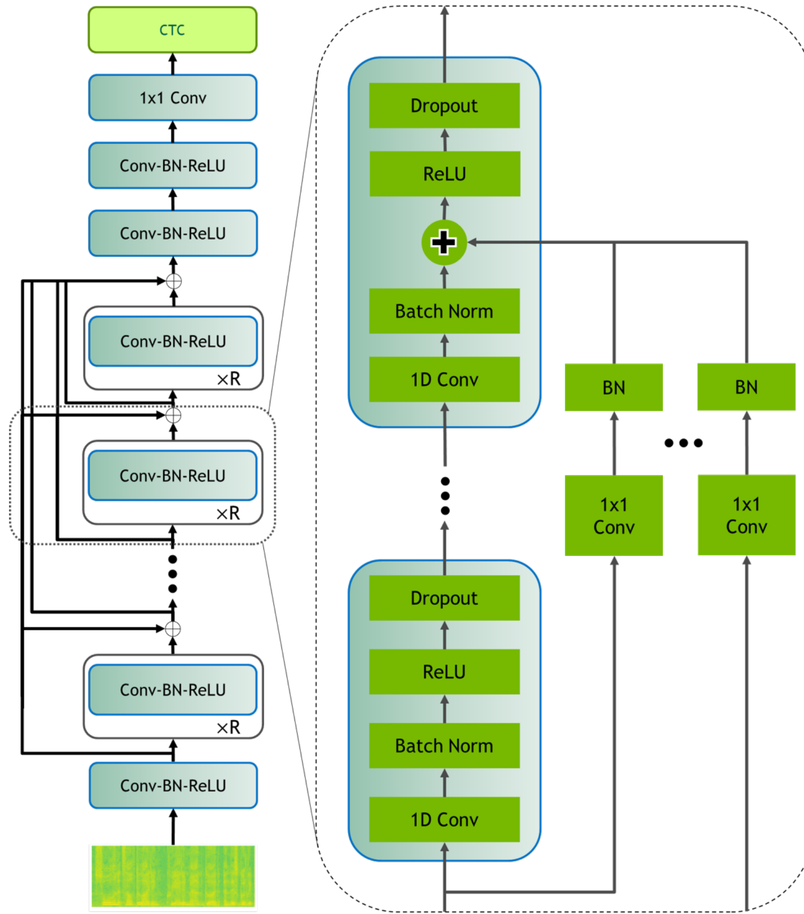


Figure 4.4: Jasper with Dense Residual. The figure is taken from [23].

The creators of Jasper tested DenseNet, DenseRNet and Dense Residual Jasper variants. The DenseNet and DenseRNet use concatenation to combine residual connections, they have a growth factor, which requires tuning for deeper models. The growth factor represents the number of filters in a Dense block of DenseNet and DenseRNet. The Dense Residual variant uses addition instead of concatenation, which means it has no growth factor to be tuned.

I used `STT En Jasper10x5dr`[6] model. It is a model with Dense Residual. with ten blocks, each with five subblocks. Input for this model must be in 16kHz Mono-channel audio.The model was trained on 7000 hours of English speech. The training data are from LibriSpeech, Wall Street Journal, Fisher English Training Speech, Switchboard, Mozilla Common Voice and NSC Singapore English (Part 1) datasets.

4.5 Autoregressive Large Language models

In the following part, I pinpoint the main characteristics of autoregressive (Section 3.15) LLM models. All of the models are based on the transformer architecture (Section 3.7).

[6] https://catalog.ngc.nvidia.com/orgs/nvidia/teams/nemo/models/stt_en_jasper10x5dr

GPT-2

GPT-2 [37] layer normalization (Section 3.11) is positioned at the input of each sub-block – before the MHA and the position-wise feed forward. To handle the accumulation on the residual path with model depth, the weights of residual layers are scaled at initialization, by a factor of $\frac{1}{\sqrt{N}}$ where N is the number of residual layers. GPT-2 has a context size of 1024 tokens.

The model uses the BPE tokenizer (Section 4.1) with a vocabulary of 50257 tokens.

There are four models of different sizes. I used 137M and 380M parameter GPT-2 models because they can be compared to BERT and RoBERTa models (Section 4.6) of approximately the same size.

Llama 2

Llama 2 [50]. Pre-normalization is applied using RMSNorm [57]. The SwiGLU activation function (Section 3.10) and RoPE (Section 3.8) are used. The primary architectural differences from Llama 1 [49] include increased context length and GQA (Section 3.9).

I used 7B and 13B parameter models of Llama 2.

TinyLlama

TinyLlama [58] was designed to explore the limits of small LLM saturation and to challenge the scaling rule.

TinyLlama is trained on around a trillion tokens. The training dataset is a mixture of natural language data from SlimPajama[7] and code data from StarCoderData.[8]

TinyLlama uses a tokenizer adopted from Llama 1 [49]. The architecture is based on the transformer architecture and is similar to Llama2's architecture (Section 4.5). To inject positional information into the model, RoPE (Section 3.8) is used for positional embedding. The input is normalized before each transformer sub-layer in pre-normalization. This normalization stabilizes training. To improve training efficiency, RMSNorm [57] is applied as a normalization technique. The activation function in TinyLlama (as in Llama2) is SwiGLU. TinyLlama uses GQA (Section 3.9), there are 32 heads for query attention and 4 groups of key-value heads.

Falcon

Falcon [4] models are released in three sizes: 7B, 40B and 180B. They are trained on diverse datasets obtained from web data.

Falcon uses GQA (Section 3.9) for scalability of inference,rotary embeddings (Section 3.8), parallel attention [28] and MLP blocks introduced in [55]. The biases from linear layers are removed to improve stability. GeLU (Section 3.10) serves as an activation function.

I used the 7B parameter version of the model.[9]

[7] https://www.cerebras.net/blog/slimpajama-a-627b-token-cleaned-and-deduplicated-version-of-redpajama

[8] https://huggingface.co/datasets/bigcode/starcoderdata

[9] https://huggingface.co/tiiuae/falcon-7b

MPT

MPT [48] does not use positional embeddings, it employs ALiBi (Section 3.9) and BPE tokenizer.[10] In contrast with the GPT-2 tokenizer, this tokenizer applies consistent space delimitation (ignores the prefix spaces and treats spaces as delimiters). The GPT-2 tokenizer tokenizes depending on the presence of prefix spaces. The MPT's tokenizer vocabulary has a size of 50257 tokens, but the model vocabulary is set to 50432 tokens. 50432 is a multiple of 128 which improves model FLOP utilization and there are tokens left for further training.

In the experiments, I utilized the 7B (more precisely 6.7B) parameter MPT model.[11]

Mistral

Mistral [20] uses Grouped-Query Attention (Section 3.9), Sliding-Window Attention (Section 3.9), Rolling Buffer Cache, Pre-fill and Chunking. The Mistral's vocabulary size is 32k.

Mistral 7B outperforms larger LLama2 13B parameters across all tested benchmarks [20].

I used the Mistral-7B-v0.1[12] model with a Byte-fallback BPE tokenizer. The Byte-fallback BPE tokenizer ensures that no character is mapped to OOV by casting unknown tokens into their byte representations.

KV caching

In autoregressive models, there is a problem in computing key and value vectors in attention from all previous steps again. This recalculation slows down inference. A mechanism called KV Cache [14] is used to mitigate this problem.

Key and value vectors from previous attention calculations are stored in the KV cache. Then they are reused to generate the next token.

The KV cache advantage is, that it makes inference faster. However, using it, a little more GPU VRAM or CPU RAM is needed for inference. Furthermore, memory consumption increases with context length and model size. The reason behind the memory consumption is that for each layer, projected k,v state stored in the memory.[13]

In the `transformers` library models use caching by default.

Rolling Buffer Cache

Rolling Buffer Cache [20] significantly reduces cache memory usage. It limits the size of the cache, which has a fixed size of W. Keys and values for timestep i are stored in the cache at position $i \mod W$. Former values are overwritten when i is larger than W. Also, memory size is not increasing after $i > W$. An example of how rolling buffer cache works is depicted in Figure 4.5.

[10] https://huggingface.co/EleutherAI/gpt-neox-20b
[11] https://huggingface.co/mosaicml/mpt-7b
[12] https://huggingface.co/mistralai/Mistral-7B-v0.1
[13] https://discuss.huggingface.co/t/generate-using-k-v-cache-is-faster-but-no-difference-to-memory-usage/31272/2

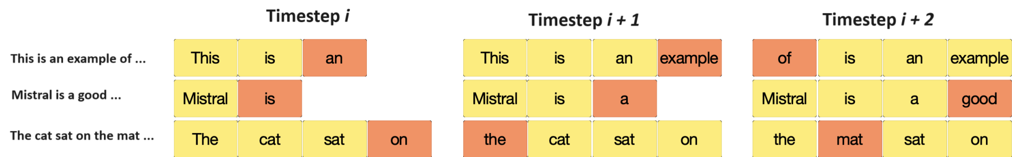


Figure 4.5: Rolling buffer cache. The fixed cache size is W = 4. When the position $i > W$, past values in the cache are overwritten. The hidden state corresponding to the latest generated tokens is in orange colour. The figure is taken from [20].

Pre-fill and Chunking

During scoring a sequence, each token is conditioned on the previous tokens, meaning the tokens need to be predicted one by one. But k, v cache can be pre-filled because the input tokens are known beforehand.

When the input is very large, the input can be chunked into smaller pieces. Then the cache is pre-filled with each chunk. The window size W determines the size of a chunk. For each chunk, the attention needs to be computed over the cache and the chunk.

4.6 Masked Language Models

In this section, I present additional transformer-based models. Specifically masked language models (Section 3.15): BERT and RoBERTa.

BERT

BERT [11] stands for Bidirectional Encoder Representations from Transformers. BERT's architecture is a multi-layer Transformer encoder based on the original Transformer (Section 3.7).

There are two sizes of the BERT model released. The smaller one called `BASE` of 110M parameters, 12 layers, 12 self-attention heads and 768 hidden size (number of features that are used to compute a hidden state) and `LARGE` of 340M parameters, 24 layers, 16 self-attention heads and 1024 hidden size.

Input for BERT can be a single sentence or a pair of question-answer separated by a special token `[SEP]`. The pair of sentences is also separated by adding a learned embedding to every token. It indicates to which sentence (the first or the second) the token belongs. Every input sequence of tokens starts with a special `[CLS]` token.

It is trained in two steps: pre-training and fine-tuning. Pre-training is performed by randomly masking some percentage of the input tokens – in the original BERT 15 % tokens were masked. Then during pre-training, the model tries to predict the masked tokens. During fine-tuning, `[MASK]` is not used. Because of this mismatch, not all masked tokens are replaced with `[MASK]`. In the training data, 15 % of the tokens are masked. Among these, 10 % are replaced with random token, 10 % are left unchanged and the remaining 80 % are replaced with `[MASK]` token. The decision of which masked token is changed, replaced with `[MASK]` or another token is random. The masking is static and it is done during data preprocessing, before training [25].

After pre-training, the model is fine-tuned for one downstream task. In fine-tuning all parameters are fine-tuned on labeled data.

RoBERTa

RoBERTa [25] stands for Robustly optimized BERT approach. RoBERTa is similar to BERT but was trained differently.

RoBERTa is trained with dynamic masking, full-sentences without Next Sentence Prediction (NSP) loss, large mini-batches and a larger byte-level BPE.

BERT uses static masking, the masks can repeat across epochs. RoBERTa used dynamic masking. It means the masking pattern is created every time a sequence is fed to the model. The advantage of dynamic masking is visible in pre-training with more steps or larger datasets.

The original BERT model batch size was small. Training with large mini-batches improves speed and end-task performance. Therefore, RoBERTa was trained with larger batches, consisting of 8K tokens.

Full-sentences mean inputs are packed with full sentences sampled contiguously from one or more documents. The total length of an input is 512 tokens. When document boundaries are crossed, a separator token is added between documents, but sampling continues without ending at the end of the document. The NSP loss, which is a binary classification loss used to predict whether two segments follow each other in the original sequence, is removed.

BERT uses a 30K token character-level BPE vocabulary, and RoBERTa uses bigger 50K tokens byte-level BPE vocabulary. This uses bytes instead of Unicode characters as the base subword units. Input text can be encoded without `unknown` tokens. For comparison, this change added 15M extra parameters to BERT `BASE` and 20M to `LARGE`. The byte-level BPE can worsen end-task performance.

Chapter 5

Experiments

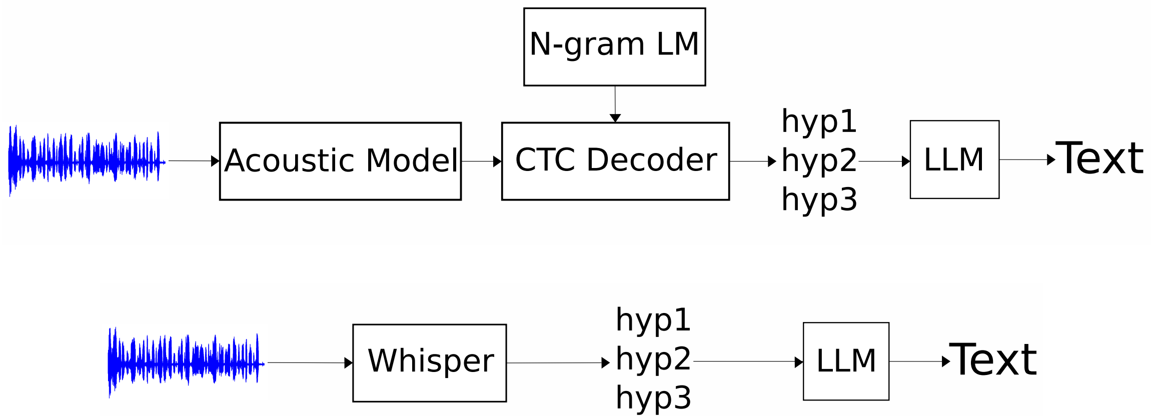


Figure 5.1: Standard pipelines shared by all experiments. The upper pipeline is the backbone of the Wav2Vec 2.0 and Jasper experiments. The lower pipeline is used in Whisper experiments, where the decoder does not use the n-gram language model. (The waveform representing audio input is genuine, it is plotted from the TED-LIUM dataset.)

The experiments share a common audio pipeline. At the start of the pipeline, there is an audio stored in a WAV file or in an array. The audio is obtained from datasets described in Section 5.1. The audio is fed to the ASR model, Whisper needs the audio to be preprocessed to mel frequency. Using a decoder with beam search, multiple transcription hypotheses are obtained. In research papers, it is a common practice to work with a hundred best hypotheses [52, 45], so I set the decoders accordingly. The transcription text is then extracted from the hypotheses, and fed to LLM (Section 3.15) to score.

$$Score_{hyp} = \alpha \cdot Score_{acoustic} + \beta \cdot Score_{LLM} + \gamma \cdot Score_{word} + \delta \cdot Score_{n-gram} \tag{5.1}$$

The Equation 5.1 describes rescoring, where α is the weight of the score obtained from the ASR model, β is the weight of the LLM model, γ is the weight of word insertion bonus, and δ is the weight of a small n-gram language model. Each score is multiplied by the corresponding weight, then the weighted scores are summed. The sum is the new hypothesis score. In the end, the hypothesis with the best (highest) new score is chosen

to be the best transcription. For each experiment, I found the best weights. The weight search is depicted in Figure 5.6.

The process of finding weights has two steps. At first, grid search is applied, and then the interval halving method is applied to obtain the best weights.

The LLM models in experiments are from huggingface transformers library.[1]

5.1 Datasets

I used LibriSpeech, GigaSpeech and TED-LIUM datasets in my experiments. These datasets contain audio of different difficulties, topics and sizes.

LibriSpeech

LibriSpeech [32] is a dataset derived from audiobooks. It contains 1000 hours of read speech. The audio is sampled at 16 kHz. All data is in English.

The dataset is divided into seven splits. There is no speaker overlap between training, development and test dataset splits. Audio in the train clean sets is of higher recording quality and speakers accents are closer to US English. To divide the data into subsets, the LibriSpeech authors trained an acoustic model on the Wall Street Journal's (WSJ) si-84 data subset and used it to recognize the audio in the dataset, using a bigram language model estimated on the text of the respective books. Then they compared the WER of the automatic transcription to the transcriptions obtained from the book texts. According to WSJ's transcriptions WER, speakers were divided into a "clean" pool with lower WER, and an "other" pool with higher WER. From each pool, 20 speakers were selected for each development and test subset. The rest of the "clean" audio was randomly split into training subsets, but in the training subset, each speaker's time was limited to 25 minutes.

For "other" development and training subsets, speakers were ranked from the least to the most difficult using WER from WSJ's models. Test and development subsets are spoken by speakers in the third quartile of the sorted list of speakers. All subsets with their sizes in hours are displayed in Table 5.1.

Table 5.1: LibriSpeech table displaying data subsets and their sizes in hours. The table is taken from [32]

subset	size [h]
train-clean-100	100.6
train-clean-360	363.6
train-clean-500	496.7
dev-clean	5.4
test-clean	5.4
dev-other	5.3
test-other	5.1

[1] https://huggingface.co/docs/transformers/index

GigaSpeech

GigaSpeech contains audio records and their transcriptions in English. Audio data are from different sources: audiobooks, podcasts and YouTube. This dataset consists of data from multiple categories: Business, Crime, History, Animals, Politics, Travel, Sports, Arts, Science, Technology, Vehicles, Film Health and many others.

The dataset is divided into seven subsets, five of which are training and are named `XS`, `S`, `M`, `L`, and `XL` from the smallest to the largest. A larger subset contains all data from all smaller subsets so M contains S and S contains XS data. Then there is a validation subset called "Dev" and a testing subset "Test". The duration of each split is depicted in Table 5.2.

Table 5.2: GigaSpeech table displaying data subsets and their sizes in hours. Data in the table are taken from github[2]

subset	size [h]
XS	10
S	250
M	1,000
L	2,500
XL	10,000
Dev	12
Test	40

The dataset text contains two groups of special tags. The first group of the special tags is called "Punctuations", there are these four tags: <COMMA>, <PERIOD>, <QUESTIONMARK>, <EXCLAMATIONPOINT>. These tags are replaced and removed before WER evaluation. The second group of special tags consists of these four tags: <SIL>, <MUSIC>, <NOISE>, <OTHER>. They are called "Garbage Utterance Tags". The audio annotated as garbage utterance, does not contain speech. Only Dev and Test dataset splits contain garbage utterances. The samples without text to transcribe are filtered at the beginning of the ASR pipeline.

Spontaneous speech in the dataset contains conversational fillers such as: 'UH', 'UHH', 'UM', 'EH', 'MM', 'HM', 'AH', 'HUH', 'HA', 'ER' [46]. The conversational fillers are recommended to be removed before WER measurement. The reason to remove them is that each ASR system may transcribe the conversational fillers differently and they do not carry valuable information. So the conversational fillers are removed in text post-processing, after rescoring.

TED-LIUM

TED-LIUM [40] is a dataset consisting of audio records from English TED talks.[3] The dataset is sampled at 16 kHz.

The TED talks have closed captions. the captions couldn't be used for ASR, because they are not verbatim transcriptions of the speech in the audio records. The closed captions do not contain repetition, and hesitations and some expressions are different in the captions. The timing is not aligned for ASR, only for humans to read it from the screen.

[3]https://www.ted.com/

Therefore, the TED-LIUM dataset was created by iteratively training acoustic models and decoding the whole dataset. The final version of the first release has 118 hours from 774 talks.

There are three releases of this dataset. I used the validation split which stays unchanged across releases, except for a special version of the third release with speaker adaptation. I also used the test split from the third release. The validation split I used contains 591 audio records and the test split contains 1467 audio records.

5.2 Fine-tuning

Across experiments, I test uncased BERT `BASE` model fine-tuned on text from `train-clean-100` subset of LibriSpeech dataset (Section 5.1). For the purpose of fine-tuning, the dataset was split into 85 % training part and 15 % validation part. The BERT model was fine-tuned for 8 epochs, after that evaluation loss rose. After the fine-tuning, evaluation loss lowered to 2.67 from 3.57 and pseudo-perplexity lowered to 14.50 from the original 35.64. The dataset has 990100 words, the number of tokens is 1083388 and the number of records is 28539. The training parameters I used were inspired by arcticle.[4]

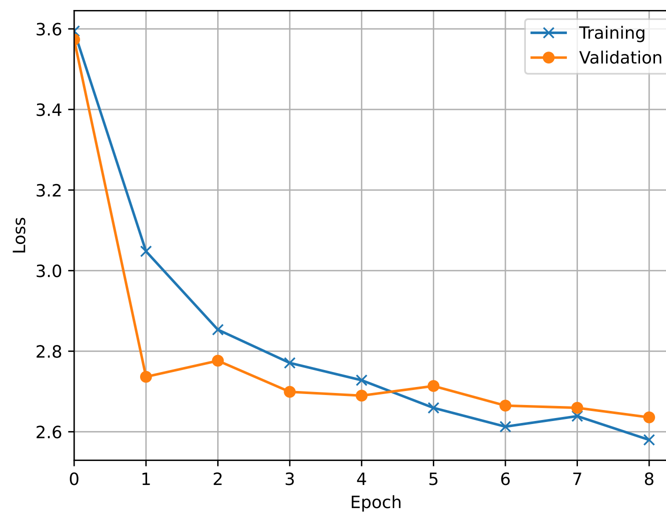


Figure 5.2: BERT `BASE` uncased fine-tuning on `train-clean-100` subset of LibriSpeech.

I fine-tuned GPT-2 on the same dataset as BERT. I tried fine-tuning with two learning rates: 10^{-3} which starts to overfit after 2 epochs and $5 \cdot 10^{-4}$, which starts to overfit after 3 epochs.

[4] https://towardsdatascience.com/fine-tuning-for-domain-adaptation-in-nlp-c47def356fd6

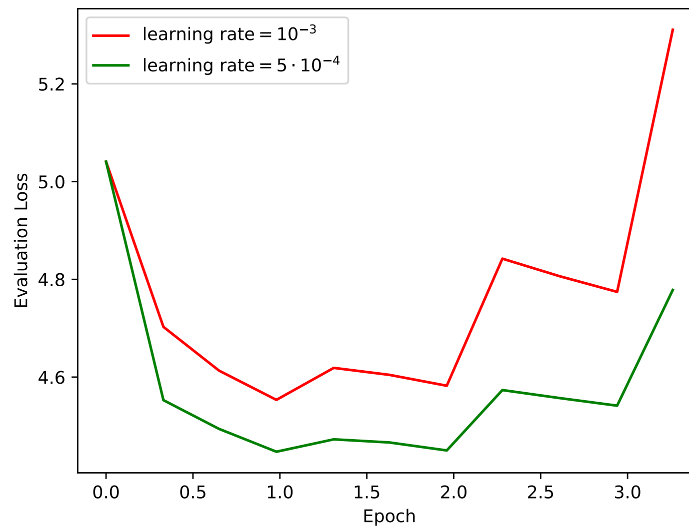


Figure 5.3: GPT-2 fine-tuning on `train-clean-100` subset of LibriSpeech.

I fine-tuned Llama2 7B on the text from the training `XL` subset of the GigaSpeech dataset. The dataset is split into 98 % training part and 2 % validation part. I used LoRA (Section 3.12) in the fine-tuning to lower resource requirements. The model was fine-tuned with three settings: $r = 8$, $r = 32$ and $r = 128$. In all settings, I used $\alpha = 16$ and dropout 0.1. I fine-tuned the Llama models for one epoch. The settings were inspired by article.[5]

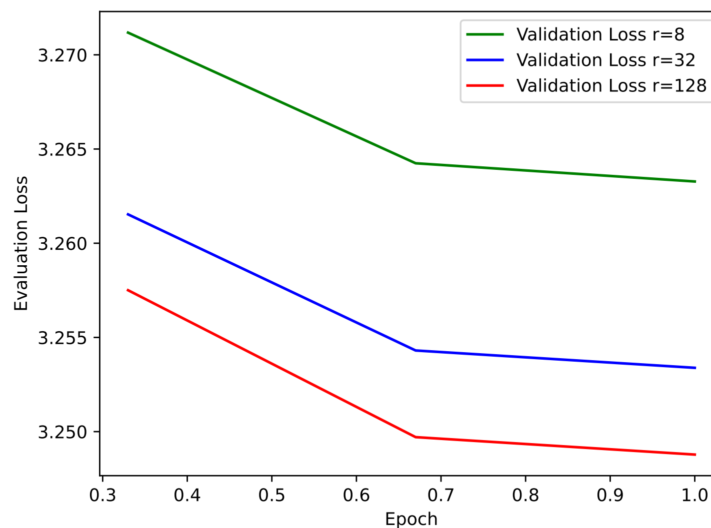


Figure 5.4: Llama 2 7B fine-tuning on `XL` subset of GigaSpeech.

[5] https://www.mldive.com/p/how-to-fine-tune-llama-2-with-lora

5.3 CTC decoder edit

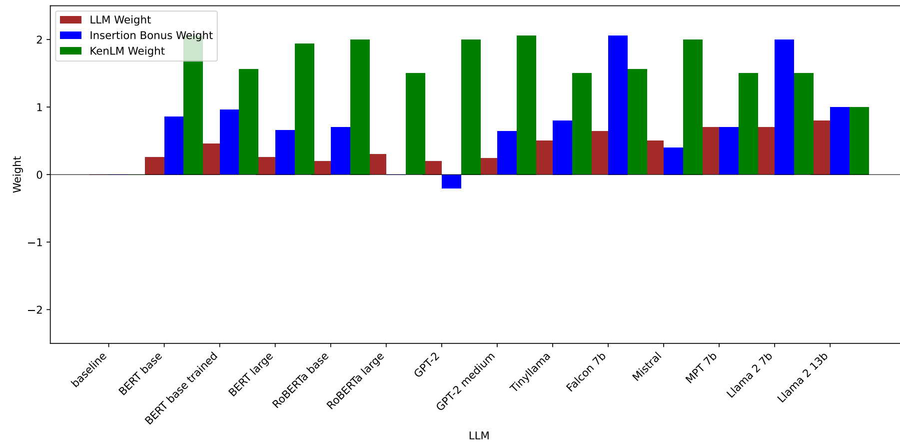
A commonly used `flashlight decoder`[6] outputs score for a hypothesis as a sum of weighted scores of n-gram, acoustic model score and so-called word score. I added to the flashlight decoder[7] printing to a file. The adjustment is in the file `Utils.h` in class `getAllHypothesis`. The order of hypothesis, emitting score, language model score and summed score are all output to a single line in the CSV file. From this file, the data about scores can be read in the rescoring process. I created a variant of the `CTCHypothesis` class which can store aforementioned score data and data about words and weights used during decoding.

5.4 Wav2Vec 2.0 experiments

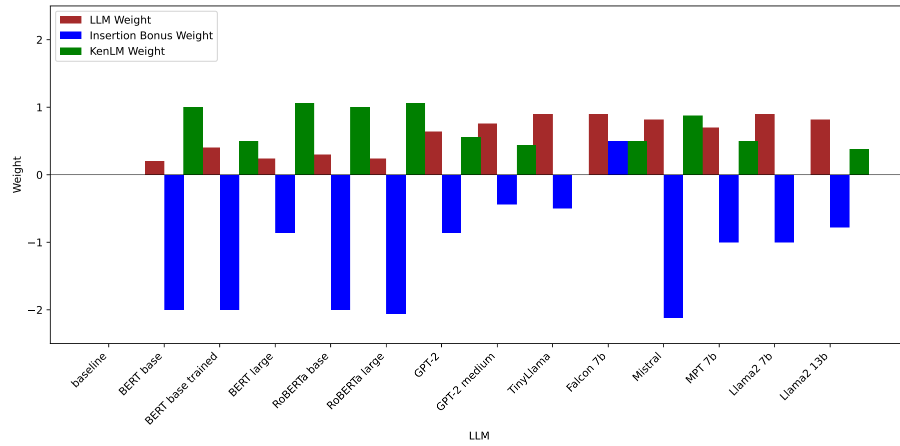
The experiments aim to compare LLMs to rescore the n-best hypothesis obtained from the decoded output of Wav2Vec 2.0. I used "dev" parts of datasets, from LibriSpeech I used "dev-other". Audio from the datasets was processed by Wav2Vec 2.0. Then, the weights for the CTC decoder had to be found. The process of finding the best weights for each dataset is following. Approximately half records from a dataset were randomly chosen. CTC decoder was constructed with a lower beam size set to 100 and an n-best set to 1. Weight of KenLM in range 0 to 4 with step 1 and word insertion score in range -2 to 2 with step 1 were searched. Then the best weights +-0.5 surrounding was searched. For each audio sample, 100 hypothetical transcriptions with scores were generated. These transcriptions are rescored using different language models. The final weights of scores are depicted in Figure 5.5.

The results of the experiments with LibriSpeech are presented in Appendix A.

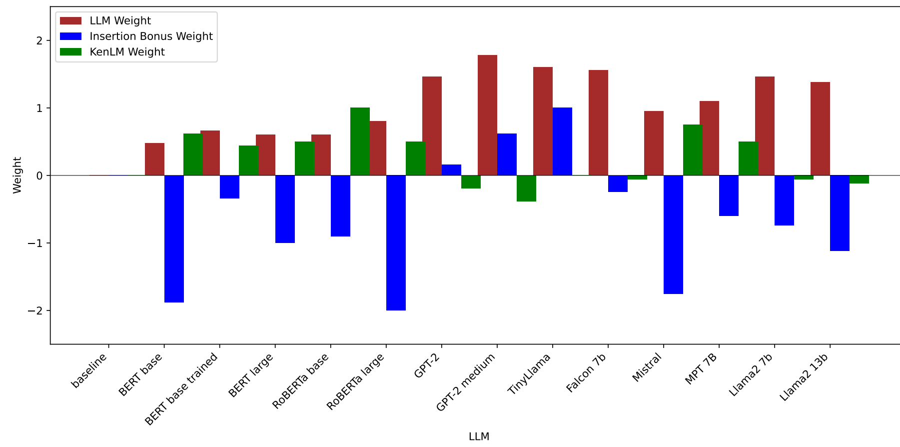
[6] https://github.com/flashlight/text/tree/main/flashlight/lib/text/decoder
[7] https://github.com/Martin-Toma/flashlightSepScores2



(a) LibriSpeech



(b) GigaSpeech



(c) TED-LIUM

Figure 5.5: Graphs of weights for rescoring Wav2Vec hypotheses on the three tested datasets.

Table 5.3: Overall WER Wav2Vec 2.0.

LLM	Parameters (in B)	LibriSpeech dev other			GigaSpeech dev			TED-LIUM dev
		KenLM Lex	Lex	No Lex	KenLM Lex	Lex	NoLex	KenLM Lex
baseline	–	8.59	9.33	9.77	28.49	30.37	31.34	17.81
BERT base	0.11	6.31	7.79	8.13	25.61	27.63	28.93	15.07
BERT base trained	0.11	**6.11**	7.45	7.87	25.32	27.28	28.59	14.93
BERT large	0.34	6.25	7.94	8.35	25.56	27.69	29.12	14.98
RoBERTa base	0.125	6.38	7.82	8.48	25.32	27.25	28.83	14.85
RoBERTa large	0.355	6.31	7.83	8.53	25.26	27.49	29.1	14.84
GPT-2	0.137	6.54	7.91	8.33	25.19	27.02	28.43	14.63
GPT-2 medium	0.380	6.44	7.64	8.16	25.05	26.93	28.34	14.57
TinyLlama	1.1	6.28	7.44	7.88	**24.52**	**26.47**	**28.19**	14.4
Falcon	7	**6.12**	**7.22**	7.77	**24.44**	26.58	**28.09**	**14.13**
Mistral	7	**6.11**	7.39	8.18	24.75	26.98	28.59	14.76
MPT	7	**6.09**	**7.26**	7.78	24.64	26.82	**28.09**	**14.08**
Llama2	7	**6.07**	**7.16**	**7.58**	24.59	**26.62**	**28.06**	**14.02**
Llama2	13	**5.92**	**7.07**	**7.51**	**24.44**	**26.5**	**28.07**	**13.99**

5.5 Jasper experiments

I generated 100 hypotheses using the Jasper model with NeMo beam search decoder.[8] The weight of n-gram language model is set by `alpha`, `beta` represents a penalty given to longer word sequences, meaning larger `beta` will result in shorter sequences. I found values of `alpha` and `beta` parameters of the decoder for each dataset by testing the WER of the randomly selected subset of the validation split. I used the interval halving method to find the best values.

The results of experiments with Jasper for each dataset are in Appendix B and Table 5.4. I tested the performance of rescoring on the test subset from the third release of TED-LIUM. The speech recognition of the test subset was run with hyperparameters found with the dev subset of TED-LIUM. The data (Table B.1) shows that rescoring with LLM reduces WER even on audio it was not tuned on. The best absolute WER improvement in the test dataset is 1 % absolute.

In experiments with Jasper, I used all fine-tuned models described in Section 5.2.

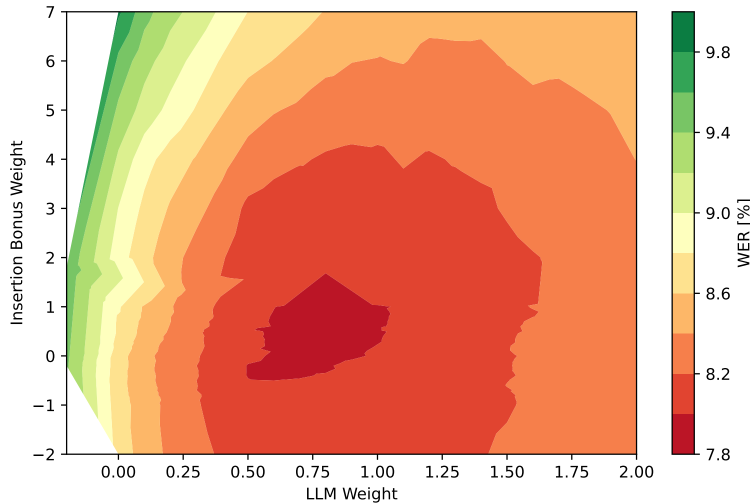


Figure 5.6: The relationship between WER values and the weights of rescoring LibriSpeech dev other using Llama2 7B with LoRA r = 32, hypotheses are obtained from Jasper. Base WER is 8.69 %

After the WER measurements, I compared the hypotheses with the highest score before and after the rescoring against reference transcriptions (Table 5.5). The change in the number of character insertions, deletions and substitutions is compared, these numbers are used to calculate WER (Section 2.2). The rescored hypotheses contain fewer inserted and substituted characters and more characters are deleted than in the referential texts from datasets compared to the non-rescored hypotheses. All measurements are in Appendix D.

[8] https://github.com/NVIDIA/NeMo/blob/main/nemo/collections/asr/modules/beam_search_decoder.py

Table 5.4: Overall WER Jasper.

LLM	Parameters (in B)	LibriSpeech	GigaSpeech	TED-LIUM	
		dev other	dev	dev	test
baseline	–	8.69	28.26	14.01	14.67
BERT base	0.11	8.44	27.88	13.53	14.67
BERT base trained	0.11	**8.22**	27.8	13.47	14.21
BERT large	0.34	8.45	27.85	13.38	14.5
RoBERTa base	0.125	8.51	27.7	13.23	14.17
RoBERTa large	0.355	8.44	27.71	13.27	14.27
GPT-2	0.137	8.59	27.77	13.24	14.15
GPT-2 trained 1	0.137	8.56	27.98	13.36	14.41
GPT-2 trained 2	0.137	8.49	27.92	13.33	14.24
GPT-2 medium	0.380	8.46	27.64	13.1	14.19
TinyLlama	1.1	8.45	27.49	13.03	**13.86**
Falcon	7	**8.26**	27.35	**12.91**	13.96
Mistral	7	**8.23**	27.52	**12.83**	14.15
MPT	7	**8.26**	27.6	13.02	14.44
Llama2	7	**8.24**	27.46	13.01	**13.81**
Llama2 LoRA r8	7	**7.98**	**27.25**	**12.76**	**13.73**
Llama2 LoRA r32	7	**7.92**	**27.26**	**12.75**	**13.62**
Llama2 LoRA r128	7	**7.94**	**27.27**	**12.76**	**13.68**
Llama2	13	**8.23**	27.46	**12.87**	**13.72**

Table 5.5: Experiment with Jasper and GigaSpeech dev dataset. Difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against dataset transcriptions.

LLM	insertions	deletions	substitutions
BERT base	1025	-1368	760
BERT base trained	1025	-1368	760
BERT large	996	-1361	696
RoBERTa base	1171	-1519	1037
RoBERTa large	889	-1086	992
GPT-2	1048	-1874	808
GPT-2 libri	1048	-1874	808
GPT-2 libri2	1048	-1874	808
GPT-2 medium	1261	-1951	1227
TinyLlama	1294	-1898	1428
Falcon 7B	1312	-1569	1540
Mistral	1118	-1094	1238
MPT 7B	1197	-1615	1364
Llama2 7B	1322	-1622	1491
Llama2 13B	1523	-2022	1658
Llama2 LoRA r8	1138	-1157	1510
Llama2 LoRA r32	1260	-1558	1703
Llama2 LoRA r128	1158	-1251	1586

5.6 Whisper experiments

Experiments were designed to test my hypothesis that using patience will improve WER. This hypothesis is based on improvement in the best possible WER in randomly chosen GigaSpeech recordings (Section 2.6). However, the use of a higher patience factor $p = 3$ in the rescoring experiments, slightly increased WER.

In the first experiments, Whisper output is fed to LLM without change. But before WER measurement, the Whisper transcripts are normalized with Whisper normalizer.[9] When the dataset is GigaSpeech or TED-LIUM, conversational fillers, hyphens and unnecessary spaces are removed. This post-processing is performed after applying Whisper normalization, before WER measurement. I applied this method to all three datasets and the results are in Appendix C, in Tables: GigaSpeech C.1, C.2, LibriSpeech C.5, C.6 and TED-LIUM C.3, C.4.

When I noticed the LLM rescoring does improve the WER of Whisper hypotheses, I decided to test if it was because of the format of the Whisper output text. So I tried four setups with the GigaSpeech dataset. The first is described above.

The second setup is the same as the first one, but input to the LLM is converted to lower-case (Tables C.7 and C.8). The third setup differs from the first in not applying the Whisper normalizer before WER measurement (Tables C.9 and C.10). The fourth setup is the same as the first one, but the input to the LLM is normalized using Whisper normalizer (Tables C.11 and C.12).

[9] https://pypi.org/project/whisper-normalizer/

In all experiments, the beam size is 40 and the patience parameter is set to 1 (without patience) and 3 (with patience).

The best weights of LLM scores that I found are in most experiments very close to zero, many times the weights are a small negative number. This indicates LLM scores do not improve the WER. In a few results, mainly in LibriSpeech experiments, I observed weights of LLM scores higher than 0.1 indicating the rescoring might improve WER. However, these weights are still very close to 0 and given all other measurements I state the n-best rescoring with LLM does not work with Whisper.

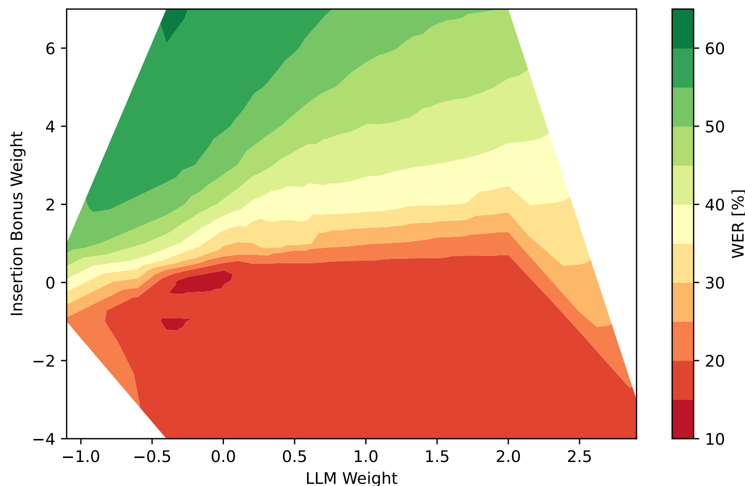


Figure 5.7: The relationship between WER values and the weights of rescoring with Llama2 13B. Hypotheses are obtained from Whisper with normalized input to LLM. Base WER is 14.52 %

5.7 Experiments summary

In the experiments, I tested three ASR models of different architectures and different settings, on three datasets, with multiple LLM models. I even fine-tuned a few of the LLM models with different settings.

The influence of LLM rescoring is determined by the quality of hypotheses obtained from ASR systems. The results of Wav2Vec experiments are depicted in Table 5.3 and Jasper experiments in Table 5.4. Whisper experiments did not demonstrate a strong correlation between WER reduction and the LLM rescoring.

The rescored hypotheses contain fewer inserted and substituted characters and more characters are deleted compared to non-rescored transcriptions, against the referential texts from datasets.

The best WER improvement measured is 4 % absolutely. The more parameters a model has, the better the results of rescoring. This is indicated by the finding that the larger the model, the bigger weight the LLM score should be multiplied with, to get the best WER. I found some exceptions in measurements with models GPT-2 and GPT-2 medium.

An interesting finding is, that after LLM in-domain fine-tuning, a smaller model (Llama 2 7B) can outperform a not fine-tuned model twice its size (Llama 2 13B). The improvement

relies on the fine-tuning data, with GPT-2 fine-tuned on LibriSpeech data, I observed WER decrease in LibriSpeech dev-other after rescoring, but WER increase in GigaSpeech and TED-LIUM after rescoring.

The rescoring depends on the ASR model, rescoring does not improve the WER of Whisper. One possible cause may be that the Whisper model does not produce very diverse hypotheses, many hypotheses differ only in punctuation. I even tried different processing of LLM input, but nothing worked. The fact LLM rescoring does not work may be caused by the Whisper hypotheses scores being very accurate.

The score separation in the CTC decoder was beneficial. Lower WERs were achieved with weighting separated scores.

When it comes to masked vs autoregressive models of the same size, WER after rescoring differs only slightly. However autoregressive (GPT-2 variants) are better for spontaneous speech (GigaSpeech, TED-LIUM) and masked (BERT and RoBERTa variants) are in most cases better for read speech (LibriSpeech). This fact is influenced by the training data. The masked models I used were trained mainly on book data. In contrast, GPT-2 models were trained on internet data. An important thing to consider when deciding between autoregressive and masked models is that the autoregressive model's rescoring is faster.

All 7B models visibly improved WER after rescoring. Surprisingly, a relatively small TinyLlama 1.1B improved WER competitively with bigger models.

Chapter 6

Conclusion

The thesis goal was to explore how to use LLMs in ASR and if they improve it. Additionally, factors that influence the results were examined. The experiments focus on improving WER by n-best rescoring using LLMs.

At first study about ASR, and LLM models and how to use them together was conducted. The n-best rescoring method was investigated further. Experiments with three ASR systems that produce multiple hypotheses, were designed. The experiments aim to test masked and autoregressive LLMs on three datasets. The datasets cover read and spontaneous speech. The observed metrics are WER and CER of the transcriptions against reference transcriptions. Three LLM models were selected for fine-tuning, and their performance in rescoring was measured. The CTC decoder used with Wav2Vec 2.0 was edited to output scores for hypotheses separately, and also the Whisper decoder was edited to output more than one hypothesis.

The results demonstrated a correlation between the number of LLM parameters and the WER reduction after rescoring. Other findings are that fine-tuned models can surpass non-fine-tuned ones twice their size. The n-best rescoring method depends on the quality of the hypotheses.

Possible extensions of this work may include further changes to the Whisper decoder to obtain more diverse hypotheses, for example by editing the temperature, because in the current Whisper implementation, the beam search can not be set with different temperatures. Another extension may be merging LLM models to obtain models with an exotic number of parameters and then testing these models' rescoring abilities.

The findings of the thesis were presented at Excel@FIT 2024.

Bibliography

[1] ABDUL, Z. K. and AL TALABANI, A. K. Mel Frequency Cepstral Coefficient and its Applications: A Review. *IEEE Access*. 2022, vol. 10, p. 122136–122158. DOI: 10.1109/ACCESS.2022.3223444.

[2] ABHANG, P. A., GAWALI, B. W. and MEHROTRA, S. C. *Introduction to EEG- and Speech-Based Emotion Recognition*. 1st ed. Amsterdam: Elsevier, 2016. ISBN 9780128044902. [cit. 2024-05-06]. Available at: https://doi.org/10.1016/C2015-0-01959-1.

[3] AINSLIE, J., LEE THORP, J., JONG, M. de, ZEMLYANSKIY, Y., LEBRÓN, F. et al. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. arXiv. 2023. DOI: 10.48550/ARXIV.2305.13245. Available at: https://arxiv.org/abs/2305.13245.

[4] ALMAZROUEI, E., ALOBEIDLI, H., ALSHAMSI, A., CAPPELLI, A., COJOCARU, R. et al. The Falcon Series of Open Language Models. arXiv. 2023. DOI: 10.48550/ARXIV.2311.16867. Available at: https://arxiv.org/abs/2311.16867.

[5] BAEVSKI, A., ZHOU, Y., MOHAMED, A. and AULI, M. Wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems*. 2020, vol. 33, p. 12449–12460.

[6] BORUAH, S. and BASISHTHA, S. *A study on HMM based speech recognition system*. IEEE, december 2013. DOI: 10.1109/iccic.2013.6724147. Available at: http://dx.doi.org/10.1109/ICCIC.2013.6724147.

[7] BRIDLE, J. Training Stochastic Model Recognition Algorithms as Networks can Lead to Maximum Mutual Information Estimation of Parameters. In: TOURETZKY, D., ed. *Advances in Neural Information Processing Systems*. Morgan-Kaufmann, 1989, vol. 2. Available at: https://proceedings.neurips.cc/paper_files/paper/1989/file/0336dcbab05b9d5ad24f4333c7658a0e-Paper.pdf.

[8] CHILD, R., GRAY, S., RADFORD, A. and SUTSKEVER, I. Generating Long Sequences with Sparse Transformers. arXiv. 2019. DOI: 10.48550/ARXIV.1904.10509. Available at: https://arxiv.org/abs/1904.10509.

[9] CHOLLET, F. *Deep Learning with Python*. 2nd ed. Manning Publications Co., 2021. ISBN 9781617296864.

[10] DAUPHIN, Y. N., FAN, A., AULI, M. and GRANGIER, D. Language Modeling with Gated Convolutional Networks. arXiv. 2016. DOI: 10.48550/ARXIV.1612.08083. Available at: https://arxiv.org/abs/1612.08083.

[11] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv. 2018. DOI: 10.48550/ARXIV.1810.04805. Available at: https://arxiv.org/abs/1810.04805.

[12] Freitag, M. and Al Onaizan, Y. *Beam Search Strategies for Neural Machine Translation*. Association for Computational Linguistics, 2017. DOI: 10.18653/v1/w17-3207. Available at: http://dx.doi.org/10.18653/v1/W17-3207.

[13] Gandhi, S., Srivastav, V., Khalusova, M. and Hollemans, M. *Evaluation metrics for ASR - hugging face audio course*. Jun 2023. Available at: https://huggingface.co/learn/audio-course/chapter5/evaluation.

[14] Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J. et al. Model Tells You What to Discard: Adaptive KV Cache Compression for LLMs. arXiv. 2023. DOI: 10.48550/ARXIV.2310.01801. Available at: https://arxiv.org/abs/2310.01801.

[15] Graves, A., Fernández, S., Gomez, F. and Schmidhuber, J. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In: *Proceedings of the 23rd International Conference on Machine Learning*. New York, NY, USA: Association for Computing Machinery, 2006, p. 369–376. ICML '06. DOI: 10.1145/1143844.1143891. ISBN 1595933832. Available at: https://doi.org/10.1145/1143844.1143891.

[16] Hannun, A. Sequence Modeling with CTC. *Distill*. 2017. DOI: 10.23915/distill.00008. https://distill.pub/2017/ctc.

[17] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. arXiv. 2015. DOI: 10.48550/ARXIV.1512.03385. Available at: https://arxiv.org/abs/1512.03385.

[18] Hu, E. J., shen yelong, Wallis, P., Allen Zhu, Z., Li, Y. et al. LoRA: Low-Rank Adaptation of Large Language Models. In: *International Conference on Learning Representations*. 2022. Available at: https://openreview.net/forum?id=nZeVKeeFYf9.

[19] Hu, K., Sainath, T. N., Li, B., Du, N., Huang, Y. et al. *Massively Multilingual Shallow Fusion with Large Language Models*. 2023.

[20] Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S. et al. Mistral 7B. arXiv. 2023. DOI: 10.48550/ARXIV.2310.06825. Available at: https://arxiv.org/abs/2310.06825.

[21] Jégou, H., Douze, M. and Schmid, C. *Product Quantization for Nearest Neighbor Search*. Institute of Electrical and Electronics Engineers (IEEE), january 2011. DOI: 10.1109/tpami.2010.57. Available at: http://dx.doi.org/10.1109/TPAMI.2010.57.

[22] Kasai, J., Sakaguchi, K., Bras, R. L., Radev, D., Choi, Y. et al. A Call for Clarity in Beam Search: How It Works and When It Stops. arXiv. 2022. DOI: 10.48550/ARXIV.2204.05424. Available at: https://arxiv.org/abs/2204.05424.

[23] Li, J., Lavrukhin, V., Ginsburg, B., Leary, R., Kuchaiev, O. et al. Jasper: An End-to-End Convolutional Neural Acoustic Model. arXiv. 2019. DOI: 10.48550/ARXIV.1904.03288. Available at: https://arxiv.org/abs/1904.03288.

[24] Liao, Y., Jiang, X. and Liu, Q. Probabilistically Masked Language Model Capable of Autoregressive Generation in Arbitrary Word Order. In: Jurafsky, D., Chai, J., Schluter, N. and Tetreault, J., ed. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, July 2020, p. 263–274. DOI: 10.18653/v1/2020.acl-main.24. Available at: https://aclanthology.org/2020.acl-main.24.

[25] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M. et al. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv. 2019. DOI: 10.48550/ARXIV.1907.11692. Available at: https://arxiv.org/abs/1907.11692.

[26] Lysandre Debut, S. G. *Perplexity of fixed-length models*. Febraury 2024. Available at: https://huggingface.co/docs/transformers/perplexity#perplexity-of-fixed-length-models.

[27] Malik, M., Malik, M. K., Mehmood, K. and Makhdoom, I. *Automatic speech recognition: a survey*. Springer Science and Business Media LLC, november 2020. DOI: 10.1007/s11042-020-10073-7. Available at: http://dx.doi.org/10.1007/s11042-020-10073-7.

[28] Medina, J. R. and Kalita, J. Parallel Attention Mechanisms in Neural Machine Translation. *ArXiv*. arXiv. 2018. DOI: 10.48550/ARXIV.1810.12427. Available at: https://arxiv.org/abs/1810.12427.

[29] Min, Z. and Wang, J. Exploring the integration of large language models into automatic speech recognition systems: An empirical study. In: Springer. *International Conference on Neural Information Processing*. 2023, p. 69–84.

[30] Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Madison, WI, USA: Omnipress, 2010, p. 807–814. ICML'10. ISBN 9781605589077.

[31] Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S. et al. A Comprehensive Overview of Large Language Models. *ArXiv*. 2023, abs/2307.06435. Available at: https://api.semanticscholar.org/CorpusID:259847443.

[32] Panayotov, V., Chen, G., Povey, D. and Khudanpur, S. Librispeech: An ASR corpus based on public domain audio books. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2015, p. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[33] Prabhavalkar, R., Hori, T., Sainath, T. N., Schlüter, R. and Watanabe, S. End-to-End Speech Recognition: A Survey. arXiv. 2023. DOI: 10.48550/ARXIV.2303.03329. Available at: https://arxiv.org/abs/2303.03329.

[34] Press, O., Smith, N. A. and Lewis, M. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. arXiv. 2021. DOI: 10.48550/ARXIV.2108.12409. Available at: https://arxiv.org/abs/2108.12409.

[35] Press, O. and Wolf, L. Using the Output Embedding to Improve Language Models. arXiv. 2016. DOI: 10.48550/ARXIV.1608.05859. Available at: https://arxiv.org/abs/1608.05859.

[36] Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C. et al. Robust Speech Recognition via Large-Scale Weak Supervision. arXiv. 2022. DOI: 10.48550/ARXIV.2212.04356. Available at: https://arxiv.org/abs/2212.04356.

[37] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. et al. Language models are unsupervised multitask learners. *OpenAI blog*. 2019, vol. 1, no. 8, p. 9.

[38] Ramachandran, P., Zoph, B. and Le, Q. V. Searching for Activation Functions. arXiv. 2017. DOI: 10.48550/ARXIV.1710.05941. Available at: https://arxiv.org/abs/1710.05941.

[39] Rosenfeld, R. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*. 2000, vol. 88, no. 8, p. 1270–1278. DOI: 10.1109/5.880083.

[40] Rousseau, A., Deléglise, P. and Estève, Y. TED-LIUM: an Automatic Speech Recognition dedicated corpus. In: Calzolari, N., Choukri, K., Declerck, T., Doğan, M. U., Maegaard, B. et al., ed. *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012, p. 125–129. Available at: http://www.lrec-conf.org/proceedings/lrec2012/pdf/698_Paper.pdf.

[41] Sennrich, R., Haddow, B. and Birch, A. Neural Machine Translation of Rare Words with Subword Units. arXiv. 2015. DOI: 10.48550/ARXIV.1508.07909. Available at: https://arxiv.org/abs/1508.07909.

[42] Shaw, P., Uszkoreit, J. and Vaswani, A. Self-Attention with Relative Position Representations. arXiv. 2018. DOI: 10.48550/ARXIV.1803.02155. Available at: https://arxiv.org/abs/1803.02155.

[43] Shazeer, N. Fast Transformer Decoding: One Write-Head is All You Need. arXiv. 2019. DOI: 10.48550/ARXIV.1911.02150. Available at: https://arxiv.org/abs/1911.02150.

[44] Shazeer, N. GLU Variants Improve Transformer. arXiv. 2020. DOI: 10.48550/ARXIV.2002.05202. Available at: https://arxiv.org/abs/2002.05202.

[45] Shin, J., Lee, Y. and Jung, K. Effective Sentence Scoring Method using Bidirectional Language Model for Speech Recognition. arXiv. 2019. DOI: 10.48550/ARXIV.1905.06655. Available at: https://arxiv.org/abs/1905.06655.

[46] SpeechColab. *SpeechColab/Gigaspeech: Large, modern dataset for speech recognition*. Available at: https://github.com/SpeechColab/GigaSpeech?tab=readme-ov-file.

[47] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. et al. RoFormer: Enhanced Transformer with Rotary Position Embedding. arXiv. 2021. DOI: 10.48550/ARXIV.2104.09864. Available at: https://arxiv.org/abs/2104.09864.

[48] Team, M. N. *Introducing MPT-7B: A New Standard for Open-Source, Commercially Usable LLMs*. 2023. Accessed: 2023-05-05. Available at: www.mosaicml.com/blog/mpt-7b.

[49] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A. et al. LLaMA: Open and Efficient Foundation Language Models. arXiv. 2023. DOI: 10.48550/ARXIV.2302.13971. Available at: https://arxiv.org/abs/2302.13971.

[50] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A. et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv. 2023. DOI: 10.48550/ARXIV.2307.09288. Available at: https://arxiv.org/abs/2307.09288.

[51] Turner, R. E. An Introduction to Transformers. arXiv. 2023. DOI: 10.48550/ARXIV.2304.10557. Available at: https://arxiv.org/abs/2304.10557.

[52] Udagawa, T., Suzuki, M., Kurata, G., Itoh, N. and Saon, G. Effect and Analysis of Large-scale Language Model Rescoring on Competitive ASR Systems. arXiv. 2022. DOI: 10.48550/ARXIV.2204.00212. Available at: https://arxiv.org/abs/2204.00212.

[53] Uday, K., Graham, K. and Emara, W. *Transformers for Machine Learning: A Deep Dive*. Crc Pr Inc, 2022. ISBN 9780367767341.

[54] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. *Attention Is All You Need*. 2023.

[55] Wang, B. *Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX* [https://github.com/kingoflolz/mesh-transformer-jax]. May 2021.

[56] Xue, L., Barua, A., Constant, N., Al Rfou, R., Narang, S. et al. ByT5: Towards a Token-Free Future with Pre-trained Byte-to-Byte Models. *Transactions of the Association for Computational Linguistics*. Cambridge, MA: MIT Press. 2022, vol. 10, p. 291–306. DOI: 10.1162/tacl_a_00461. Available at: https://aclanthology.org/2022.tacl-1.17.

[57] Zhang, B. and Sennrich, R. Root Mean Square Layer Normalization. arXiv. 2019. DOI: 10.48550/ARXIV.1910.07467. Available at: https://arxiv.org/abs/1910.07467.

[58] Zhang, P., Zeng, G., Wang, T. and Lu, W. TinyLlama: An Open-Source Small Language Model. arXiv. 2024. DOI: 10.48550/ARXIV.2401.02385. Available at: https://arxiv.org/abs/2401.02385.

Appendix A

Wav2Vec measurements

I created tables from each experiment with Wav2Vec 2.0. The following tables contain WER and CER after rescoring with LLM and also weights: β represents the weight of LLM score, γ is the weight of word insertion bonus and δ is the weight of KenLM score. The weight of the acoustic model is set to 1. The results of experiments with LibriSpeech are presented in Tables A.2, A.3, A.4, GigaSpeech in Tables A.5, A.6, A.7 and TED-LIUM in Table A.1.

Table A.1: WER and CER of Wav2Vec 2.0 with KenLM and lexicon on TED-LIUM dev. Decoded with KenLM weight = 2.0, word score = 0.0. According to the hypotheses, the best WER is 11.84 %, the worst WER is 26.37 %, the best CER is 4.40 % and the worst CER is 11.02 %.

LLM	β	γ	δ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	0	17.81	6.69
BERT base	0.48	-1.88	0.62	15.07	6.25
BERT base trained	0.66	-0.34	0.44	14.93	6.11
BERT large	0.6	-1.0	0.5	14.98	6.25
RoBERTa base	0.6	-0.9	1.0	14.85	6.14
RoBERTa large	0.8	-2.0	0.5	14.84	6.22
GPT-2	1.46	0.16	-0.19	14.63	6.22
GPT-2 medium	1.78	0.62	-0.38	14.57	6.18
TinyLlama	1.6	1.0	0.0	14.4	6.07
Falcon 7B	1.56	-0.24	-0.06	14.13	5.97
Mistral	0.95	-1.75	0.75	14.76	5.89
MPT 7B	1.1	-0.6	0.5	14.08	5.95
Llama2 7B	1.46	-0.74	-0.06	14.02	5.89
Llama2 13B	1.38	-1.12	-0.12	13.99	5.88

Table A.2: Compare WER and CER of Wav2Vec 2.0 on Librispeech dev-other with KenLM and lexicon. According to the hypotheses, the best WER = 2.99 %, the worst WER = 26.34 %, the best CER = 1.33 % and the worst CER = 10.36 %. Decoded with KenLM weight = 2.0, word score = 0.0.

LLM	parameters (B)	β	γ	δ	best WER[%] ↓	best CER[%] ↓
baseline	–	0	0	0	8.59	3.39
BERT base	0.11	0.26	0.86	2.06	6.31	2.89
BERT base trained	0.11	0.46	0.96	1.56	6.11	2.8
BERT large	0.34	0.26	0.66	1.94	6.25	2.87
RoBERTa base	0.125	0.2	0.7	2.0	6.38	2.94
RoBERTa large	0.355	0.3	0.0	1.5	6.31	2.87
GPT-2	0.137	0.2	-0.2	2.0	6.54	3.03
GPT-2 medium	0.38	0.24	0.64	2.06	6.44	3.0
TinyLlama	1.1	0.5	0.8	1.5	6.28	2.9
Falcon 7B	7	0.64	2.06	1.56	6.12	2.84
Mistral	7	0.5	0.4	2.0	6.11	2.8
MPT 7B	7	0.7	0.7	1.5	6.09	2.86
Llama 2 7B	7	0.7	2.0	1.5	6.07	2.8
Llama 2 13B	13	0.8	1.0	1.0	5.92	2.73

Table A.3: Compare WER and CER of Wav2Vec 2.0 on Librispeech dev-other with only lexicon. According to the hypotheses, the best WER = 4.96 %, the worst WER =27.34 %, the best CER = 1.92 %, the worst CER = 8.61 %. Decoded with word score = -1.5.

LLM	parameters (B)	β	γ	δ	best WER[%] ↓	best CER[%] ↓
baseline	–	0	0	0	9.33	3.51
BERT base	0.11	0.4	−0.7	0.0	7.79	3.2
BERT base trained	0.11	0.44	0.54	0.0	7.45	3.11
BERT large	0.34	0.36	−0.84	0.0	7.94	3.22
RoBERTa base	0.125	0.3	−0.2	0.0	7.82	3.21
RoBERTa large	0.355	0.4	0.3	0.0	7.83	3.19
GPT-2	0.137	0.6	−0.5	0.0	7.91	3.3
GPT-2 medium	0.380	0.64	0.06	0.0	7.64	3.2
TinyLlama	1.1	0.7	−0.3	0.0	7.44	3.15
Falcon 7B	7	0.9	0.0	0.0	7.22	3.07
Mistral	7	0.66	−0.34	0.0	7.39	3.06
MPT 7B	7	0.7	0.6	0.0	7.26	3.09
Llama 2 7B	7	0.8	−0.3	0.0	7.16	3.05
Llama 2 13B	13	0.9	−0.6	0.0	7.07	3.03

Table A.4: WER and CER of Wav2Vec 2.0 on LibriSpeech dev-other without lexicon. According to the hypotheses, the best WER is 5.58 %, the worst WER is 27.84 %, the best CER is 2.11 % and the worst CER is 8.04 %.

LLM	parameters (B)	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	0	0	9.77	3.68
BERT base	0.11	0.353125	0.484375	8.13	3.29
BERT base trained	0.11	0.446875	0.446875	7.87	3.24
BERT large	0.34	0.33125	0.53125	8.35	3.33
RoBERTa base	0.125	0.2	0.6	8.48	3.37
RoBERTa large	0.355	0.23125	-0.63125	8.53	3.4
GPT-2 medium	0.380	0.6625	0.7625	8.16	3.3
GPT-2	0.137	0.615625	0.984375	8.33	3.38
TinyLlama	1.1	0.6	0.9	7.88	3.25
Falcon 7B	7	0.715625	0.284375	7.77	3.18
Mistral	7	0.4375	0.8625	8.18	3.27
MPT 7B	7	0.684375	-0.484375	7.78	3.19
Llama2 7B	7	0.9	1.0	7.58	3.1
Llama2 13B	13	0.984375	0.753125	7.51	3.1

Table A.5: WER and CER of Wav2Vec 2.0 with KenLM and lexicon on GigaSpeech dev. Decoded with KenLM weight = 1.5, word score = -1.5. According to the hypotheses, the best WER is 18.57 %, the worst WER is 42.51 %, the best CER is 10.15 % and the worst CER is 22.96 %.

LLM	β	γ	δ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	0	28.49	14.99
BERT base	0.2	-2.0	1.0	25.61	15.0
BERT base trained	0.4	-2.0	0.5	25.32	14.82
BERT large	0.24	-0.86	1.06	25.56	14.81
RoBERTa base	0.3	-2.0	1.0	25.32	14.99
RoBERTa large	0.24	-2.06	1.06	25.26	14.97
GPT-2	0.64	-0.86	0.56	25.19	14.91
GPT-2 medium	0.76	-0.44	0.44	25.05	14.86
TinyLlama	0.9	-0.5	0.0	24.52	14.61
Falcon 7B	0.9	0.5	0.5	24.44	14.45
Mistral	0.82	-2.12	0.88	24.75	14.68
MPT 7B	0.7	-1.0	0.5	24.64	14.66
Llama2 7B	0.9	-1.0	0.0	24.59	14.79
Llama2 13B	0.82	-0.78	0.38	24.44	14.79

Table A.6: WER and CER of Wav2Vec 2.0 with only lexicon on GigaSpeech dev. Decoded with word score = -2.5. According to the hypotheses, the best WER is 22.89 %, the worst WER is 43.37 %, the best CER is 11.42 % and the worst CER is 21.19 %.

LLM	β	γ	δ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	0	30.37	15.26
BERT base	0.38	-2.12	-0.12	27.63	15.14
BERT base trained	0.56	-2.06	-0.06	27.28	15.05
BERT large	0.3	-2.0	0.0	27.69	15.08
RoBERTa base	0.38	-2.12	-0.12	27.25	14.97
RoBERTa large	0.36	-2.06	-0.06	27.49	15.06
GPT-2	0.78	-1.12	-0.12	27.02	14.92
GPT-2 medium	0.8	-1.0	0.0	26.93	14.96
TinyLlama	0.9	-1.0	0.0	26.47	14.85
Falcon 7B	0.78	-1.02	-0.12	26.58	14.7
Mistral	0.72	-2.12	-0.12	26.98	14.78
MPT 7B	0.9	-1.0	0.0	26.82	14.88
Llama2 7B	0.84	-1.94	-0.19	26.62	14.96
Llama2 13B	0.64	-1.94	-0.06	26.5	14.84

Table A.7: WER and CER of Wav2Vec 2.0 without KenLM and lexicon on GigaSpeech dev. According to the hypotheses, the best WER is 24.54 %, the worst WER is 45.00 %, the best CER is 11.90 % and the worst CER is 20.32 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	31.34	15.31
BERT base	0.515625	-1.984375	28.93	15.03
BERT base trained	0.36875	-2.03125	28.59	14.92
BERT large	0.26875	-0.83125	29.12	15.01
RoBERTa base	0.26875	-1.96875	28.83	14.96
RoBERTa large	0.2	-2.0	29.1	14.94
GPT-2	0.7	-2.0	28.43	14.87
GPT-2 medium	0.7375	-1.9375	28.34	14.82
TinyLlama	0.646875	-0.978125	28.19	14.83
Falcon 7B	0.76875	-1.96875	28.09	14.7
Mistral	0.3375	-0.8375	28.59	14.84
MPT 7B	0.653125	-1.953125	28.09	14.68
Llama2 7B	0.8	-2.0	28.06	14.78
Llama2 13B	1.309375	-3.109375	28.07	14.77

Appendix B

Jasper measurements

I created tables from each experiment with Jasper. The following tables contain WER and CER after rescoring with LLM and also weights: β represents the weight of the LLM score and γ is the weight of the word insertion bonus. The weight of the acoustic model is set to 1. The results of experiments with Jasper for each dataset are Tables: TED-LIUM B.1, LibriSpeech B.2 and GigaSpeech B.3.

Table B.1: Jasper with KenLM on TED-LIUM dev and test release 3. According to the hypotheses, the best WER is 10.68 %, the worst WER is 25.36 %, the best CER is 3.57 % and the worst CER is 8.48 %. Decoder settings: `alpha` = 0.5, `beta` = 0.0.

LLM	β	γ	best WER[%] ↓		best CER[%] ↓	
			dev	test	dev	test
baseline	0	0	14.01	14.67	4.96	4.52
BERT base	0.115625	-3.984375	13.53	14.67	4.88	4.69
BERT base trained	0.3375	-3.9375	13.47	14.21	4.94	4.55
BERT large	0.415625	-0.984375	13.38	14.5	4.85	4.61
RoBERTa base	0.315625	-0.784375	13.23	14.17	4.82	4.38
RoBERTa large	0.3	0.0	13.27	14.27	4.75	4.39
GPT-2 base	0.5375	-1.0625	13.24	14.15	4.84	4.49
GPT-2 l.r. 10^{-3}	1.03125	6.03125	13.36	14.41	4.74	4.44
GPT-2 l.r. $5 \cdot 10^{-4}$	0.93125	4.03125	13.33	14.24	4.76	4.38
GPT-2 medium	1.315625	1.984375	13.1	14.19	4.8	4.49
TinyLlama	0.9	-0.6	13.03	13.86	4.75	4.46
Falcon 7B	1.884375	2.015625	12.91	13.96	4.73	4.51
Mistral 7B	0.653125	-1.953125	12.83	14.15	4.58	4.45
MPT 7B	1.778125	-1.078125	13.02	14.44	4.95	4.76
Llama2 7B	1.5625	2.0625	13.01	13.81	4.71	4.45
Llama2 7B LoRa r8	1.215625	0.184375	12.76	13.73	4.61	4.32
Llama2 7B LoRa r32	1.0375	-0.8375	12.75	13.62	4.63	4.3
Llama2 7B LoRa r128	1.7	-0.2	12.76	13.68	4.65	4.33
Llama2 13B	0.9	0.1	12.87	13.72	4.6	4.32

Table B.2: Jasper with KenLM on LibriSpeech dev-other. According to the hypotheses, the best WER is 5.80 %, the worst WER is 28.08 %, the best CER is 2.82 % and the worst CER is 10.18 %. Decoder settings: `alpha` = 1.5, `beta` = 0.5.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	8.69	4.22
BERT base	0.284375	-0.815625	8.44	4.14
BERT base trained	0.5	-1.0	8.22	4.06
BERT large	0.175	-0.575	8.45	4.13
RoBERTa base	0.215625	-0.715625	8.51	4.16
RoBERTa large	0.3	-0.8	8.44	4.14
GPT-2	0.3	0.1	8.59	4.21
GPT-2 l.r. 10^{-3}	0.3	-0.7	8.56	4.21
GPT-2 l.r. $5 \cdot 10^{-4}$	0.4	-0.2	8.49	4.2
GPT-2 medium	0.484375	0.053125	8.46	4.18
TinyLlama	0.46875	-0.66875	8.45	4.16
Falcon 7B	0.684375	0.915625	8.26	4.07
Mistral	0.584375	-1.015625	8.23	4.06
MPT 7B	0.384375	-0.515625	8.26	4.11
Llama2 7B	0.53125	-0.43125	8.24	4.11
Llama2 LoRA r8	0.7375	0.7375	7.98	3.98
Llama2 LoRA r32	0.846875	0.846875	7.92	3.98
Llama2 LoRA r128	0.715625	0.653125	7.94	3.98
Llama2 13B	0.6	0.3	8.23	4.08

Table B.3: Jasper with KenLM on GigaSpeech. According to the hypotheses, the best WER is 24.49 %, the worst WER is 40.76 %, the best CER is 24.22 % and the worst CER is 0.30294534943653106. Decoder settings: `alpha` = 0.5, `beta` = 0.5.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	0	0	28.26	26.49
BERT base	0.1	-2.0	27.88	26.55
BERT base trained	0.253125	-0.715625	27.8	26.47
BERT large	0.1	-1.0	27.85	26.48
RoBERTa base	0.246875	-1.015625	27.7	26.46
RoBERTa large	0.184375	-0.484375	27.71	26.44
GPT-2	0.4625	0.3625	27.77	26.48
GPT-2 l.r. 10^{-3}	0.215625	-0.984375	27.98	26.55
GPT-2 l.r. $5 \cdot 10^{-4}$	0.2375	-0.8375	27.92	26.52
GPT-2 medium	0.6	0.3	27.64	26.48
TinyLlama	0.6	-0.1	27.49	26.44
Falcon 7B	0.8	0.6	27.35	26.37
Mistral	0.484375	-0.915625	27.52	26.36
MPT 7B	0.33125	-0.46875	27.6	26.43
Llama2 7B	0.53125	-0.16875	27.46	26.4
Llama2 LoRA r8	0.56875	0.26875	27.25	26.31
Llama2 LoRA r32	0.7	-0.1	27.26	26.35
Llama2 LoRA r128	0.6	0.2	27.27	26.32
Llama2 13B	0.53125	-0.73125	27.46	26.42

Appendix C

Whisper measurements

I created tables from each experiment with Whisper medium. The following tables contain WER and CER after rescoring with LLM and also weights: β represents the weight of the LLM score and γ is the weight of the word insertion bonus. The weight of the acoustic model is set to 1. The results of experiments in which text fed to LLMs is not normalized, but after rescoring is normalized, are in Tables: GigaSpeech C.1, C.2, LibriSpeech C.5, C.6 and TED-LIUM C.3, C.4. Except for the first setup described above, three other setups were tried for the GigaSpeech dataset. The second setup is the same as the first one, but input to the LLM is converted to lower-case (Tables C.7 and C.8). The third setup differs from the first in not applying the Whisper normalizer before WER measurement (Tables C.9 and C.10). The fourth setup is the same as the first one, but the input to the LLM is normalized using Whisper normalizer (Tables C.11 and C.12). In all experiments, the beam size is 40 and the patience parameter is set to 1 (without patience) and 3 (with patience).

Table C.1: Whisper on GigaSpeech dev, without patience $p = 1$. According to the hypotheses, the best WER is 9.11 %, the worst WER is 24.70 %, the best CER is 5.60 % and the worst CER is 18.22 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.62	10.10
BERT base	-0.015625	0.384375	13.01	8.19
BERT base trained	-0.015625	0.384375	12.96	8.15
BERT large	-0.015625	0.384375	13.02	8.2
RoBERTa base	-0.015625	0.384375	12.98	8.16
RoBERTa large	0.03125	0.43125	13.0	8.28
GPT-2	-0.1	0.0	12.99	8.07
GPT-2 medium	-0.03125	0.36875	12.96	8.14
TinyLlama	-0.015625	0.353125	12.98	8.18
Falcon 7B	-0.015625	0.353125	12.96	8.18
Mistral	-0.015625	0.353125	12.98	8.19
MPT 7B	-0.015625	0.353125	12.95	8.17
Llama2 7B	-0.015625	0.353125	12.97	8.18
Llama2 13B	-0.015625	0.353125	12.96	8.17

Table C.2: Whisper on GigaSpeech dev, with patience $p = 3$. According to the hypotheses, the best WER is 8.11 %, the worst WER is 66.99 %, the best CER is 4.81 % and the worst CER is 61.78 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.52	9.96
BERT base	-0.0625	0.1375	12.93	8.1
BERT base trained	-0.13125	0.03125	13.01	8.01
BERT large	-0.046875	0.184375	13.05	8.23
RoBERTa base	-0.015625	0.184375	13.2	8.4
RoBERTa large	-0.015625	0.184375	13.19	8.42
GPT-2	-0.1	0.0	12.83	7.86
GPT-2 medium	-0.084375	0.015625	12.91	7.96
TinyLlama	-0.084375	0.015625	13.0	8.05
Falcon 7B	-0.06875	0.03125	13.02	8.09
Mistral	-0.084375	0.046875	13.09	8.14
MPT 7B	-0.1	0.0	12.99	8.0
Llama2 7B	-0.084375	0.046875	13.05	8.08
Llama2 13B	-0.053125	0.046875	13.06	8.2

Table C.3: Whisper on TED-LIUM dev, without patience $p = 1$. According to the hypotheses, the best WER is 4.59 %, the worst WER is 12.49 %, the best CER is 3.23 % and the worst CER is 9.67 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	8.78	6.89
BERT base trained	0.246875	3.046875	5.91	4.08
BERT base	-0.015625	0.415625	5.98	4.07
BERT large	-0.015625	0.384375	5.99	4.08
RoBERTa base	0.03125	0.43125	6.0	4.12
RoBERTa large	0.03125	0.36875	5.98	4.13
GPT-2 medium	-0.015625	0.384375	5.99	4.09
GPT-2	-0.078125	0.353125	5.99	4.05
Falcon 7B	0.0	0.4	6.02	4.13
Mistral	-0.1	1.0	6.0	3.94
MPT 7B	0.0	0.4	6.02	4.13
Llama2 7B	0.0	0.4	6.02	4.13
Llama2 13B	0.015625	0.446875	5.99	4.1

Table C.4: Whisper on TED-LIUM dev, with patience $p = 3$. According to the hypotheses, the best WER is 4.02 %, the worst WER is 40.87 %, the best CER is 2.79 and the worst CER is 37.25 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	8.79	6.89
BERT base trained	-0.1	0.0	6.43	4.35
BERT base	-0.090625	0.171875	6.24	4.15
BERT large	-0.10625	0.09375	6.21	4.17
RoBERTa base	-0.028125	0.171875	6.2	4.24
RoBERTa large	-0.046875	0.178125	6.24	4.24
GPT-2 medium	-0.115625	0.046875	5.92	3.94
GPT-2	-0.13125	0.03125	5.88	3.87
Falcon 7B	-0.215625	0.015625	6.06	4.0
Mistral	-0.240625	-0.109375	6.2	3.95
MPT 7B	-0.2	0.0	6.15	3.98
Llama2 7B	-0.2	0.0	6.02	3.92
Llama2 13B	-0.16875	0.03125	6.14	4.03

Table C.5: Whisper on LibriSpeech dev-other, without patience $p = 1$. According to the hypotheses, the best WER is 5.83 %, the worst WER is 16.30 %, the best CER is 3.07 % and the worst CER is 9.64 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	9.37	5.22
BERT base trained	0.1	0.7	8.86	4.99
BERT base	0.084375	0.315625	8.68	4.83
BERT large	0.084375	0.284375	8.6	4.85
RoBERTa base	0.03125	0.33125	8.96	4.87
RoBERTa large	0.1	1.0	8.94	5.01
GPT-2 medium	-0.015625	0.315625	9.0	4.76
GPT-2	-0.015625	0.315625	8.98	4.75
Falcon 7B	0.1	1.0	8.99	5.05
Mistral	0.1	0.9	8.97	5.04
MPT 7B	0.115625	0.915625	8.95	5.05
Llama2 7B	0.2	2.0	8.89	4.99
Llama2 13B	0.2625	2.0625	8.8	5.01

Table C.6: Whisper on LibriSpeech dev-other, with patience $p = 3$. According to the hypotheses, the best WER is 5.35 %, the worst WER is 49.61 %, the best CER is 2.81 % and the worst CER is 43.74 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	9.39	5.25
BERT base trained	-0.03125	-0.06875	9.02	4.93
BERT base	0.0	-0.1	9.08	5.16
BERT large	0.0	-0.1	9.08	5.16
RoBERTa base	0.0	-0.1	9.08	5.16
RoBERTa large	-0.015625	-0.084375	9.05	5.05
GPT-2 medium	-0.0625	-0.1625	8.89	4.67
GPT-2	-0.0625	-0.1625	8.86	4.66
Falcon 7B	-0.015625	-0.084375	8.93	4.97
Mistral	-0.046875	-0.115625	8.96	4.79
MPT 7B	-0.046875	-0.146875	9.03	4.85
Llama2 7B	-0.046875	-0.115625	8.97	4.78
Llama2 13B	-0.015625	-0.084375	8.98	4.99

Table C.7: Whisper on GigaSpeech dev, without patience $p = 1$. Lower-cased input to the LLM. According to the hypotheses, the best WER is 9.11 %, the worst WER is 24.70 %, the best CER is 5.60 % and the worst CER is 18.22 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.62	10.10
BERT base	-0.015625	0.384375	13.01	8.19
BERT base trained	-0.015625	0.384375	12.96	8.15
BERT large	-0.015625	0.384375	13.02	8.2
RoBERTa base	-0.015625	0.384375	12.99	8.19
RoBERTa large	-0.015625	0.384375	12.99	8.19
GPT-2	-0.1	0.0	12.99	8.13
GPT-2 medium	-0.03125	0.36875	12.95	8.12
TinyLlama	-0.015625	0.446875	12.86	8.04
Falcon 7B	-0.015625	0.353125	12.96	8.18
Mistral	-0.015625	0.353125	12.99	8.21
MPT 7B	-0.015625	0.353125	12.96	8.17
Llama2 7B	-0.015625	0.353125	12.96	8.18
Llama2 13B	-0.03125	0.36875	12.96	8.14

Table C.8: Whisper on GigaSpeech dev, with patience $p = 3$. Lower-cased input to the LLM. According to the hypotheses, the best WER is 8.11 %, the worst WER is 66.99 %, the best CER is 4.81 % and the worst CER is 61.78 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.52 9.96	
BERT base	-0.0625	0.1375	12.93	8.1
BERT base trained	-0.13125	0.03125	13.01	8.01
BERT large	-0.046875	0.184375	13.05	8.23
RoBERTa base	-0.078125	0.184375	13.1	8.19
RoBERTa large	-0.015625	0.184375	13.16	8.4
GPT-2	-0.084375	0.015625	12.79	7.97
GPT-2 medium	-0.084375	0.046875	12.82	7.95
TinyLlama	-0.13125	0.03125	12.96	7.96
Falcon 7B	-0.084375	0.015625	13.0	8.02
Mistral	-0.084375	0.046875	13.06	8.08
MPT 7B	-0.084375	0.046875	12.91	7.98
Llama2 7B	-0.084375	0.046875	12.97	8.03
Llama2 13B	-0.06875	0.09375	12.92	8.02

Table C.9: Whisper on GigaSpeech dev, without patience $p = 1$. Without normalization. According to the hypotheses, the best WER is 15.89 %, the worst WER is 35.61 %, the best CER is 6.97 % and the worst CER is 20.55 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	24.45	12.11
BERT base	-0.215625	0.109375	20.77	9.93
BERT base trained	0.4	3.0	20.87	10.29
BERT large	-0.246875	0.046875	21.14	10.06
RoBERTa base	-0.015625	0.384375	22.76	10.14
RoBERTa large	-0.115625	0.984375	22.66	10.39
GPT-2	-0.090625	0.109375	22.61	10.1
GPT-2 medium	-0.246875	-0.046875	22.49	10.17
TinyLlama	-0.03125	0.36875	22.77	10.09
Falcon 7B	-0.015625	0.446875	22.78	10.08
Mistral	-0.253125	0.046875	21.3	10.01
MPT 7B	-0.046875	0.384375	22.75	10.07
Llama2 7B	-0.0625	0.3375	22.71	10.1
Llama2 13B	-0.015625	0.446875	22.77	10.07

Table C.10: Whisper on GigaSpeech dev, with patience $p = 3$. Without normalization. According to the hypotheses, the best WER is 15.27 %, the worst WER is 80.06 %, the best CER is 6.36 % and the worst CER is 68.39 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	24.65	12.14
BERT base	-0.16875	0.09375	21.1	9.91
BERT base trained	0.015625	0.184375	23.42	10.9
BERT large	-0.115625	0.015625	21.52	10.12
RoBERTa base	-0.115625	0.046875	23.34	10.7
RoBERTa large	-0.115625	0.109375	22.63	10.42
GPT-2	-0.115625	-0.015625	22.85	10.14
GPT-2 medium	-0.13125	-0.09375	22.77	10.15
Falcon 7B	-0.06875	0.03125	23.24	10.34
TinyLlama	-0.084375	0.015625	23.23	10.32
Mistral	-0.153125	0.078125	21.91	9.99
MPT 7B	-0.1	0.0	23.13	10.27
Llama2 7B	-0.13125	-0.03125	23.12	10.29
Llama2 13B	-0.053125	0.046875	23.25	10.43

Table C.11: Whisper on GigaSpeech dev, without patience $p = 1$. Input to the LLM is normalized. According to the hypotheses, the best WER is 9.11 %, the worst WER is 24.70 %, the best CER is 5.60 % and the worst CER is 18.22 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.62	10.01
BERT base	-0.015625	0.384375	12.95	8.16
BERT base trained	-0.015625	0.384375	12.95	8.16
BERT large	-0.015625	0.384375	12.98	8.19
RoBERTa base	-0.015625	0.384375	13.0	8.2
RoBERTa large	-0.015625	0.384375	12.97	8.18
GPT-2	-0.084375	0.046875	12.97	8.15
GPT-2 medium	-0.03125	0.36875	12.92	8.1
TinyLlama	-0.015625	0.353125	12.96	8.19
Falcon 7B	-0.03125	0.36875	12.97	8.14
Mistral	-0.015625	0.353125	12.97	8.2
MPT 7B	-0.015625	0.353125	12.95	8.18
Llama2 7B	-0.03125	0.36875	12.95	8.13
Llama2 13B	-0.015625	0.353125	12.97	8.19

Table C.12: Whisper on GigaSpeech dev, with patience $p = 3$. Input to the LLM normalized. According to the hypotheses, the best WER is 8.11 %, the worst WER is 66.99 %, the best CER is 4.81 % and the worst CER is 61.78 %.

LLM	β	γ	best WER[%] ↓	best CER[%] ↓
baseline	–	–	14.52	9.96
BERT base	-0.1	0.0	13.05	8.17
BERT base trained	-0.115625	0.015625	13.09	8.16
BERT large	-0.046875	0.153125	13.08	8.26
RoBERTa base	-0.0625	0.1375	13.09	8.21
RoBERTa large	-0.015625	0.184375	13.18	8.42
GPT-2	-0.084375	0.046875	12.76	7.93
GPT-2 medium	-0.06875	0.03125	12.81	7.98
TinyLlama	-0.06875	0.03125	12.94	8.09
Falcon 7B	-0.084375	0.015625	13.0	8.04
Mistral	-0.084375	0.015625	13.02	8.09
MPT 7B	-0.084375	0.015625	12.96	8.05
Llama2 7B	-0.084375	0.046875	12.98	8.05
Llama2 13B	-0.053125	0.046875	13.03	8.2

Appendix D

Difference in insertions, deletions and substitutions

ASR hypotheses order changes after rescoring with LLM, which changes the final speech transcriptions from the ASR pipeline. To find out how the transcriptions change, I measured the difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against reference transcriptions.

Table D.1: Experiment with Jasper and LibriSpeech dev-other dataset. Difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against dataset transcriptions.

LLM	insertions	deletions	substitutions
BERT base	149	-103	39
BERT base trained	140	-92	42
BERT large	91	-17	-18
RoBERTa base	99	-87	64
RoBERTa large	145	-91	73
GPT-2	153	-217	172
GPT-2 libri	129	-195	177
GPT-2 libri2	129	-195	177
GPT-2 medium	153	-217	172
TinyLlama	193	-184	135
Falcon 7B	218	-143	136
Mistral	153	-30	11
MPT 7B	194	-137	108
Llama2 7B	244	-170	130
Llama2 13B	222	-129	121
Llama2 LoRA r8	235	-82	91
Llama2 LoRA r32	246	-101	106
Llama2 LoRA r128	234	-78	86

Table D.2: Experiment with Jasper and TED-LIUM dev dataset. Difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against dataset transcriptions.

LLM	insertions	deletions	substitutions
BERT base	131	-247	32
BERT base trained	136	-210	73
BERT large	104	-86	70
RoBERTa base	86	-66	85
RoBERTa large	80	-6	92
GPT-2	109	-62	115
GPT-2 libri	38	45	60
GPT-2 libri2	38	45	60
GPT-2 medium	109	-62	115
TinyLlama	145	-77	136
Falcon 7B	129	-27	140
Mistral	125	-43	133
MPT 7B	157	-105	152
Llama2 7B	109	-16	136
Llama2 13B	134	-12	151
Llama2 LoRA r8	150	-26	157
Llama2 LoRA r32	148	-48	149
Llama2 LoRA r128	158	-57	163

Table D.3: Experiment with Jasper and GigaSpeech dev dataset. Difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against dataset transcriptions.

LLM	insertions	deletions	substitutions
BERT base	1025	-1368	760
BERT base trained	1025	-1368	760
BERT large	996	-1361	696
RoBERTa base	1171	-1519	1037
RoBERTa large	889	-1086	992
GPT-2	1048	-1874	808
GPT-2 libri	1048	-1874	808
GPT-2 libri2	1048	-1874	808
GPT-2 medium	1261	-1951	1227
TinyLlama	1294	-1898	1428
Falcon 7B	1312	-1569	1540
Mistral	1118	-1094	1238
MPT 7B	1197	-1615	1364
Llama2 7B	1322	-1622	1491
Llama2 13B	1523	-2022	1658
Llama2 LoRA r8	1138	-1157	1510
Llama2 LoRA r32	1260	-1558	1703
Llama2 LoRA r128	1158	-1251	1586

Table D.4: Experiment with Jasper and TED-LIUM test dataset. Difference in number of insertions, deletions and substitutions between non-rescored and rescored transcriptions against dataset transcriptions.

LLM	insertions	deletions	substitutions
BERT base	244	-480	67
BERT base trained	245	-352	83
BERT large	174	-159	100
RoBERTa base	169	-90	108
RoBERTa large	102	-46	129
GPT-2	200	-200	157
GPT-2 libri	48	68	111
GPT-2 libri2	48	68	111
GPT-2 medium	200	-200	157
TinyLlama	241	-240	170
Falcon 7B	234	-154	200
Mistral	192	-143	177
MPT 7B	300	-327	199
Llama2 7B	194	-76	210
Llama2 13B	234	-120	189
Llama2 LoRA r8	236	-85	183
Llama2 LoRA r32	264	-126	191
Llama2 LoRA r128	268	-150	200