BEYOND AUDIO QUALITY: UNDERSTANDING AND IMPROVING VOICE COMMUNICATION WITH LOW-RESOURCE DEEP LEARNING

By

Quchen Fu

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May 12th, 2023

Nashville, Tennessee

Approved:

Jules White, Ph.D.

Douglas C. Schmidt, Ph.D.

Aniruddha Gokhale, Ph.D.

Peng (Dana) Zhang, Ph.D.

Maria Powell, Ph.D.

ACKNOWLEDGMENTS

I would like to extend my sincere gratitude to my advisor, Dr. Jules White, for his dedicated support and guidance throughout my four-year PhD journey. His expertise and encouragement were instrumental in helping me to identify and pursue a research topic that aligns with my passions and interests. I am also grateful to my labmates for fostering a supportive and caring team culture. The experiences shared and memories created will be cherished forever.

I extend my appreciation to my dissertation committee members, Dr. Douglas C. Schmidt, Dr. Aniruddha Gokhale, Dr. Peng (Dana) Zhang, and Dr. Maria Powell, for their valuable feedback and insights during my defense. Their guidance and support were crucial to the success of my research.

I am honored to have had the opportunity to collaborate and co-author papers with some of the world's most reputable companies, IBM Cooperation, Intel Cooperation, and Microsoft Cooperation, and am excited for the knowledge and experiences gained through these partnerships.

Lastly, I express my deepest thanks to my family and friends for their unwavering support throughout my PhD journey. Their encouragement and belief in me were invaluable in helping me to grow and achieve success.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . . **v**

LIST OF FIGURES . . . . **vi**

1 Introduction **1**

1.1 Problem Overview 1
1.2 Challenge 1: Integrate traditional method with deep learning for faster training, a case study of Spoof Speech Detection for communication security 2
1.3 Challenge 2: Real-time Interruption Detection on mobile device with light-weight DL model inference for communication inclusiveness 3
1.4 Challenge 3: How to generate high quality training data for more intuitive communication, a case study with NL2Command 4
1.5 Challenge 4: A methodology guide for efficient DL model training on CPU 6

2 Faster training algorithm: Learnable Audio Front-End for Spoof Speech Detection **8**

2.1 Problem Overview 8
2.2 Related Work and Challenges 8
2.3 Experiment and Dataset 9
2.3.1 Dataset 9
2.3.2 Metrics 10
2.3.3 Back-end 10
2.3.4 Experimental Setup 11
2.4 Results and Analysis 11
2.4.1 Learnable frontend performance 11
2.4.2 Frontend design 12
2.4.3 Optimal filterbank constraints 12
2.4.4 Learned feature interpretation 13
2.4.5 FastAudio Usage 14
2.5 Conclusion 14

3 Light-weight model inference: Real-time Speech Interruption Analysis **16**

3.1 Problem Overview 16
3.2 Model and Method 17
3.2.1 Baseline Model 17
3.2.2 Model Structure 17
3.2.3 Structural Exploration 18
3.2.4 Quantization 19
3.3 Experiments 19
3.3.1 Data statistics 19
3.3.2 Result and findings 20
3.3.3 Memory and Energy Analysis 21
3.3.4 Error Analysis 21
3.4 Conclusion 22

4 High quality data generation: Translating Natural Language to Bash Commands **23**

4.1 Problem Overview . . . 23
4.2 Background and Related Work . . . 24
4.3 Key Research Challenges . . . 25
4.3.1 Ambiguity . . . 26
4.3.2 Many-to-many mapping . . . 26
4.3.3 Scarce Paired data . . . 26
4.3.4 Environment dependent . . . 27
4.4 Research Questions . . . 27
4.4.1 Summary of the Highest Performing Architecture . . . 31
4.4.2 Parsing and Tokenization . . . 32
4.4.3 Model Details . . . 33
4.4.4 Post-processing . . . 34
4.5 Corpus Construction . . . 35
4.5.1 An Updated Pipeline . . . 35
4.5.2 Bash Command Synthesis . . . 36
4.5.3 Bash Command Validation . . . 39
4.5.4 English Text Synthesis . . . 40
4.5.5 Corpus Description and Statistics . . . 40
4.5.6 Data Quality . . . 41
4.5.7 Comparison to Existing Dataset . . . 42
4.6 Metrics and Error Analysis . . . 44
4.6.1 Metrics . . . 44
4.6.2 Synthesized Dataset Results . . . 45
4.6.3 Error Analysis . . . 46
4.7 Conclusion . . . 47

5 Efficient training: A Methodology for Deep Learning Models on CPUs . . . 49

5.1 Problem Overview . . . 49
5.2 Method Summary . . . 50
5.2.1 Profile and Tracing . . . 51
5.2.2 Data Discrepancy . . . 53
5.2.3 Projection and Toolkit structure . . . 53
5.2.4 Dataloader and Memory Layout . . . 55
5.2.5 Library Optimization . . . 55
5.2.6 Low-precision Training . . . 56
5.2.7 Layer Fusion and Optimizer Fusion . . . 57
5.2.8 Custom Operation Kernel . . . 57
5.2.8.1 Theoretical deduction . . . 57
5.2.8.2 Implementation and Assessment . . . 58
5.3 Distributed Training . . . 59
5.3.1 Distributed Training Performance . . . 59
5.3.2 Training Convergence . . . 60
5.4 Conclusion . . . 60

6 Appendix . . . 63

6.1 Summary of Publications . . . 63
6.2 Reference Focal Loss Code [1] . . . 64
6.3 Focal Loss Derivative . . . 64
6.4 Custom Focal Loss Kernel Code . . . 64

References . . . 66

LIST OF TABLES

Table Page

2.1 Filter Comparison of Learnable Front-end . . . 9
2.2 Description of ASVspoof 2019 dataset (LA) . . . 10
2.3 X-vector and ECAPA-TDNN . . . 10
2.4 A stage-wise comparison of the different Front-ends' performance on the ASVspoof 2019 LA dataset . . . 12

3.1 Complexity and RTF drop by various parameters . . . 18
3.2 Number of Clips and Labels . . . 19

4.1 Model Performance Comparison . . . 28
4.2 The NLC2CMD Leaderboard . . . 28
4.3 Parameter Replacement . . . 30
4.4 Command Generation Process . . . 38
4.5 New vs. Existing Dataset . . . 42
4.6 Comparison of Model Performance Across Datasets . . . 46

5.1 Comparison of DL Profiling Tools . . . 51
5.2 Summary of benchDNN Parameters . . . 55
5.3 Scalability (Normalized Throughput) . . . 59

LIST OF FIGURES

Figure		Page
1.1	Paper domain distribution	2
2.1	Heatmap of the magnitude of the frequency response for initialization filters (up) and learned filters (down).	11
2.2	Visualization of Learnable Front-ends	12
2.3	Cumulative frequency response of the FastAudio filters	13
3.1	Failed speech interruption model (WavLM-SI) architecture and training pipeline	17
3.2	Number of layers vs model size	18
3.3	Layer Order vs True Positive Rate (TPR)	20
3.4	True Positive Rate (TPR) is higher for thinner models	20
3.5	Memory usage of the model with different ONNX parameter settings	21
3.6	T-SNE	22
4.1	Example of a Bash Command	30
4.2	Pipeline of the NLC2CMD Workflow	32
4.3	Visualization of the Tokenization Process	33
4.4	Model Structure	34
4.5	Updated Pipeline of the Dataset Generation and Translation	36
4.6	Valid Command Rates For Generated Commands	39
4.7	Utility Distribution in the Generated Dataset	42
4.8	Utility Distribution in the Original NL2Bash Dataset	43
4.9	Percentage of Utility and Flag Errors on Original Dataset	46
4.10	(a) Distribution of Reference Utilities that are Wrongly Predicted. (b) Distribution of Wrongly Predicted Utilities	47
5.1	DL Workflow Method Decomposition	51
5.2	Comparison of Primitive Operations Across Models	52
5.3	The vTune Design	53
5.4	Open Image vs COCO Training Time Ratio Breakdown	54
5.5	Toolkit Structure and Flow Pipeline	54
5.6	Comparison of Custom Focal Loss Time vs Default	58
5.7	Convergence Ratio vs Global Batch Size (Normalized)	61
6.1	Backward Kernel Equation	64
6.2	Simplified Backward Kernel	65

CHAPTER 1

Introduction

1.1 Problem Overview

Over the past few decades, Artificial Intelligence (AI) has advanced significantly and has become a prevalent technology that impacts various aspects of our daily lives. Deep learning (DL) models, which serve as the cornerstone of AI, have found numerous applications in areas such as computer vision, natural language processing, and speech. One key aspect of these application is voice quality service. Despite the various means of communication available today, such as phone calls, emails, and social media posts, voice communication remains an important and intuitive means of exchanging information. While research has focused on improving the physical properties of voice service like noise suppression [2] (more clarity), de-reverberation [3] (echo cancellation), and packet loss concealment [4] (stalling prevention), there has been less attention on the content and intention of conversations. Previous research [5] has summarized the main tasks for speech communities into four aspects: content, speaker, semantics, and paralinguistics. Content tasks include Automatic Speech Recognition (ASR), Keyword Spotting (KS), and Phoneme Recognition (PR) which aims to answer what one says and ignores other properties of speech. Speaker tasks include Speaker Identification (SID), Automatic Speaker Verification (ASV), and Speaker Diarization (SD) which focus solely on answering who is speaking and when. Semantics tasks include Slot Filling (SF) and Intent Classification (IC) and tries to answer why one says this and if is it correct. Paralinguistics tasks are all about how one says it, e.g., Emotion Recognition (ER). In this paper, we choose three areas out of the four and aligned them with three foundational elements of AI: Data, Algorithms, and Computing Power. We found that DL models excel in each of the tasks (achieved state-of-the-art) and further discussed training on low-resource hardwares.

In this dissertation, we will investigate the use of low-resource deep learning models to improve the quality of voice communication with data generation, efficient algorithm, training and inference hardware utilization. We propose the creation of Voice Analysis as a Service (VAaaS), which offers spoof detection, interruption detection, voice-to-command generation, low-resource training to enhance the quality of voice communication. Specifically, we will investigate the following four challenges: 1. Protect conversation participants from spoofing attacks to enhance security. 2. Classify speech overlaps to promote more inclusive and engaging conversations for online meetings. 3. Use English speech for asking machines to perform complex tasks intuitively. 4. Use CPU for training Deep Learning models.

Our research question is: **How do we use low-resource deep learning to create voice analysis services**

that improve the quality of conversations? Through this research, we aim to provide a comprehensive understanding of how to use low-resource deep learning to enhance the quality of voice communication and facilitate more effective interactions between humans and machines.

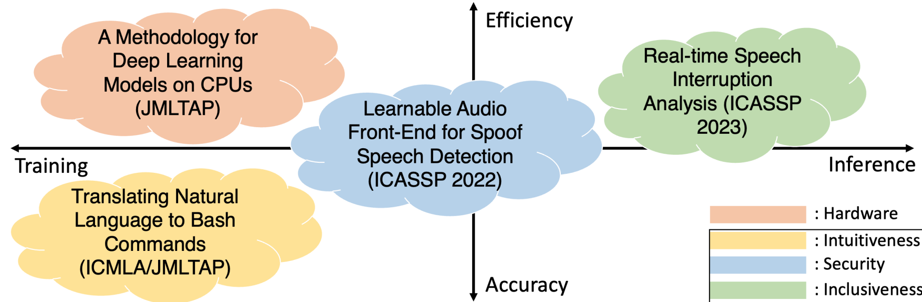


Figure 1.1: Paper domain distribution

1.2 Challenge 1: Integrate traditional method with deep learning for faster training, a case study of Spoof Speech Detection for communication security

The increasing popularity of smart appliances has streamlined our daily chores but also brought security concerns. It is currently estimated that over 35% of the US adult population owns a smart speaker at home [6]. These voice assistants are becoming increasingly versatile and can automate various tasks, ranging from making a phone call to placing an order. However, many of these tasks require different levels of privileges that are tied to the identity of the person interacting with the voice assistant. Thus, identifying ”who is speaking” has become a crucial aspect of personalized voice services. In comparison to Speaker Identification, which focuses on personalization, Speaker Verification is a binary classification of the validity of user identity claims for biometric security.

Spoofing attacks against speaker verification systems can be broadly categorized into three groups: Text-To-Speech (TTS), Voice Conversion (VC), and Replay Attacks. Among these, replay attacks, which involve the playback of a recorded sample of the victim's speech to fool the identification system, are the most prevalent as they require the least technological sophistication. However, this type of attack can be mitigated by incorporating random prompt words into the system. With the rapid advancement of deep learning techniques, TTS and VC models have seen significant improvement in their ability to deceive speaker verification systems. For instance, models such as Tacotron2 [7] are capable of generating high-quality synthetic speech from text that is almost indistinguishable from human speech.

Audio files are typically stored as 1D vectors that are extremely large in size (1 second of audio recording with a sampling rate of 16kHz contains 16000 data points). Due to their length, they are traditionally pre-processed to create a compressed representation that is smaller in size, while attempting to preserve as many

important features as possible, prior to the application of spoof detection. The component responsible for this preprocessing step is known as the front-end. Front-ends can be either handcrafted or learnable, and the process of selecting appropriate handcrafted front-ends is commonly referred to as feature selection. Both types of front-ends include filter layers and constraints can be applied to the filters.

Although handcrafted front-ends have been shown to be a strong baseline for a variety of tasks, the underlying concept behind the design of these features is based on the non-linearity of the human ear's sensitivity to frequency (Mel scale) and loudness (Log compression). Therefore, they may not always represent the most salient features for audio classification across all domains. Empirical studies have shown that learnable front-ends generally outperform handcrafted front-ends in 7 out of 8 audio classification tasks [8]. However, CNN-based front-ends often have large kernels that require extensive training and inference time. This chapter discussed the possibility of integrating traditional methods with standard deep learning methods, specifically proposing an algorithm that considers model size, inference time, and accuracy. This is particularly relevant in the context of developing a front-end for audio, as they are often part of the model running on smart speakers which have limited resources and require real-time response.

1.3 Challenge 2: Real-time Interruption Detection on mobile device with light-weight DL model inference for communication inclusiveness

Meetings are an essential form of communication for all types of organizations, and remote collaboration systems have become extremely common since the COVID-19 pandemic. One of the major challenges faced during remote meetings is the difficulty for remote participants to interrupt and speak [9]. In fact, the inability of virtual meeting participants to effectively interrupt and speak has been identified as the primary obstacle to achieving more inclusive online meetings [9]. Remote participants may often try to join a discussion, but are unable to do so as other speakers are talking. We refer to these attempts as *failed interruptions*.

Too many failed interruptions may alienate participants and lead to a less effective and inclusive meeting environment, which can further impact the overall working environment and employee retention at organizations [10]. Our previous research [10] has explored the feasibility of mitigating this issue by creating a failed interruption detection model that prompts the failed interrupter to raise their virtual hands and gain attention, a feature that is beneficial for enhancing meeting inclusiveness but rarely used.

The WavLM-based Speech Interruption Detection model (WavLM_SI) [10] was deployed in the Azure cloud and has proven to be a useful feature. Through client integration, the reach of this model can be expanded to a broader range of customers without incurring additional costs associated with cloud deployment, as well as reducing the environmental impact by eliminating the need for a dedicated service. However, it should be noted that the original model is based on a pre-trained speech model, which is computationally

intensive and therefore not suitable for deployment on client devices. To be deployed on clients, the model must be optimized to meet the constraints of memory, computation, and energy efficiency. A demonstration video of the WavLM_SI model is available here.

We developed the first speech analysis model that detects failed speech interruptions, which exhibits very promising performance and is being deployed in the cloud [10]. While it is economically desirable to run this model on client devices, its large size and computational complexity present a challenge for deployment on such devices. In the field of speech analysis, there is a prevalent emphasis on achieving high levels of accuracy, without sufficient attention given to trade-offs such as model size, inference time, hosting costs and environmental impact. Therefore, there is a pressing need for solutions that enable the deployment of speech analysis models on client devices, in order to ensure privacy and low latency.

In this chapter, we present a methodology for improving the performance of a failed speech interruption detection model. Specifically, we describe how we were able to enhance the True Positive Rate (TPR) from 50.93% to 68.63% by training on a larger dataset and fine-tuning the model. Furthermore, we demonstrate how the model size was reduced from 232.4 MB to 9.8 MB with only a slight decrease in accuracy, and the computational complexity was reduced from 31.2 GMACs to 4.3 GMACs. Additionally, we provide an estimation of the environmental impact of moving the model from the cloud to client devices, which can serve as a general guideline for large machine learning model deployment and make these models more accessible with a reduced environmental footprint.

1.4 Challenge 3: How to generate high quality training data for more intuitive communication, a case study with NL2Command

Automating the conversion of natural language to executable computer programs is a long-coveted goal that has recently experienced a resurgence of interest amongst researchers and practitioners. In particular, converting natural language to Bash (which is a shell scripting language for UNIX systems) has emerged as an area of interest, with the goal of automating repetitive tasks, such as file manipulation, search, and application-specific scripting. In the near term, natural language to Bash Commands translation is unlikely to replace discussion groups or help forums completely. They can, however, provide a quick reference mechanism that may improve on-demand code suggestions and popups generated by integrated development environments (IDEs). This type of AI-based approach complements other prior work, such as SOFix [11], which can fix bugs in code by mining postings in Stack Overflow.

The NL2Bash problem can be described as a semantic parsing challenge, i.e., creating a mapping from natural language to a formal, executable representation [12]. The NLC2CMD competition, held at the NeurIPS 2020 conference, has been a driving force in the advancement of efforts to address this problem.

Our recent participation in this competition resulted in the creation of an architecture that significantly improved the state-of-the-art performance in the translation of natural language to Bash Commands, increasing the accuracy from 13.8% to 53.2% [13]. Our transformer model, developed for the NLC2CMD competition, has been acknowledged as the current leading architecture for this problem [14].

The task of natural language to Bash command translation, commonly referred to as the NL2Bash problem, has traditionally been reliant on the availability of a specific dataset, namely the NL2Bash corpus [15]. This corpus, which comprises of over 9000 English-command pairs, was created through the manual scraping of frequently used Bash Commands from various sources such as forums, tutorials, tech blogs, and course materials. The construction of the NL2Bash corpus was achieved through the hiring of freelance software engineers, who were tasked with manually searching, browsing, and entering data through a web interface. These freelancers were able to construct approximately 50 English-command pairs per hour, prior to the filtering and cleaning of the dataset [15]. This manual approach, although effective in generating a dataset, is resource-intensive, as it requires specialized labor from freelancers, which can be time-consuming and costly. Additionally, this approach is not scalable as the marginal cost of labor does not significantly decrease as the size of the dataset increases.

Data scarcity is a common issue in the field of machine translation, as it often necessitates a large volume of parallel data, which can be difficult to obtain. While there are typically larger datasets available for single language corpus, to the best of our knowledge, we did not find a Bash dataset that was sufficient for our task. In light of this, we devised a pipeline for the synthesis of a Bash dataset.

The utilization of dataset synthesis and augmentation has been widely explored in various translation tasks. Nguyen *et al.* [16] explored the use of combining augmented data with the original dataset to boost the accuracy of neural machine translation between human languages. Zhao *et al.* [17] also explored data augmentation in neural machine translation to improve dataset diversification. Notably, Agarwal *et al.* [18] proposed using document similarity methods to create noisy parallel datasets of code, enabling the advancement of machine translation with monolingual datasets.

In our work on Bash command generation, we adopted a two-step approach: (1) Scraping syntax and flag structures from the Bash manual pages for efficient command generation, and (2) Training a back-translation model for accurate command summarization. This methodology allowed us to construct a dataset of English-command pairs that is over six times larger than the original NL2Bash dataset.

Bash manual pages provide an overview of Bash features and are considered as the authoritative reference on shell behavior [19], offering comprehensive and accurate guidance for Bash usage. Recent research has examined the utilization of manual page data for assistance in Bash to natural language translation [20] by processing the page descriptions to aid the translation model. However, in our work, we discovered that the

manual pages also offered additional insight and sufficient context into utility-flag relationships to generate a new dataset from scratch.

1.5 Challenge 4: A methodology guide for efficient DL model training on CPU

Deep learning (DL) models have been extensively utilized in a variety of domains such as computer vision, natural language processing, and speech-related tasks, as reported in literature [21; 22; 23; 24]. The DL models are implemented using popular frameworks such as PyTorch [25], TensorFlow [26], and OpenVINO [27]. The DL models are executed on a wide range of hardware platforms, including general-purpose processors such as CPUs and GPUs, as well as customizable processors such as FPGA and ASICs, which are often referred to as XPUs [28]. The diversity of hardware poses a challenge for proposing a universal methodology for efficient training of DL models. While GPUs have been the dominant hardware for deep learning tasks, comparatively little research has been conducted on optimizing DL models for execution on CPUs, particularly for training [29]. Previous research on DL models on CPUs have primarily focused on performance comparisons between CPUs and GPUs [30; 31; 32; 33] or solely on CPU inference [34].

For organizations with limited resources or existing CPU servers, the training of deep learning models can be a time-consuming and computationally intensive task. One important question to address when optimizing training performance on CPUs is determining the appropriate metrics to guide the optimization process. Several metrics and benchmarks have been proposed in literature to evaluate the performance of deep learning workloads and training. For instance, Multiply-Accumulate (MAC) operations have been utilized as a proxy for FLOPs to measure the computational complexity of Convolutional Neural Network (CNN) models [35]. Time-to-Train (TTT) is a widely adopted metric for measuring the training performance of deep learning models, which is measured by the time taken for a model to reach a certain level of accuracy. NetScore [36] is a universal metric proposed for deep learning models, which balances the trade-off between information density and accuracy.

Until recently, a widely accepted benchmark for deep learning models that encompassed a broad range of domain tasks, frameworks, and hardware had yet to be established. The MLPerf benchmarking suite [37] was proposed as a comprehensive benchmark to cover a variety of tasks and hardware. This effort has been supported by many major technology companies who have participated in the MLPerf challenge to improve the training performance of deep learning models across various domains. Intel, in particular, has been actively participating in the MLPerf challenge to improve the training performance of deep learning models across multiple domains.

As the pursuit of model performance leads to the training of larger and more complex models using larger datasets, the cost in terms of hardware requirements also increases. While deep learning models are

becoming more powerful, they also require more resources. There is a growing concern about the potential for AI monopolies by a few large companies (Nvidia has 80% share of AI processors in 2020), as they have access to exclusive large datasets, the expertise to create cutting-edge algorithms, and the resources to utilize high-end computing hardware at scale [38]. Additionally, the trend towards multi-modality models [39] and foundational models [40], which enable a single model to perform a wide range of tasks across domains and are trained on vast datasets, is exacerbating the gap between large companies and the general public. Therefore, it is crucial to adapt deep learning to low-resource settings, in order to promote more research and advancements in conversation improvement.

CHAPTER 2

Faster training algorithm: Learnable Audio Front-End for Spoof Speech Detection

2.1 Problem Overview

Spoof speech can be used to attempt to deceive speaker verification systems that determine the identity of the speaker based on voice characteristics. This chapter compares popular learnable front-ends on this task. We categorize the front-ends by defining two generic architectures and then analyzing the filtering stages of both types in terms of learning constraints. We propose replacing fixed filterbanks with a learnable one that can better adapt to anti-spoofing tasks. The proposed FastAudio front-end is then tested with two popular back-ends to measure its performance on the LA track of the ASVspoof 2019 dataset. The FastAudio front-end demonstrated a relative improvement of 27% when compared with fixed front-ends, outperforming all other learnable front-ends on this task. This chapter provides the following contributions to the study of defending against audio spoofing attacks:

1. It proposes a light-weight[1] learnable front-end called FastAudio that achieved the lowest min t-DCF in spoof speech detection compared to other front-ends,

2. It provides a comparison of feature selections for spoofing countermeasures, with a special focus on learnable audio front-ends, and demonstrates how incorporating shape constraints in the filterbank layer improves performance while reducing the number of parameters, and

3. It describes the architecture that achieved top performance on the ASVspoof 2019 [41] dataset.

2.2 Related Work and Challenges

Previous research has explored the feasibility of learnable filterbanks. For example, the work presented in nnAudio [42] implemented a set of unconstrained learnable filterbanks; however, T. Sainath *et al.* [43] reported limited improvement from unconstrained filterbank learning. DNN-FBCC [44] explored some constraints over filters by adopting a mask matrix. Zhang and Wu [45] described a detailed study on the shape and positiveness constraint's effect on the filterbanks. Despite these efforts, to date, no comprehensive study has been conducted on the impact of constraining filterbank shape in the context of STFT-based spoof speech detection.

As shown in Table 2.1, all current FST based front-ends put shape constraints on the band-pass filters; however, STFT based front-ends, like DNN-FBCC, do not constrain the filter shape. Instead, a mask is put

[1]Front-end trains faster and has the least computational complexity as estimated by multiply–accumulate operations (MACs) compared to other learnable front-ends. See Table 2.4, https://pypi.org/project/ptflops/.

Table 2.1: Filter Comparison of Learnable Front-end

Type	Name	Filter/BandWidth		Center Frequency		Gain
		Shape	Clamp	Sorted	Clamp	
FST based	TD-FBanks	Gabor	-	Yes	Yes	-
	SincNet	Sinc	Yes	No	Yes	Fixed
	LEAF	Gabor	Yes	No	Yes	Fixed
STFT based	nnAudio	-	No	No	No	-
	DNN-FBCC	-	Yes	Yes	Yes	-
	FastAudio	Triang	Yes	No	Yes	Fixed

on the filters so that the bandwidth is clamped and the filters are sorted by center frequencies. Therefore, we designed a learnable front-end, named FastAudio, which aims to address the following research questions:

1. Is a shape constraint necessary for spoof detection, if so, what specific shape constraint yields the lowest minimum t-DCF?

2. Is the sorting of center frequencies necessary for spoof detection?

3. What do trained filterbanks learn about spoof detection in comparison to handcrafted FBanks?

2.3 Experiment and Dataset

The ASVspoof 2019 corpus is composed of two distinct components: Logical Access (LA) and Physical Access. In this study, we focus specifically on the LA task. The LA dataset comprises of synthetic speech, generated through various text-to-speech and voice conversion techniques, referred to as *spoof* speech and true speech audio files referred to as *Bona fide* speech. As there are already established Automatic Speech Verification (ASV) systems that provide some level of protection against spoofing attacks, the objective of this research is to design a system, referred to as Countermeasures (CM), that can effectively complement existing ASV systems. The performance of the proposed system is evaluated using the tandem detection cost function (min t-DCF), which is considered to be a metric that best reflects the real-world protection effects.

2.3.1 Dataset

The performance of the FastAudio learnable front-end is evaluated on the ASVspoof 2019 LA dataset. As shown in Table 3.2, the dataset was partitioned into three parts where the evaluation set is three times the size of the training set. The training and development sets contain data generated from the same algorithms; however, to ensure the spoof detection system can generalize well to audio of unseen types, the evaluation set also contains attacks that are generated from different algorithms.

Table 2.2: Description of ASVspoof 2019 dataset (LA)

Subset	**#Speaker**		**#Utterances**	
	Male	**Female**	**Bona fide**	**Spoofed**
Training	8	12	2580	22800
Development	8	12	2548	22296
Evaluation	21	27	7355	63882

2.3.2 Metrics

The primary metric for evaluating the performance of spoof speech detection systems is the minimum normalized tandem detection cost function (min t-DCF), as shown in Equation 2.1. The min t-DCF measures the overall protection rate for combined CM and ASV systems, where β depends on application parameters (priors, costs) and ASV performance (miss, false alarm, and spoof miss rates), while $P^{\text{cm}}_{\text{miss}}(s)$ and $P^{\text{cm}}_{\text{fa}}(s)$ are the CM miss and false alarm rates at threshold s [41]. A secondary metric, Equal Error Rate (EER), is also used to facilitate comparison with earlier datasets such as ASVSpoof 2017. EER is defined as the value at which the False Acceptance Rate and False Rejection Rate are equal.

$$\text{t} - \text{DCF}^{\text{min}}_{\text{norm}} = \min_{s} \{\beta P^{\text{cm}}_{\text{miss}}(s) + P^{\text{cm}}_{\text{fa}}(s)\} \tag{2.1}$$

2.3.3 Back-end

Our FastAudio front-end consists of an STFT transform followed by a learnable filterbank layer, and finally a log compression layer to mimic the non-linearity of human sensitivity to loudness. We integrated the front-ends with two of the most popular back-ends for audio classification: X-vector [46] [47] and ECAPA-TDNN [48] [47]. The back-end converts the filterbank variant into a 256-dimensional embedding vector, which is then fed into a linear classifier.

Table 2.3: X-vector and ECAPA-TDNN

X-vector		**ECAPA-TDNN**	
Layer	**Output**	**Layer**	**Output**
Input	(N, T′)	Input	(N, T′)
TDNN X 5	(1500, T′)	Conv1D + ReLU + BN	(C, T′)
Stats Pool	(3000, 1)	SE-Res2Block X 3	(3, C, T′)
Linear	(256, 1)	Conv1D + ReLU	(1536, T′)
		Atten Stats Pool + BN	(3072, 1)
		FC + BN	(256, 1)

2.3.4 Experimental Setup

The model was trained on 2 Nvidia 2080 Ti GPUs for 100 epochs and the batch size was set to 12 (except for TD-filterbank whose batch size was 4 to stay within memory limits). We also compared the performance of our front-end with other STFT-based and FST-based front-ends, both under learnable and fixed settings. To make the comparison fair, we keep the hyperparameters across all experiments the same so that the front-end outputs have the same dimensions. The sampling rate was set to 16kHz, window length to 25ms, window stride to 10ms, and the number of filters to 40. All learnable front-ends were initialized to mimic Mel-FBanks, as previous research [49] has shown that random initialization has worse performance. Detailed hyperparameters and data augmentation are available on our GitHub repository[2]

2.4 Results and Analysis

2.4.1 Learnable frontend performance

How do learnable front-ends perform on min t-DCF compared with handcrafted front-ends for spoof speech detection? We conducted an updated comparison of front-ends for spoof detection, as the most recent systematic comparison available was from 2015 [50]. Our experiments aimed to establish a new baseline that incorporates learnable front-ends. We selected a combination of FST and STFT front-ends, with both fixed and learnable settings, to provide a comprehensive examination. As demonstrated in Table 2.4, our findings indicate that FST-based learnable front-ends require a longer training time compared to hand-crafted features in the task of spoof speech detection and cannot beat the performance of CQT.

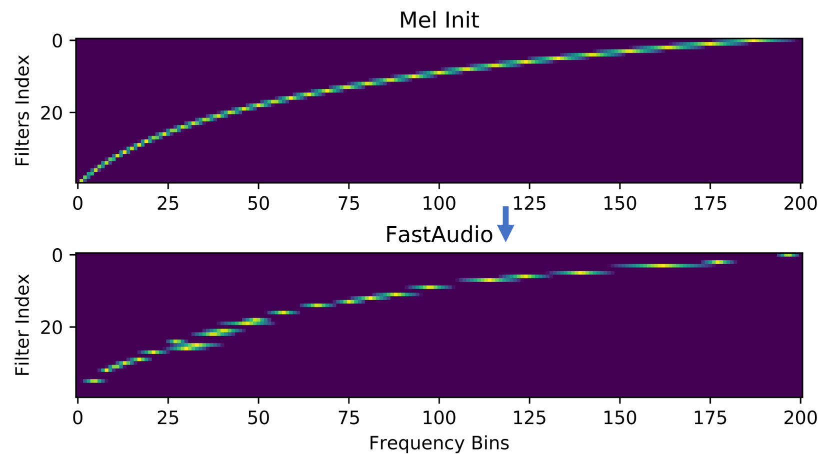


Figure 2.1: Heatmap of the magnitude of the frequency response for initialization filters (up) and learned filters (down).

[2]https://github.com/magnumresearchgroup/Fastaudio

Table 2.4: A stage-wise comparison of the different Front-ends' performance on the ASVspoof 2019 LA dataset

Front-end	#Params	Constraint	ECAPA-TDNN EER	ECAPA-TDNN min t-DCF	X-vector EER	X-vector min t-DCF	MACs	Train Time/Epoch
CQT	0	Fixed	1.73	0.05077	3.40	0.09510	0	10:58 min
Fbanks	0	Fixed	2.11	0.06425	2.39	0.06875	0	10:53 min
FastAudio-Tri	**80**	**Shape+Clamp**	**1.54**	**0.04514**	**1.73**	**0.04909**	**0.00GMac**	**13:02 min**
FastAudio-Gauss	80	Shape+Clamp	1.63	0.04710	1.67	0.05158	0	12:51 min
FastAudio-Sort	80	Shape+Clamp+Order	1.89	0.05204	1.69	0.05235	0	12:59 min
LEAF	282	Shape+Clamp	2.49	0.06445	3.28	0.07319	0.01GMac	34.45 min
nnAudio	8.04k	No	3.63	0.08929	5.56	0.14707	0	13:00 min
TD-filterbanks	31k	Shape+Clamp	1.83	0.05284	3.18	0.08427	1.32GMac	22.48 min
Front-end	**Name**	**Constraint**	**EER**	**min-tDCF**	**Backend**			**Baseline**
SincNet	RawNet2	Fixed	5.13	0.1175	-			-

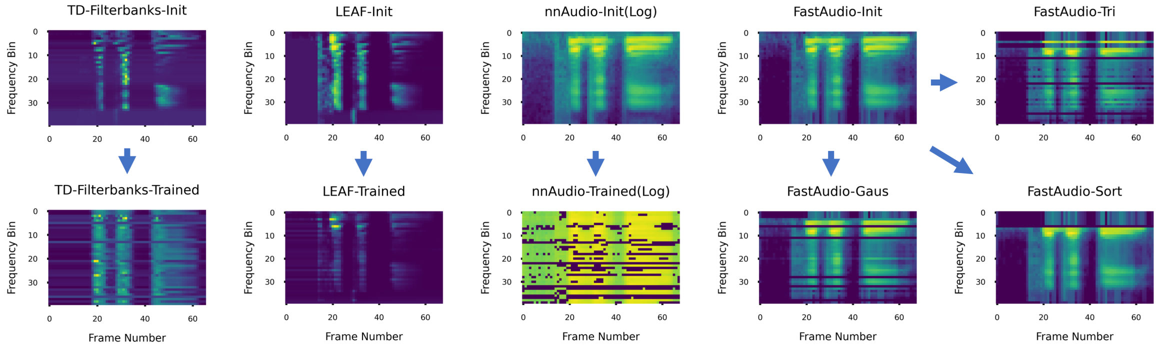


Figure 2.2: Visualization of Learnable Front-ends

2.4.2 Frontend design

Can we design an STFT-based front-end for spoof speech detection that is learnable and can it beat the performance of CQT? As the FST-based learnable front-ends were unable to surpass the performance of CQT, we developed a new front-end approach that adheres to the traditional STFT-based method and limits the number of trainable parameters. This approach, which we refer to as FastAudio, is designed to train faster than FST-based front-ends. We hypothesized that by making the filterbank layer learnable, we could improve the performance of the fixed STFT-based approach without the need for a complete change in the front-end architecture. Our experimentation with FastAudio under three different constraint settings revealed that the best configuration achieved a 27% decrease in minimum t-DCF compared to FBanks, outperforming CQT (see Table 2.4).

2.4.3 Optimal filterbank constraints

Which set of constraints for filterbank learning performs best in spoof speech detection? Our findings indicate that the implementation of shape constraint plays a crucial role in enhancing spoof detection accuracy. However, we did not observe a significant difference in performance when constraining the shape of the

filters to be Gaussian or Triangular. We also found that sorting the filterbanks by center frequency does not improve accuracy, which confirms the conclusion from a previous study in LEAF[8]. As illustrated in Figure 2.1, the learned filterbank distribution closely follows the hand-crafted filterbanks in both center frequency and bandwidth. The similarity in c_n and b_n can help explain the superior performance of handcrafted features compared to the learnable front-end, particularly in comparison to the FST-based front-ends.

The visualization of the front-end output is presented in Figure 2.2. All of the outputs exhibit "horizontal lines" that correspond to specific frequencies, indicating filter selectiveness. Our analysis revealed that the front-end outputs of LEAF, TD-filterbanks, and nnAudio underwent significant changes after training due to the number of trainable parameters. As depicted in nnAudio, when the shape of the filters is not constrained, the trained front-end exhibits signs of overfitting (many randomly distributed dots) and demonstrates the worst performance. Given that nnAudio has no constraint for filter shape, the learned filter shape is determined by 201 points, which can result in very sharp peaks and the selection of frequencies of very narrow ranges, resulting in the irregular dots.

2.4.4 Learned feature interpretation

What does FastAudio learn about spoof speech detection and how can we interpret what it learns? The formants, which are the result of the acoustic resonance of the human vocal tract, have been found to be present in the spectral peaks of speech signals. Due to the fact that English vowels possess a higher energy content than consonants, it is expected that the center frequencies of learned filters will converge around the formants of typical English vowels, as stated in previous research [51]. In our study, we have analyzed the cumulative frequency response of the FastAudio and have observed that there are two peaks in the lower frequency range and one peak in the higher frequency range. These peaks, located around 320Hz to 440Hz and 1120Hz, are believed to correspond to the first and second formants, as reported in [52]. This adaptation to human speech suggests that the FastAudio has been able to effectively learn the characteristics that are crucial for spoof speech detection tasks. Similar adaptation has also been observed in the FST-based front-end for speech identification tasks [52].

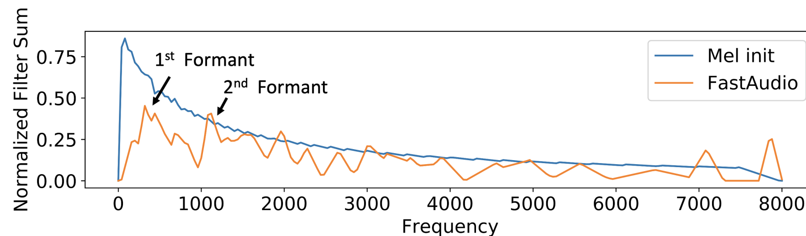


Figure 2.3: Cumulative frequency response of the FastAudio filters

An interesting observation in our study was the presence of peaks in the high pitch regions near the sampling boundary. This suggests that spoof speech may differ from real speech in frequencies that are often ignored by handcrafted front-ends such as the Mel-scale. Historically, high-frequency energy has been considered to be of less importance and has been underrepresented in Mel-scales. However, in the context of spoof speech detection, we suspect that because these high frequencies are not crucial to human hearing, the spoof speech generator may not generate realistic imitations in these high frequencies. Thus, the representation of high-frequency data may serve as an effective indicator for the detection of spoof speech.

Together, these findings indicated that:

1. The learned FastAudio filters exhibit greater selectivity compared to their initialization.
2. FastAudio places emphasis on frequencies around the first and second formants, which may be crucial for distinguishing between spoof and authentic speech.
3. The learned FastAudio filters exhibit increased sensitivity to high-frequency energy, which may be a distinctive feature in the detection of spoof speech.
4. Through the application of end-to-end training, FastAudio can adapt to spoof detection tasks. The front-end has successfully adapted to the downstream task and has been able to learn the phonetics of human speech.

2.4.5 FastAudio Usage

How can people use FastAudio for spoof detection and suggestion for model fusion People involved in the design of back-end systems for spoof speech detection can utilize the FastAudio as a substitute for their current front-end. Our experiments demonstrate that FastAudio is superior to the Constant-Q Transform (CQT) in spoof detection, despite previous research reporting CQT as the best front-end [53]. From the information theory's perspective, the fusion of results from models whose front-end outputs are least similar tend to result in improved performance. Therefore, the visualizations of the output of learnable front-ends presented in this chapter can provide guidance for feature selection and can serve as a complementary tool to handcrafted front-ends in spoof detection [50].

2.5 Conclusion

This chapter presents an examination of the performance of learnable front-ends in spoof detection, and introduces a Short-Time Fourier Transform (STFT)-based audio front-end, named FastAudio. The proposed front-end was evaluated under various constraint settings and demonstrated the ability to successfully adapt to spoof detection tasks. The proposed front-end achieved outstanding results on the ASVspoof 2019 dataset,

outperforming its fixed equivalent by 27%, and surpassing the performance of the Constant-Q Transform (CQT), which has been previously reported as the best hand-crafted feature for spoof speech detection.

CHAPTER 3

Light-weight model inference: Real-time Speech Interruption Analysis

3.1 Problem Overview

Improving meeting inclusiveness has significant financial incentive (such as higher employee retention rate) for organizations [9], and speech interruption detection [10] can help address the top issue by empowering participants to interrupt and speak in virtual meetings, promoting greater participation and engagement. By deploying a speech interruption detection model on the client-side, the feature can be made more widely accessible in a cost-effective and environmentally sustainable manner. However, it should be noted that model size and complexity have been identified as key challenges in the implementation of this technology.

Previous research has investigated various techniques to reduce the size of models for improved inference performance, including Structural Change [54; 55], Teacher-Student Knowledge Distillation [56; 57; 58; 59], Parameter Sharing [60; 61], Layer Drop [62], Post Training Quantization (PTQ) [63; 64] and a combination of the above [65; 66]. Recently, Microsoft open sourced a library called DeepSpeed [67] that supports more efficient inference for transformer-based Pytorch models. The ONNX runtime [68] is another popular inference accelerator that supports PTQ and cross-platform deployment, enabling the deployment of these models in a variety of environments.

Inference is estimated to account for 90% of the cost compared to training (which are done only once) [69], the energy associated with it therefore is of major concern, especially if the inference runs as a background service and serves a large user base of over 100 million. To mitigate energy consumption, Henderson [70] proposed a framework for tracking and assessing the environmental impact of deep learning models. Additionally, specialized studies on the energy consumption of deep learning models for natural language processing tasks have also been conducted [71].

This chapter presents two key contributions to the field of self-supervised learning (SSL) model deployment. The first contribution is an examination of the utility of pruning as a technique to decrease the size and complexity of models while maintaining an acceptable level of performance. Specifically, we demonstrate that pruning can be used to achieve a 23X reduction in model size and a 9X reduction in complexity with good performance. Furthermore, we show that this size-complexity trade-off is smooth and can be adapted to a wide range of client devices. The second contribution is an improvement in the true positive rate (TPR) for speech interruption detection tasks, increasing it from 50.9% to 68.6%, which represents the current state-of-the-art performance.

3.2 Model and Method

Recent research has demonstrated the effectiveness of learnable audio frontends in audio tasks, as they are gradually replacing traditional fixed frontends, such as spectrogram, in various applications [23]. Semi-supervised learning (SSL) speech models, including Hubert [72], Wav2Vec [73], and WavLM [74], have achieved state-of-the-art performance on a wide range of tasks [75] and have dominated the leaderboard of the SUPERB benchmark [5]. The SUPERB benchmark is a comprehensive evaluation of the performance of shared models across a wide range of speech processing tasks with minimal architecture changes and labeled data. Among these SSL speech models, WavLM [74] currently holds the top position on the SUPERB benchmark.

3.2.1 Baseline Model

In our previous work [10], we utilized the WavLM model as the embedding layer for the speech interruption detection task and achieved a true positive rate (TPR) of 50.93% at a false positive rate (FPR) of 1%. The model had a size of 1.2 GB and required an inference time of 2.5 seconds on an Intel Xeon E5-2673 CPU (2.40GHz), corresponding to a Real-Time Factor (RTF) of 0.5. Our current goal is to significantly reduce the model size to 10 MB while maintaining a TPR above 40% while also achieving an RTF of 0.1. This will enable us to detect at least two failed interruptions per meeting on average, based on meeting statistics and assuming 40 interruptions per half-hour meeting [10].

3.2.2 Model Structure

The architecture of our failed speech interruption model is illustrated in Figure 3.1. It consists of a WavLM-based embedding module followed by an attention pooling classifier [76]. The WavLM model was pre-trained on a large in-house dataset and acts as a feature extractor. To fine-tune the model for the downstream task, a weight matrix was applied to the output of all layers and the model was trained on our dataset.

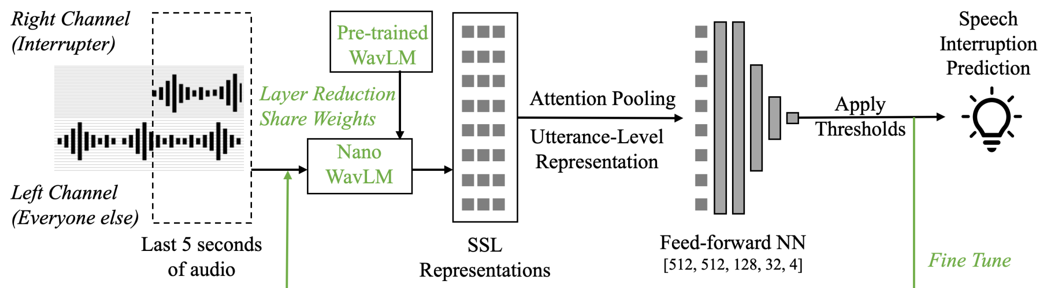


Figure 3.1: Failed speech interruption model (WavLM-SI) architecture and training pipeline

3.2.3 Structural Exploration

The most intuitive and efficient approach for reducing the model size that we identified is structural exploration. To determine the optimal starting model for achieving the goal of 10 MB, we compared different WavLM models with varying widths (CNN channel from 512 to 64, transformer hidden size from 288 to 768) and applied weight sharing to the most promising models, as shown in Table 3.1. We kept the number of layers constant at 12 and found that the complexity and RTF decreased linearly with the number of parameters when weight-sharing was not enabled, as demonstrated in Table 3.1.

Table 3.1: Complexity and RTF drop by various parameters

	Base	**Small**	**Tiny**	**Tiny_SW**	**Nano**	**Nano_SW**	**Pico**
# Conv_ch	512	386	256	128	128	128	64
# Trans_hi	768	576	384	384	288	288	288
Param (M)	95.5	54.04	24.36	8.14	13.58	4.93	12.71
MACs (G)	55.36	31.18	13.9	12.19	7.87	7.55	6.36
RTF	0.51	0.32	0.16	0.12	0.09	0.08	0.07

To further reduce the model size, we applied reductions to both the width and depth of the transformer layers and evaluated the resulting model size. As illustrated in Figure 3.2, when weight sharing and quantization were applied, both the *tiny* and *nano* models were able to reach the target size of around 10 MB when the number of transformer layers was less than four. The weight sharing technique effectively reduces the number of parameters and thus the model size. We implemented weight sharing by allowing every three neighboring transformer layers to share the same weights, which resulted in a step-like pattern in the size versus number of layers as shown in Figure 3.2.

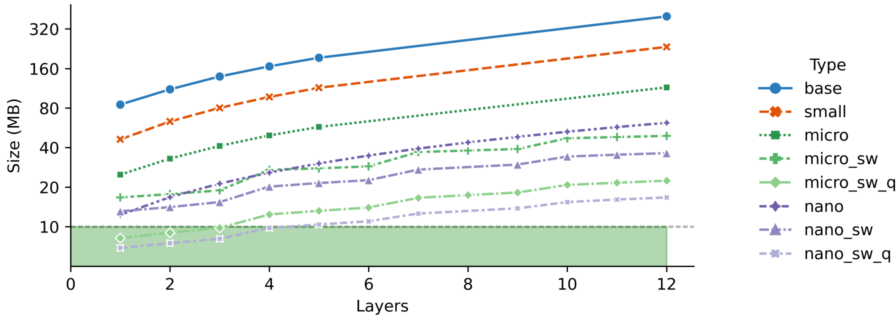


Figure 3.2: Number of layers vs model size

After analyzing the size of each component of our model, we found that the positional convolution layer is 6 MB. Based on this observation, we hypothesized that simply removing this component might be suitable for our specific task for the following two reasons: first, the transformer layers within the model are capable of encoding positional information, and second, the nature of our data, which is clipped in such a way that the start of any interruption is always at the beginning of the 5-second clip, renders positional information

unnecessary. These considerations are further discussed in Section 3.3.2.

3.2.4 Quantization

Quantization has been proven as an effective way to reduce model size. It can be broadly classified into two categories: post-training quantization (PTQ) and quantization-aware training (QAT). The former approach involves mapping the model weights from float32 to a lower-precision format, such as int8, after the training process has been completed. This typically results in a higher degree of accuracy loss. On the other hand, QAT employs pseudo-quantization/dequantization during the training process, which can help mitigate the impact on accuracy. Additionally, the DeepSpeed framework has introduced a novel quantization method called MoQ [77].

3.3 Experiments

3.3.1 Data statistics

Our dataset comprises 40,068 clips, each of which is 5 seconds in duration and contains two channels. The right channel stores the speech of the potential interrupter, while the left channel contains a merger of speech from all other speakers. As shown in Table 3.2, we randomly split the dataset into the train, validate and holdout subset, each consisting of 75%, 19%, 6% of the total size, respectively. The holdout subset contains only speakers that are never seen in other subsets, in order to simulate a real-world evaluation scenario. We plan to make our dataset publicly available as part of a future challenge.

Table 3.2: Number of Clips and Labels

	Backchannel	**Failed Interruption**	**Interruption**	**Laughter**	**Total**
Train	14591	3292	9622	2455	29960
Validate	3693	889	2478	623	7683
Holdout	1378	211	588	248	2425
All	19662	4392	12688	3326	40068

We used crowd-sourcing to label the majority of our dataset, which is divided into four categories: *backchannel* (utterances that do not intend to interrupt, e.g. "uh-huh", "yeah"), *failed interruption*, *successful interruption*, and *laughter* [10]. Each sample in the dataset was labeled by 7 individuals, and a 70% agreement level was adopted, which required a majority of at least 5 individuals to reach consensus. The test set and holdout set were labeled by experts to ensure the highest quality labeling for evaluation purposes. In order to increase the representation of the *failed interruption* class within our data set, we propose incorporating a synthetic data augmentation strategy. Specifically, we suggest combining samples from the *failed interruption* class with randomly selected samples from other channels within the same class. By implementing this approach, we aim to effectively double the overall size of the *failed interruption* data set.

3.3.2 Result and findings

We trained our models on eight NVIDIA V100 GPUs for a total of 70 epochs, with a batch size of 16. To optimize performance, we utilized a polynomial loss function with an epsilon value of 4. Our experiments were conducted multiple times, with an average reported. Additionally, we defined *thinness* as the number of channels for convolution layers and hidden size for transformer layers. Our findings, as illustrated in Figure 3.4, indicate that a model with a smaller width dimension tends to yield better accuracy, given a fixed model size.

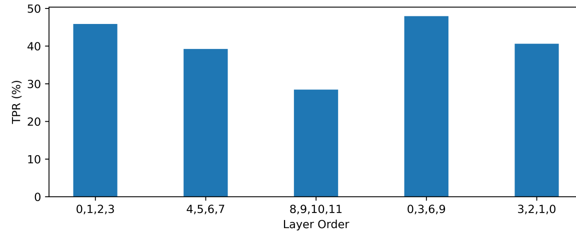


Figure 3.3: Layer Order vs True Positive Rate (TPR)

We found that reducing the number of layers and removing certain components can significantly decrease the size of the model while only incurring a minimal decrease in accuracy. As demonstrated in Figure 3.3, the position of the layers appears to have a greater impact than the order in which they are arranged, which may be attributed to the presence of residual connections within transformer layers. Previous research [54] has suggested that skip layers perform better than other combinations, however, we observed only a slight improvement. This may be due to the fact that we applied fine-tuning after reducing the number of layers. We also found that models that utilize shared weights exhibit similar performance to their non-shared counterparts, while significantly reducing model size. Additionally, quantization proved to be an effective method for reducing model size, with minimal or no impact on accuracy when implemented correctly. We also evaluated the use of teacher-student distillation, apply layer drop during pre-training as potential techniques for improving speech interruption detection task, however, these methods did not yield significant improvements.

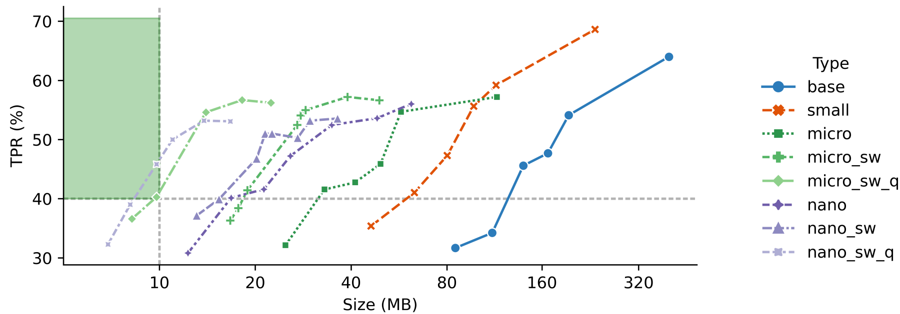


Figure 3.4: True Positive Rate (TPR) is higher for thinner models

3.3.3 Memory and Energy Analysis

In order to simulate the memory usage of our model, we utilized the ONNX runtime on an Intel Xeon E5-2673 CPU (2.40GHz) with 6GB of memory. Our results, illustrated in Figure 3.5, indicate that disabling certain optimization options during runtime, such as *arena_mem*, can significantly reduce the amortized memory usage.

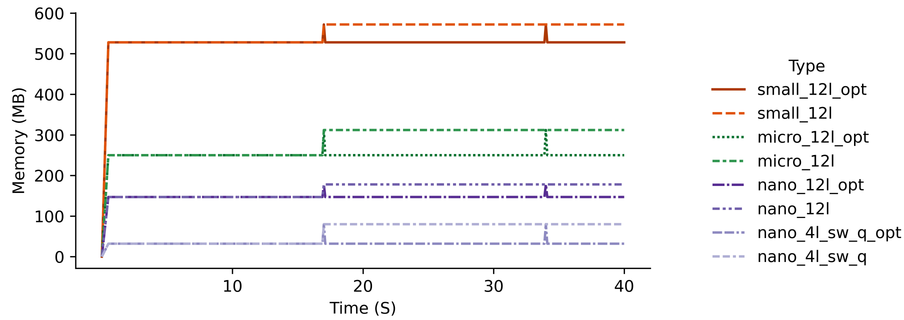


Figure 3.5: Memory usage of the model with different ONNX parameter settings

To analyze energy usage, we utilized inference time as a proxy for CPU usage. We employed a Voice Activity Detector (VAD) [78] to detect 57,450 instances of human speech lasting longer than one second within 272 hours of meeting audio. This resulted in an average inference trigger every 17 seconds. Compared to a scenario in which inference is conducted every 5 seconds, which is less effective in capturing interruption clips, utilizing a VAD to detect potential interruptions results in a 3.4x reduction in inference calls. Since VAD is a widely available service that is already present on the client, the energy consumption for this component can be assumed to be negligible. Furthermore, by reducing the model complexity, we were able to decrease inference time from 1.6 seconds to 200 milliseconds, resulting in a 7.3x reduction in inference time. Overall, this results in a combined energy reduction of 25 times.

3.3.4 Error Analysis

As illustrated in Figure 3.6, we utilized T-SNE to plot the output of the model's pooling layer. Our analysis revealed that the back-channel class is most frequently confused with the failed interruption class, which is consistent with the observations made during the manual labeling process. Additionally, we observed that the successful interruption class is similar to the failed interruption class, but can be relatively easily distinguished. Furthermore, the laughter class is distinct from all other classes and can be easily separated.

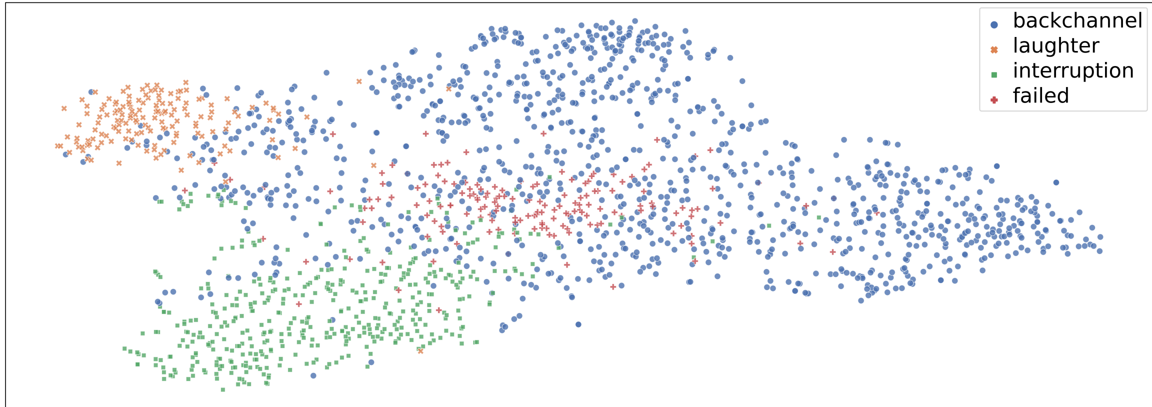


Figure 3.6: T-SNE

3.4 Conclusion

In this chapter, we examined various methods for reducing the size of SSL models. We were able to successfully decrease the in-memory model size of a 234 MB model to 32 MB while maintaining an acceptable level of accuracy. Additionally, we were able to reduce the real-time factor (RTF) from 0.32 to 0.04, which is an important step towards deploying the SSL model to the client. As future work, we plan to enhance the performance of the model by utilizing neural processing units (NPUs) and continue to improve the true positive rate.

CHAPTER 4

High quality data generation: Translating Natural Language to Bash Commands

4.1 Problem Overview

Translating natural language into source code for software or scripts can assist developers in finding ways to accomplish tasks in languages they are not familiar with, similar to how help forums such as Stack Overflow are used today. As early as 1966, Sammet [79] envisioned a future of automated code generation where individuals can program in their native language. While generating software templates from configuration files is now a common practice, research on translating natural language into code is still in a relatively nascent stage. Past research mainly focused on scripting languages or small code snippets, and most efforts have been directed towards developing more accurate translation models. Various datasets have been created to aid research on generating code from natural languages, such as WikiSQL for SQL [80], CoNaLa for Python [81], and NL2Bash for Bash [82]. Here we focus on the task of translating natural language into Commands in the Bash scripting language. Translating natural language into Bash Commands is an example of semantic parsing, which means natural language is translated into logical forms that can be executed [83]. For example, the phrase "how do I compress a directory into a bz2 file" can be translated to the Bash command: `tar -cjf FILE_NAME PATH`. To the best of our knowledge, only two datasets are available for this task, with one based on the other. Both datasets involve scraping through known data sources (through platforms like stack overflow, crowdsourcing, etc.) and hiring experts to validate and correct either the English text or Bash Commands.

This chapter presents two contributions to the research on synthesizing Bash commands from scratch. Firstly, we describe a state-of-the-art translation model used to generate Bash commands from the corresponding English text. Secondly, we introduce a new NL2CMD dataset that is automatically generated, involves minimal human intervention, and is over six times larger than prior datasets. Since the generation pipeline does not rely on existing Bash commands, the distribution and types of commands can be custom-adjusted. Our empirical results show how the scale and diversity of our dataset can offer unique opportunities for semantic parsing researchers. As commonly observed in this research domain, only marginal improvements in the accuracy of solutions to the NL2Bash problem have occurred since progress has been impeded due to the limited amount of annotated data. This chapter extends our prior published work [22] on translating natural language to Bash commands and provides the following new contributions beyond our prior work:

1. A membership query synthesis technique is used to generate a large dataset of Bash commands, ex-

panding the available data to solve this problem.

2. A back-translation technique is presented that takes the generated Bash commands and creates corresponding natural language pairs.

3. A validation and verification technique for generated Bash commands is discussed by converting them into an executable form and running them in an isolated environment, which doubles as a data quality metric.

4. A new dataset called NLC2CMD is presented, which is the largest dataset for translating natural language to Bash commands available to researchers and practitioners.

5. A post-processing addition to the original workflow is adopted, replacing placeholder values with actual arguments and making many of the translated commands executable in the Linux environment.

Most importantly, our work suggests that the future of addressing this challenging problem lies in the automation of Bash command synthesis and back-translation to natural language representations. We have developed a workflow that can be improved upon and used to maximize model performance with minimal labeling costs. Our approach has numerous advantages over prior work, including notable improvements in time efficiency, labeling costs, diversity of the dataset, and practicality.

The remainder of this chapter is organized as follows: Section 4.2 introduces the NLC2CMD problem and provides an overview of recent developments in semantic parsing; Section 4.3 summarizes the challenges associated with translating natural language to Bash commands; Section 4.4 examines the performance of different model structures and training techniques; Section 4.5 describes our data generation and validation technique, as well as providing statistics on data quality and comparisons with existing datasets; Section 4.6 discusses different metrics and error analysis for the state-of-the-art model on our new dataset; and finally, Section 4.7 presents concluding remarks and outlines our future work.

4.2 Background and Related Work

Our contributions focus on advancing machine translation through the use of dataset synthesis. This research approach is novel in the domain of Bash command translation. Various architectures have been explored for different tasks of program synthesis from natural language. For example, Lin *et al.* [84] achieved state-of-the-art generation of shell scripts using Recurrent Neural Networks (RNNs) [85]. Similarly, Zeng *et al.* [86] utilized the Bert [87]-based encoder and a pointer-generator [88] decoder to generate SQL code from text. Additionally, ValueNet [89] (Transformer encoder + LSTM decoder with pointer networks [90]) was the first Text-to-SQL system incorporating values. Furthermore, Xu *et al.* [91] improved upon the TranX [92]

transition-based neural semantic parser to translate natural language into general programming languages, such as Python.

The best results in prior work on the problem of translating natural language to Bash commands were produced by Tellina [84]. Tellina used the Gated Recurrent Unit (GRU) network [93], which is an RNN, and achieved 13.8% accuracy on the NLC2CMD metrics proposed by IBM [94]. The Tellina [84] paper produced the NL2Bash [82] dataset and new semantic parsing methods that established the baseline for mapping English sentences to Bash commands. Transformer models have been shown to generally have better accuracy and parallelism [95] than RNNs [85] on machine translation tasks. Prior research on machine translation has largely focused on the GRU architecture to translate natural language to Bash commands. This paper enhances prior research by investigating the performance of several architectures on the NLC2CMD dataset.

Our experiments with applying Transformer models to the natural language to Bash task show that they outperform other approaches, such as (1) RNNs that show an 18.4% improvement and (2) Bidirectional RNN (BRNN) that show up to 4.4% improvement [96]. Investigating how model structural choices and prediction strategies affect model performance in the natural language to command translation task [94] is a key contribution of this paper. Since the energy and accuracy metrics for model evaluation were specifically designed for the NL2CMD competition, potential improvements for the metrics are also discussed. Bash is a widely used command line scripting language, thus it offers a unique opportunity to generate diverse, and more importantly, executable commands more easily due to its relatively short and simple nature. To increase programmer productivity, the Bash Commands suggested by a tool should be both syntactically and semantically correct. If suggestions are not syntactically correct and cannot execute, programmers may simply ignore them as they distract from the task at hand. Moreover, if translations are not semantically correct, programmers may execute Bash Commands that do not accomplish the goal they want to achieve, or worse, have negative impacts on the system, such as deleting important files or directories.

Our approach differs from previous research in that we focus on dataset synthesis, utilizing transformer-based models for parallel corpus mining in the domain of machine translation. While previous research has employed classification techniques, such as document similarity [18], to identify translations from pre-existing corpora, our approach emphasizes the generation of a new, high-quality dataset to improve performance in the natural language to Bash Commands translation task.

4.3 Key Research Challenges

In this section, we will succinctly outline the significant research challenges encountered in the process of translating natural language to Bash commands and highlight the general obstacles that the machine transla-

tion community is currently facing in regards to this topic.

4.3.1 Ambiguity

Translating from an ambiguous language to precise Bash Commands is hard. Translating human language into code is a challenging task, due in part to the inherent ambiguity of natural language. The Winograd Schema Challenge, as exemplified by the sentence "The trophy would not fit in the brown suitcase because it was too big", `it` can either mean trophy or suitcase, illustrates the complexity of resolving pronoun references, a task that requires both knowledge and commonsense reasoning [97].

In the context of Bash Commands, ambiguities can present in two forms [98]. **Genuine ambiguities** occur when a sentence has multiple valid interpretations for an intelligent listener, as in the example "merge file A with B in folder C", which could be interpreted as "merge file A with B if B is in folder C" or "merge file A with B and put the result in folder C". **Computer ambiguities**, on the other hand, occur when a sentence is clear to a listener but generates multiple parse trees for a computer, leading to undefined word order for input. Both types of ambiguities can negatively impact the performance of natural language to Bash Command translations.

4.3.2 Many-to-many mapping

Translation tasks are often characterized by many-to-many mappings, which indicate that there may be multiple correct translations for a given input sentence. Additionally, the input sentence itself may have various possible expressions. As the size of the dictionary increases, the number of possible translations for a given input will also increase. The process of creating target sentences requires a significant amount of human effort.

Natural language is inherently flexible and Bash commands may have functional overlap between different utilities. For example, when translating natural language to Bash, phrases such as `find the word "foo" in file "bar"` and `search in "bar" for "foo"` have the same meaning. Similarly, both `grep -w foo bar` and `cat bar | grep -w foo` are valid translations.

4.3.3 Scarce Paired data

Machine translation models require a substantial amount of training data in order to effectively generate translations. The collection of such a corpus can be challenging, particularly for supervised learning approaches that require paired data, as it necessitates a thorough understanding of both the source and target languages. Without a large number of training examples, the model's ability to generalize beyond the small samples present in the training set may be limited.

Translating natural language to Bash commands poses a unique challenge, as there are a vast number of English sentences and Bash commands. However, obtaining paired data (i.e., English sentences with their corresponding Bash commands) is not easily achievable. While sources such as coding help forums like Stack Overflow may provide some paired data, the questions posed on such platforms are often detailed descriptions of commands which are subsequently summarized succinctly by humans. Furthermore, writing Bash commands requires a high level of coding expertise, making it difficult to crowd-source this task.

4.3.4 Environment dependent

Bash Commands change environments and are generally computer specific. Bash commands are commonly executed on the command line and are utilized for tasks such as file manipulation, searching, and application-specific scripting. When these commands are generated randomly in large quantities, they can lead to unintended consequences such as deleted files, undefined behavior, and excessively large searches. These outputs not only put a strain on the environment in which they are executed, but can also cause damage to the system itself by deleting important files and directories.

In addition, the variability in file systems and the different methodologies used by humans for organizing their file systems results in an infinite number of unique configurations. Each configuration has its own distinct directory structure, permissions, and file names. Therefore, commands that are generated or scraped from the internet may not be guaranteed to execute correctly on different machines. Furthermore, what may achieve the desired result on one machine, may cause another to crash.

4.4 Research Questions

Which deep learning architectures perform best when translating between natural language and Bash Commands?

As there is a limited amount of literature on the topic of translating natural language to Bash commands, a crucial concern is determining which architectures, developed in other domains, perform optimally for this task. Specifically, Sequence-to-Sequence [99] models have been extensively studied in the context of translations, thus we investigated their performance on this specific task. These models consist of two primary components: an *encoder* and a *decoder*. The encoder converts the inputs into vectors while the decoder reverses this process. We evaluated various combinations of encoder-decoder layers, including RNN, BRNN, and Transformer, to translate natural language to Bash commands.

According to the findings of Chen et al. in their study [100], it was determined that the Transformer encoder was primarily responsible for the improvements in quality observed in Transformer models. Additionally, it was noted that Recurrent Neural Network (RNN) decoders often exhibit faster inference times.

In light of these discoveries, we conducted an investigation in which various combinations of encoder and decoder types were utilized and evaluated. The results of this study are summarized in Table 4.1, which presents a comparison of the performance (measured in seconds) across the different model structures.

Table 4.1: Model Performance Comparison

Encoder	Decoder	Accuracy	Train	Inference
Transformer	Transformer	**0.522***	1625	0.126
Transformer	RNN	-	-	-
RNN	Transformer	0.486	1490	0.116
RNN	RNN	0.336	**1151***	0.069
BRNN	Transformer	0.495	1411	0.120
BRNN	RNN	0.476	1218	**0.065***

Table 4.2: The NLC2CMD Leaderboard

Team	Model	Data Augment	Accuracy	Power	Latency
Magnum	Transformer	No	**0.532***	682.3	0.709
Hubris	GPT-2	No	0.513	809.6	14.87
Jb	Classifier+Transformer	Yes	0.499	828.9	3.142
AICore	Two-stage Transformer	No	0.489	**596.9***	0.423
Tellina [84]	BRNN (GRU)	No	0.138	916.1	3.242

The results presented in Table 4.1 indicate that in this specific case, utilizing the Transformer architecture for both the encoder and decoder yields the highest accuracy. However, it is worth noting that the model with an RNN decoder has the potential to significantly reduce inference time by approximately 50%. To provide a comprehensive understanding of how model architecture can impact performance, we conducted an analysis of the architectures of the top-performing teams in the NLC2CMD competition. Table 4.5 provides a summary of the top 4 teams and the baseline model as listed on the NLC2CMD Challenge leaderboard [94]. The Transformer architecture discussed in Section 4.4.1 was derived from the analysis of the architecture utilized by our team, Magnum, which was the winner of the accuracy competition.

The AICore system, as described in the study by Agarwal et al. (2021) [94], won the energy track of the competition through the implementation of a two-stage prediction design, comprising of two 2-layer Transformers. The first model in this system was responsible for predicting the template, while the second model filled in the arguments. The relatively low energy consumption of the AICore system is thought to be a result of its smaller model sizes, in comparison to the Magnum team's model which consisted of six layers. However, it should be noted that this reduction in energy consumption was accompanied by a decrease in accuracy of 4.3%.

Team Hubris [94] adopted a fine-tuned ensemble of GPT-2 as the language model for their approach and achieved second place in terms of accuracy. However, it should be noted that GPT-2 models are known to

be large in size, with an average memory requirement of more than 5 GB, and are also known to be power-intensive. This poses a significant challenge when it comes to their application as a background program, running continuously in a terminal, to suggest translations of Bash commands. Furthermore, the inference time of GPT-2 ensembles (774M params) is also a significant concern, as it is not feasible for real-world deployment scenarios that require fast response time and low energy consumption to run continuously in the background. Considerable effort is required to compress and deploy GPT-2 ensembles in order to compete with other solutions.

Team Jb [94] augmented the training data using back-translation [101] and created 78,000 augmented training samples. They also utilized the manual pages of Linux Bash Commands [102] to concatenate utilities with corresponding flags, resulting in the generation of an additional 200,000 new samples. Similar to Team AICore, they also employed a two-stage model, which consisted of a classifier for utility prediction and a transformer for command generation. Interestingly, the large number of additional training samples generated by Team Jb was not sufficient to overcome the architectural improvements implemented by other teams.

The results presented in Table 4.5 provide several noteworthy observations. Firstly, it is apparent that Transformer models were the most commonly utilized architecture in this task. Additionally, it was observed that two-stage models performed worse than single-stage-and-larger models. Secondly, GPT-2 based approaches demonstrated near state-of-the-art accuracy, however, they resulted in much larger models in comparison to Transformers and exhibited longer inference times. Thirdly, data augmentation techniques were found to improve accuracy, as evidenced by Team Jb's 1% improvement in accuracy over Team AICore. However, it is important to note that the impact of data augmentation was less pronounced than the impact of model architecture in this task, with the caveat that the two teams had similar, but not identical, models. In light of these findings, the experiments in the remainder of this paper employ Transformer models, as they were determined to be the best-performing architecture in the NLC2CMD competition.

How do Bash Command parameters affect the performance of natural language to Bash translation?

As previously discussed in Section 4.3.4, obtaining a sufficient quantity of training data consisting of paired English and Bash Commands is a challenging task. The lack of adequate training data may result in the model being unable to fully learn the entire vocabulary that it must translate between. Therefore, identifying methods to reduce the vocabulary size is crucial for the development of more accurate models.

Bash Commands are typically composed of three elements: (1) utilities, which specify the primary objectives of the command (e.g., `ls`), (2) flags, which provide metadata concerning the execution of the command (e.g., `-verbose`), and (3) parameters, which specify directories, strings, or other values that the command should operate on (e.g., `/usr/bin`). Each utility is associated with a limited set of flags that can be passed

to it. In contrast, parameters have a much broader range of values. Training examples for natural language to Bash Command translation typically provide values for the parameters, which can vary significantly between translated examples of the same command.

We hypothesized that including the actual parameter values, such as `ls /usr/bin` and `ls /etc`, from the training examples would vastly increase the overall vocabulary size and decrease model accuracy. The basis for this hypothesis is the limited availability of paired examples of natural language and Bash commands, which can lead to a larger vocabulary size and limited training data, thus negatively impacting the performance of translation models.

In order to investigate the validity of our hypothesis, we used the English and Bash tokenizers from the Tellina model [84] with a modification to suit our needs. As depicted in Figure 4.1, Bash tokens can be classified into three categories: utilities, flags, and parameters (i.e., arguments, such as a specific file path). The English tokenizer was used to lowercase all letters and to replace specific parameter values with generic forms. The Bash tokenizer, on the other hand, parsed the commands into syntax trees, with each element labeled as a utility, flag or parameter.

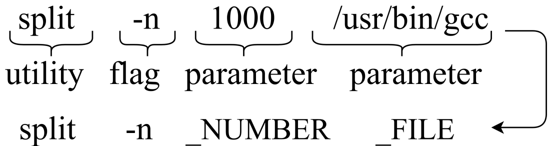


Figure 4.1: Example of a Bash Command

In our assessment of the accuracy metric, we placed a primary emphasis on the structural and syntactical correctness of Bash commands. To achieve this, we replaced all parameters in Bash with their corresponding generic representations. For instance, a folder path such as `/usr/bin` was replaced with the token `PATH`. This transformation resulted in a reduction of the Bash vocabulary size from 8,184 to 776 tokens, and a subsequent increase in the accuracy of the Transformer models we evaluated by 1.3%. As demonstrated in Table 4.3, we observed increases in accuracy and performance across all architectures, particularly for those with lower accuracy.

Table 4.3: Parameter Replacement

Encoder	**Decoder**	**Accuracy**	**Accuracy (NP)**
Transformer	Transformer	0.509	**0.522***
Transformer	RNN	-	-
RNN	Transformer	0.448	**0.486***
RNN	RNN	0.151	**0.336***
BRNN	Transformer	0.483	**0.495***
BRNN	RNN	0.301	**0.476***

How to expand the amount of available Bash Command English language pairs without the hiring of external freelancers?

As previously discussed, further innovation and developments when attempting to solve the natural language to Bash Command translation problem are severely restricted by the limited number of command-natural language pairs provided in the original dataset. This issue not only pertains to the most efficient means of incorporating new and valid Bash Commands, but also encompasses the creation of corresponding natural language pairs.

Obtaining Bash commands by scraping online forums such as Stack Overflow is a viable method for expanding the dataset, however, it presents several challenges including the identification of invalid commands, the elimination of duplicate commands and the differentiation of commands written in different programming languages. An alternative approach could be to enhance the existing training data by making slight modifications to commands, such as the removal of a flag. However, this strategy also poses difficulties in distinguishing between similar commands and provides minimal diversity in terms of function within the dataset.

In order to address the aforementioned limitations and challenges, we chose to develop a Bash Command generator and utilize manual page data to synthesize entirely new Bash Commands. This approach enabled us to curate the synthesized dataset, preserving similarities to the existing dataset, while simultaneously incorporating novel data points. To generate corresponding natural-language components, we employed a back-translation model with a transformer architecture. Transformer models have been demonstrated to be highly effective in summarization tasks, which our back-translation process resembled.

4.4.1 Summary of the Highest Performing Architecture

In this study, we evaluated a variety of data processing, architectural, and post-processing techniques as previously outlined. We now present the best-performing model that we evaluated using the NLC2CMD competition data. While this model will likely be refined through future work, it serves as a starting point for researchers investigating natural language to Bash Command translation. Our results indicate that the Transformer model is a strong foundation for continued research in this field.

Our Transformer model pipeline consisted of the following six stages, as depicted in Figure 4.5 and described in detail below:

1. **Parsers and filters** – The raw paired data were first processed through various parsers to convert English sentences and Bash Commands into syntax trees, with data that could not be parsed being removed.

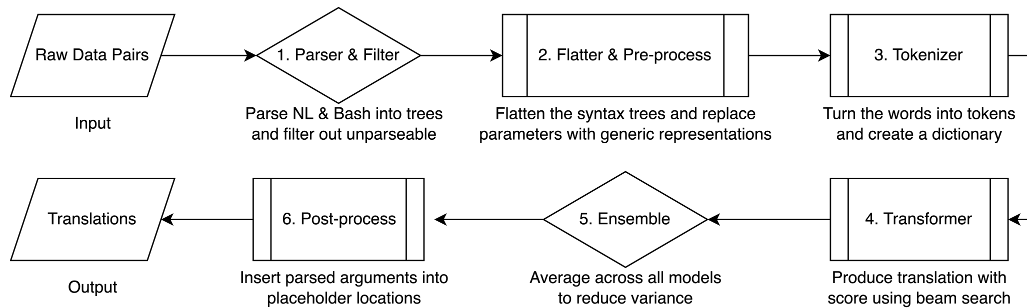


Figure 4.2: Pipeline of the NLC2CMD Workflow

2. **Flatten and pre-process** – The syntax trees were flattened, and the parameters were replaced with their generic representations.

3. **Tokenization** – The flattened sentence pairs were tokenized, and dictionaries were created for English sentences and Bash Commands.

4. **Transformer models** – - The tokenized sentences were input into Transformer models, with Beam Search enabled to produce multiple translations.

5. **Ensemble** – The best-performing models on the validation dataset were selected to create an ensemble.

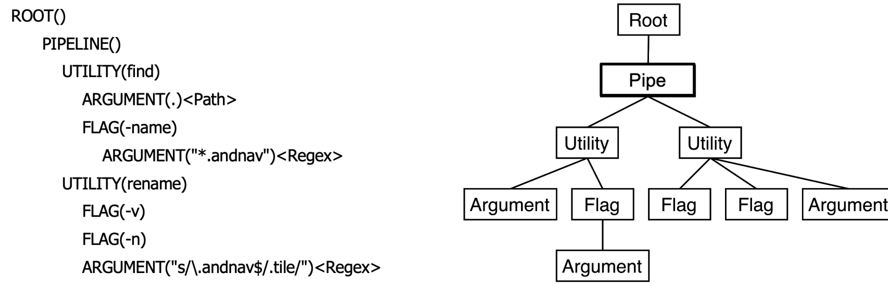
6. **Post-process** – The translations produced by the ensemble model were post-processed by removing the placeholder arguments and inserting the values that were originally removed by the parser.

4.4.2 Parsing and Tokenization

In our study, we utilized both the NLC2CMD dataset, which comprises 10,347 pairs of English sentences and their corresponding Bash Commands, and our synthesized dataset, which consisted of 71,705 English sentence-Bash Command pairs. Of the 10,347 pairs of data in the original dataset, 29 were excluded due to grammar issues. The size of this public dataset is relatively small in the field of natural language processing, in comparison to other datasets such as WMT-14 en-de, which comprises 4.5 million sentence pairs. Therefore, our aim for data processing was to create a small vocabulary and make use of as much data as possible. With our synthesized dataset being significantly larger, our focus shifted to achieving higher-quality commands, rather than utilizing as many of the generated commands as possible.

Bash Commands can exhibit complex and nested structures, as illustrated in Figure 4.3. This complexity can make it challenging for programmers to both create and understand Bash Commands, highlighting the necessity for a customizable parser. Additionally, Bash Commands can be concatenated using pipes, which allows commands to consist of multiple parts, with the output of one part serving as the input for the next.

cmd: find . -name "*.andnav" | rename -vn "s/\.andnav$/.tile/”



tokens: ['find', 'Path', '-name', 'Regex', '|', 'rename', '-v', '-n', 'Regex']

Figure 4.3: Visualization of the Tokenization Process

We constructed our parser using the Tellina parser [84], which was developed on top of the Bashlex parser [103] in previous work. This parser is able to parse a Bash Command into an abstract syntax tree (AST) composed of utility nodes, each of which may include multiple corresponding flags and parameters. During the tokenization stage, utilities and flags are retained in their original form while parameters are classified and replaced with tokens such as `_NUMBER`, `_PATH`, `_FILE`, `_DIRECTORY`, `_DATETIME`, `_PERMISSION`, `_TIMESPAN`, `_SIZE`, with the default option being `_REGEX`.

In the pre-processing of natural language sentences, stop words (e.g., “a”, “is”, “the”) which carry minimal meaning, are filtered out. The remaining words are then converted to lowercase and lemmatized (transformed to their base form) in order to create a relatively smaller dictionary mapping.

Our generator used a different parameter categorization strategy than the parser, with the objective of aligning more closely with the interpretation of manual pages and the generation of high-quality commands. While the categorizations were comparable, the generator’s categorization could easily be converted to the parser’s representation for training and inference with the Transformer-based model.

4.4.3 Model Details

The model with the highest accuracy used a Transformer as both the encoder and the decoder, as illustrated in Figure 4.4. The encoder and decoder were both composed of six layers. The model was trained for 2,500 steps and an ensemble of the four top-performing single models was employed.

The first positional weight was set to 1.0, and the remaining weights were set to the exponential of beam scores, with a maximum value of 0.5. Our aim was to train an efficient and robust model that could be easily deployed. As a result, the need for modifications to the network structure was relatively low. Instead of using FairSeq [104], which allows users to alter the low-level network structure, we selected OpenNMT [105], an open-source neural sequence learning framework, to implement our Transformer model.

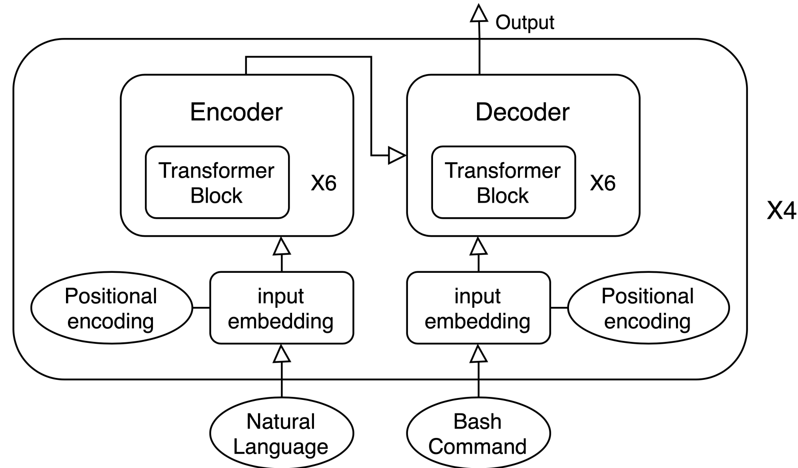


Figure 4.4: Model Structure

Our findings indicate that the Transformer model is sensitive to the learning rate, and that larger batch sizes tend to yield superior results. The specific training hyperparameters can be found on our GitHub repository [106]. Our tuning strategy was guided by the principles outlined by Popel et al. [107].

We trained our model on 2 Nvidia 2080 Ti Graphic cards with 64GB memory. Our model achieved a 53.2% accuracy on the hidden test dataset for the NLC2CMD competition and demonstrated superior performance in both inference time and energy consumption. To address the Challenge 4.3.1 of ambiguity, we implemented a technique of masking specific parameters. Additionally, the dataset used in our study was designed to limit ambiguity by restricting the natural language description to a single sentence and the Bash Command to a single line [82]. Furthermore, to address the Challenge 4.3.4 of low resource, the dataset was collected with the same Bash Command paired with multiple English descriptions, in order to increase language diversity [82].

4.4.4 Post-processing

The initial results of our model translation incorporated placeholders as a means of reducing the size of the dictionary mapping and improving the accuracy of the model. However, these results, while accurate, lacked practicality and were often difficult for humans to understand and execute. For example, the inclusion of placeholders such as _REGEX in a command translated from English made the command vague and difficult for users to utilize as a tool. To address this issue, we implemented a post-processing step in our workflow to convert the translated commands into a more executable form. This approach ensures that both inexperienced users and machines can easily utilize the commands by running them in a terminal, as opposed to commands that are nested with vague placeholders which may be less useful.

In the pre-processing stage of natural language processing, we utilized a parser to extract parameters from the original corpus. These parameters were then utilized to populate placeholders in the generated translations. In many cases, there was a direct one-to-one correspondence between the extracted parameters and the placeholders in the translated commands, resulting in fully executable Bash commands. However, in some instances, there were more placeholders than extracted parameters, resulting in partially populated commands that were not executable but were not necessarily incorrect translations.

In the process of translating natural language descriptions of tasks into executable commands, it may be necessary to utilize placeholders for certain values that are not explicitly provided in the original description. For instance, the command `find . -inum Quantity exec rm` may be generated in response to the instruction "remove all files in the current directory with a specific inode number". In this example, the specific inode number is represented by the placeholder `Quantity` as it is not specified in the original description. This scenario illustrates that the use of partially filled-in commands can still be accurate translations, and that efforts towards partial replacement are valuable in terms of usability and practicality of the translation pipeline. By converting Bash command templates into executable or nearly executable commands that more closely align with the intended purpose described in the natural language, the pipeline is made more user-friendly and useful.

4.5 Corpus Construction

We successfully constructed a corpus of "Bash Command and natural language" pairs that is six times larger than the original dataset. This was achieved through the development of a generator for synthesizing a vast number of Bash Commands, which were subsequently validated and scaled. These commands were then processed through our back-translation model to generate the corresponding natural language pairs.

4.5.1 An Updated Pipeline

In our updated pipeline, we have integrated the most effective elements of our prior pipeline to generate a novel dataset. Subsequently, we have trained and evaluated our top-performing model on this new dataset. To achieve this, we have employed a state-of-the-art transformer-based model, which has been demonstrated to be effective in machine translation tasks. This model has been incorporated into both our dataset generation and training processes, as outlined in Figure 6. Our updated pipeline comprises the following steps:

1. **Manual Page Scraping** – We extracted data from manual pages to identify the syntax usage of 38 utilities. Additionally, we determined the flags associated with each utility and the categorization of the parameter, if any, associated with each of those flags.

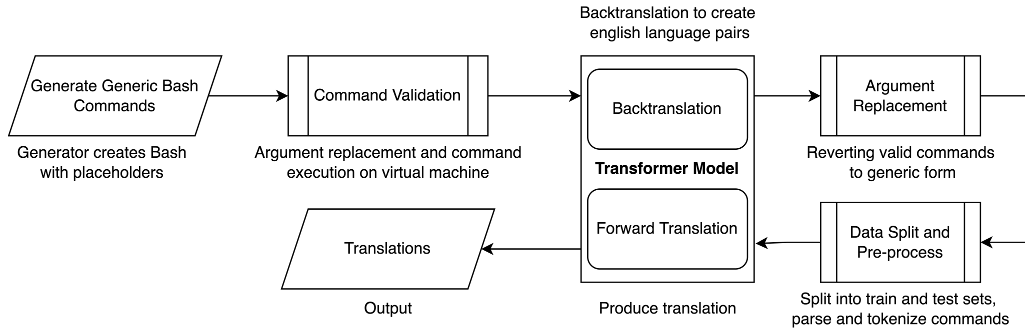


Figure 4.5: Updated Pipeline of the Dataset Generation and Translation

2. **Generation** – Utilizing the syntactical structures, flags, and arguments for each utility, we generated over 1 million Bash Commands from different combinations of flags and piped commands.

3. **Validation** – The generated commands were then replaced with actual arguments. For example, `[File]` was replaced with `temp.txt`. These commands were executed on a virtual machine, and we discarded all commands that did not execute successfully with exit statuses of zero within a given time frame.

4. **Scaling** – The validated commands were then converted into a form comprehensible by the parser, parsed, and scaled. This step involved maintaining a similar proportion of commands with the `find` utility to the original dataset and ensuring a diversity of other utilities in the new dataset. Additionally, we discarded commands of over-represented utilities and commands that were parsed incorrectly by the parser.

5. **Back-translation** – The validated commands were then converted into a form comprehensible by the parser and fed to the back-translation model. This model was the same transformer-based model utilized on the original dataset, except trained in the reverse direction, using Bash Commands to predict natural-language sentences. This step created the corresponding natural language pairs for the generated dataset.

6. **Forward translation** – The new dataset was then divided into training and testing sets, and used to train and evaluate the model. For the validation, the best-performing models on the validation dataset were chosen to create an ensemble.

4.5.2 Bash Command Synthesis

Our generation stage involved scraping manual page information and assembling together commands from individual components. We used data gathered from the Linux manual pages to form syntax structures to

help our generator understand the relationship between the different components from which commands are formed. Commands in Bash typically consist of utilities, flags, and arguments, but may exhibit increased complexity through the use of piped and nested commands. Our scraping methods involved sophisticated techniques and manual oversight to create a comprehensive mapping of utilities to their corresponding flags and arguments. Likewise, each flag had corresponding arguments, as the introduction of flags often increased the number of arguments in a command.

In addition to mapping utilities to their corresponding flags and arguments, we also extracted the syntax for each utility. This allowed us to create templates that provide insight into the context in which each utility is utilized and the order in which flags and arguments are presented. Our data collection efforts focused on the 38 most commonly used utilities in the original NL2Bash dataset. Through the utilization of this data, we were able to generate thousands of commands that include combinations of zero to three flags. The potential number of commands that our generator is capable of producing is in the billions. However, for the purposes of our study, we exercised restraint and limited the number of commands generated for various reasons:

- **Quality preservation**. While the generation of commands with a large number of pipes and flags may result in a significant number of valid and executable commands, such commands are relatively rare and are therefore not commonly found in the original training dataset. To preserve the characteristics of the original dataset, we aimed to maintain similar distributions of utilities and ratios of commands with pipes to those without.

- **Practicality**. Both the validation and back-translation processes, as described in our methodology, required a significant amount of time and resources to process the dataset. As the number of commands increases, the amount of time and resources required also increases. Therefore, after generating several hundred thousand commands, we deemed it infeasible to devote further resources towards additional command generation.

In the scaling stage, we aimed to ensure that the distribution of utilities in the generated commands closely resembled that of the original NL2Bash dataset. To achieve this, we scaled the number of generated commands for each utility to match the proportion of that utility in the original dataset. For example, the original dataset consisted of 63.44% commands that began with the utility `find`, so we scaled the number of `find` commands generated to represent a similar percentage in our generated dataset.

We attempted to do this scaling for all generated utilities, although we achieved varying results. The original dataset comprised 117 distinct utilities, whereas the generator only supported 38, as depicted in Table 4.4. Furthermore, the manual pages for a significant number of these 38 utilities lacked comprehensive

Table 4.4: Command Generation Process

	Generation	**Validation**	**Scaling**
Utility Count	38	35	35
Non-piped Commands	570,436	60,926	38,557
Single-piped Commands	500,000	81,787	33,148

or consistent documentation, which made it challenging to accurately gather all relevant flags and arguments and generate a sufficient number of commands to meet the desired distributions.

Another constraint in the distribution matching process was the challenge of generating valid commands for certain utilities. As discussed in a subsequent section, each generated command was later evaluated for validity to determine its inclusion in the dataset. The probability of commands for certain utilities being deemed invalid was substantial. Consequently, generating a substantial number of commands for these utilities negatively impacted their representation in the dataset.

Certain utilities had a limited number of available flags and a high frequency of duplicate instances in the original dataset. For instance, the command `cd [Directory]` was observed 13 times in the training data as it originally appeared without placeholders. In the generated dataset, however, there were no duplicate commands, hence the command only appeared once, which resulted in a limited representation of that utility in the generated data.

The generated dataset included not only commands that utilized a single utility, but also piped commands. These commands are composed of multiple utilities or commands that are concatenated, enabling the sharing of information during execution. To support the generation of piped commands, an analysis of the training data for common utility pairs was conducted, followed by the independent generation of commands for each utility and their subsequent concatenation using the pipe symbol.

The utility pair that was found to be most prevalent in the training data was (`find`, `xargs`). Specifically, it was observed that a command utilizing the `find` utility was frequently followed by an instruction utilizing the `xargs` utility in a piped sequence. An analysis of the original training dataset revealed that 31.36% of the commands contained one or more pipes, with an average of 1.45 per piped command.

In light of the added complexity involved in the support of piped commands, simplifications were made in the generation of such commands. Specifically, it was observed that 70.33% of the piped commands in the training data comprised of a single pipe. Therefore, the generation of piped commands was limited to those containing a single pipe. Additionally, it was noted that the distribution of utilities in piped commands was more limited in comparison to those without pipes. Specifically, 43.71% of the single pipe commands in the training data consisted of a sequence of `find` commands followed by `xargs`, `grep`, and `sort`

commands. Given this information, the decision was made to support these specific combinations of utilities in the generation of piped commands.

The inclusion of piped commands into the generated dataset significantly expanded the number of available commands for generation. The introduction of pipes resulted in an exponential increase in the number of potential commands, thereby adding pressure to the validation process which involved the sequential execution of all generated commands. As a result, the generation of piped commands was restricted to combinations of 500 distinct `find` commands concatenated with 1,000 distinct `xargs`, `grep`, and `sort` commands, resulting in a total of 500,000 commands synthesized in the initial stage.

4.5.3 Bash Command Validation

To ensure the validity of the generated commands, each command was replaced with valid placeholder arguments and executed in an isolated environment. For instance, `cd [Directory]` would be converted to `cd abc`, where the directory `abc` is available on the machine. To protect against undefined behavior, commands were executed in a virtual machine environment.

The commands were executed in a sequential manner and the program recorded the exit status of each command. The commands that completed execution and returned with exit statuses of zero were retained, while those with non-zero exit statuses were discarded. The valid commands that were retained were then converted back to their generic forms, this time utilizing the placeholders available in the original NL2Bash dataset. To prevent the commands from hanging, a timeout of 0.5 seconds was implemented for each command during the validation process, after which the command was deemed invalid. However, it was observed that the percentage of commands deemed invalid due to timeouts was negligible.

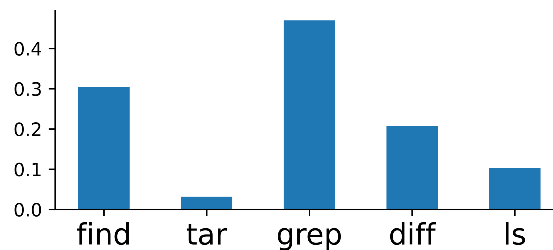


Figure 4.6: Valid Command Rates For Generated Commands

The rate of commands that completed execution and returned with exit statuses of zero varied significantly among different utilities. Notably, commands such as `grep` and `ls` exhibited lower error rates, with approximately 50% of them returning with exit statuses of zero. The most common utility in the training data, `find`, had a validity rate of 30.4%. Two utilities, `rev` and `rename`, did not have any generated commands that executed with exit statuses of zero and were thus removed from the generated dataset altogether. Overall,

13.3% of the generated commands were deemed valid, with validity rates of 10.7% and 16.4% for non-piped and piped commands, respectively.

An exit status of zero, indicating successful execution, does not necessarily indicate the effectiveness or practicality of a command. For instance, a command such as `cd .`, which changes the working directory to the current one, may execute without error, yet it serves no functional purpose as it does not result in a change to the environment.

It is important to note that commands that fail validation are not necessarily incorrect. The generated commands are generic in nature and arguments are replaced during the validation stage. Therefore, command failure can occur as a result of the replacement of specific arguments. Additionally, the virtual environment may not possess the same directories, libraries, and files that are manipulated in a given command, which can lead to command failure. While execution can be an effective method for eliminating a large number of incorrectly generated commands, it does not necessarily provide an accurate assessment of command validity in all cases.

4.5.4 English Text Synthesis

In order to generate natural language for our generated Bash commands, we utilized the transformer-based model that was previously employed in our initial translation task. This model architecture has demonstrated exceptional performance in machine translation tasks, and therefore we decided to reuse it for the back-translation process instead of opting for a less effective model. For this purpose, we trained the model using data from the NL2Bash dataset, but in the reverse direction, specifically, by attempting to predict natural language from Bash commands.

After training the model, we employed inference to predict the natural language equivalent for each command in the generated dataset. This step completed the natural language component for every validated Bash command and marked the end of the dataset generation process.

4.5.5 Corpus Description and Statistics

Our corpus consisted of a total of 71,705 validated Bash commands, each accompanied by its corresponding English text. Of these commands, 69.9% utilized the `find` utility, while the remaining commands were distributed among 34 other utilities, resulting in a total of 35 utilities represented. The most frequently used utilities were `find`, `tar`, `grep`, `diff`, `ls`, `file`, `du`, and `cp`.

All our generated commands contained between zero and three flags for each utility within the command. Typically, commands without a pipe included one utility, while commands with a pipe incorporated two utilities. However, for certain utilities such as `xargs`, nested commands allowed for the inclusion of additional

utilities. As a result, the generated commands contained between one and four utilities, with the vast majority containing just one or two. Out of the generated commands, 46.22% (33,148) included a single pipe, while the remaining commands did not include any pipes. Every command generated was executed in a virtual machine command-line environment and was able to complete execution with an exit status of zero within 0.5 seconds.

4.5.6 Data Quality

In order to enhance the quality of the generated commands, we made a deliberate decision to not utilize the same argument-type placeholders as the parser during command generation. As previously discussed, the parser typically defaulted to classifying parameters as `_REGEX`, which a difficult placeholder for executable command generation as it can have various forms.

In lieu of utilizing the same argument-type placeholders as the parser, we employed 15 argument-type placeholders which, while similar to those of the parser, were specifically chosen to achieve the goal of generating valid and executable commands. Our objective was to select argument-type placeholders that were as specific as possible, yet still able to be automatically scraped and classified from manual pages. Rather than relying on a single argument type, `_REGEX`, we incorporated `Pattern`, `FormattedString`, and `Separator` to differentiate between the various forms of regular expressions and enhance the specificity of the generated commands. This approach was feasible as manual pages also differentiate between these argument types, enabling us to classify these flags based on keywords in the manual pages with minimal additional effort.

While our strategy for handling parameters and placeholders diverged during the command generation stage, it was important to ensure that this inconsistency did not carry over to the translation process. The parser and its associated placeholders had been demonstrated to be effective in preprocessing, thus, after command generation, we converted the placeholders to align with those of the parser for inclusion in our synthesized dataset. This conversion process was relatively straightforward as each of our chosen placeholders was mapped to a single parser placeholder.

Moreover, our validation process, which involved injecting actual arguments into the placeholders and executing the generated commands in a virtual machine environment, ensured a high level of quality for the generated commands. Of the 570,476 no-pipe commands generated, 60,926 commands (10.7%) were able to complete execution with a successful exit status of zero.

It is worth noting that such a level of validation was not applied to the initial dataset. Our analysis revealed that a significant proportion of the commands present in the original NL2Bash dataset were unable to complete execution with a successful exit status of zero in our testing environment. This finding suggests

the potential for Bash Command generation to yield a higher percentage of valid commands compared to those that have been manually verified by expert freelancers.

4.5.7 Comparison to Existing Dataset

As previously discussed in Section 4.5.2, efforts were made to align the distribution of generated commands with that of the training dataset. Specifically, the majority of training commands utilized the `find` utility, and this proportion was taken into consideration when determining the size of the generated dataset. However, despite these measures, notable disparities were observed between the existing and generated datasets, as illustrated in Table 4.5.

Table 4.5: New vs. Existing Dataset

Category	Original	Generated (Raw)	Generated (Valid)
Total Commands	10,348	1,070,436	71,705
Piped Commands	3246	500,000	55,931
Distinct Utilities	117	36	36

The original dataset displayed a wide range of command variations, with 117 distinct utilities represented. Given that our generator only supported 38 utilities, a higher proportion of certain utilities were included to compensate for the absent utilities. In most cases, the quantity of generated commands for a given utility significantly exceeded the number of corresponding commands in the original dataset. The distribution of utilities in the generated and original datasets are illustrated in Figures 4.7 and 4.8, respectively, where the most prevalent utilities in the generated dataset are compared to those in the original dataset.

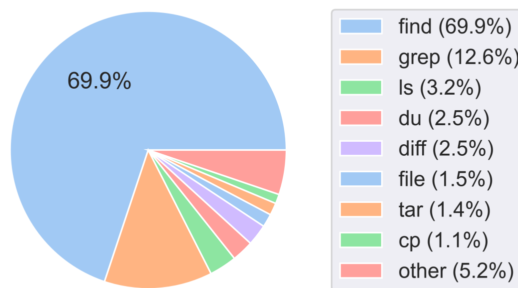


Figure 4.7: Utility Distribution in the Generated Dataset

There are various factors that contribute to the dissimilarity in the composition of the two datasets. One of the reasons is that the original dataset comprises of many duplicate commands or commands with similar structures applied to different files, directories or other arguments. Conversely, this level of repetition did not occur in the synthesized dataset, as each generated Bash Command is unique. This duplication also

resulted in the underrepresentation of popular commands with specific use cases, such as the command `cd [Directory]`, in the generated dataset in comparison to the original dataset. More generally, the original dataset comprised of a significant number of popular commands and underrepresented less common commands or those with uncommon flags, as a result of the data collection strategy of sourcing from online forums, as demonstrated in Figure 4.8. This approach yielded a heavily skewed dataset and inadequate

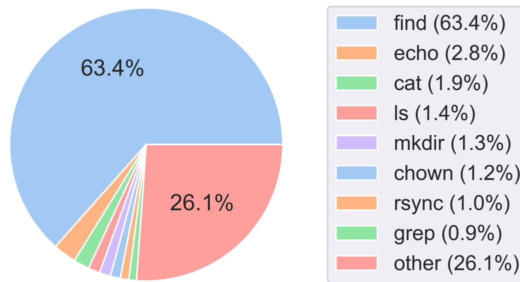


Figure 4.8: Utility Distribution in the Original NL2Bash Dataset

coverage of some powerful but infrequently used flags. In contrast, our generated dataset did not prioritize popularity and treated all flags equitably, although those less likely to cause errors or prolonged execution times were often screened during validation stages. Consequently, our dataset was significantly more diverse, featuring a variety of unusual flag combinations not present in the original dataset.

Furthermore, our synthesized dataset incorporated up to three flags for each utility in the command, and at most one pipe within the command. This configuration resulted in every generated command having a maximum of two utilities and six flags, with the exception of a small proportion of nested commands. While the majority of commands in the existing dataset conform to this demographic, many commands in the training data had multiple pipes or more than three flags following a utility. This configuration implies that while the generated dataset included a vast array of diverse commands, the commands were generally of shorter length and relatively simple in nature.

Another significant difference pertains to the validity of the commands. Since the entire generated dataset underwent validation and only valid commands were retained, the resulting dataset had 100% validity. In contrast, when the original dataset was subjected to the same conditions, only 2,360 commands were able to execute with exit statuses of zero within the 0.5-second time frame, resulting in a validity rate of 22.8%.

In summary, the analysis above illustrates that although the two datasets shared several similarities in terms of utility composition, there were key differences in terms of utility and flag diversity, as well as validity rates.

4.6 Metrics and Error Analysis

This section presents an examination of various metrics and performances of the transformer-based Magnum model on both datasets. Section 4.6.1 explains the accuracy metric and introduces an enhanced energy metric, Section 4.6.2 provides an overview of the accuracy performance of the Magnum model on the novel NL2CMD dataset, and Section 4.6.3 examines the distribution of various error types on the original dataset.

4.6.1 Metrics

Below, we describe the accuracy metric and propose an improved energy metric.

Accuracy: The ideal evaluation metric would verify if the predicted Bash Command produces the same outcome as the reference answer. However, this is not a practical approach since simulating 10K variant situations falls outside the scope of this study. Instead, our scoring mechanism specifically checks for structural and syntactic correctness, which "incentivizes precision and recall of the correct utility and its flags, weighted by the reported confidence" [94]. The metric defines two terms: Flag score S_F^i and Utility score S_U^i.

As shown in Equation 4.1 [94], the flag score is defined as twice the union of reference flags and predicted flags number minus the intersection, scaled by the max number of either reference flags or predicted flags. The range of flag scores is between -1 and 1.

$$S_F^i(F_{\text{pred}}, F_{\text{ref}}) = \frac{1}{N}\Big(2 \times |F_{\text{pred}} \cap F_{\text{ref}}| - |F_{\text{pred}} \cup F_{\text{ref}}|\Big) \tag{4.1}$$

As shown in Equation 4.2 [94], the utility score is defined as the number of correct reference utilities scaled by capping flag score between 0 and 1, minus the number of wrong utilities, scaled by the maximum number of either reference utilities or predicted utilities.

$$S_U = \sum_{i \in [1,T]} \frac{1}{T} \times \left(|U_{\text{pred}} = U_{\text{ref}}| \times \frac{1}{2}\Big(1 + S_F^i\Big) - |U_{\text{pred}} \neq U_{\text{ref}}| \right) \tag{4.2}$$

By taking the sum of all the utility scores within a predicted command, the range of normalized utility scores is between -1 and 1.

Energy: The assessment and documentation of energy consumption in natural language processing (NLP) models is an emerging field of research [108] [109]. As highlighted by Henderson et al. [70] in their systematic review, a significant contributing factor to this lack of attention is the complexity of data collection. Specifically, according to Appendix B of Henderson *et al.* [70], out of 100 papers from the 2019 NeurIPS proceedings, only 1 paper included measurements of energy consumption, while 45 papers reported on runtime performance.

In an effort to address the lack of attention towards energy consumption in NLP models, the NeurIPS 2020

conference proposed the use of "energy" as a more direct metric for evaluating the environmental impact of these models. However, upon examination of the current energy metric used in the NL2CMD competition, it was determined that this metric, which relied on estimated attributable power draw in milliwatts to compute scores, was not optimal. This approach disproportionately penalized models with shorter inference times.

One example of the limitations of the current energy metric used in the NL2CMD competition is the discrepancy in energy consumption between the GPT-2 model and a smaller model such as a GRU. Despite its larger model size and longer inference time of 14.87 seconds, the GPT-2 model's score on the power metric is lower than the baseline GRU model, which has a much shorter inference time of 3.24 seconds, as shown in Table 4.5. Additionally, the energy consumption in milliwatt-hours (mWh) can be easily manipulated by extending the inference time, for instance, by adding a delay of 3 seconds after each batch. This results in an improvement of the test submission score from 682 to 88 on the leaderboard. To address these issues, an alternative approach would be to measure the total energy consumed, rather than power, as this would take into account both the model size and inference time in determining energy consumption.

Validity: Another metric that can be used to evaluate the quality of the commands in a dataset is a measure of the validity rate. The validity rate metric was defined as the percentage of commands that were able to execute to completion with exit statuses of zero within a 0.5-second time frame, when replaced with standard replacement values. While this metric is not entirely reliable due to the complexities of different computer systems and file systems, it does provide valuable insight into the general correctness of the commands in the dataset.

Commands that *passed* the validity test indicate a baseline level of error aversion and demonstrate that they do not result in undefined behavior under the specified conditions. However, it is important to note that commands that do *not pass* the validity test are not necessarily incorrect. They may execute correctly on a different machine or may require more time than the allotted 0.5 seconds to execute.

4.6.2 Synthesized Dataset Results

The generated dataset was partitioned in a 80:20 ratio for the training and testing sets, respectively. The transformer-based model was trained on the training dataset. The results were subsequently evaluated, yielding an accuracy score of 31.63%, as depicted in Table 4.6.

The relatively low accuracy score obtained serves as an indication of the complexity of the dataset. It is hypothesized that this outcome was a result of the presence of utility-flag combinations that were not present in the original dataset. This led to inaccuracies in the back-translations for the natural language components, thereby rendering predictions for the model substantially challenging.

In addition, it is suspected that the model trained on the NL2Bash dataset may have exhibited overfitting,

Table 4.6: Comparison of Model Performance Across Datasets

Training Dataset	Test Dataset	Accuracy
NL2Bash	NL2Bash	52.3%
NLC2CMD	NLC2CMD	31.6%
NLC2CMD	NL2Bash	-13%
NL2Bash	NLC2CMD	-6.67%
NLC2CMD+NL2Bash	NL2Bash	48%
NLC2CMD+NL2Bash	NLC2CMD	31.7%

due to its relatively small size. The results of cross-training the model on both datasets revealed satisfactory performance on both test sets, whereas the model trained on a single dataset demonstrated inadequate performance on the test set of the other dataset. This outcome further emphasizes the significance of utilizing large and diverse datasets for enhancing the robustness of the model.

4.6.3 Error Analysis

Previous studies, such as Lin et al. [82], have identified sparse training data, utility errors, and flag errors as the top three causes of inaccurate predictions in natural language-to-bash (NL2Bash) systems. However, as the extent of training data sparsity is subjective, this study focuses specifically on the analysis of incorrect utility and flag predictions. A separate, independently-created testing dataset consisting of 1,867 samples was utilized for evaluation, as opposed to the manual analysis of 100 samples from the development dataset, as previously employed. The results, depicted in Figure 4.9, indicate that a majority of errors, over two-thirds, are attributed to utility errors, indicating that a sufficient quantity of data for each utility is more critical than a diverse set of flags.

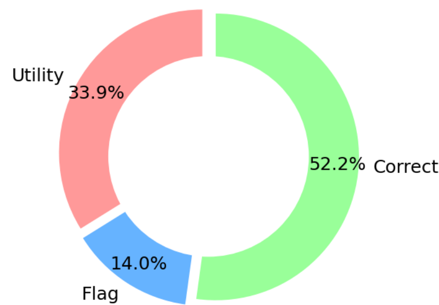


Figure 4.9: Percentage of Utility and Flag Errors on Original Dataset

The results of this study reveal the underlying cause of the lower accuracy score observed in our new dataset. Specifically, utilities that are less prevalent in the original dataset exhibit a greater number of commands. As depicted in Figure 4.10, among the top six incorrectly predicted utilities, the utilities `ls`, `grep` and `find` were found to be the most frequently confused. This is not unexpected as the functionalities of

these utilities overlap significantly, and they are also among the most commonly used Bash commands. A manual examination of the incorrect predictions also revealed that these three utilities are frequently utilized in piped commands, which can partially explain their high proportion among all incorrectly predicted utilities. Our synthesized dataset contained both a larger absolute and relative number of piped commands, further highlighting the challenges faced by the model in correctly predicting them.

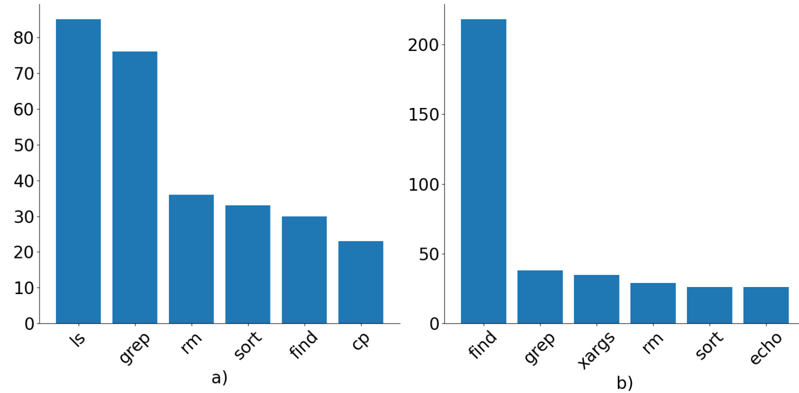


Figure 4.10: (a) Distribution of Reference Utilities that are Wrongly Predicted. (b) Distribution of Wrongly Predicted Utilities

4.7 Conclusion

The present study presents several significant contributions to the field of semantic parsing research. Firstly, a refined workflow for a state-of-the-art machine translation model was proposed to generate accurate and practical commands. Additionally, a post-processing technique was introduced to replace placeholders in translated Bash commands with the original parameters provided in natural language. Lastly, a new dataset was generated from scratch and an accompanying method for generating additional data was also presented. These contributions provide valuable insights and tools for the semantic parsing community to improve the accuracy and practicality of their models.

Our research has established crucial foundations for the development of an automated system for translating natural language to Bash commands. We were the first to create a comprehensive and valid dataset for Bash commands from scratch, and also established a baseline accuracy of 31.6% for translating natural language to Bash commands using this dataset. The code for this research project is available on the GitHub repository [110]. The following is a summary of the key insights and findings that were gleaned from this research project:

- **It is feasible to synthesis a large dataset of Bash Commands and corresponding English pair by adopting back-translation**. Generation from scratch is a major milestone and provides significant

advantages over prior augmentation strategies. Our approach provided new opportunities for generating additional datasets of natural language to computer code translations and enhancing the performance of machine translation for Bash command

- **To make translated commands practical they must be executable, therefore validity testing is important**. The conversion from a Bash Command template to an executable (or nearly executable) command shows both progress and promise of the usability and practicality of our translation pipeline. A more complete and streamlined process of converting natural language to valid, executable commands will become a larger focus as model accuracy continues to improve.

- **It is necessary to establish a hold-out dataset[1] to evaluate the generalizability of the model.** Despite our dataset being six times larger and more diverse than the original dataset, the model performed well on the test set but failed to generalize to the original dataset and vice versa. This highlights that the current datasets of Natural Language to Bash Commands are still relatively small and insufficiently diverse to create robust models that generalize well to hold-out sets. This outcome can inform further developments and testing of different models to optimize performance on both our dataset and the original dataset.

[1] A hold-out dataset is one that is sourced differently from the original data, making it more challenging in comparison to a test set, which is sourced in the same manner as the training data and is likely to have a similar distribution.

CHAPTER 5

Efficient training: A Methodology for Deep Learning Models on CPUs

5.1 Problem Overview

The utilization of GPUs for the training of deep learning models has become prevalent due to their highly parallelized architecture. Consequently, research on optimization techniques for training primarily centers on GPU-based systems. However, it is important to consider the trade-off between cost and efficiency when determining the appropriate hardware for training. Utilizing CPU servers may be a viable option, as they may result in greater efficiency and entail lower costs for hardware updates, while also effectively utilizing existing infrastructure.

This chapter presents three contributions to the field of training deep learning models using CPUs. Firstly, the chapter describes a method for optimizing the training of deep learning models on Intel CPUs, along with the development of a toolkit called ProfileDNN, which is designed to enhance performance profiling. Secondly, the chapter introduces a generic training optimization method that guides the workflow and examines various case studies in which performance issues were identified and resolved through the optimization of the Intel® Extension for PyTorch, resulting in a 2x improvement in training performance for the RetinaNet-ResNext50 model. Thirdly, the chapter demonstrates the utilization of the visualization capabilities of ProfileDNN, which facilitated the identification of bottlenecks and the creation of a custom focal loss kernel that was two times faster than the official reference PyTorch implementation.

To address the challenges of portability associated with the deployment of deep learning (DL) models across different hardware platforms, Intel has open-sourced the oneAPI Deep Neural Network Library (oneDNN)[111]. OneDNN is a cross-platform performance library of basic deep learning primitive operations and includes a benchmarking tool called *benchDNN*. Additionally, Intel has developed optimized versions of popular frameworks, such as Intel® Optimizations for TensorFlow and Intel® Extensions for PyTorch[112], utilizing oneDNN. Despite these efforts, few guidelines exist guidelines for profiling and optimizing DL model training on CPUs.

Several fundamental research challenges must be addressed when training DL models on CPUs, including the following:

1. **Identifying performance bottlenecks**. Frameworks with CPU-optimized kernels, such as Intel® Extension for PyTorch, are relatively new, and generic model-level profilers [113] , such as the PyTorch Profiler [25], are not oneDNN-aware. Furthermore, low-level profilers, such as *benchDNN*, can only

benchmark performance at the operational level. Identifying the primitive operations that are most critical for a specific model/framework/hardware combination is essential so that low-level optimizations, such as those provided by oneDNN, can significantly accelerate performance.

2. **Addressing performance bottlenecks**. While GPUs have well-established platforms, such as CUDA, for kernel implementations, libraries for CPUs are less well-known. It is therefore crucial to understand how to rectify performance bottlenecks, such as by locating and implementing custom operation kernels for both forward and backward propagation and by adopting proper low-precision training to reduce computing time without sacrificing accuracy for CPUs.

3. **Establishing achievable goals**. Projections for CPUs are often made in a crude way by dividing CPU performance in FLOPs over FLOPs required for model training. In a computation-bounded scenario, however, it is essential to create an experiment-based projection for DL models so that the goal is realistically achievable, taking into account hardware limits and kernel optimizations.

In order to address the challenges present in training deep learning (DL) models on CPUs, we have developed a structured, top-down method for prioritizing optimization options. Utilizing this approach, we have also created a DL performance profiling toolkit, called ProfileDNN, which is specifically designed to be compatible with oneDNN and supports both profiling and projection at the model level. This toolkit serves as a bridge between oneDNN-specific model-level projection and the optimization process. An example of a model that can be optimized using our method is RetinaNet [114].

The present study is organized as follows: In Section 5.2.1, we provide an overview of various profile tools and their significance in identifying performance bottlenecks and inconsistencies. Section 5.2.2 elaborates on this subject further. In Section 5.2.3, we describe the objectives and procedures of projection, as well as the architecture and workflow of ProfileDNN. Section 5.2.4 to Section 5.2.8 discuss strategies and techniques for achieving efficient training while maintaining accuracy. The impact of distributed training on efficiency and convergence is analyzed in Section 5.3. Finally, in Section 5.4, we provide concluding remarks and outline our future research. All the experiments reported in this paper were conducted on Intel Xeon Cooper Lake processors.

5.2 Method Summary

In this section, we present the methodological component of our contribution for optimizing training of deep learning (DL) models on CPUs. Our objective is to establish a structured framework for users to optimize the training of DL models on CPUs. The method we propose adopts a top-down approach, similar to that described in [115], which aims to efficiently and effectively identify critical performance bottlenecks.

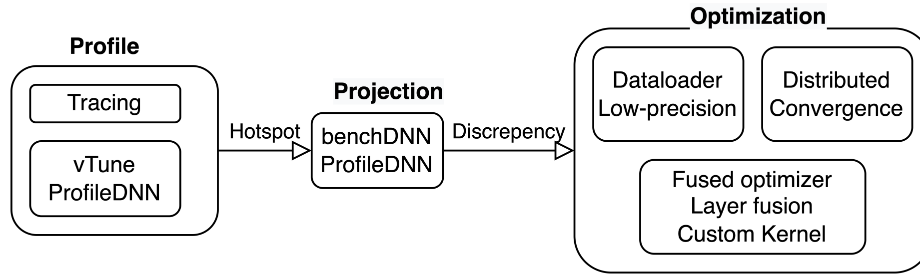


Figure 5.1: DL Workflow Method Decomposition

In our experience, DL workflows can be divided into three stages: profiling, projection, and optimization. Figure 5.1 illustrates how each stage can be broken down into various sub-components. Users are advised to follow the stages in the order presented, as each stage builds upon the results of the previous stage. Our toolkit, ProfileDNN, can function as both a profiling tool and a projection tool. Framework-level profilers, such as Tensorflow Profiler and Torch Profiler, are popular tools in part due to their hardware independence (they work on any hardware that supports Pytorch or Tensorflow), however, they lack support for executing low-level primitive kernel operations, which are crucial for performance projection. Low-level profiling/projection tools, on the other hand, can measure kernel execution time, and are traditionally hardware-specific. For example, Deep Learning Profiler (DLProf) developed by Nvidia maps the correlation between profile timing, kernel information and a Deep Learning model; ZenDNN developed by AMD support CPU profiling; BenchDNN developed by Intel went one step further by supporting primitive operation benchmarking, and thus can potentially be adapted into a projection tool. Our ProfileDNN, as shown in Table 5.1, has support for both high-level profiling and low-level (kernel) projection, and thus can act as a bridge between framework and kernel operations, thereby enabling execution-based DL model performance projection.

Table 5.1: Comparison of DL Profiling Tools

	ProfileDNN	**BenchDNN**	**TF Profiler**	**DLProf**	**ZenDNN**
Developer	Ours	Intel	Google	Nvidia	AMD
Devices Support	OneDNN hardware	CPU	CPU / GPU / TPU	GPU	CPU
Result Format	Chart / Log / Table	Log / Table	UI / Log	UI / Log	Log
Mode	Observe / Execution	Execution	Observe	Observe	Observe
Kernel Level	High / Low	Low	High	Low	Low

5.2.1 Profile and Tracing

During the profiling stage, it is important for users to examine the breakdown of operation kernel components of the DL model and evaluate their relative significance. Particular attention should be paid to any discrepancies between the user's model and data versus the reference implementation and intended use case. For

instance, it is important to verify if all the major kernel operations of the reference model are present in the user's model, and whether the percentage of kernel components remains roughly the same. If the answer to either question is negative, it may indicate that the code may perform poorly due to inadequate adoption of oneDNN kernel. ProfileDNN facilitates the comparison of kernel component distribution through the use of intuitive visualizations. This approach is similar to the one adopted by vTune [116]. ProfileDNN supports all primitive kernels, such as convolution, pooling, matrix multiplication, reordering, etc, from benchDNN.

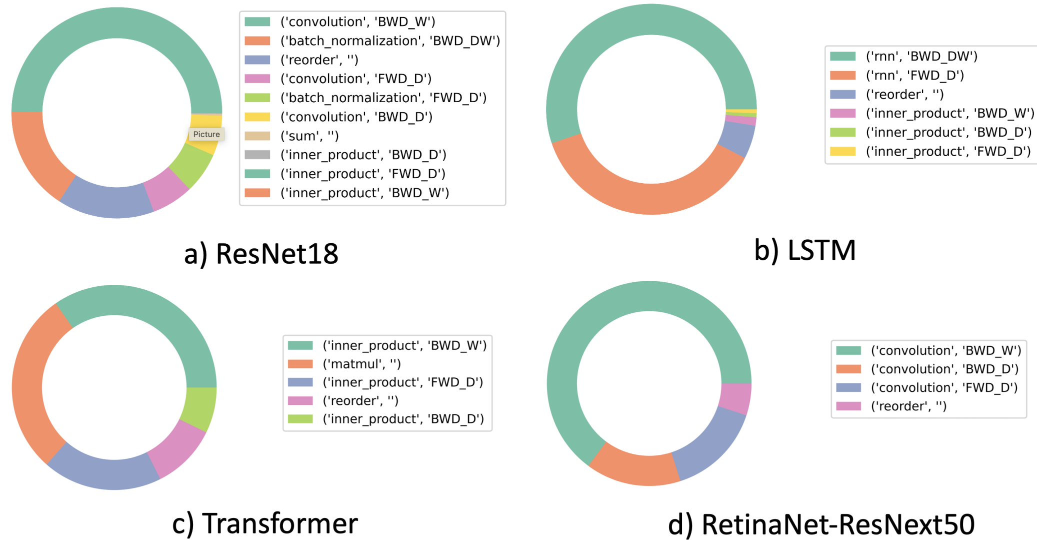


Figure 5.2: Comparison of Primitive Operations Across Models

Convolutional Neural Networks (CNNs)[117], Recurrent Neural Networks (RNNs)[118], and Transformers [119] are some of the most widely used Neural Network models today. ProfileDNN can provide a breakdown of primitive operations by type and direction, as illustrated in Figure 5.2a-c. We have found that both CNN and RNN models spend a greater proportion of their time on back-propagation than forward-propagation. Transformer models are primarily composed of inner product and matrix multiplication operations, which correspond to the `softmax` operation that is often a performance bottleneck for transformer-based models [120]. Figure 5.2d also shows the breakdown of the RetinaNet-ResNext50 model, which is a complex object detection model whose distribution is similar to that of the CNN in Figure 5.2a.

A breakdown of primitive operations is often sufficient for identifying performance bottlenecks in DL training tasks as many of them are computation-bound. However, in scenarios where memory or cache utilization is the limiting factor, trace analysis is required to examine the sequence in which operations are executed. A trace is a ordered set of span sequences, where each span includes an operation name, start and end timestamps, and relations to other spans (such as child processes, etc.). If a trace is highly fragmented, it suggests a high degree of context switching, which can be addressed by using custom merged operators to

improve performance.

Frontend Bound		Backend Bound		Speculation &Retiring
Latency	BandWidth	Core	Memory	

Figure 5.3: The vTune Design

VTune is another powerful tool for profiling CPU performance based on a top-down approach [115]. vTune divides the CPU workflow pipeline into frontend and backend, with the former being constrained by latency and bandwidth, and the latter being constrained by core (computation) and memory (cache), as illustrated in Figure 5.3. The initial step in profiling should be a generic hotspot analysis on the model training process to identify the most computationally expensive operations. The profiling process can then be followed by micro-architecture exploration to measure CPU utilization rate (spinning time), memory bandwidth, and cache (L1, L2, or L3) miss rate. Once the primitive operation with the heaviest computational footprint has been identified, algorithm- or implementation-level optimizations can be applied. If memory utilization is identified as the bottleneck, memory access and IO analysis can also be performed on individual operations.

5.2.2 Data Discrepancy

A commonly overlooked discrepancy is the difference between the reference dataset and the custom dataset used. The data distribution can not only impact the performance of the same model but it can also affect the structure of the model itself. For instance, RetinaNet-ResNext50 is a classification model that alters its structure based on the number of classes in the dataset.

After we switched the dataset from COCO [121] to OpenImage [122], the training time increased significantly. We observed that while the dataset size increased by a factor of 10, the training time per epoch increased by a factor of 20, which is not proportional. This increase can be partly attributed to a larger fully-connected (FC) layer in the backbone. Specifically, we found that the major contributor to the increased time was the focal loss calculation caused by the threefold increase in the number of classes, as demonstrated in the detailed breakdown in Figure 5.4. Our findings were further supported by trace analysis, which revealed that approximately one-third of the backward calculation time was spent on the focal loss calculation. To address this issue, we implemented a custom focal loss kernel as discussed in Section 6.4.

5.2.3 Projection and Toolkit structure

The projection of DL models is a process that aims to determine the theoretical performance ceiling of a specific model/framework/hardware combination. Intel has an internal tool that can perform projection for DL models, but it currently requires extensive manual configuration and tuning. *BenchDNN* can be used to

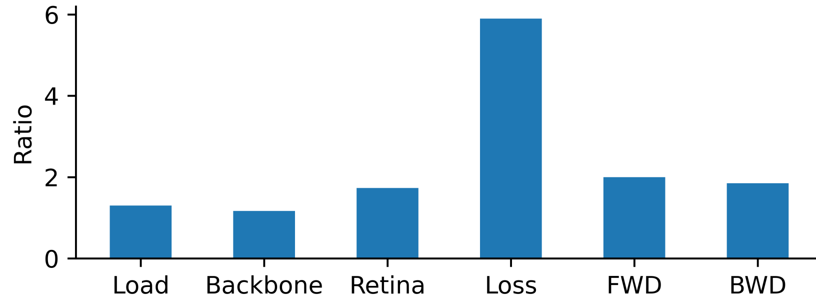


Figure 5.4: Open Image vs COCO Training Time Ratio Breakdown

predict performance on specific hardware automatically, but it can only perform predictions for individual operations at a time. In light of this, we designed ProfileDNN to combine the benefits of both existing tools by allowing for predictions for the entire DL model with minimal manual effort.

As depicted in Figure 5.5, ProfileDNN accepts an arbitrary log file generated by running deep learning models on a platform that supports oneDNN with `DNNL_VERBOSE` set to `1`. The *stats.py* script then processes the raw log file by collecting and cleaning it into CSV format, generating a template parameter file, calculating, and plotting the distribution of primitive operation components. The *benchDNN.sh* script runs each primitive operation multiple times and calculates the average. The *efficiency.py* script subsequently takes a weighted sum of the execution time of all operations based on the number of calls, and produces an efficiency ratio number.

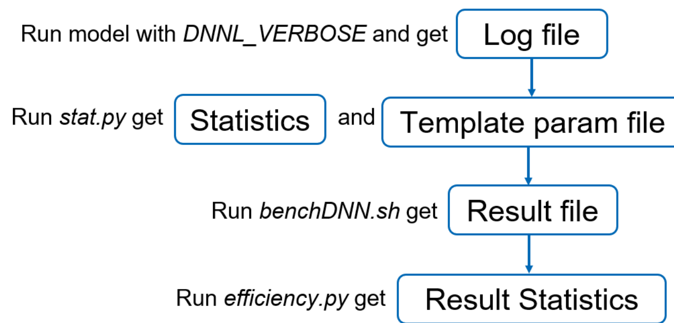


Figure 5.5: Toolkit Structure and Flow Pipeline

In order to ensure the accuracy of our toolkit in reproducing the behavior of the kernels from the original model, we employ a method that involves controlling both the computation resources and problem descriptions to ensure they are identical. To achieve this, we utilize the tool *numactl* to manage the number of CPU cores and memory binding. Additionally, we set the mode to *p (performance)* in *benchDNN* to optimize performance. These parameters are carefully monitored and documented in Table 5.2 for reference.

Name	Example
Driver	conv, relu, matmul, rnn, bnorm
Configuration	u8s8u8, s8f32
Directory	FWD_I, BWD_D, BWD_W
Post_Ops	sum+eltwise_relu
Algorithm	DIRECT
Problem_batchsize	mb1, mb32
Problem_input	id4ih32iw32
Problem_output	id16ih16iw16
Problem_stride	sd2sh4sw4
Problem_kernel	kd2kh3kw3
Problem_padding	pd1ph1pw1
Problem_channel	ic16oc32

Table 5.2: Summary of benchDNN Parameters

5.2.4 Dataloader and Memory Layout

By analyzing the DL training process from the same vTune top-down perspective shown in Fig 5.3, the dataloader can be seen as a frontend bounded by bandwidth and latency. Through examination, we identified three primary sources of bottlenecks for the data loader: input/output (I/O), decoding, and preprocessing. Our analysis revealed that there was minimal difference in performance between data stored in NVMe or loaded into RAM, indicating that I/O overhead is negligible. Additionally, we found that adopting Pillow-SIMD and accimage as the backend in torchvision resulted in improved decoding performance.

A PyTorch dataloader parameter controls the number of worker processes, which are usually set to prevent blocking the main process when training on GPUs. For training on CPUs, however, this number should not be set to minimize memory overhead. This is because the RAM memory on CPUs, while generally larger than that of GPUs, has a smaller bandwidth. As a result, training on CPUs has the advantage of allowing for larger batch sizes and the ability to train larger models as reported in [30].

Here we define the variables n as the batch size, c as the number of channels, h as the height, and w as the width. The recommended memory layout in the Intel® Extension for PyTorch is *nhwc* (channel last) for more efficient training, though the default layout in *benchDNN* is *nchw*. To align with established best practices, the default behavior of ProfileDNN is set to adopt the *nchw* layout. If the log input specifies a different memory layout, ProfileDNN will automatically override the default setting.

5.2.5 Library Optimization

We observed significant performance improvements by replacing slow operation implementations with more efficient libraries, as demonstrated by substituting the official PyTorch implementation with its counterpart in the Intel® Extension for PyTorch. Through the use of ProfileDNN, we were able to identify a discrepancy

in the number of backward convolution calls between the official PyTorch library and the Intel® Extension for PyTorch. By conducting a detailed analysis of the computation graph and utilizing our ProfileDNN-based visualization, we discovered that these calls originated from the frozen layers in the pre-trained model (ResNext backbone).

Our analysis led to a significant improvement in the performance of the RetinaNet-ResNext50 model training with 2 fixed layers, resulting in a 16% increase. We also identified that the primitive operation `frozenbachnorm2d` was missing in Figure 5.2d, and that the operation `torchvision.ops.misc.FrozenBatchNorm2d` was being interpreted as separate `mul` and `add` operations, which meant it was not a single oneDNN kernel operation. Our analysis revealed that bandwidth-limited operations made the `torchvision.ops.misc.FrozenBatchNorm2d` operation inefficient and unable to be fused with other operations to reduce memory accesses. Training performance was further improved by 29.8% after we replaced the `torchvision.ops.misc.FrozenBatchNorm2d` operation with `IPEX.nn.FrozenBatchNorm2d`.

5.2.6 Low-precision Training

Low-precision training has proven to be a highly effective method for high-performance computing and BF16 (Brain Floating Point) is widely supported by various DL hardware platforms. BF16 is unique in that it has the same range as float32, but uses fewer bits to represent the fraction (7 bits). This characteristic of BF16 can be beneficial in situations where computation speed is a priority, but can also result in a loss of accuracy when compared to float32 in the calculation of loss. As illustrated in Figure 5.6, the computation time is nearly halved when using BF16 compared to float32. (Note that the improvements plotted are relative and not absolute, in compliance with Intel's data policy).

Our analysis revealed a substantial discrepancy between the forward/backward training time ratio in comparison to the bare-bone kernel time, which indicates highly inefficient non-kernel code in the forward pass. Further examination revealed that the loss function does not scale well and comprises a significant portion of computation time. By identifying the focal loss as having significant overhead, we implemented our version of the focal loss kernel, as described in Section 5.2.8. However, we observed a difference in loss results compared to the original implementation. Through further investigation, we identified that the loss of accuracy occurred during low-precision casting to BF16 by the `torch.cpu.amp.autocast`. Therefore, unless convergence can be guaranteed, it is advisable to avoid casting data into BF16 for loss calculation, particularly when reduction operations are involved.

5.2.7 Layer Fusion and Optimizer Fusion

In inference mode, certain layers can be fused for a forward pass in order to save cache copying operations, as intermediate results are not needed. However, in training mode, the layers containing trainable weights must save the intermediate results for backpropagation. When oneDNN is running in inference mode, it enables the fusion of `batchnorm+relu` and `conv+relu` respectively, but not the fusion of `frozenbatchnorm (FBN)+relu`. OneDNN already supports the use of `eltwise (linear, relu)` post-operations for `conv` and chaining of post-operations. As a result, we treat `FBN` as a per-channel linear operation to enable the fusion of `conv+FBN+relu`. This fusion has the potential to increase performance by 30% and is currently work-in-progress (WIP).

Modern deep learning frameworks typically support automatic differentiation and modularity of deep learning building blocks, which simplify the creation of deep learning models by lowering the entry barrier. However, it is common knowledge in software development that there exists a trade-off between modularity and performance. As noted by Jiang et al. in 2021 [123], eager execution, which executes forward propagation, gradient calculation, and parameter updating in serialized stages, may negatively impact model performance. In contrast, optimizer fusion aims to improve locality and parallelism by reordering these procedures. The Intel® Extension for PyTorch currently supports the fusion of the SGD [124] and Lamb [125] optimizers, in part by fusing operations and separating the gradients, parameters, and intermediates into small groups for improved caching mechanism. We conducted an experiment comparing a fused and unfused Lamb optimizer with RetinaNet and found a 5.5X reduction in parameter updating time when the optimizer was fused.

5.2.8 Custom Operation Kernel

Custom operation kernels play a crucial role in optimizing performance by eliminating computation overhead, such as unnecessary copying and intermediates. It is essential that these kernel implementations are mathematically equivalent to the reference code. Moreover, these custom kernels have the potential to show significant performance gains in most or all circumstances, as discussed in the following section.

5.2.8.1 Theoretical deduction

Instead of utilizing the PyTorch implementation (Appendix 6.2) for the forward pass of the focal loss and relying on the default generated backward pass, we implemented a custom kernel for both the forward and backward pass (the backward kernel implementation is optional, as implicit autograd can be generated). The focal loss can be represented mathematically as shown in Equation 5.1 and we adopt $\gamma = 2$ and $\alpha = 0.25$ in our implementation. The forward pass can be further simplified by assuming that x and y are real numbers in Equation 5.2. Additionally, since y is a binary matrix, all terms that contain `y(y-1)` are equal to `0` and can

be removed as shown in Equation 5.3. The backward equation is presented in Appendix 6.3.

$$FL(p) = \begin{cases} -\alpha(1-p)^{\gamma}\log(p), & y = 1 \\ -(1-\alpha)p^{\gamma}\log(1-p), & \text{otherwise} \end{cases} \tag{5.1}$$

$$FL = (a(2y-1)-y+1)\left(\frac{-e^{x}y+e^{x}+y}{e^{x}+1}\right)^{\gamma}(\log(e^{x}+1)-xy) \tag{5.2}$$

$$FL_{sp} = \left(\frac{-e^{x}y+e^{x}+y}{e^{x}+1}\right)^{\gamma}((\alpha(2y-1)-y+1)\log(e^{x}+1)-\alpha xy) \tag{5.3}$$

5.2.8.2 Implementation and Assessment

The operators in ATEN of PyTorch can be broadly classified into two categories: in-place operations and standard operations. In-place operations are denoted by a suffix of _, such as in the case of *add_*. Since in-place operations modify the Tensor directly, the overhead of copying or creating new spaces in the cache is eliminated. As observed in the implementation presented in Appendix 6.4, we have made extensive use of in-place operations to enhance efficiency.

After verifying that our kernel implementation is mathematically equivalent to the reference implementation, we conducted an experimental evaluation of our kernel against the reference code using both float32 and BF16 settings. As illustrated in Figure 5.6, the custom forward kernel demonstrated a 2.6-fold improvement in performance over the default implementation under the BF16 setting.

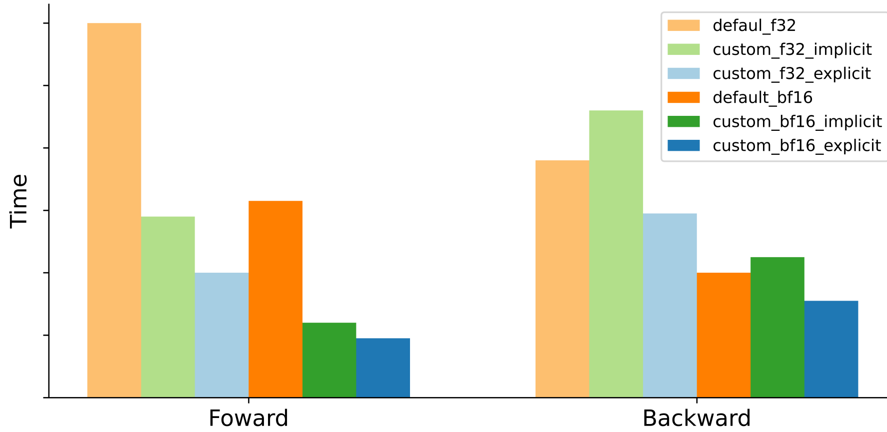


Figure 5.6: Comparison of Custom Focal Loss Time vs Default

Although the PyTorch framework can generate implicit autograd for our custom kernel, its performance is suboptimal. Our custom backward kernel demonstrated a 1.3-fold improvement in performance over the reference implementation and a 1.45-fold improvement over the generated implicit autograd kernel. Addi-

tionally, we observed that the custom backward kernel can enhance the performance of the forward kernel and we suspect that the explicit backward kernel prevents the forward kernel from retaining unnecessary intermediates. The overall improvement from the custom focal loss kernel is a two-fold increase in performance. Our code has been integrated into the Intel® Extension for PyTorch and will be available in that library in the near future.

5.3 Distributed Training

In contrast to inference, which can be scaled out among independent nodes, training deep learning models often requires more powerful computing resources that work in synchronization. This requirement can be met by scaling up nodes with additional CPU resources or by scaling out among multiple nodes. When training a system at scale, whether it involves multiple nodes, multiple sockets, or even a single socket, it is necessary to distribute the workload across multiple workers. Coordination among distributed workers necessitates communication between them. The distribution of workloads on CPUs can be achieved through various protocols and middleware, such as MPI (Message Passing Interface)[126] and Gloo [127]. In the subsequent sections, we will use MPI terminology.

5.3.1 Distributed Training Performance

To achieve optimal training performance, a training workload should aim to utilize one thread per CPU core of each system node. For instance, an 8-socket system with 28 cores per socket should target 224 total threads. The total threads can be divided among several workers identified by their *rank*, such as 8 ranks of 28 threads, 16 ranks of 14 threads, or 32 ranks of 8 threads, etc. It is important to note that the selection of ranks and threads should not result in any rank spanning multiple sockets.

In practice, it is observed that better performance can be achieved by using more ranks with fewer threads each, as opposed to fewer ranks with more threads each, at the same global batch size. Table 5.3 illustrates how throughput increases diagonally from the bottom-left to the top-right. However, the number of available ranks is constrained by the available system memory, model size, and batch size. Since system memory is divided among the ranks, each rank must have sufficient memory to support the model and host functions to avoid workload failure.

	Number of Workers				
Threads/Worker	**1**	**2**	**4**	**8**	**16**
7	1.00	2.00	3.82	7.04	11.87
14	1.86	3.7	6.8	11.59	-
28	3.27	6.51	11.20	-	-
56	5.11	10.18	-	-	-

Table 5.3: Scalability (Normalized Throughput)

While high-end CPUs can already perform comparably with their GPU counterparts (e.g. Intel 4th Gen Xeon processors were able to train a ResNet-50 model in under 90 minutes [128]), the real advantage lies in the democratization of DL model training for individuals or companies without access to GPUs or those with existing CPU clusters and limited budgets. The inefficiency of CPU training is largely attributed to limited bandwidth, but this can be addressed by utilizing better software optimization (such as Intel® Extension for PyTorch) and low-level kernel support (such as oneDNN) to break and group operations into more manageable chunks for improved caching. These optimizations can lead to significant performance improvements, as we observed a 2X performance boost with Intel® Extension for PyTorch compared to the default PyTorch. Another potential platform for training is AI accelerators, as the latest MLPerf benchmark suggests that the Gaudi2 processor has 2X the throughput of the A100 on ResNet-50 and BERT [129].

5.3.2 Training Convergence

As a training system is scaled-out to more nodes, sockets, or ranks, two factors that have a negative impact on the model's convergence time are weak scaling efficiency and convergence point. Weak scaling efficiency is defined as the ratio of the performance of a system to N systems doing N times as much work, and tends to fall behind the linear rate at which resources are added. This phenomenon and its underlying causes have been widely studied [130] across different hardware types and will not be further explored in this paper.

A model's convergence point is the second factor that affects convergence time as a training system scales. Specifically, as a distributed system scales out, the global batch size increases, even though the local batch size per worker remains constant. For example, if a 2-socket system launches a combined 8 ranks with a global batch size of 64 (BS=8 per rank), when scaled out to 8-sockets, the global batch size becomes 256, despite the fact that each rank has the same local batch size.

As the number of epochs required to converge to a model's target accuracy increases, the global batch size of a training workload also increases, as shown in Fig 5.7. This increase in the number of epochs to reach a convergence point can have a significant negative impact on the benefits of increased resources. When planning a system scale-out, it is therefore crucial to take into account the resulting convergence point and attempt to mitigate it by reducing the local batch size if possible [131].

5.4 Conclusion

In this chapter, we investigate various techniques for optimizing the training of deep learning (DL) models on CPUs, along with a comprehensive method guide. We present a DL profiling and projection toolkit called ProfileDNN that assisted in identifying several issues related to the training of RetinaNet-ResNext50. The resolution of these issues led to a significant improvement in efficiency, increasing performance by a factor

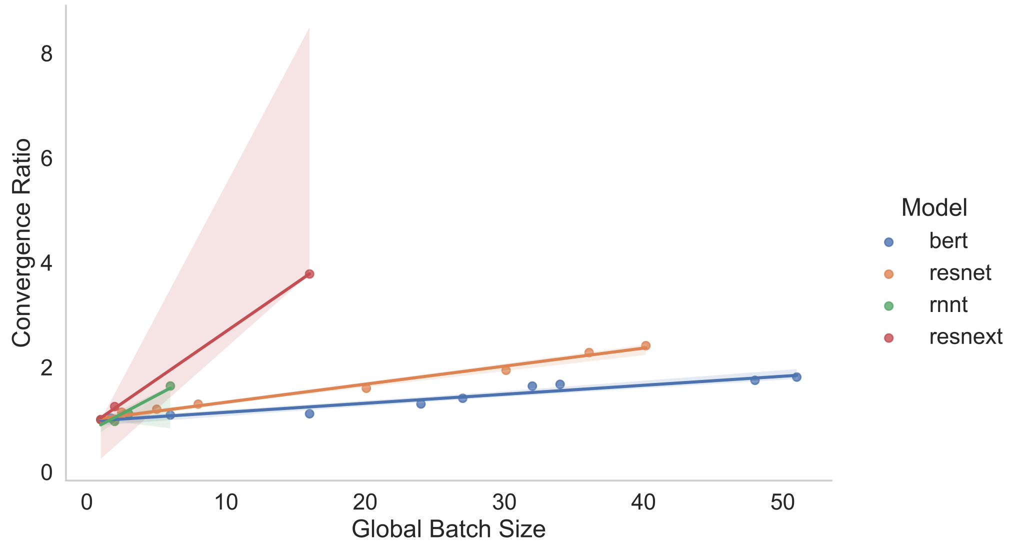


Figure 5.7: Convergence Ratio vs Global Batch Size (Normalized)

of two. Additionally, we developed a custom Focal Loss kernel that demonstrated a 1.5 times improvement in performance over the PyTorch reference implementation when executed on CPUs.

The results of our study on the training of deep learning models using CPUs have yielded several key insights. Specifically, the utilization of efficient deep learning frameworks optimized for CPUs, such as the Intel extension for PyTorch, can significantly reduce training time with minimal additional cost. Additionally, it is crucial to conduct model profiling on both reference code and custom implementations, particularly when changes to the data set have been made. This analysis can aid in identifying discrepancies between different implementations and the corresponding low-level operation distributions, which can serve as indicators of potential bottlenecks. Furthermore, it has been found that implementing both the forward pass and backward pass explicitly for custom kernels can lead to the most optimal training performance. Lastly, our research has demonstrated that there is a strong correlation between local batch size and the convergence point, and thus it is essential to properly reduce batch size when scaling out the system.

Ammar et al.[132] have demonstrated that software libraries have a greater impact on the performance of deep learning (DL) model training compared to hardware architecture. As such, there is a need for further research in the area of hardware-software co-design. Recently, DeepMind[133] introduced a novel algorithm that utilizes reinforcement learning to optimize matrix multiplication through the simultaneous profiling and fine-tuning of both hardware and software. This hardware-dependent algorithm resulted in a significant increase in performance and could not have been discovered through traditional algorithm analysis. This approach could also be applied to discover more hardware-specific efficient kernel implementations.

As highlighted by Sparsh et al. [134], basing hardware comparisons for DL model training solely on Time-to-Train (ToT) metrics may not provide a comprehensive understanding of the system. Other factors such as energy efficiency, throughput, and latency must also be considered when selecting a system for DL training.

In future work, our research will focus on testing our proposed method and ProfileDNN toolkit on other popular models and conducting a more detailed study on optimizing the training of DL models with distributed CPU clusters. Additionally, we will work towards improving MLperf by incorporating more comprehensive metrics for DL model training.

CHAPTER 6

Appendix

6.1 Summary of Publications

1. Quchen Fu, Szu-Wei Fu, Yaran Fan, Yu Wu, Zhuo Chen, Jayant Gupchup, Ross Cutler, Real-time Speech Interruption Analysis: From Cloud to Client Deployment, ICASSP 2023

2. Quchen Fu, Zhongwei Teng, Jules White, Maria Powell, and Douglas Schmidt, FastAudio: A Learnable Audio Frontend for Spoof Speech Detection, ICASSP 2022

3. Quchen Fu, Zhongwei Teng, Marco Georgaklis, Jules White and Douglas Schmidt, NL2CMD: An Updated Workflow for Natural Language to Bash Commands Translation, JMLTAP

4. Quchen Fu, Ramesh Chukka, Keith Achorn, Thomas Atta-fosu, Deepak R. Canchi, Zhongwei Teng, Jules White, Douglas C. Schmidt, Deep Learning Models on CPUs: A Methodology for Efficient Training

5. Quchen Fu, Zhongwei Teng, Jules White and Douglas Schmidt, A Transformer-based Approach for Translating Natural Language to Bash Commands, ICMLA 2021

6. Zhongwei Teng, Quchen Fu, Jules White, Maria Powell, and Douglas C. Schmidt, SA-SASV: An End-to-End Spoof-Aggregated Spoofing-Aware Speaker Verification System. INTERSPEECH 2022

7. Zhongwei Teng, Quchen Fu, Jules White, Maria Powell, and Douglas C. Schmidt, Complementing Handcrafted Features with Raw Waveform Using a Light-weight Auxiliary Model. ICPR 2022

8. Zhongwei Teng, Quchen Fu, Jules White, and Douglas C. Schmidt, Sketch2Vis: Generating Data Visualizations from Hand-drawn Sketches with Deep Learning. ICMLA 2021

9. Agarwal, Mayank, Tathagata Chakraborti, Quchen Fu, David Gros, Xi Victoria Lin, Jaron Maene, Kartik Talamadupula, Zhongwei Teng, and Jules White. Neurips 2020 nlc2cmd competition: Translating natural language to bash commands. In NeurIPS 2020 Competition and Demonstration Track, pp. 302-324. PMLR, 2021.

10. Yu Yao, Maria Powell, Jules White, Jian Feng, Quchen Fu, Peng Zhang, and Douglas C. Schmidt. An Exploration of Rare Voice Disorder Diagnosis with Multi-stage Transfer Learning.

6.2 Reference Focal Loss Code [1]

```python
import torch
import torch.nn.functional as F
import time
def sigmoid_focal_loss(
    inputs: torch.Tensor,
    targets: torch.Tensor,
    alpha: float = 0.25,
    gamma: float = 2,
    reduction: str = "none",
):
    inputs = inputs.to(dtype=torch.float32)
    targets = targets.to(dtype=torch.float32)
    p = torch.sigmoid(inputs)
    ce_loss = F.binary_cross_entropy_with_logits(
        inputs, targets, reduction="none"
    )
    p_t = p * targets + (1 - p) * (1 - targets)
    loss = ce_loss * ((1 - p_t) ** gamma)
    if alpha >= 0:
        alpha_t = alpha * targets + (1 - alpha) * (1 - targets)
        loss = alpha_t * loss
    if reduction == "mean":
        loss = loss.mean()
    elif reduction == "sum":
        loss = loss.sum()
    return loss
```

6.3 Focal Loss Derivative

$$
\begin{gathered}
\frac{\partial}{\partial x}\left(\left(1-\left(y\times\frac{1}{1+e^{-x}}+\left(1-\frac{1}{1+e^{-x}}\right)(1-y)\right)\right)^{\gamma}\right. \\
\left.\left(-y\log\left(\frac{1}{1+e^{-x}}\right)-(1-y)\log\left(1-\frac{1}{1+e^{-x}}\right)\right)(ay+(1-a)(1-y))\right)= \\
-\left(\left(\left(a(2y-1)-y+1\right)\left(\frac{-e^{x}y+e^{x}+y}{e^{x}+1}\right)^{\gamma}\right.\right. \\
\left(e^{2x}y^{2}-\gamma e^{x}\left(2y^{2}-3y+1\right)\log\left(\frac{1}{e^{x}+1}\right)-2e^{2x}y+\gamma e^{x}(2y-1)\right. \\
\left.\left.\left.y\log\left(\frac{1}{e^{-x}+1}\right)+e^{2x}-y^{2}\right)\right)/((e^{x}+1)(e^{x}(y-1)-y))\right)
\end{gathered}
$$

Figure 6.1: Backward Kernel Equation

6.4 Custom Focal Loss Kernel Code

$$
\begin{aligned}
&-(e^x+1)^{-\gamma-1}(y-e^x(y-1))^{\gamma-1} \\
&\quad \left(-\alpha\gamma e^x xy+\gamma e^x \log(e^x+1)(\alpha+y-1)+(\alpha-1)e^{2x}(1-y)+\alpha y\right)
\end{aligned}
$$

Figure 6.2: Simplified Backward Kernel

```
at::Tensor _focal_loss_forward(const at::Tensor& input, const at::Tensor& target, const
    float alpha, const float gamma, const int64_t reduction) {
    at::Tensor loss;
    loss=(((alpha*(-input).mul_(target)).add_(((2*alpha-1)*target+(1-alpha)).mul_(((input.
    exp_() + 1).log_()))))).mul_(((target -1).mul_(input).add_(-target)).pow_(gamma))).div_
    ((input + 1).pow_(gamma));
    return apply_loss_reduction(loss, reduction);
}

at::Tensor _focal_loss_backward(const at::Tensor& grad, const at::Tensor& input, const at
    ::Tensor& target, const float alpha, const float gamma, const int64_t reduction) {
    at::Tensor grad_input;
    grad_input=-((input.exp() + 1).pow(-gamma-1)).mul((target.add((1-target).mul(input.exp
    ()))).pow(gamma - 1)).mul(((-alpha*gamma*input).mul(target).mul(input.exp())).add(
    gamma*(target+alpha-1).mul(input.exp()).mul(((input.exp()+1).log()))).add(alpha*target
    ).add((alpha-1)*(1-target).mul((input.exp()).pow(2))))).mul(grad);
    if (reduction == at::Reduction::Mean) {
        return grad_input / input.numel();
    }
    return grad_input;
}
```

References

[1] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1485–1488, 2010.

[2] Harishchandra Dubey, Vishak Gopal, Ross Cutler, Ashkan Aazami, Sergiy Matusevych, Sebastian Braun, Sefik Emre Eskimez, Manthan Thakker, Takuya Yoshioka, Hannes Gamper, et al. Icassp 2022 deep noise suppression challenge. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 9271–9275. IEEE, 2022.

[3] Ross Cutler, Ando Saabas, Tanel Parnamaa, Marju Purin, Hannes Gamper, Sebastian Braun, Karsten Sørensen, and Robert Aichner. Icassp 2022 acoustic echo cancellation challenge. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 9107–9111. IEEE, 2022.

[4] Lorenz Diener, Sten Sootla, Solomiya Branets, Ando Saabas, Robert Aichner, and Ross Cutler. Interspeech 2022 audio deep packet loss concealment challenge. *arXiv preprint arXiv:2204.05222*, 2022.

[5] Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. Superb: Speech processing universal performance benchmark. *arXiv preprint arXiv:2105.01051*, pages 1194–1198, 2021.

[6] Bret Kinsella. U.s. smart speaker growth flat lined in 2020, 2021.

[7] Jonathan Shen, Ruoming Pang, Ron J. Weiss, M. Schuster, Navdeep Jaitly, Zongheng Yang, Z. Chen, Yu Zhang, Yuxuan Wang, R. Skerry-Ryan, R. Saurous, Yannis Agiomyrgiannakis, and Yonghui Wu. Natural tts synthesis by conditioning wavenet on mel spectrogram predictions. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4779–4783, 2018.

[8] Neil Zeghidour, O. Teboul, F. D. C. Quitry, and M. Tagliasacchi. Leaf: A learnable frontend for audio classification. *ArXiv*, abs/2101.08596, 2021.

[9] Ross Cutler, Yasaman Hosseinkashi, Jamie Pool, Senja Filipi, Robert Aichner, Yuan Tu, and Johannes Gehrke. Meeting effectiveness and inclusiveness in remote collaboration. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW1):1–29, 2021.

[10] Szu-Wei Fu, Yaran Fan, Yasaman Hosseinkashi, Jayant Gupchup, and Ross Cutler. Improving meeting inclusiveness using speech interruption analysis. In *ACM Multimedia*, pages 887–895, 2022.

[11] Xuliang Liu and H. Zhong. Mining stackoverflow for program repair. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129, 2018.

[12] Raymond J Mooney. Semantic parsing: Past, present, and future. In *Presentation slides from the ACL Workshop on Semantic Parsing*, 2014.

[13] Quchen Fu, Zhongwei Teng, Jules White, and Douglas C Schmidt. A transformer-based approach for translating natural language to bash commands. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1245–1248. IEEE, 2021.

[14] Shikhar Bharadwaj and Shirish Shevade. Explainable natural language to bash translation using abstract syntax tree. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 258–267, 2021.

[15] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *arXiv preprint arXiv:1802.08979*, 2018.

[16] Xuan-Phi Nguyen, Shafiq Joty, Kui Wu, and Ai Ti Aw. Data diversification: A simple strategy for neural machine translation. *Advances in Neural Information Processing Systems*, 33:10018–10029, 2020.

[17] Yuekai Zhao, Haoran Zhang, Shuchang Zhou, and Zhihua Zhang. Active learning approaches to enhancing neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1796–1806, Online, November 2020. Association for Computational Linguistics.

[18] Mayank Agarwal, Kartik Talamadupula, Fernando Martinez, Stephanie Houde, Michael Muller, John Richards, Steven I Ross, and Justin D Weisz. Using document similarity methods to create parallel datasets for code translation. *arXiv preprint arXiv:2110.05423*, 2021.

[19] Bash reference manual, 2020.

[20] Shikhar Bharadwaj and Shirish Shevade. Explainable natural language to bash translation using abstract syntax tree. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 258–267, Online, November 2021. Association for Computational Linguistics.

[21] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[22] Quchen Fu, Zhongwei Teng, Jules White, and Douglas C. Schmidt. A transformer-based approach for translating natural language to bash commands. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1245–1248, 2021.

[23] Quchen Fu, Zhongwei Teng, Jules White, Maria E. Powell, and Douglas C. Schmidt. Fastaudio: A learnable audio front-end for spoof speech detection. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3693–3697, 2022.

[24] Jee weon Jung, Hemlata Tak, Hye jin Shim, Hee-Soo Heo, Bong-Jin Lee, Soo-Whan Chung, Ha jin Yu, Nicholas W. D. Evans, and Tomi H. Kinnunen. Sasv 2022: The first spoofing-aware speaker verification challenge. *Interspeech 2022*, 2022.

[25] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[26] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. {TensorFlow}: A system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[27] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. Openvino deep learning workbench: Comprehensive analysis and tuning of neural networks inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[28] James R Reinders. Sycl, dpc++, xpus, oneapi. In *International Workshop on OpenCL*, pages 1–1, 2021.

[29] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing deep learning recommender systems training on cpu cluster architectures. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[30] Yu Emma Wang, Gu-Yeon Wei, and David Brooks. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[31] Ebubekir Buber and DIRI Banu. Performance analysis and cpu vs gpu comparison for deep learning. In *2018 6th International Conference on Control Engineering & Information Technology (CEIT)*, pages 1–6. IEEE, 2018.

[32] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104. IEEE, 2016.

[33] Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. In *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, pages 148–155. IEEE, 2019.

[34] Yanli Qian. *Profiling and characterization of deep learning model inference on CPU*. PhD thesis, 2020.

[35] Jiho Chang, Yoonsung Choi, Taegyoung Lee, and Junhee Cho. Reducing mac operation in convolutional neural network with sign prediction. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 177–182. IEEE, 2018.

[36] Alexander Wong. Netscore: towards universal metrics for large-scale performance analysis of deep neural networks for practical on-device edge usage. In *International Conference on Image Analysis and Recognition*, pages 15–26. Springer, 2019.

[37] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark, 2019.

[38] Nur Ahmed and Muntasir Wahed. The de-democratization of ai: Deep learning and the compute divide in artificial intelligence research. *arXiv preprint arXiv:2010.15581*, 2020.

[39] Carey Jewitt, Jeff Bezemer, and Kay O'Halloran. *Introducing multimodality*. Routledge, 2016.

[40] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[41] M. Todisco, X. Wang, Ville Vestman, Md. Sahidullah, H. Delgado, A. Nautsch, J. Yamagishi, N. Evans, T. Kinnunen, and Kong-Aik Lee. Asvspoof 2019: Future horizons in spoofed and fake audio detection. *ArXiv*, abs/1904.05441, 2019.

[42] K. Cheuk, Hans Anderson, Kat R. Agres, and Dorien Herremans. nnaudio: An on-the-fly gpu audio to spectrogram conversion toolbox using 1d convolutional neural networks. *IEEE Access*, 8:161981–162003, 2020.

[43] T. Sainath, Brian Kingsbury, Abdel rahman Mohamed, and B. Ramabhadran. Learning filter banks within a deep neural network framework. *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 297–302, 2013.

[44] Hong Yu, Z. Tan, Yiming Zhang, Zhanyu Ma, and Jun Guo. Dnn filter bank cepstral coefficients for spoofing detection. *IEEE Access*, 5:4779–4787, 2017.

[45] Teng Zhang and Ji Wu. Discriminative frequency filter banks learning with neural networks. *EURASIP Journal on Audio, Speech, and Music Processing*, 2019:1–16, 2019.

[46] David Snyder, Daniel Garcia-Romero, Gregory Sell, Daniel Povey, and Sanjeev Khudanpur. X-vectors: Robust dnn embeddings for speaker recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5329–5333. IEEE, 2018.

[47] Mirco Ravanelli, Titouan Parcollet, Peter Plantinga, Aku Rouhe, S. Cornell, Loren Lugosch, Cem Subakan, Nauman Dawalatabad, A. Heba, Jianyuan Zhong, Ju-Chieh Chou, Sung-Lin Yeh, Szu-Wei Fu, Chien-Feng Liao, Elena Rastorgueva, Franccois Grondin, William Aris, Hwidong Na, Yan Gao, R. Mori, and Yoshua Bengio. Speechbrain: A general-purpose speech toolkit. *ArXiv*, abs/2106.04624, 2021.

[48] Brecht Desplanques, Jenthe Thienpondt, and Kris Demuynck. Ecapa-tdnn: Emphasized channel attention, propagation and aggregation in tdnn based speaker verification. *arXiv preprint arXiv:2005.07143*, 2020.

[49] Neil Zeghidour, Nicolas Usunier, I. Kokkinos, Thomas Schatz, Gabriel Synnaeve, and Emmanuel Dupoux. Learning filterbanks from raw speech for phone recognition. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5509–5513, 2018.

[50] X. Xiao, Xiaohai Tian, Steven Du, Haihua Xu, Chng Eng Siong, and Haizhou Li. Spoofing speech detection using high dimensional magnitude and phase features: the ntu approach for asvspoof 2015 challenge. In *INTERSPEECH*, 2015.

[51] B. Lindblom. Explaining phonetic variation: A sketch of the h&h theory. 1990.

[52] Mirco Ravanelli and Yoshua Bengio. Speaker recognition from raw waveform with sincnet. *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 1021–1028, 2018.

[53] Xu Li, N. Li, Chao Weng, Xunying Liu, Dan Su, Dong Yu, and H. Meng. Replay and synthetic speech detection with res2net architecture. In *ICASSP*, 2021.

[54] Yeonghyeon Lee, Kangwook Jang, Jahyun Goo, Youngmoon Jung, and Hoirin Kim. Fithubert: Going thinner and deeper for knowledge distillation of speech self-supervised learning. *arXiv preprint arXiv:2207.00555*, pages 3588–3592, 2022.

[55] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[56] Sanyuan Chen, Yu Wu, Zhuo Chen, Jian Wu, Takuya Yoshioka, Shujie Liu, Jinyu Li, and Xiangzhan Yu. Ultra fast speech separation model with teacher student learning. pages 3026–3030, 08 2021.

[57] Heng-Jui Chang, Shu-wen Yang, and Hung-yi Lee. Distilhubert: Speech representation learning by layer-wise distillation of hidden-unit bert. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7087–7091, 2022.

[58] Wenhui Wang, Hangbo Bao, Shaohan Huang, Li Dong, and Furu Wei. Minilmv2: Multi-head self-attention relation distillation for compressing pretrained transformers. In *FINDINGS*, 2021.

[59] Alexei Baevski, Wei-Ning Hsu, Qiantong Xu, Arun Babu, Jiatao Gu, and Michael Auli. Data2vec: A general framework for self-supervised learning in speech, vision and language. *arXiv preprint arXiv:2202.03555*, 2022.

[60] Sho Takase and Shun Kiyono. Lessons on parameter sharing across layers in transformers. *ArXiv*, abs/2104.06022, 2021.

[61] Tao Ge and Furu Wei. Edgeformer: A parameter-efficient transformer for on-device seq2seq generation. *arXiv preprint arXiv:2202.07959*, 2022.

[62] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. *ArXiv*, abs/1909.11556, 2020.

[63] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *ArXiv*, abs/2106.08295, 2021.

[64] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

[65] Rui Wang, Qibing Bai, Junyi Ao, Long Zhou, Zhixiang Xiong, Zhihua Wei, Yu Zhang, Tom Ko, and Haizhou Li. LightHuBERT: Lightweight and configurable speech representation learning with once-for-all hidden-unit BERT. In *Proceedings of the Interspeech*, 2022.

[66] Zilun Peng, Akshay Budhkar, Ilana Tuil, Jason Levy, Parinaz Sobhani, Raphael Cohen, and Jumana Nassour. Shrinking bigfoot: Reducing wav2vec 2.0 footprint. *CoRR*, abs/2103.15760, 2021.

[67] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[68] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[69] Radosvet Desislavov, Fernando Martínez-Plumed, and José Hernández-Orallo. Compute and energy consumption trends in deep learning inference. *arXiv preprint arXiv:2109.05472*, 2021.

[70] Peter Henderson, Jieru Hu, Joshua Romoff, Emma Brunskill, Dan Jurafsky, and Joelle Pineau. Towards the systematic reporting of the energy and carbon footprints of machine learning, 2020.

[71] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. *arXiv preprint arXiv:1906.02243*, 2019.

[72] Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdelrahman Mohamed. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:3451–3460, 2021.

[73] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in Neural Information Processing Systems*, 33:12449–12460, 2020.

[74] Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, et al. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 2022.

[75] Sree Hari Krishnan Parthasarathi and Nikko Strom. Lessons from building acoustic models with a million hours of speech. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6670–6674, 2019.

[76] Wei-Cheng Tseng, Chien-yu Huang, Wei-Tsung Kao, Yist Y Lin, and Hung-yi Lee. Utilizing self-supervised representations for mos prediction. *arXiv preprint arXiv:2104.03017*, pages 2781–2785, 2021.

[77] Reza Yazdani Aminabadi, Samyam Rajbhandari, Minjia Zhang, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Jeff Rasley, Shaden Smith, Olatunji Ruwase, et al. Deepspeed inference: Enabling efficient inference of transformer models at unprecedented scale. *arXiv preprint arXiv:2207.00032*, 2022.

[78] Sebastian Braun and Ivan Tashev. On training targets for noise-robust voice activity detection. In *2021 29th European Signal Processing Conference (EUSIPCO)*, pages 421–425. IEEE, 2021.

[79] J. Sammet. The use of english as a programming language. *Commun. ACM*, 9:228–230, 1966.

[80] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

[81] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM, 2018.

[82] Xi Victoria Lin, C. Wang, Luke Zettlemoyer, and Michael D. Ernst. Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system. *ArXiv*, abs/1802.08979, 2018.

[83] Jonathan Berant, A. Chou, Roy Frostig, and Percy Liang. Semantic parsing on freebase from question-answer pairs. In *EMNLP*, 2013.

[84] Xi Victoria Lin. Program synthesis from natural language using recurrent neural networks. 2017.

[85] Tomas Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, 2010.

[86] Jichuan Zeng, Xi Victoria Lin, S. Hoi, R. Socher, Caiming Xiong, Michael R. Lyu, and Irwin King. Photon: A robust cross-domain text-to-sql system. *ArXiv*, abs/2007.15280, 2020.

[87] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, 2019.

[88] A. See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *ACL*, 2017.

[89] Ursin Brunner and Kurt Stockinger. Valuenet: A neural text-to-sql architecture incorporating values. *ArXiv*, abs/2006.00888, 2020.

[90] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *NIPS*, 2015.

[91] Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. Incorporating external knowledge through pre-training for natural language to code generation. *ArXiv*, abs/2004.09015, 2020.

[92] Pengcheng Yin and Graham Neubig. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *EMNLP*, 2018.

[93] J. Chung, Çaglar Gülçehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *ArXiv*, abs/1412.3555, 2014.

[94] Mayank Agarwal, T. Chakraborti, Q. Fu, David Gros, Xi Victoria Lin, Jaron Maene, Kartik Talamadupula, Zhongwei Teng, and J. White. Neurips 2020 nlc2cmd competition: Translating natural language to bash commands. *ArXiv*, abs/2103.02523, 2021.

[95] Isaac Caswell and Bowen Liang. Recent advances in google translate, 2020.

[96] Mike Schuster and K. Paliwal. Bidirectional recurrent neural networks. *IEEE Trans. Signal Process.*, 45:2673–2681, 1997.

[97] H. Levesque. On our best behaviour. *Artif. Intell.*, 212:27–35, 2014.

[98] Ernest Davis. Notes on ambiguity.

[99] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[100] M. Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, G. Foster, Llion Jones, Niki Parmar, M. Schuster, Zhi-Feng Chen, Yonghui Wu, and Macduff Hughes. The best of both worlds: Combining recent advances in neural machine translation. In *ACL*, 2018.

[101] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, M. Srivastava, and Kai-Wei Chang. Generating natural language adversarial examples. In *EMNLP*, 2018.

[102] Free software foundation (2018) linux, 2018.

[103] Idan Kamara. Bashlex, 2014.

[104] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, S. Gross, Nathan Ng, David Grangier, and M. Auli. fairseq: A fast, extensible toolkit for sequence modeling. *ArXiv*, abs/1904.01038, 2019.

[105] G. Klein, Yoon Kim, Y. Deng, Jean Senellart, and Alexander M. Rush. Opennmt: Open-source toolkit for neural machine translation. *ArXiv*, abs/1701.02810, 2017.

[106] Magnum-nlc2cmd, 2020.

[107] M. Popel and Ondrej Bojar. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110:43–70, 2018.

[108] Emma Strubell, Ananya Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. *ArXiv*, abs/1906.02243, 2019.

[109] Qingqing Cao, Aruna Balasubramanian, and Niranjan Balasubramanian. Towards accurate and reliable energy measurement of nlp models. *ArXiv*, abs/2010.05248, 2020.

[110] Bash gen, 2022.

[111] Oneapi deep neural network library (onednn). https://github.com/oneapi-src/oneDNN.

[112] Intel extension for pytorch. https://github.com/intel/intel-extension-for-pytorch.

[113] Luis A Torres, Carlos J Barrios, and Yves Denneulin. Computational resource consumption in convolutional neural network training–a focus on memory. *Supercomputing Frontiers and Innovations*, 8(1):45–61, 2021.

[114] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[115] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[116] James Reinders. *VTune performance analyzer essentials*, volume 9. Intel Press Santa Clara, 2005.

[117] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[118] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[119] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[120] Jiachen Lu, Jinghan Yao, Junge Zhang, Xiatian Zhu, Hang Xu, Weiguo Gao, Chunjing Xu, Tao Xiang, and Li Zhang. Soft: Softmax-free transformer with linear complexity. *Advances in Neural Information Processing Systems*, 34, 2021.

[121] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[122] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, et al. The open images dataset v4. *International Journal of Computer Vision*, 128(7):1956–1981, 2020.

[123] Zixuan Jiang, Jiaqi Gu, Mingjie Liu, Keren Zhu, and David Z Pan. Optimizer fusion: Efficient training with better locality and parallelism. *arXiv preprint arXiv:2104.00237*, 2021.

[124] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[125] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[126] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[127] Gloo. https://github.com/facebookincubator/gloo.

[128] Intel delivers leading ai performance results on mlperf v2.1 industry benchmark for dl training, 2022.

[129] Ml commons v2.1 result, 2022.

[130] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, et al. On scale-out deep learning training for cloud and hpc. *arXiv preprint arXiv:1801.08030*, 2018.

[131] Mlcommons rcp. https://github.com/mlcommons/logging/tree/master/mlperf_logging/rcp_checker/training_2.0.0.

[132] Ammar Ahmad Awan, Hari Subramoni, and Dhabaleswar K Panda. An in-depth performance characterization of cpu-and gpu-based dnn training on modern architectures. In *Proceedings of the Machine Learning on HPC Environments*, pages 1–8. 2017.

[133] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[134] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. A survey of deep learning on cpus: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.