LEVERAGING LARGE PRETRAINED MODELS FOR LINE-BY-LINE SPOKEN PROGRAM RECOGNITION

Sadia Nowrin, Keith Vertanen

Michigan Technological University

ABSTRACT

Spoken programming languages significantly differ from natural English due to the inherent variability in speech patterns among programmers and the wide range of programming constructs. In this paper, we employ Wav2Vec 2.0 to enhance the accuracy of transcribing spoken programming languages like Java. Adapting a model with just one hour of spoken programs that had prior exposure to a substantial amount of natural English-labeled data, we achieve a word error rate (WER) of 8.7%, surpassing the high 28.4% WER of a model trained solely on natural English. Decoding with a domain-specific N-gram model and subsequently rescoring the N-best list with a fine-tuned large language model tailored to the programming domain resulted in a WER of 5.5% on our test set.

Index Terms— low resource speech recognition, large pre-trained models, voice programming, language modeling

1. INTRODUCTION

The traditional approach for automatic speech recognition (ASR) relies on a supervised approach where ASR models are trained with a vast amount of audio examples paired with their corresponding transcriptions. One of the major challenges of building an ASR system for a certain domain is the need for ample labeled training data. In recent years, ASR systems have made significant progress due to the emergence of semi-supervised and unsupervised learning paradigms for low-resource domains [1, 2, 3]. In this paper, we aim to address one such domain, dictating a Java program in a line-by-line manner. While some work has tried to infer an entire function or class from a single utterance [4, 5, 6], we think line-by-line dictation may better match how programmers incrementally build complex and novel programs. Further, even if a block of code is generated from a single utterance, the user may need to correct errors in the generation. Finally, for existing code bases, programmers may need to modify individual lines of code to, for example, fix bugs.

The speech patterns among different programmers and programming constructs is highly variable compared to natural language. For instance, programmers might verbalize the loop `for(i=0;i<n;i++)` in a natural way such as "create a for loop that iterates from one to n" or they may opt for a more literal approach such as "for open paren i equal one i less than n i plus plus close paren". A truly user-friendly approach would allow programmers to use whichever speaking style they prefer [7, 8]. This distinctive dialect and hybrid language style along with limited data in this domain make accurate recognition challenging. Nevertheless, the potential benefits of achieving accurate speech recognition in programming are substantial. Such advancements would allow individuals with motor impairments to input programs by voice rather than by typing. We intend to pursue a two-step pipeline for converting speech into code. The first step is to recognize the literal words spoken by the programmers. The second step is to convert those literal words into the target programming language. This second step could be done via machine translation guided by knowledge of the target programming language and the current code base. Different models could be swapped in for the second step using the same recognition system for the first step. We focus on the first step here.

In this work, we leverage Wav2vec 2.0 [9] which can learn representations directly from audio without requiring a large amount of labeled training data. We adapt Wav2vec using just one hour of spoken Java programs from novice and expert programmers and explore the impact of models trained in general spoken English and spoken programming language on recognition accuracy, along with the impact of different data quantities. To the best of our knowledge, this is the first work on recognizing line-by-line spoken programs.

Large language models like CodeBert [4], CodeGPT [6], and PLBART [10] have made significant advancements in code generation, primarily focusing on single-turn code generation where users express the intent of a whole block of code (e.g. an entire function) in one utterance. Nijkamp et al. [11] demonstrated how multi-turn program synthesis can be effective for improving program synthesis quality where users articulate their intentions for subprograms. While existing models leverage comments that programmers write to express the functionality of a section of code, these comments might lack information or fail to perfectly align with the written code. In our prior research, we found significant variations in how a simple method was verbalized by various programmers. Some phrased it naturally as "create a public method called cube that takes one parameter num and return the cube of num", while others verbalized literally: "public int cube

open paren int num close paren new line return num times num times num". In this work, we demonstrate how speech recognition accuracy on such spoken lines of code benefits from using a small corpus of spoken code to adapt both the neural speech recognizer and the decoder's language models. Our contributions are as follows:

- We conduct the first study that recognizes line-by-line spoken programs using pre-trained models.
- We show how to improve line-by-line spoken Java recognition by adapting a large pre-trained language model that was trained on various programming languages.
- We release[1] the human transcripts of the spoken programs, filtered comments from the CodexGlue dataset, and our best N-gram language model for spoken Java.

2. EXPERIMENTS

2.1. Datasets and Preprocessing

Previously, we conducted a data collection study involving programmers speaking Java to a hypothetical system [7]. Each participant was presented with 20 programs, from a pool of 30 unique programs. They were asked to verbalize a single line (16 programs) or a multi-line block of code (4 programs). To introduce variability and to investigate how to best collect data of this type, we designed half of the programs with a missing line, prompting users to invent it based on the surrounding code. In the other half, participants were asked to speak a highlighted line. In total, there were 41 Java programmers, 16 had 4+ years of experience and 25 had 1–2 years of experience. All were native English speakers (24 male, 17 female). The audio recordings were captured using a single audio channel sampled at 16 kHz. For data transcription, we transcribed audio including all spoken words, symbols, and spaces, e.g. "items at index i is equal to scan dot next int". Audio recordings that were mostly empty were excluded. The average audio duration was 13.6 seconds.

In this paper, we extended our SpokenJava dataset by recruiting more participants as part of our current research efforts. We split our data into training, development, and test sets, using an 80/10/10 split. We split the dataset by speakers and by the target line of code; our aim is to measure recognition performance on unseen programs spoken by unseen users. We split the 20 programming statements into 19/10/10 and included the corresponding utterances in the train/dev/test set. Our relatively compact SpokenJava dataset comprises 269/27/28 utterances, 29/6/6 users, and 60/5.7/6.9 minutes of audio for the train/dev/test sets. The dataset includes various programming constructs, including method signatures, if-else statements, loops, input-output statements, arrays, single and multi-line comments, decrement operations, math statements, and variable declarations. This dataset is unique because it focuses on spoken code, unlike existing datasets with written code comments or function descriptions. We have made the text transcripts of SpokenJava available[1]. Unfortunately, we did not have ethics approval to release the audio recordings.

In our language modeling experiment, we used the CodeSearchNet [12] dataset, encompassing six programming languages totaling 908K/45K/53K examples in the train/dev/test sets. We used the filtered version of the CodeSearchNet dataset from the CodeXGLUE [6] dataset. We extracted only the docstrings or comments associated with Java code snippets. We further refined the dataset to align it more closely with our spoken code dataset by 1) breaking multi-line descriptions into single lines, 2) splitting camel-case names such as variable names into separate words, and 3) discarding lines with symbols like angle brackets, hyphens, or underscores. The resulting dataset contained 495 K examples.

2.2. Fine-Tuning Wav2vec2

We conducted fine-tuning experiments using the open-source Wav2vec2 model [9], which was originally trained on natural English speech. Our primary objective was to investigate whether this model could adapt to spoken Java. Additionally, we wanted to assess the impact of data diversity and amount on performance. We used the Fairseq toolkit [13] with the settings from [9]. We used a learning rate of 1×10^{-4} and used the Adam optimizer with a tri-state rate schedule to improve convergence. This schedule warmed up the learning rate for the first 10% of updates, keeping it constant for the next 40%, and then linearly decaying it. We applied a layer dropout rate of 0.1. We fine-tuned models using 16-bit precision on two NVIDIA RTX 2080 Ti GPUs with a batch size of 4 samples per GPU. We updated all network parameters except for the feature encoder. We use time-step and channel mask probabilities of 0.65 and 0.25 respectively following [9].

Initially, we started with the Wav2vec2 base model, pretrained on 960 hours of unlabeled spoken English data from the Librispeech dataset. This model consisted of 12 transformer layers, each with 8 attention heads. We fine-tuned this base model with SpokenJava training data for 10K updates using the configuration in [9] and observed no significant improvement beyond that. We validated the performance on the Java dev set. We fine-tuned for 10K steps following [9] Subsequently, we leveraged a fine-tuned checkpoint from the base model, which had been trained on the LibriSpeech corpus with text labels for a maximum of 300K updates. Building upon this checkpoint, we performed an additional round of fine-tuning for 10K updates using our in-domain labeled data, aiming to leverage the knowledge gained from labeled natural English data during the initial fine-tuning stage.

2.3. Training N-grams and Beam Search Decoding

We trained 4-gram word language models using SRILM [14]. Our first data set consisted of human-generated transcripts

[1] https://osf.io/h6nk4

of people speaking lines of Java code. We reserved 5% of the data as a held-out development set leaving 255 sentences (4K words) in the training set. We trained separate 4-gram language models on single-line comments from CodexGlue [6] (495 K sentences, 4 M words) and LibriSpeech normalized (40 M sentences, 803 M words). We uppercased all text. All models used modified Kneser-Ney smoothing with a vocabulary of 203 K words with out-of-vocabulary words mapped to an unknown word. Our vocabulary merged a 200 K word vocabulary from LibriSpeech[2], all words appearing in our training corpus, and words appearing at least five times in CodexGlue. Requiring five occurrences helped exclude idiosyncratic non-camel-case multi-word combinations. We found that higher-order N-gram models did not provide gains.

Next, we created a mixture model by linearly interpolating 4-gram models from the SpokenJava, CodeXGlue, and LibriSpeech datasets. We tuned the mixture weights to optimize the perplexity on the held-out development set. We present our evaluation results by computing the Word Error Rate (WER) through beam search decoding with both the LibriSpeech 4-gram model and our best 4-gram mixture model across various Wav2Vec2 models. We use a lexicon-based beam search decoder available in the Flashlight framework [15]. The decoder aims to maximize:

$$\log p_{\text{AM}}(\hat{y}|x) + \alpha \log p_{\text{NGRAM}}(\hat{y}) + \beta|\hat{y}| \quad (1)$$

where $\hat{y}$ represents the output sequence, p_{AM} corresponds to the acoustic score, p_{NGRAM} represents the language model score, and $|\hat{y}|$ is the characters in the transcription (including spaces). α is the language model weight and β is the word insertion penalty both of which were optimized based on minimizing WER on the Java dev set. The word insertion penalty helps balance the contribution of the acoustic and language models. Our hyperparameter tuning searched over language model weights in [0, 5] and word insertion penalties in [-5, 5] using Bayesian optimization[3] for 128 trials. We explored beam widths in [5, 500] and found a beam width of 35 decoded at 11.3 sentences/second with no loss in accuracy.

2.4. Rescoring With Transformer Language Models

Motivated by the success in enhancing ASR recognition by rescoring with a transformer language model [16], we employ a similar strategy. We utilize the CodeGen-NL model with 350 M parameters from the Hugging Face Transformers Library [17] on two NVIDIA RTX 2080 Ti GPUs. CodeGen was trained on the Pile dataset[4], a portion of which includes GitHub code repositories. We fine-tuned the model with our SpokenJava corpus using a causal language modeling objective. We conducted a grid search to optimize hyperparameters based on minimizing per-token perplexity on the held-out development set. We searched over learning rates in [1e-5, 5e-5], weight decays in [0.001, 0.1], epochs in [1-10], and training batch sizes of 2 and 4.

We rescored the top 50 candidates from our first pass search, calculating a weighted linear combination following [18]. The re-estimated final ranking for each candidate was:

$$\log P_{\text{AM}}(\hat{y}|x) + \alpha_1 \log p_{\text{NGRAM}}(\hat{y}) + \alpha_2 \log p_{\text{NLM}}(\hat{y}) + \beta|\hat{y}| \quad (2)$$

Here, p_{AM} corresponds to the acoustic score, p_{NGRAM} and p_{NLM} represent the N-gram and transformer language model scores respectively, α_1 and α_2 are the corresponding weights.

3. RESULTS AND DISCUSSION

We evaluate our models on the development and test sets, reporting WER in three scenarios: without external language model decoding, with beam search decoding using an N-gram language model, and with beam search decoding followed by rescoring with a transformer language model. We selected the model checkpoint with the lowest WER on the Java development set for all evaluations.

The pre-trained Wav2Vec 2.0 base model fine-tuned on 960 hours of labeled data had a high WER on both the dev and test sets. As shown in Table 2, with just one hour of SpokenJava, we saw a 42.1% improvement in WER on the dev set but only an 8.8% improvement on the test set relative to LibriSpeech fine-tuned model. The model's difficulty in generalizing well on the test set, likely because of more diverse programming statements, can be attributed to overfitting caused by limited training data. Adapting a model already fine-tuned on 960 hours of labeled LibriSpeech data resulted in a WER of 8.1% and 8.7% on the dev and test sets respectively. The addition of a large amount of natural English data seemed to enhance the handling of diverse speakers and spoken content.

In our N-gram language modeling experiment, we found that a mixture model produced the best results. Our best N-gram model was a combination of 4-gram SpokenJava (weighted at 0.7), 4-gram LibriSpeech (weighted at 0.1), and 4-gram CodexGlue (weighted at 0.1). We think this improvement can be attributed to the inherent natural English language in spoken Java programs, which benefited from LibriSpeech data, as well as learning written comments from a partially in-domain CodexGlue dataset.

Table 3 shows the impact of different language models on the fine-tuned models, with the 4-gram LibriSpeech model as the baseline. Despite having a limited amount of in-domain text data, we achieved a substantial improvement in the WER decoding with our 4-gram mixture model. Specifically, for the 960h-Libri fine-tuned model, the WER decreased by 49.8% and 53.1% relative, and adding one hour of SpokenJava, it decreased by 13.8% and 21% relative on dev and test sets respectively, both in comparison to the LibriSpeech 4-gram model. Rescoring with our adapted transformer model further improved accuracy, resulting in a 19.6% and 8.3% rela-

[2]https://www.openslr.org/11/

[3]https://github.com/bayesian-optimization/

[4]https://pile.eleuther.ai/

Table 1. Example human transcripts and predictions from the 960h-libri model decoded with a 4-gram LibriSpeech language model (base model) and our rescored model. Word errors are underlined and in red.

Model	Text
Human	`items at index i is equal to scan dot next int`
Base model	`items a index eyes equal to scan dot next int`
Best adapted model	`items at index is equal to scan dot next int`
Human	`constructor public employee int age comma double salary`
Base model	`instructor public employ n comet double salary`
Best adapted model	`instructor public employ int age comma double salary`
Human	`for int i equal zero i less than five i plus plus`
Base model	`or in equal zero eye less than five eye plus plus`
Best adapted model	`for int i equals zero i less than five i plus plus`
Human	`create a public static method called print phrase that takes two arguments the first is a string phrase and the second is a double called num`
Base model	`create a public setoc method called print phrase that takes two arguments the first is a string phrase and the second is a double called numb`
Best adapted model	`create a public static method called print phrase that takes two arguments the first is a string phrase and the second is a double called num`

Table 2. WER on the development and test sets varying the labeled data used for fine-tuning. Results obtained by CTC greedy decoding without a language model.

Labeled Data	Dev	Test
960h LibriSpeech	31.1	28.4
1h SpokenJava	18.0	25.9
+ 960h LibriSpeech	**8.1**	**8.7**

Table 3. WER on the Java development and test sets of different Wav2Vec and language model combinations.

Model	LM	Dev	Test
960h-Libri	4-gram-libri	22.3	24.1
960h-Libri	4-gram-mixture	11.2	11.3
960h-Libri + 1h-Java	4-gram-libri	6.5	7.6
960h-Libri + 1h-Java	4-gram-mixture	5.6	6.0
	+ rescoring	4.5	5.5

tive reduction in WER on the dev and test sets respectively compared to the beam search results.

To the best of our knowledge, there are no existing methods purpose-built for recognizing line-by-line programming utterances. In [7], we tested performance using Google's speech-to-text on a different subset of our collected data which resulted in a high WER of 25%. Despite language model adaptation, the WER remained high at 19%, emphasizing the need for tailored approaches. As an initial exploration, our study establishes the groundwork for future comparisons in this domain.

We compared the predictions from our best model to the base model trained only on natural English (Table 3). Our best model seemed to have learned some key Java terms such as "for" or "int" and common variable names like "i" indicating effective domain-specific learning. The base model learned the natural description of a small method but struggled with fundamental Java keywords like "static" and abbreviated terms like "num". Our best model still struggled with text containing a blend of natural language words like "employee" and code-related words like "public" or "int". This suggests potential issues arising from sparse labeled data. Potential solutions could include obtaining more labeled data, employing data augmentation techniques (such as generating text-to-speech audio from back-translated text), or enhancing the model's ability to recognize user-defined names.

4. CONCLUSIONS

In our study, we explore the effectiveness of utilizing Wav2Vec 2.0 for recognizing spoken line-by-line Java programs, a task distinctly different from natural English speech. We demonstrate that even a modest addition of just one hour of domain-specific data to a model enriched with extensive knowledge of English yields remarkable results, with a substantial 74.1% relative reduction in WER compared to a model trained exclusively on standard English. This significant improvement holds promise for broader applicability beyond Java, potentially extending to other spoken programming languages as well. Our low final error rate of 5.5% provides a solid basis for further work that converts the literal words spoken into actual code and then investigates performance in actual use.

5. REFERENCES

[1] H. Zhu, L. Wang, J. Wang, G. Cheng, P. Zhang, and Y. Yan, "Wav2vec-S: Semi-Supervised Pre-Training for Low-Resource ASR," June 2022, arXiv:2110.04484 [cs, eess].

[2] P. Ortiz and S. Burud, "BERT Attends the Conversation: Improving Low-Resource Conversational ASR," Jan. 2022, arXiv:2110.02267 [cs, eess].

[3] G. Zheng, Y. Xiao, K. Gong, P. Zhou, X. Liang, and L. Lin, "Wav-BERT: Cooperative Acoustic and Linguistic Representation Learning for Low-Resource Speech Recognition," Oct. 2021, arXiv:2109.09161 [cs, eess].

[4] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[5] Y. Wang, W. Wang, S. Joty, and S.C.H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic, Nov. 2021, pp. 8696–8708, Association for Computational Linguistics.

[6] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," Mar. 2021, arXiv:2102.04664 [cs].

[7] S. Nowrin and K. Vertanen, "Programming by Voice: Exploring User Preferences and Speaking Styles," in *Proceedings of the 5th International Conference on Conversational User Interfaces*, New York, NY, USA, July 2023, CUI '23, pp. 1–13, Association for Computing Machinery.

[8] S. Nowrin, P. Ordóñez, and K. Vertanen, "Exploring Motor-impaired Programmers' Use of Speech Recognition," in *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility*, Athens Greece, Oct. 2022, pp. 1–4, ACM.

[9] A. Baevski, Y. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: A framework for self-supervised learning of speech representations," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, Eds. 2020, vol. 33, pp. 12449–12460, Curran Associates, Inc.

[10] W. Ahmad, S. Chakraborty, B. Ray, and K.W. Chang, "Unified Pre-training for Program Understanding and Generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Online, June 2021, pp. 2655–2668, Association for Computational Linguistics.

[11] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2022.

[12] H. Husain, H. Wu, T. Gazit, M. Allamanis, , and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," June 2020, arXiv:1909.09436 [cs, stat].

[13] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A Fast, Extensible Toolkit for Sequence Modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, Minneapolis, Minnesota, June 2019, pp. 48–53, Association for Computational Linguistics.

[14] A. Stolcke, "SRILM - An extensible language modeling toolkit," in *Proc. 7th International Conference on Spoken Language Processing (ICSLP 2002)*, 2002, pp. 901–904.

[15] V. Pratap, A. Hannun, Q. Xu, J. Cai, J. Kahn, G. Synnaeve, V. Liptchinsky, and R. Collobert, "Wav2letter++: A fast open-source speech recognition system," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 6460–6464.

[16] K. Li, Z. Liu, T. He, H. Huang, F. Peng, D. Povey, and S. Khudanpur, "An empirical study of transformer-based neural language model adaptation," in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2020, pp. 7934–7938.

[17] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and et al., "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[18] H. Huang and F. Peng, "An empirical study of efficient ASR rescoring with transformers," *CoRR*, vol. abs/1910.11450, 2019.