Introduction to Speech and Audio Processing – Final Project

Eitan Stern and Niv Eckhaus

https://github.com/niveck/Speech-Processing-Project

"It's like an LLM whispering in the STT model's ear…"

Abstract

In this work, we explore the integration of an LLM into a Speech-to-Text network, trained with CTC loss, to investigate the effectiveness of weak supervision in low-resource settings. Inspired by the success of weakly supervised models like WHISPER, we propose using an LLM to infer labels for half of our training data, simulating non-annotated samples. This approach allows the Speech-to-Text model to be trained with a mixture of true and LLM-generated labels. We evaluate our method on a digit name recognition task, comparing the performance of our weakly supervised model with fully supervised baselines. Despite technical challenges and resource limitations, our results show that the LLM-generated labels allow the Speech-to-Text model to maintain competitive accuracy, while benefiting from a larger dataset. We further analyze the LLM's labeling accuracy and the model's exact-match accuracy, highlighting the potential of this approach for extending Speech-to-Text models to low-resource environments. Our findings suggest that LLMs can play a valuable role in improving speech recognition models where labeled data is limited.

Introduction

During the course, we learned in class about the Connectionist Temporal Classification (CTC) loss for training Speech-to-Text (STT) models, and had the chance to implement a neural network of our own using this loss, in exercise 3. We were impressed by the high results that we received with our network over the digits dataset, achieving accuracy of around 90% with our suggested architecture! (9 times better than a random classifier) We liked the idea that CTC loss allows robustness of different decoding methods to the same output word – we focused on the fact that it enables punishing the network in the proper way, without missing correct possible matching, along the temporal axis. We wanted to explore and investigate more aspects that CTC loss can enable and be robust to. We were inspired by the big amount of weakly supervised data that was used to train WHISPER, as we learned in class. We decided to research whether our CTC network can be also robust to weak supervised learning. Both of us also research in NLP-oriented labs in the university, and we thought of combining our passion for LLMs (Large Language Models) with our motivation and curiosity in this course. That's how we came up with the idea of harnessing the power of LLMs to create a setting of weak supervision for a CTC loss STT model.

Model architecture

For our network, we decided to use our implementation from exercise 3 as our baseline, as it achieved great results. We came up with these exact layers, functions and hyper parameters after many iterations and experiments, as we explained in detail in exercise 3.

In summary, our network architecture consists of:

- Input: MFCC extraction (without normalization).
- Layer 1: Convolution from the MFCC dimension to the hidden state dimension.
- Activation function: LeakyReLU (chosen over ReLU, GELU, and Tanh due to better performance).
- Layer 2: Convolution from the hidden state dimension to the number of possible letters (+1 for epsilon).
- Output function: LogSoftmax (instead of Softmax; used for numerical stability, which improved our accuracy dramatically).

Regarding hyper parameters that worked best for us:

- Convolution kernel size: 5
- Number of MFCC coefficients: 13
- Hidden state dimension: 512
- Batch size: 64
- Learning rate: 1e-3
- Optimizer: AdamW

Our method

In class we learned about the revolution that the WHISPER model had brought with it, and among its novelties is its large-scale usage of weakly supervised data. Inspired by it, our main goal was to examine working in low-resource environments, mainly regarding annotated data (which is expensive to collect). Knowing the abilities and strengths of recent LLMs, our idea was to **use an LLM to annotate the labels through the training process**.

During the training of the original model in exercise 3, we added running printings of the current output of the model to each training sample. That allowed us to see its improvement over time. Viewing these outputs, we noticed that as humans, when looking at the disrupted outputs, we could often understand which digit was closest to it, and most times it was actually the right label. For example, when the model outputted “eiygt”, we could use our reasoning to infer the digit was “eight”; and the same with “sikt” and “six”, “seroe” and “zero”, “sevn” and “seven”, “forf” and “four”, and so on. This is exactly where we thought an LLM can become useful.

We split the original training set into two, and used the labels only for one half, simulating non-labeled data. We started the training of the model over only the labeled half dataset, for the first 5 epochs. This allowed the model to gain a certain ability of speech recognition, before we start using its output to infer labels. Then, for the rest of the training epochs, we used both halves. When a sample had its label, the CTC loss was computed as usual; In contrast, when a sample belonged to the “unlabeled” half, we **used an LLM over the current decoded output** of the model – we asked it to **infer which original digit was probably the source** of this speech recognition, and **used it as the target label in the CTC loss**.
Our assumption was that a strong LLM can “understand”

Choosing the right LLM

Our naive approach was to take a small local LLM, as we assumed that the task of recognizing only one of the 10 digits is not too complicated. We also preferred using a small LLM since we acknowledged that our resources were very limited.

We started experimenting with Microsoft’s **Phi-3-Mini-4K-Instruct**, a model we had worked with in the past, and had served us well for previous text generation tasks. We didn’t have problems running it with our resources, but unfortunately its performance was very poor. It had problems even outputting a name of a digit, and after a long prompt engineering process, it stuck with the choice of “nine”, no matter which input it was given.

Then we moved to examine bigger models (around 7-8B parameters), such as Meta’s **Llama-3.1-8B-Instruct**, models from the the **Qwen** series (by the chinese company with the same name), and also some of OpenAI’s **GPT** models. Intentionally we wanted to examine different and independent models, to prevent ourselves from sticking with our habits or our bias to one of them. Unfortunately, most of the recent LLMs require the use of the recent **Flash-Attention** package, and as we only learned after too many frustrating attempts, the school’s system group was conducting a large-scale update over the cluster’s machines and versions. Therefore, we couldn’t install the right packages to run those models successfully (specifically when we wanted to use them to generate answers over tens of thousands of training samples).

Reluctantly, and after spending too much time (over 3 full days of work) over technical issues (such as environments, packages and versions), we had to compromise and decided to use the simpler but more expensive way: **we used the LangChain API**. Now we also had a financial perspective to our consideration, so with our constraints we chose **GPT-3.5-turbo-0125**. It was our choice since it was cheap-enough per token, had great performance in generating reasonable answers for potential outputs, and was reasonably fast.

Our technical difficulties (and bad luck) didn't end there, and sadly after more than 30 hours of training, our code crashed due to a global problem in the LangChain API access (see appendix for their detailed apology email). In response we came up with a defense mechanism, wrapping the API calls in a try-except statement, allowing it a delayed second try, and if it fails too, then choosing a digit name randomly (by the same error-protection mechanism described in the "Prompt engineering & robustness to output" section).

Prompt engineering & robustness to output

We wanted our prompt to be as simple as possible, but still reflect the setting that the outputs are coming from – during the process of an STT model training, using the CTC loss, where the ground truth can be only one out of the 10 digits names.

Our original prompt was: (where *{decoded_output}* was replaced each time with the current decoded output of the model)

"We're training a CTC speech to text model and we decoded the probability matrix to get the word: '{decoded_output}'. Tell us what you think the word was:"

However, our experiments showed that the model was consistently answering with a sentence instead of a single word, for example like: *"Certainly! The word was probably 'nine'!"* We kept adding more and more emphasis on outputting nothing but the answer digit name, until we saw it indeed outputs nothing but a digit name. The result was a pretty cumbersome prompt:

"We're training a CTC speech to text model and we decoded the probability matrix to get the word: '{decoded_output}'. Tell us what you think the word was:" Give us the answer with one word only (very important to only use one word, because I don't want your answer to be more than one word):"

Our next problem was that the single LLM answer was not always a digit name. Sometimes its output was the single word "None". Therefore, we continued experimenting with different phrasings, eventually fixating on a version with explicitly giving the options for the LLM to choose from:

“My Speech-to-Text model is in the middle of training, and predicted the following text for a recording of a digit name: '{decoded_output}'. Which digit name is the most likely the true label of this recording? Reply with only one word out of the following: zero, one, two, three, four, five, six, seven, eight, nine (remember to only output one word out of them).”

Notice that we left out the “CTC loss” mention, as we saw that it did not contribute to the model’s success in deciphering the original word, and we wanted to keep the prompt as simple as possible.

Since an LLM samples from a token probability while generating, we knew we couldn’t guarantee that its answer will always be valid. To ensure robustness of the code while training, we came up with a validation mechanism for the LLM’s performance:

- After getting the response from the LLM, we lowered its case and stripped it from whitespaces.
- We then iterated over the 10 digit names. For each one we checked whether the response contained it. It was meant to cover the possibility of the response being in the format of “the answer is <name>”, but it also covered the simple format of just “<name>”.
- In the unlikely (but possible) case where more than one of the names was in the response, we would have chosen the first one, as we assumed it is the token which was prioritized by the LLM distribution.
- If none of the names were in the response (for example if the output was “None”), we treated it as if the LLM didn’t succeed in understanding the word. We didn't want that case to disrupt the training process, so if that case occurred, we would randomly choose one of the digit names, and log it as an event of an LLM failure.
    - After training, we checked the LLM failure rate, so we can understand if our experiment was relevant at all (if it failed too many times, it means we could just randomly choose a label instead of using an LLM). **Luckily, the failure rate was negligible**: the mean failure rate for a training epoch was 0.000520915 of the samples, with a standard deviation of 0.000136833.

Results

We should start by acknowledging that by the time we were finally able to run a successful training with our LLM loss, the training took much longer than expected. Unfortunately, we have reached our maximum resources time on the university’s cluster after 3 days of running, in the middle of the 12th training epoch (out of our planned 20 epochs). Since we saved checkpoints only after every 3 epochs, our last checkpoint was after the 9th, which was not good enough for testing. That's why we compare the model's results with the control models, only by their evaluation on the validation set, up to the 11th epoch.

We first compare our model's performance, to the performance of the same model without the use of the LLM for weak supervision. It means that the control model had all the labels for the training set, and did not have to use the LLM before the CTC loss. The control model was run in the same way, where the first 5 epochs were run only with the LLM-model's original "labeled" samples. The models' difference starts at the 6th epoch, where the LLM-model used the LLM to label the other samples, while the control used their true labels. So as expected, until the 6th epoch, all metrics show the same results (can be seen in figures 1 & 2).

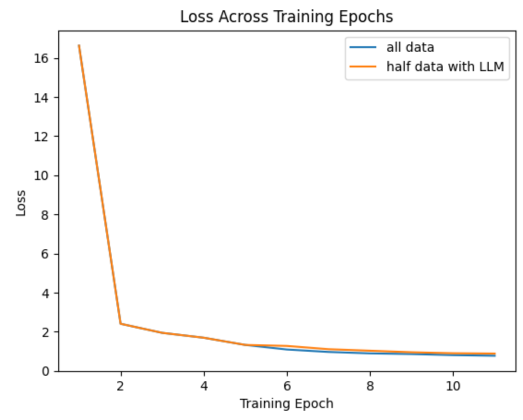


Figure 1

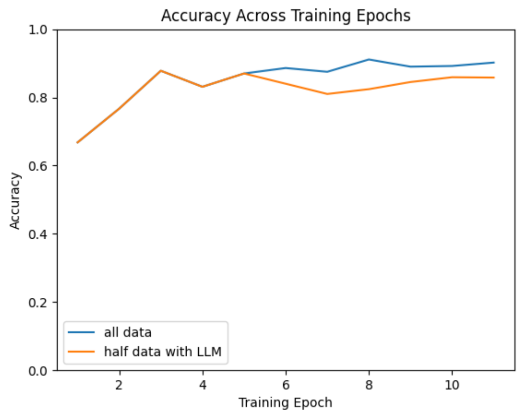


Figure 2

Starting from the 6th epoch and onward, we can see that both the loss and the accuracy of the models are still similar and have the same trends, even if our LLM-model's performance is a bit poorer (see zoomed-in plot of the loss in figures 3). The slightly worse results are reasonable, since the difference between the models is that the control has all the labels, while the LLM-model is trying to label some of the samples itself. So even if it labeled them perfectly, it could only be as good as the control model.

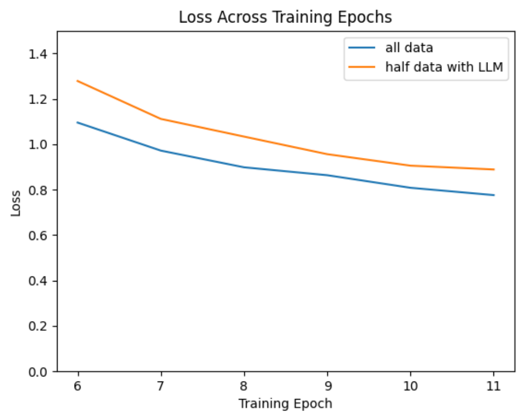


Figure 3

We also analyzed the LLM performance by calculating the accuracy of the labels it generates for the samples. We can see in blue in figure 4 that as our STT model learn, the LLM's performance improves – during the 6th epoch there are still not a few samples where the STT model outputs scrambled characters, so the LLM struggles with understanding their right label, achieving around 66% accuracy. It is seen that this performance is good enough to allow the model to learn, and with every epoch the LLM's accuracy improves as well – it rises monotonically until it gets to around 77% at epoch 11 (before our training was cut off).

When examining the LLM's output "with our eyes" we could see that the samples where the LLM failed, had usually outputs that even we as humans could not interpret as their right label. Few examples can be seen in table 1. It was mainly in the earlier epochs. Our conclusion is that training the model for more epochs, would likely result in clearer decoded outputs from the STT model, thus also improving the LLM performance over time.

Decoded output from the STT model	**LLM's prediction**	**True label**
f	five	four
tn	ten	three
sev	seven	six
hron	three	zero
i	nine	seven

tih	three	eight
e	three	zero
hr	four	six
tfvne	five	four

Table 1: examples of wrong LLM predictions on ambiguous raw outputs

To ensure that the LLM is indeed needed, we also compared its accuracy to the accuracy of the direct output of the STT model – as can be seen in orange figure 4, in the majority of the samples across all epochs, the direct output is not the right label (and in our qualitative analysis we saw it's not even a digit name). This shows the power of the addition of the LLM to the process.

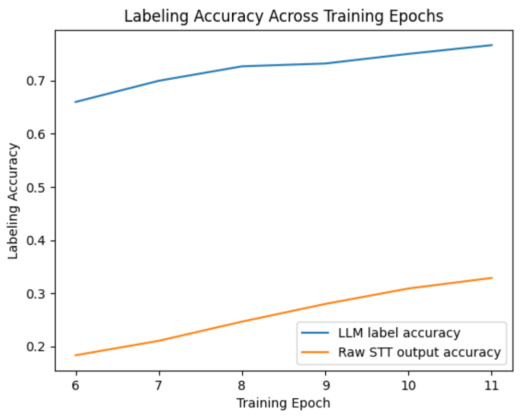


Figure 4

Moreover, we analyze the exact-match accuracy of our model. The regular accuracy is calculated by choosing the digit name with the lowest CTC score for the output. In contrast, in exact-match accuracy, we calculate the percentage of samples where the decoded output of the model was exactly the right digit name. We can see in figure 5 that as we start injecting noisy labels to the training process at the 6th epoch, the LLM-model indeed decreases in the exact-match accuracy. However, as the training proceeds, it manages to catch up with the control model. This is evidence that the weak supervision works as we hypothesized.

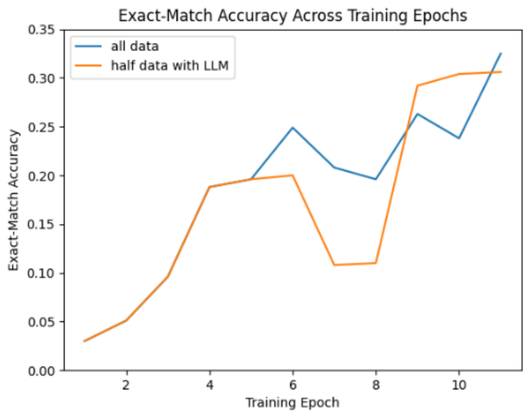


Figure 5

We also wanted to compare the model's results to a setting where we only had the original half of "labeled" data (without the half that we used the LLM for). We assumed that its performance should be worse, then both our LLM-model and the original control, as it trains over less data. In figure 7 we can see that its loss is indeed higher than the other two across all epochs (before epoch 6 the training is exactly the same). However, its accuracy was a bit higher than our weakly supervised LLM-model, as can be seen in figure 8. We can assume that it is due to a certain overfit that the model faces, and the use of less training data allows it to be avoided. We can see the same trend also in the exact-match accuracy metric (as can be seen in figure 9).

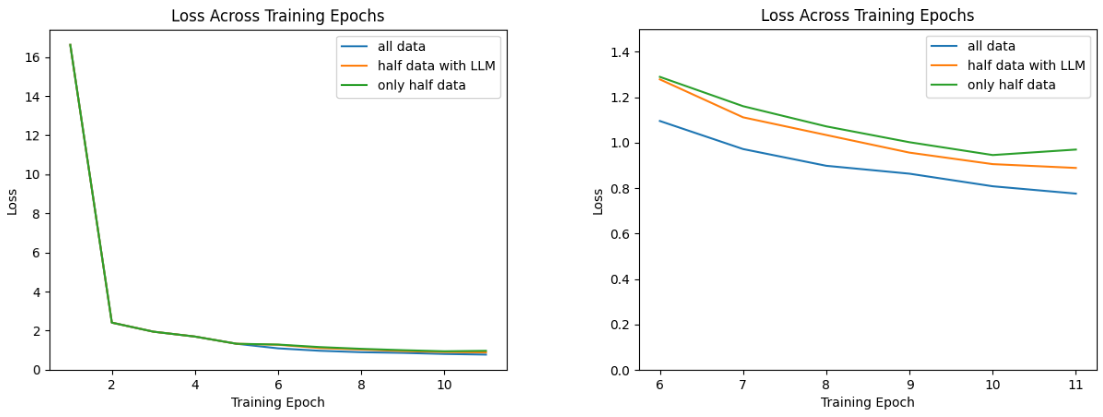


Figure 7

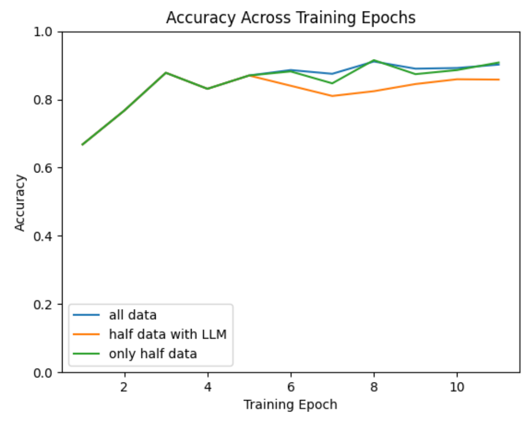


Figure 8

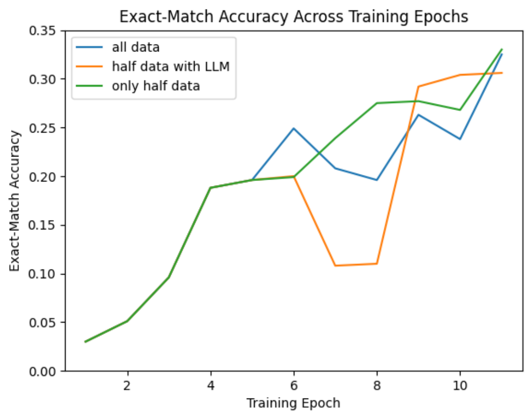


Figure 9

Conclusions

In this research, we explored the application of weakly supervised learning for an STT model trained using the CTC loss. By integrating an LLM to assist with label generation on non-annotated data, we aimed to simulate a low-resource environment and evaluate the potential of leveraging LLMs for weak supervision in STT tasks.

Despite facing several technical challenges, particularly related to resource constraints and API failures, our experimental results indicate that the use of an LLM can offer a viable solution for generating approximate labels. The LLM's performance improved as the underlying STT model advanced in training, suggesting a synergy between the evolving accuracy of the STT model's output and the LLM's capacity to infer labels. However, the slightly lower accuracy of the LLM-based model compared to a fully supervised model highlights the trade-off between weak supervision and access to labeled data.

While our experiments were cut short, preventing us from fully training the model to completion, we observed trends that suggest further training could improve the model's performance, particularly as clearer outputs from the STT model lead to more accurate predictions from the LLM. The exact-match accuracy of the weakly supervised model also caught up with the fully supervised baseline, demonstrating that weak supervision can still be effective over time.

Suggestions for future work

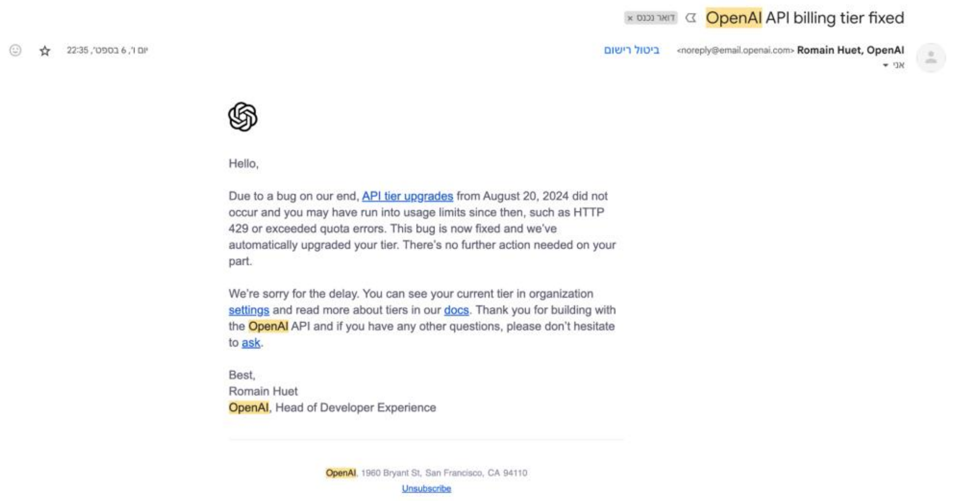
As seen throughout our report, our suggested method has opened a window for ruther research. Our results were limited by some technical and external resources, so we believe new insights can arise from the following research directions:

Training with more epochs: Our findings indicate that integrating the LLM for the label generation in a later stage of the training, would allow us to utilize its power better. Training for additional epochs before applying the LLM label generation, would likely result in clearer, more accurate outputs from the STT network. In turn, this could enhance the LLM's ability to infer the correct labels. We believe this can increase the LLM's accuracy.

Expanding to more diverse label spaces: This study was restricted to a digit recognition task, which is limited in scope. To understand the broader applicability of weak supervision using LLMs, future experiments should explore more complex label spaces, such as recognizing full words, phrases, or even domain-specific vocabularies. This would test the relevance of our approach in real-world applications where STT systems must handle more varied and nuanced output spaces.

Character-level tokenization for LLM inference: One potential improvement to the LLM-based label generation process could involve splitting the possible label outputs into character-level tokens rather than full words. By prompting the LLM with character-level options, it may be better equipped to infer the correct label, especially when dealing with outputs that are close but not exact matches. This could reduce ambiguity and improve the LLM's performance in weak supervision settings.

Appendix



OpenAI API billing tier fixed דואר נכנס ×

Romain Huet, OpenAI <noreply@email.openai.com> ביטול רישום

אני ▾

יום ו', 6 בספט', 22:35

Hello,

Due to a bug on our end, API tier upgrades from August 20, 2024 did not occur and you may have run into usage limits since then, such as HTTP 429 or exceeded quota errors. This bug is now fixed and we've automatically upgraded your tier. There's no further action needed on your part.

We're sorry for the delay. You can see your current tier in organization settings and read more about tiers in our docs. Thank you for building with the OpenAI API and if you have any other questions, please don't hesitate to ask.

Best,
Romain Huet
OpenAI, Head of Developer Experience

OpenAI, 1960 Bryant St, San Francisco, CA 94110
Unsubscribe