BERFA: Binary Estimator for Robust Filler Analysis

Andrew Ash
Analog VLSI Laboratory
Oklahoma State University
andrew.ash@okstate.edu

Marcus Mellor
VLSI Computer Architecture Research Laboratory
Oklahoma State University
marcus@infinitymdm.dev

***Abstract*—This work presents BERFA, a Binary Estimator for Robust Filler Analysis. The neural network is designed to detect filler words as a tool for public speaking organizations, such as Toastmasters, that value accurately tracking the verbal disfluencies of speakers. To achieve this goal, a neural network was developed that attained a maximum F0.5 score of 0.915 using the Wav2Vec2 Basic model for feature extraction along with a custom classifier head for the specific task. The final trained model was then used in a command-line application for real-time tracking of filler words with latencies under 1 second, making it a suitable tool for the intended filler word tracking task.**

I. Introduction

Public speakers strive to avoid interruptions in the flow of their speech. Normal speech disfluencies such as pauses, repetitions, and filler words, are considered unprofessional and distract from the goal of the speech. However, learning to recognize and avoid disfluencies is a difficult process for many speakers, and often requires intentional coaching.

One such coaching method, as applied by the Toastmasters public speaking organization, is to assign a dedicated "Ah-Counter" role [1]. At each meeting, the Ah-Counter listens specifically for filler words, often ringing a bell and keeping a tally. This near-instant feedback helps speakers break the habit of using filler words during short pauses.

This paper applies transfer learning methods to train a Binary Estimator for Robust Filler Analysis (BERFA) with the goal of filling a similar role: identifying filler words and providing real-time feedback to speakers. Several models are developed and discussed with a cumulative approach leading up to the final architecture. Top-performing models are identified and demonstrated with real-time speech input. The code used to train each model, as well as real-time speech input scripts, are freely available at https://github.com/infinitymdm/Final-Project-1.

A. Paper Overview

The remainder of this paper is organized into several sections. Section II discusses engineering requirements, design decisions on datasets and transfer models used, and performance evaluation strategies. Section III details the model structure and walks through the process of training several models with incremental improvements and design changes, culminating in the final BERFA model. This section also provides training and testing results. Section IV considers the practical challenges of providing real-time filler identification and feedback, including a look at the inference code. Finally, Section V discusses future improvements for further research to potentially produce better results.

II. Design

This section describes the decisions made during the design of the BERFA model and real-time input system. The design process is guided by a number of engineering requirements. These requirements led us to specific decisions on which datasets to use for training, which models to select for transfer learning, and which metrics to use for evaluating model performance.

A. Requirements

Each of the following requirements served as a goal during the process of developing the BERFA system.

a) *The system should detect obvious filler words:* While the primary purpose of the system is to detect filler words, some fillers are not obvious in speech. The word "so", for example, is frequently misused as a filler word but it is not obvious how to distinguish legitimate uses compared to filler words. The system should focus on obvious fillers such as "um" and "uh".

b) *The system should run on commodity hardware:* A model that can only run inference on large, expensive hardware has limited use. As presenters often travel to venues, carrying around inference hardware that is not designed for portability is prohibitively impractical. It should be possible to run BERFA inference on modest hardware, such as a thin-and-light laptop. We used a first-generation Lenovo Z13 laptop as our testing platform for this requirement.

c) *The system should provide feedback within no more than 5 seconds.:* In order to help speakers notice their use of filler words, the BERFA system should provide feedback as quickly as possible. This helps speakers interrupt their own habits and be more intentional about their speech. BERFA can only be useful as a speaking aid if it provides rapid feedback, ideally on the order of a few hundred milliseconds. Feedback may be visual, auditory, or a combination of both.

d) *The system should continue to identify fillers and provide feedback for as long as desired.:* Public speech varies widely in duration. Toastmasters clubs typically encourage 5 to 7 minute speeches, but lecturers often speak for an hour or more. The BERFA system must be capable of handling either use case. The system should be designed to handle arbitrary runtimes.

e) *The system should produce more false negatives than false positives.:* Ah-Counters in Toastmasters certainly fail to

identify and count a few filler words at each meeting. Our system should behave in a similar manner: it is more acceptable to fail to identify multiple filler words than to misclassify one nonfiller.

B. Dataset Selection

We considered multiple training datasets in designing the BERFA system. This section describes each dataset and why it was determined to be suitable or unsuitable for our purposes.

a) *TED-LIUM:* The TED-LIUM dataset is a corpus of over 450 hours of English-language TED Talk audio with detailed transcripts [2]. These transcripts include labeled filler words using a specific format, making it potentially suitable for our purposes. This dataset was designed for automatic speech recognition training, and a portion of the dataset is labeled using these methods.

Unfortunately, filler words in this dataset are often mislabeled. While sampling the dataset at random we found that although the transcripts accurately reflected the spoken words, non-speech labels were frequently incorrect or entirely missing. These problems persist across all three releases of the dataset, making it unsuitable for training filler detection.

b) *PodcastFillers:* The PodcastFillers dataset is a corpus of over 145 hours of gender-balanced English-language podcast audio designed for filler word location and removal tasks [3]. The dataset is broken down into several subsets, with some data used for speaker identification and other portions used for filler detection. Labels are conveniently divided into specific fillers such as "um", "you know", and "uh", or nonfiller categories such as "words", "breath", and "noise". Filler detection data consists of 1-second audio clips centered on the labeled item. The dataset contains approximately 35,000 fillers and 50,000 nonfillers.

While the dataset is slightly imbalanced, this can easily be overcome using some simple data augmentation. The most significant challenge with using this dataset for real-time audio is that while the dataset clips are centered on the labeled event, real-time audio can make no such guarantee. Data augmentation methods to compensate for this are discussed in Section III.

C. Transfer Model Selection

The Wav2Vec2 Basic model [4] was chosen for feature extraction. It was trained on a large dataset to allow for good generalization with a model intended to work on general speech processing tasks without additional complex layers required for specialized tasks. This lighter form of Wav2Vec2 still has many layers to perform feature extraction, using the following architecture:

- 1x Conv1D with kernel size 10, stride 5
- 4x Conv1D with kernel size 3, stride 2
- 2x Conv1D with kernel size 2, stride 2
- 1x Linear with 512 inputs, 768 outputs
- 1x Conv1D with kernel size 128, stride 1, padding 64
- 12x Encoder layers each with
  - 1x K projection 768 inputs, 768 outputs
  - 1x V projection 768 inputs, 768 outputs
  - 1x Q projection 768 inputs, 768 outputs
  - 1x Out projection 768 inputs, 768 outputs
  - 1x Linear with 768 inputs, 3072 outputs
  - 1x Linear with 3072 inputs, 768 outputs
- Normalization and Dropout layers throughout

The first seven Conv1D layers act as feature extractors that also reduce the size of the output to improve speed with smaller layers deeper in the network. The linear layer then changes the dimensionality to match the target size of the upcoming encode layers. A final Conv1D layer reinserts temporal information without reductions to dimensionality. The 12 encoder layers then behave as a standard transformer to complete the feature extraction with more temporal context for our customized classifier designs, which will be discussed in Section III.

D. Performance Metrics

Several metrics were evaluated throughout the training process to help us tune the behavior of the BERFA model. This section provides a brief justification for the selection of each metric.

a) *Accuracy:* Accuracy is calculated as the number of samples the model correctly classifies out of the total number of samples in the set.

$$\text{accuracy} = \frac{\text{correctly classified samples}}{\text{all samples}} \tag{1}$$

Typically this is evaluated using a held-out validation set consisting of a small fraction of the complete dataset. We used the provided validation split from the PodcastFillers dataset. This metric gives us a simple measure of the overall performance of our models.

b) *Precision:* Precision refers to the fraction of samples classified as a particular class which are correctly classified.

$$\text{precision} = \frac{\text{correctly classified samples of class k}}{\text{all samples classified as k}} \tag{2}$$

Higher precision is associated with a lower false positive rate relative to the false negative rate. This is typically calculated for each class, but since our model is a binary classifier (i.e. only one class exists) we calculate it only for the case where filler words are present.

c) *Recall:* Recall refers to the fraction of samples of a particular class correctly classified by the model.

$$\text{recall} = \frac{\text{correctly classified samples of class k}}{\text{all samples of class k}} \tag{3}$$

As with precision, we measure recall only for the case when a filler word is present.

d) $F_{0.5}$*-score:* F_β score is a weighted harmonic mean of the precision and recall metrics.

$$F_\beta = (1+\beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \tag{4}$$

The β parameter allows the sum to be tuned towards either precision or recall, with $\beta < 1$ resulting in a larger weight for precision. Because high precision is more important in our application, we set $\beta = 0.5$.

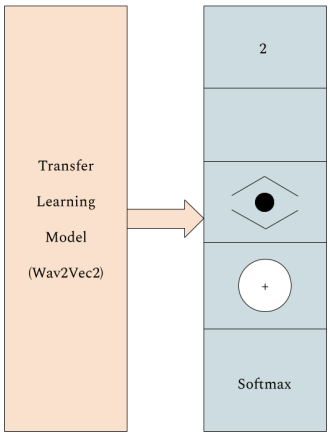

Fig. 1: The Wav2Vec2 basic model is used for feature selection, followed by a single fully connected layer with two neurons and a softmax output to be fed into a cross-entropy loss function.

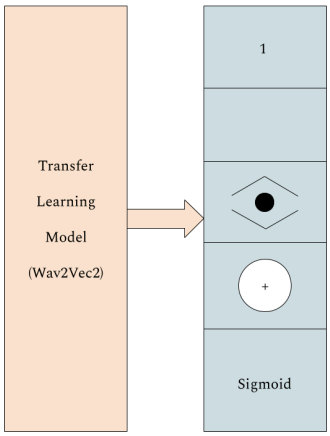

Fig. 2: The Wav2Vec2 basic model is used for feature selection, followed by a single fully connected layer with one neuron and a sigmoid output to be fed into a binary cross-entropy loss function.

III. Model Tuning & Training Results

All models we trained used the Wav2Vec2 Basic model for voice audio feature extraction without fine-tuning, as this would have greatly increased the time required to train new models between tests and could have led to overfitting due to the much smaller training dataset compared with the Wav2Vec2 training data.

A. Initial Model

Fig. 1 shows the basic structure of our first model with a single linear layer classifier with two neurons fed into a softmax activation function to create confidence scores for the filler and non-filler classes. The confidence values are then used by a cross-entropy loss function.

This basic classifier trained in under 2 minutes per epoch, allowing for many epochs of training while maintaining quick evaluation. PyTorch's built-in CrossEntropyLoss method includes an optional weight value to help with class imbalance problems [5]. For our application, this weight allows us to punish loss related to the positive class more heavily than the negative class by increasing the weight for Filler class loss and leaving Non-Filler unchanged. The loss equation is shown in (5).

$$\text{Cross Entropy Loss}(x, y) = -w_y \cdot \log\left(\frac{e^{x_y}}{\sum_j e^{x_j}}\right) \quad (5)$$

Giving double the weight to Filler classifications in the loss function, this model achieved an F0.5 score of 0.865 after training for 49 epochs. At this point we determined exploring other loss functions may improve our model's performance and streamline our training process. The slowly climbing F0.5 score made training take longer than necessary. In addition, cross-entropy loss depends on the logits of all classes in the denominator. Our custom weight is only intended to punish false positives, but dependence on all classes' logits leads our weight to interact indirectly with the non-filler class decision.

B. Using Custom BCE Loss

Based on recommendations from a Stack Overflow question [6] and answer [7], the next model we attempted was a custom binary cross-entropy (BCE) loss. To accommodate the change in loss function, the model needed slight modifications, shown in Fig. 2. A single neuron outputs a logit that is passed through a sigmoid activation function and then given to the custom BCE function, calculated as shown in (6). The ε here is just a small value to ensure that log has a non-zero value for stability. w_{fn} and w_{fp} are the custom weights for the negative and positive classes, y is the target class, and p is the sigmoid output (a value between 0 and 1).

$$\begin{aligned}\text{Custom BCE} = -\big(&w_{\text{fn}} \cdot y \cdot \log(p + \varepsilon) + \\ &w_{\text{fp}} \cdot (1 - y) \cdot \log(1 - p + \varepsilon)\big)\end{aligned} \quad (6)$$

Training for this network was highly unstable. Fig. 3 shows the F0.5, accuracy, precision, and recall all wildly oscillating between minimum and maximum values from one epoch to the next. This is because the custom loss function does not implement any maximum magnitude for the output, while PyTorch's built-in BCEloss function caps the loss at a mag-

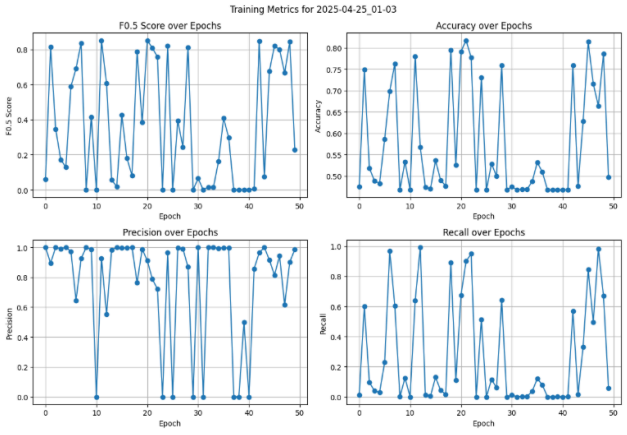

Fig. 3: The F0.5, accuracy, precision, and recall of the model using custom BCE Loss. Training is highly unstable with the model oscillating between classifying everything as a filler and then as a non-filler.

nitude of 100 for stability while training [8]. A maximum magnitude could have been easily implemented, but further exploration of the PyTorch documentation [9] and forums [10] revealed that BCEWithLogitsLoss could meet our needs without writing additional custom code.

C. Using Weighted BCE Loss

The next model is identical to that of Fig. 2, but the loss function is replaced with PyTorch's BCEWithLogitsLoss method. The pos_weight variable allows us to punish false positives more heavily than false negatives by setting a weight between 0 and 1. Weights greater than 1 would punish false negatives instead of false positives. This is equivalent to (6) where w_{fn} is set to 1 and w_{fp} is pos_weight, which is determined based on the intended ratio of the two weights. Fig. 4 shows the baseline performance for the model with this loss function; a pos_weight of 1.0 ensures standard performance without weighting to punish false positives. Fig. 5 shows the minor changes that start to appear with a small emphasis on the positive class using a pos_weight of 0.75. Finally, Fig. 6 shows the differences in the model's behavior with a pos_weight of 0.5. The recall occasionally drops below 0.8, but the F0.5 is getting closer to 0.9 due to precision gains. The model's accuracy is slighter higher with this pos_weight because the dataset is slightly weighted towards fillers.

At this stage in the model design, we implemented learning rate scheduling to help with stability while training. Fig. 7 shows 50 training epochs with a learning rate scheduler that reduces the rate by a factor of 0.33 after 2 epochs without improving the F0.5 score. The training has largely stopped around 20 epochs as the learning rate hits its minimum. Fig. 8 shows the performance differences with a pos_weight of 0.75 with the same learning rate scheduler. The model now overemphasizes recall for our purposes, resulting in a slightly lower F0.5 maximum. In this stage of design, we achieved a maximum F0.5 of 0.881 in the 10th epoch of training using a pos_weight of 0.5 (0.023 better than the cross-entropy loss and 39 epochs faster).

After this, we added early stopping to avoid wasting unnecessary training time if no improvements were noted as the learning rate decreased to identify a near-optimal network. From this stage onward, we tested with three different pos_weight values to explore the small performance differences between a weighting of 0.75 (small extra punishment for false positives), 0.5 (heavy false positive punishment), and 0.625 (a balance between the first two weights).

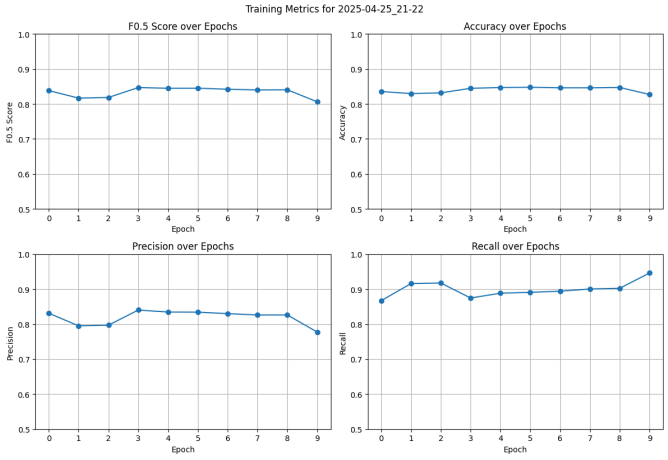

Fig. 4: A baseline example of BCEWithLogitsLoss using a pos_weight of 1. The loss function has the default behavior in this configuration.

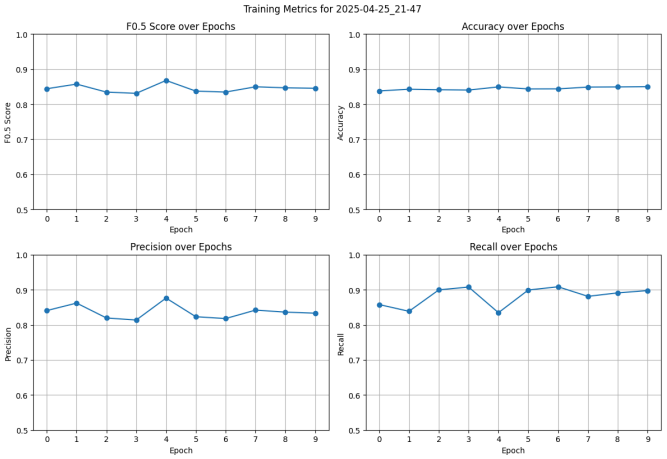

Fig. 5: BCEWithLogitsLoss using a pos_weight of 0.75. The fifth epoch shows the training tradeoffs between precision and recall are impacting performance with minimal change in accuracy compared to the baseline.

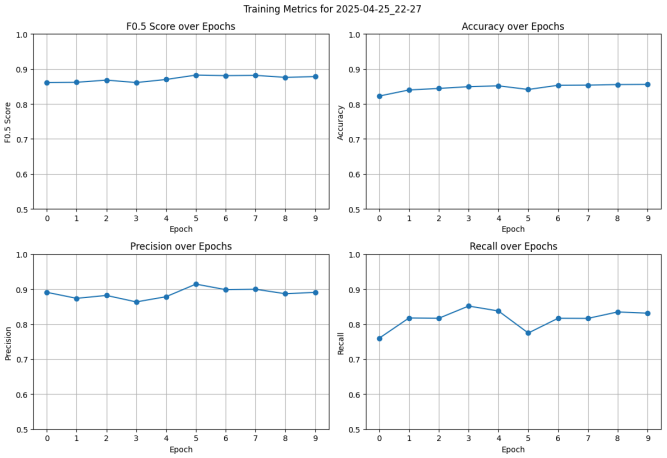

Fig. 6: BCEWithLogitsLoss using a pos_weight of 0.5. The later epochs show that the tradeoffs between precision and recall slowly increase F0.5 at the cost of lower recall compared to the baseline.

D. Adding Data Augmentation

Reading through the full details of the PodcastFillers dataset [3] revealed a potential overfitting issue in the data. The audio clips that include a filler are always centered on the filler; this quirk of the dataset could easily lead the network to assume a filler should be in the middle of sampled audio. To resolve this issue, we added data augmentation code that takes a random amount of audio off the front or back of the clip and then appends the same amount of silence onto the clip's other side. This effectively moves the remaining audio forward or backward in time to attain more diverse filler locations, thereby achieving better translation invariance in the time dimension. Fig. 9 shows the best model with data

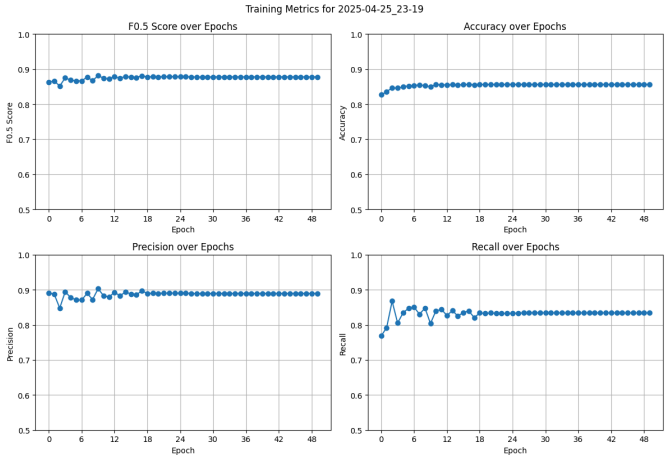

Fig. 7: A pos_weight of 0.5 with learning rate scheduled to reduce by 1/3 after 2 epochs without F0.5 score improvement. Results have plateaued after ~20 epochs.

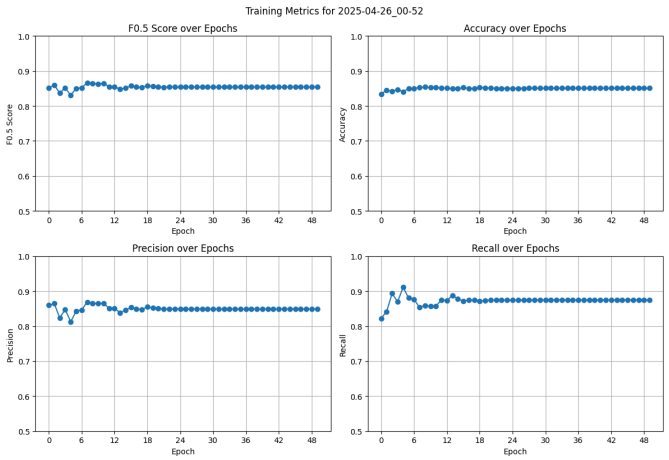

Fig. 8: A pos_weight of 0.75 with learning rate scheduling. Results emphasize recall more than precision and plateau after ~20 epochs.

augmentation used a pos_weight of 0.625 and random shifting between $\pm$ 150 milliseconds to achieve an F0.5 of 0.882 in 7 epochs (0.001 higher and 3 epochs faster).

E. Adding a Dense Layer

The final modification to the model was to add another linear layer to the classifier. The new model is shown in Fig. 10. The classifier head is now a small multi-layer perceptron with additional feature extraction capabilities from the added hidden layer. The extra layer adds 10-15 seconds onto each epoch, maintaining efficient training and testing.

Fig. 11 shows the new model trained with a pos_weight of 0.75; this model achieves a good balance of precision and recall. However, the recall values are consistently greater than precision, indicating this may not be the best value for our use case. Fig. 12 shows a better balance of precision and recall, with both values staying much closer together and a slightly better precision than recall. Fig. 13 shows the best-performing model achieved an F0.5 score of 0.903 in 7 epochs using a pos_weight of 0.5. This comes at the cost of a lower recall that is far less stable throughout the training.

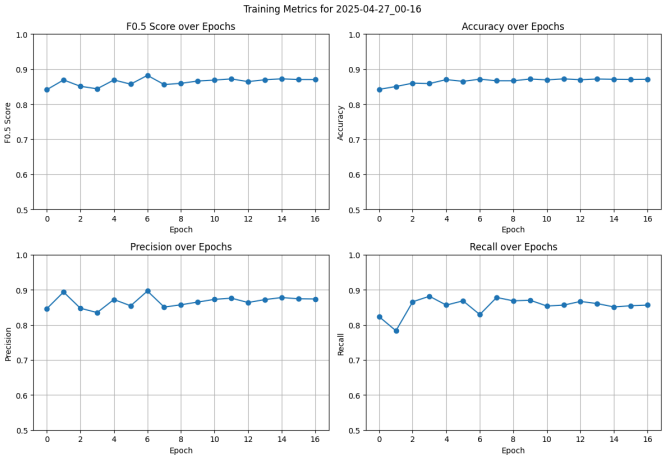

Fig. 9: Using a pos_weight of 0.625 and adding time-shifting data augmentation yields slightly better performance than prior models. Early stopping prevented unnecessary training after epoch 17.

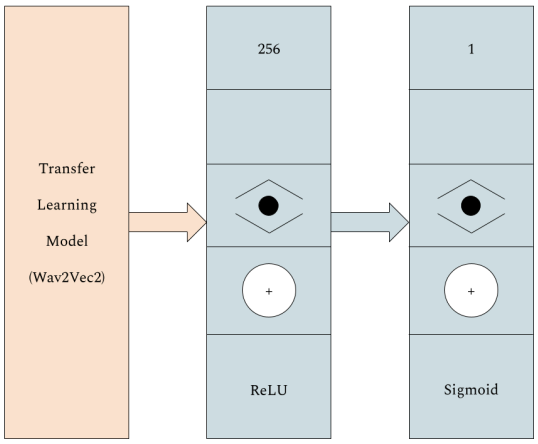

Fig. 10: The Wav2Vec2 basic model is used for feature selection, followed by a simple multi-layer perceptron. The first fully connected layer extracts more feature information, and the second is the classifier with a sigmoid output to be fed into a cross-entropy loss function.

Regardless of pos_weight, the accuracy is largely unaffected, slowly climbing towards 0.9.

F. Adjusting Decision Thresholds

The final improvement came from manually adjusting the threshold delineating filler and non-filler classifications. By default, the model assumes a threshold of 0.5 and then rounds up or down to the nearest classification label. The F0.5 score can be further tuned by allowing a custom threshold value to filter out additional false positives. SkLearn's precision_recall_curve method tests all different threshold values that would make a change in classification accuracy [11]. For example, if a test dataset included three samples that yielded logits of 0.3, 0.56, and 0.8 then the method would test precision and recall at these threshold values as well as one threshold value less than the minimum and one threshold greater than the maximum to force all classifications to each of the two classes. The precision, recall, and list of thresholds checked are all returned for easy plotting.

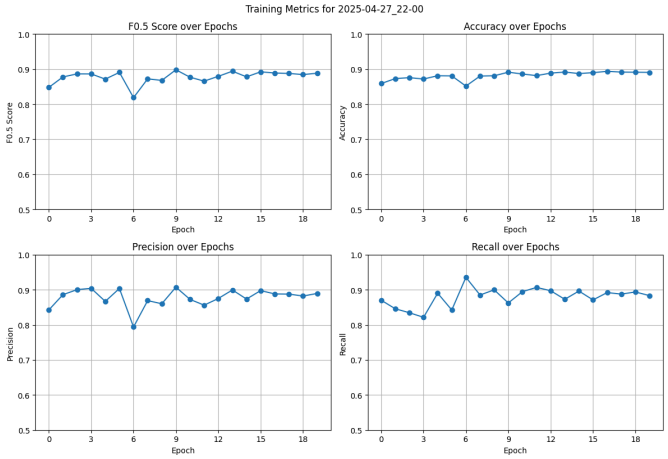

Fig. 11: A pos_weight of 0.75 provides a better balance of precision and recall than past models using this value. Both metrics stay close to 0.9. The F0.5 also closely follows this trend.

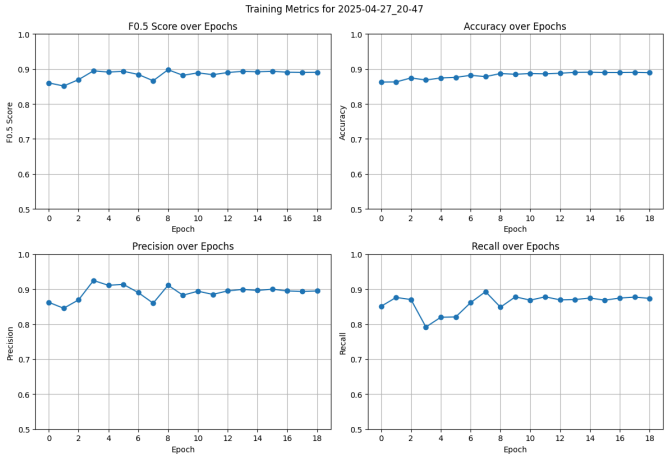

Fig. 12: A pos_weight of 0.625 benefits precision well, with values consistently near or above 0.9. Despite oscillations in recall, the F0.5 remains steady between 0.89 and 0.9.

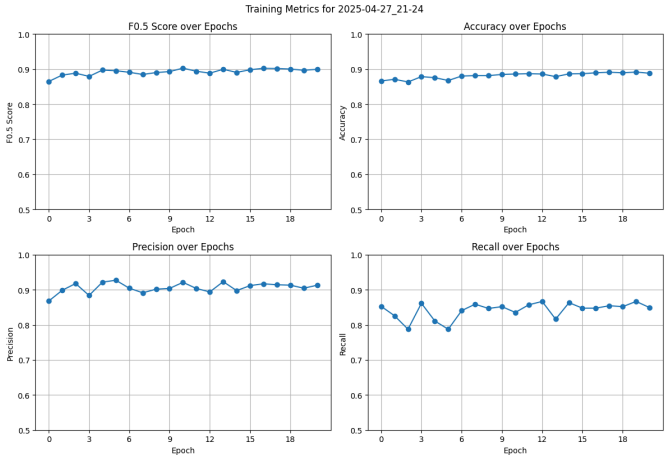

Fig. 13: A pos_weight of 0.5 benefits precision well, with values consistently above 0.9. F0.5 climbs above 0.9 despite recall scores staying between 0.8 and 0.85.

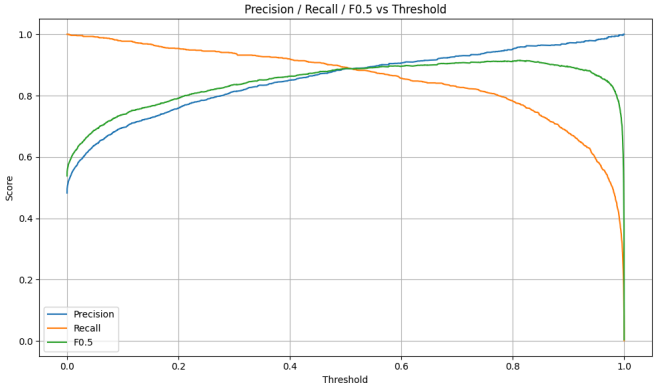

Fig. 14: Threshold shifting for epoch 16 with a pos_weight of 0.75 (see Fig. 11) achieves a maximum of 0.915 at a threshold of 0.809.

Fig. 14 shows the result of threshold shifting using epoch 16 of the model trained with a pos_weight of 0.75. It is interesting to note that the precision and recall curves intersect at a threshold of approximately 0.5, indicating the default threshold is well balanced for cases where both metrics are equally important. Shifting the threshold to 0.809 yields an F0.5 of 0.915, an improvement of 0.026 from the default threshold at the cost of much lower recall.

Fig. 15 shows the improvements possible from epoch 15 of the 0.625 pos_weight model. A balanced precision and recall would require a threshold near 0.425, clearly showing the model's preference to improve precision even before threshold shifting. The maximum F0.5 reaches 0.909 at a threshold of 0.713, yielding a 0.016 improvement.

Fig. 16 shows the improvements possible at epoch 16 of the 0.5 pos_weight model. The threshold for balancing precision and recall now drops below 0.4, indicating the expected extreme bias towards precision in this model. Shifting the threshold to 0.764 leads to a maximum F0.5 of 0.910, an improvement of 0.008 from the default threshold.

This final hyperparameter adjustment allowed for a maximum F0.5 of 0.915 by using a pos_weight of 0.75 and a threshold of 0.809. This is not recommended as the false negative rate quickly increases with a threshold greater than 0.6. For a practical system, the small precision increases are not worth the large recall losses. Instead, a threshold between 0.5 and 0.75 should improve any of these models moderately at a lower cost to recall.

IV. Real-time Filler Detection

A major challenge in applying the BERFA system was handling real-time audio input. If the system were to stall for inference each second, it could miss significant portions of speech audio that elapse during classification of the previous segment. This section describes the strategies we used to overcome this challenge.

A. *Generating Network Inputs from Real-time Audio*

To handle real-time inference and audio, we made use of Python's built-in `asyncio` module. This module provides the usual handful of real-time processing primitives, including

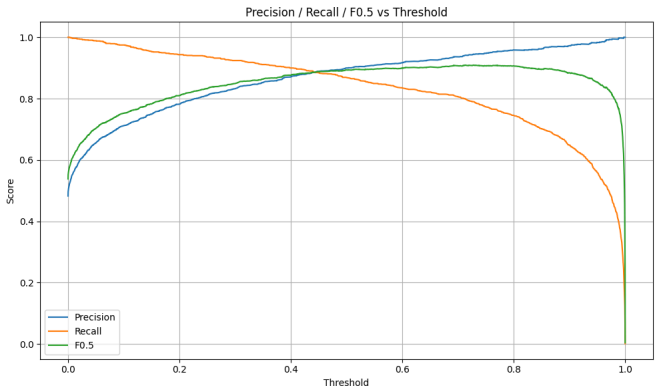

Fig. 15: Threshold shifting for epoch 15 with a pos_weight of 0.625 (see Fig. 12) achieves a maximum of 0.909 at a threshold of 0.713.

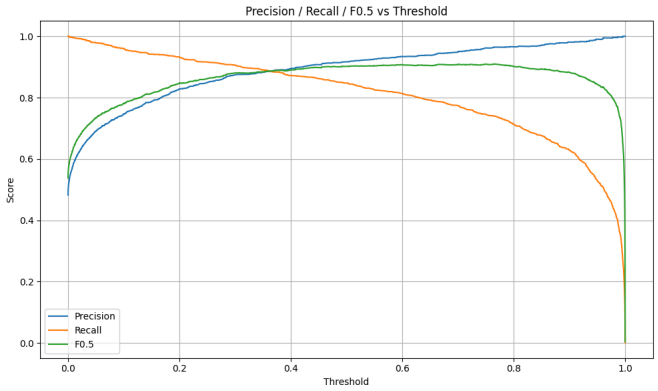

Fig. 16: Threshold shifting for epoch 16 with a pos_weight of 0.5 (see Fig. 13) achieves a maximum of 0.910 at a threshold of 0.764.

semaphores, queues, mutexes, and much more. Particularly applicable to our use case was the asynchronous iterator feature, which we used to yield audio for inference in segments appropriately sized for the BERFA model.

Since the PodcastFillers dataset contains 1-second audio clips sampled at 16 kHz, we elected to use the same sample rate and duration for real-time input. The `sounddevice` Python library provides an abstract interface to audio hardware, allowing us to simply pass the sample rate and number of samples as keyword arguments. Psuedocode representing this process is displayed in Listing 1.

B. *Identifying Fillers in Segmented Audio*

Once the audio is broken into appropriately sized segments, it is simple to pass it through the trained BERFA model for inference. The model outputs raw logits, which are passed through a sigmoid function and finally compared to a user-defined threshold to produce the final classification.

Similar to generating segments, this has to happen asynchronously. The program can't pause for inference; it must

```
def input_generator():
  while True:
    for segment in inputstream(fs, size):
      yield segment
```

Listing 1: Pseudocode demonstrating the process of asynchronously generating appropriately-sized audio clips from real-time audio.

```
for segment in input_generator():
  logit = classifier(segment)
  result = (sigmoid(logit) > threshold)
```

Listing 2: Pseudocode demonstrating the asynchronous classification loop. Note how the code iterates over segments returned by the `input_generator` defined in Listing 1.

continue to sample audio in the meantime. We also apply the `asyncio` module here to allow inference to be dispatched to a different thread while audio sampling continues unimpeded. Pseudocode for the classification process is displayed in Listing 2.

V. Conclusion

While significant work remains before the BERFA system is ready for practical application, our design satisfied all requirements and exceeded our expectations. The resulting model and inference scripts are surprisingly robust and usable, though more refinement to both would be necessary prior to real-world deployment.

Several improvements could make the system perform even better for the intended task. First, we would begin exploring which layer(s) of Wav2Vec2 should be included in fine-tuning. While we were concerned about overfitting to the PodcastFillers dataset, there may be layers that could extract more meaningful features with a well-controlled learning rate after some initial training of the classifier head. Though fine-tuning the transfer model would likely incur a large increase in training time due to the many additional parameters, it would likely yield an increase in precision and/or recall. Identifying additional data augmentation beneficial to the dataset may help reduce concerns of overfitting. The dataset tends towards slower speaking speeds common to podcasts; augmenting the speaker speed may help identify faster or longer, drawn-out filler words. The dataset is gender-balanced, but minor shifts in speaker pitch may help reduce any overfitting to the voices represented. In addition, our testing of the real-time system determined that the speaker's accent has some impact on accuracy. Identifying additional audio data with a greater diversity of accents may also improve the system's performance in real-world applications.

The project could better serve in a Toastmasters meeting with a more complex system. Adding a neural network focused on automatically identifying different speakers would allow for a final system that can run for a full Toastmasters meeting. When the Ah-Counter is ready to generate the final report, the system could then report filler word counts for each individual who spoke throughout the meeting.

The BERFA system meets all design requirements and demonstrates practical usefulness in real-time speech testing. Several models are trained and compared with a variety of performance metrics, with the final model demonstrating favorable precision and accuracy characteristics for filler word identification. The real-time infrastructure code for applying BERFA is briefly analyzed and potential possible future improvements are discussed. The results speak for themselves: BERFA has significant potential as an aid for speakers.

References

[1] "Toastmasters Club Meeting Roles." [Online]. Available: https://www.toastmasters.org/membership/club-meeting-roles

[2] F. Hernandez, V. Nguyen, S. Ghannay, N. Tomashenko, and Y. Estève, "TED-LIUM 3: Twice as Much Data and Corpus Repartition for Experiments on Speaker Adaptation," in *Speech and Computer*, Springer International Publishing, 2018, pp. 198–208. doi: 10.1007/978-3-319-99579-3_21.

[3] G. Zhu, J.-P. Caceres, and J. Salamon, "Filler Word Detection and Classification: A Dataset and Benchmark," in *23rd Annual Cong.~of the Int.~Speech Communication Association (INTERSPEECH)*, Incheon, Korea, Sep. 2022. [Online]. Available: https://arxiv.org/abs/2203.15135

[4] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations." Accessed: Apr. 29, 2025. [Online]. Available: http://arxiv.org/abs/2006.11477

[5] "CrossEntropyLoss — PyTorch 2.7 documentation." Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

[6] clueless, "Using weights in CrossEntropyLoss and BCELoss (PyTorch)." Accessed: Apr. 24, 2025. [Online]. Available: https://stackoverflow.com/q/67730325

[7] J. Vuurens, "Answer to "Using weights in CrossEntropyLoss and BCELoss (PyTorch)"." Accessed: Apr. 24, 2025. [Online]. Available: https://stackoverflow.com/a/67778392

[8] "BCELoss — PyTorch 2.7 documentation." Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html

[9] "BCEWithLogitsLoss — PyTorch 2.7 documentation." Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html

[10] "BCELoss with class weights - vision." Accessed: Apr. 23, 2025. [Online]. Available: https://discuss.pytorch.org/t/bceloss-with-class-weights/196991

[11] "precision_recall_curve." Accessed: May 06, 2025. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.metrics.precision_recall_curve.html