Filler Word Detection Using Transfer Learning

Marcus Mellor and Andrew Ash

Outline

- Problem Introduction
- Dataset Selection
- Training Process
- Live Demonstration
- Q&A

Problem Introduction

Initial Concept: “Ah Counter”

- Inspired by the Toastmasters “Ah Counter” role
- Purpose: identify and count filler words in speech audio
  - Provide real-time feedback to speakers
  - Help speakers improve by reducing filler words
- Similar to keyword detection tasks
  - E.g. using “Alexa” or “Hey Google” to start talking to a personal assistant
  - Detect a very small subset of speech with high precision
  - Ignore everything else

Requirements

- Detect obvious filler words
    - Don’t worry about “like” and “so”; these are not always fillers
- Run on commodity hardware
- Provide feedback within no more than a few seconds
- Keep running for as long as necessary without exploding
    - No crashing, run out of memory, etc.
- Heavily punish false positives
    - It’s acceptable to miss a few filler words, but not acceptable to mislabel non-fillers

Performance Metric

- Precision: Of all samples classified as X, how many are actually class X?
- Recall: Of all class X in the data, how many were classified as X?
- F_1-score: balance precision and recall

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}}$$

- F_β-score: tune towards precision or recall by a factor β
  - Generalized form of F_1-score
  - Using $\beta < 1$ favors precision over recall

$$F_\beta = \frac{\beta^2 + 1}{(\beta^2 \cdot \mathrm{recall}^{-1}) + \mathrm{precision}^{-1}}$$

Dataset Selection

TED-LIUM Dataset

- A dataset designed for training automatic speech recognition [1]
- 118 hours of English language TED talk audio with detailed transcriptions including speech disfluencies and silence
  - Filler words are specifically marked as {FILL}
  - Conveniently available in the Hugging Face datasets module
- Major challenge: filler words are imperfectly labeled
  - Many "ums" labeled incorrectly or completely absent from labeling

[1] A. Rousseau, P. Deléglise, and Y. Estève, "TED-LIUM: an automatic speech recognition dedicated corpus", in Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12), May 2012.

PodcastFillers Dataset

- A dataset specifically designed for filler word detection tasks [2]
- 145 hours of gender-balanced English language audio with more than 350 speakers represented split into 1-second clips
  - 35,000 filler samples: “uh”, “um”, “like”, “you know”, and other
  - 50,000 non-fillers: breathing, music, noise, laughter, normal speech, repeated words, etc.
- Major challenge: fillers are always at the center of audio clips

[2] G. Zhu, J. Caceres, and J. Salamon, “Filler Word Detection and Classification: A Dataset and Benchmark” 23rd Ann. Cong. Int. Speech Comm. Association (INTERSPEECH), Incheon, Korea, Sep. 2022.

Training Process

Transfer Model Selected

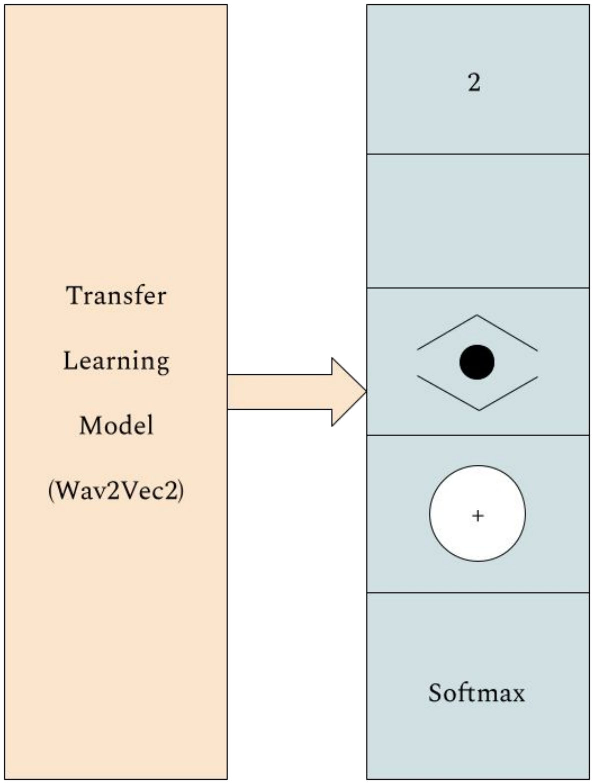
- Wav2Vec2 Basic Model [3]
  - Trained for general speech processing tasks
  - (Comparatively) small model to allow for low latency real-time applications

- General structure
  - 7 Conv1D layers: 512 kernels in each layer, kernel size from 10 to 3 to 2, stride from 5 to 2
  - Linear layer changes input shape from 512 to 768
  - 1 Conv1D layer: 768 kernels, kernel size is 128 with 64 padding, stride of 1
  - 12 encoder layers
    - 768 inputs and outputs to the self attention layers
    - 768 -> 3072 -> 768 in the final two feedforward layers
  - Custom classifier head for training as a filler word detector

[3] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, “wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations,” Oct. 22, 2020, arXiv: arXiv:2006.11477. doi: 10.48550/arXiv.2006.11477.

First Model - Cross Entropy Loss

- Each neuron creates a confidence score of filler or non-filler
- Max F0.5 achieved: 0.865 (49/50 epochs)
- Pro: Custom loss weights available for each class to help discourage false positives
- Con: Cross entropy loss is overkill for our two class problem, the calculation is more meaningful in a multiclass problem



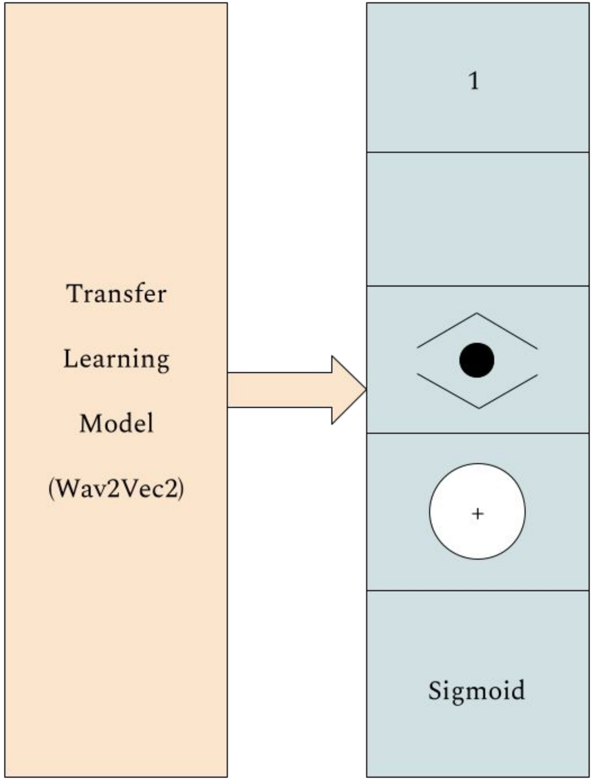
Second Model - Custom Binary Cross Entropy

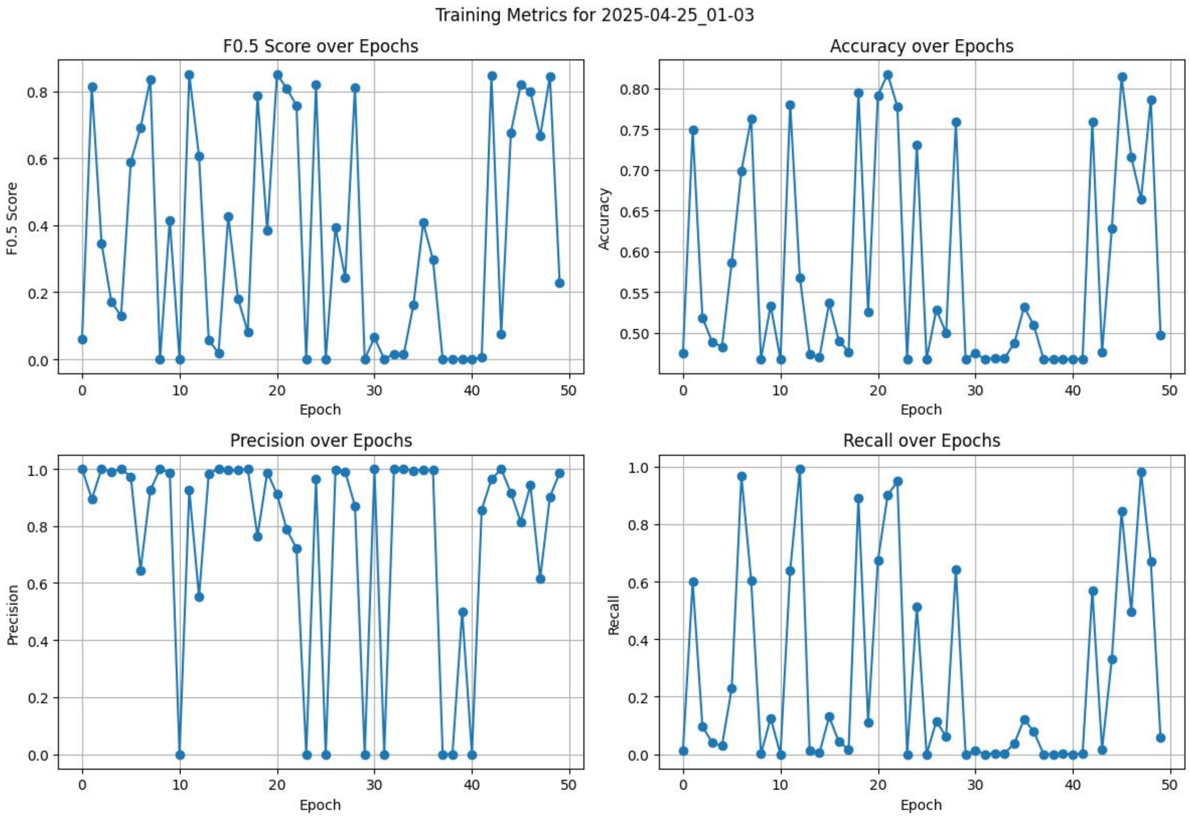
- Only one neuron, after a sigmoid activation function a custom binary cross entropy loss function is used

```
loss = -((fn_weight * targets * torch.log(probs + eps)) + (fp_weight * (1 - targets) * torch.log(1 - probs + eps)))
```

- Max F0.5 achieved: 0.851 (21/50 epochs)

- Unstable training

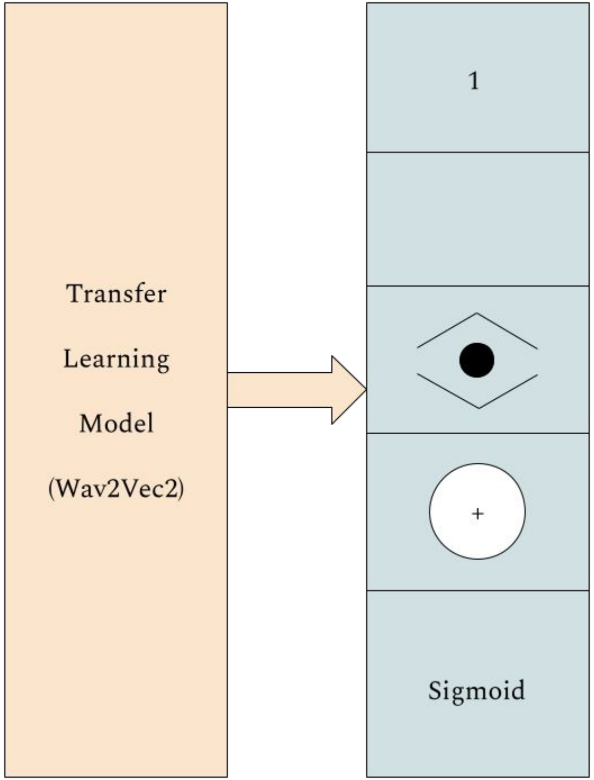


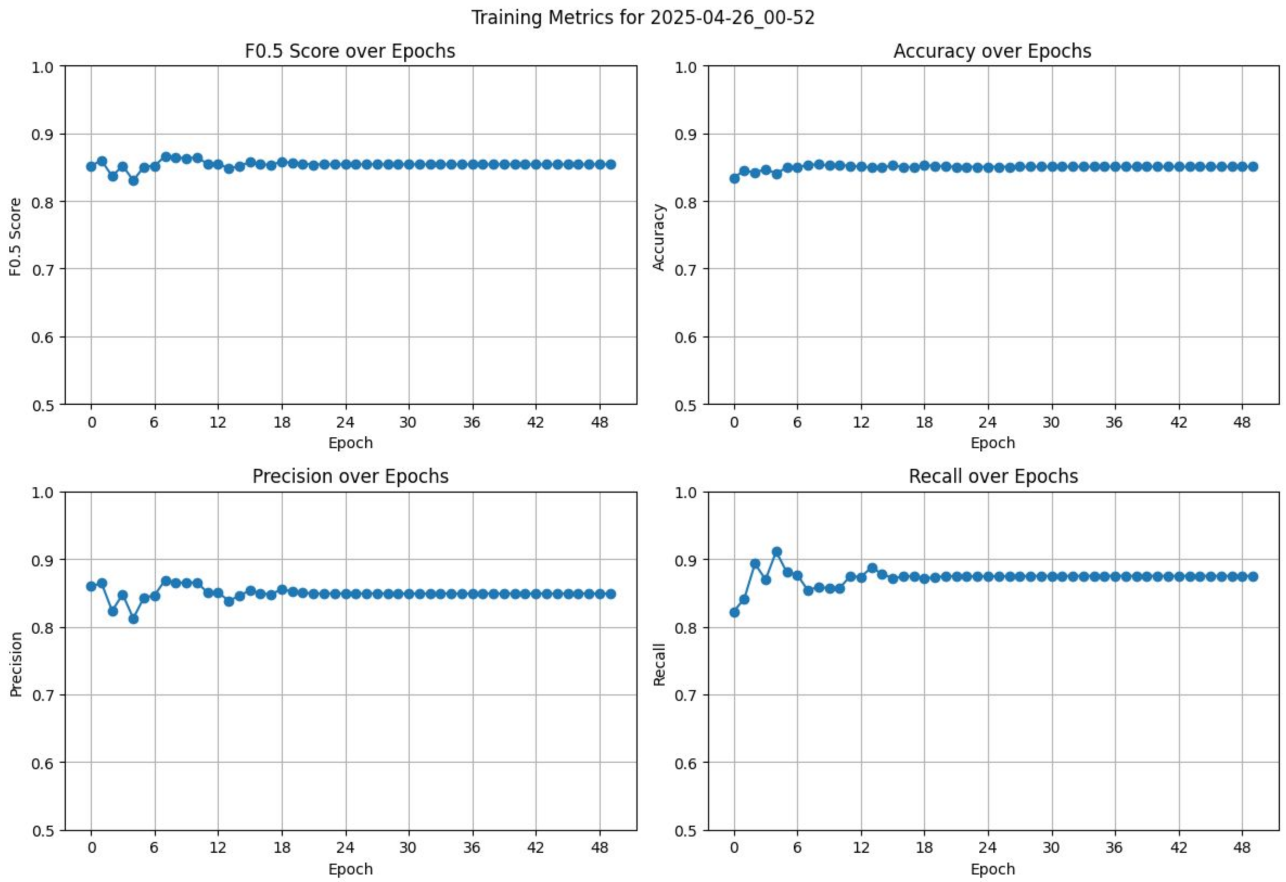


Model 2 - Unstable Training

Third Model - BCE with Logits

- Identical architecture using built in BCEwithLogitsLoss function
- pos_weight variable allows us to discourage false positives as desired
- Max F0.5 achieved: 0.866 (7/50 epochs) equivalent to CEL but faster to achieve
- Implemented a learning rate scheduler to stabilize training and an early terminate to save training time





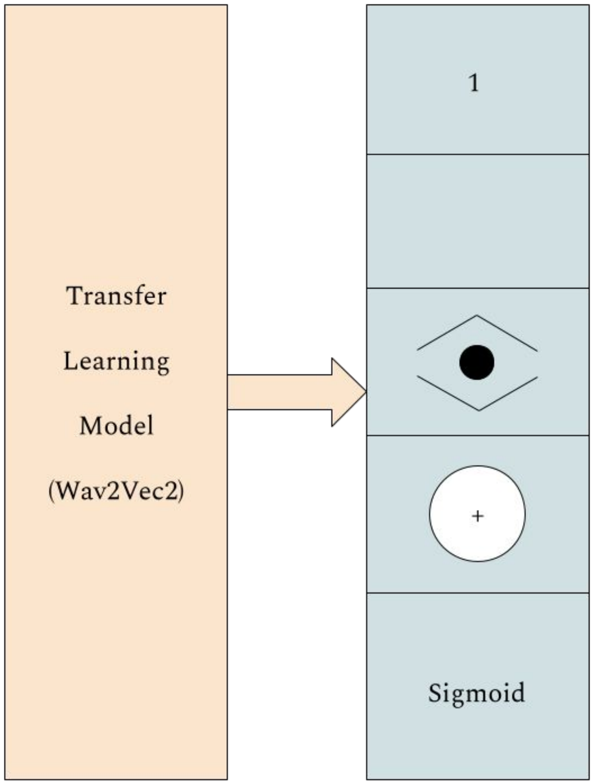
Model 3 - Training stabilizes due to LR Scheduler and standard BCE

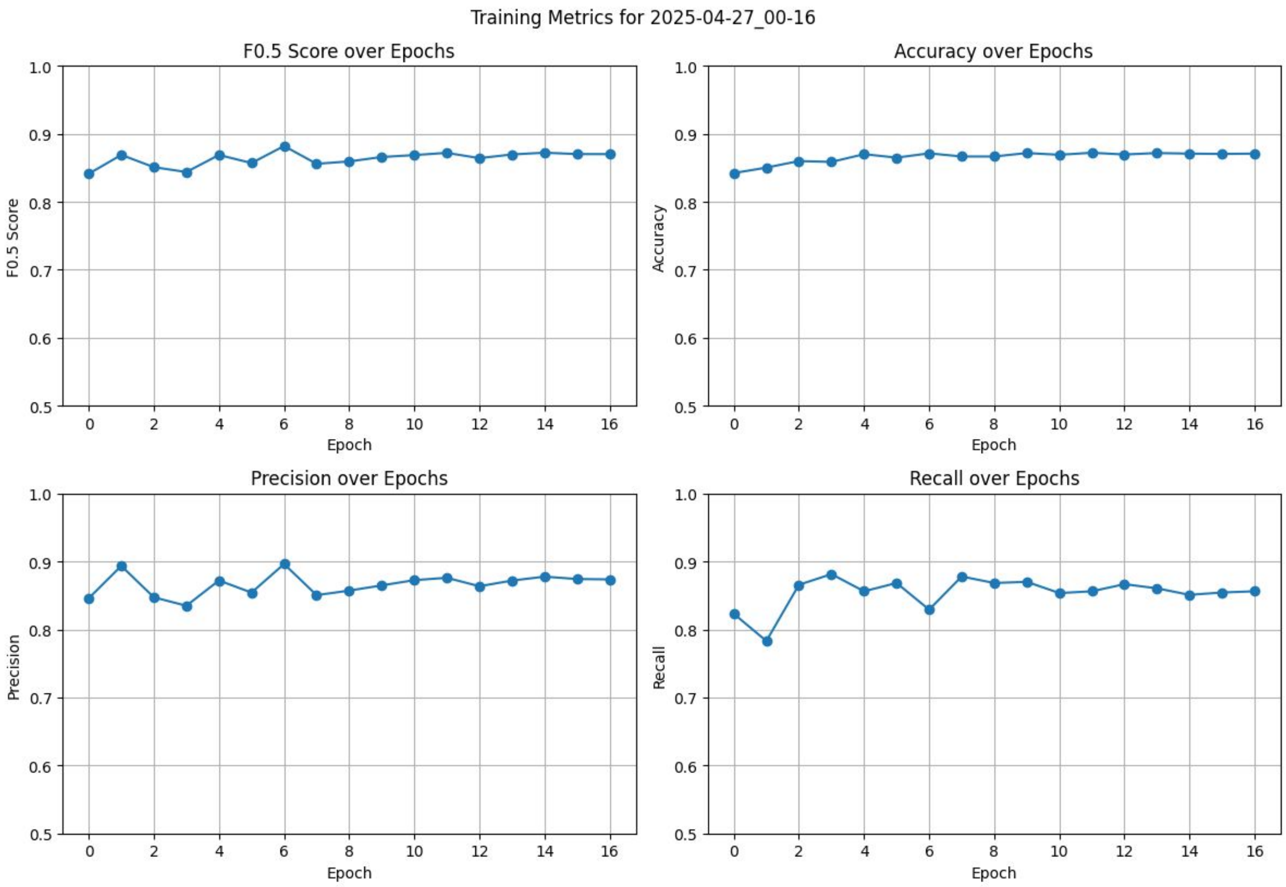
Data Augmentation Code

```python
def random_shift_audio(self, audio):  1 usage  andrew-ash
    """
    This cuts some audio from the front or back of the audio to force fillers to not be in the exact middle of samples.
    """
    shift = random.randint(-self.max_shift, self.max_shift)

    # The random value is between -max_shift and max_shift negative values pad the end and cuts audio from the front
    # the positive values pad the start and cut audio off the end.
    if shift > 0:
        audio = torch.cat([
            torch.zeros(shift, dtype=audio.dtype),
            audio
        ])[:16000]
    elif shift < 0:
        audio = torch.cat([
            audio[-shift:],
            torch.zeros(-shift, dtype=audio.dtype)
        ])[:16000]

    return audio
```

Fourth Model - Add Augmentation

- Augmentation reduces overfitting to fillers expected only in the middle of audio clips
- Max F0.5 achieved: 0.882 (7/17 epochs) an improvement of 0.016 from default data
- Performance has largely plateaued with the selected classifier head

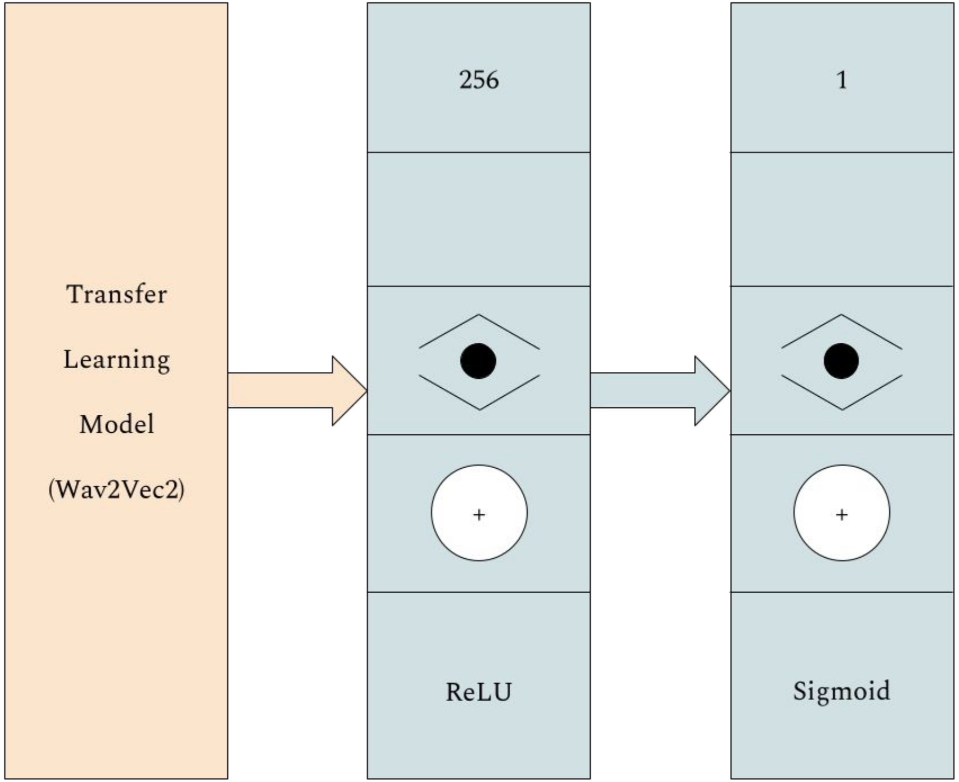


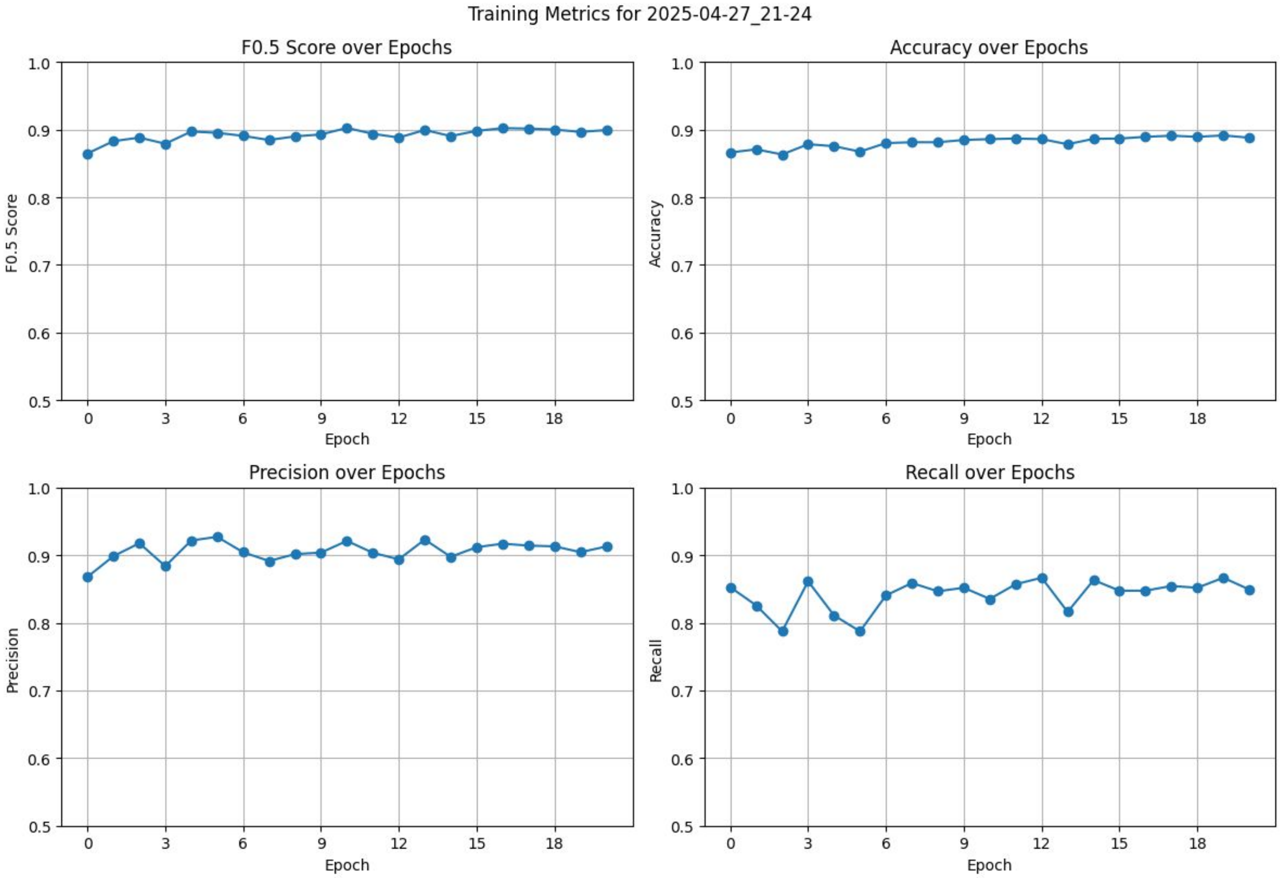


Model 4 - Augmentation yields small improvement

Final Model - Additional Linear Layer

- An additional linear layer allows for more feature extraction for the final decision layer
- Max F0.5 achieved: 0.903 (11/21 epochs) an improvement of 0.0201
- Precision/recall tradeoffs finally lead to an F0.5 > 0.9





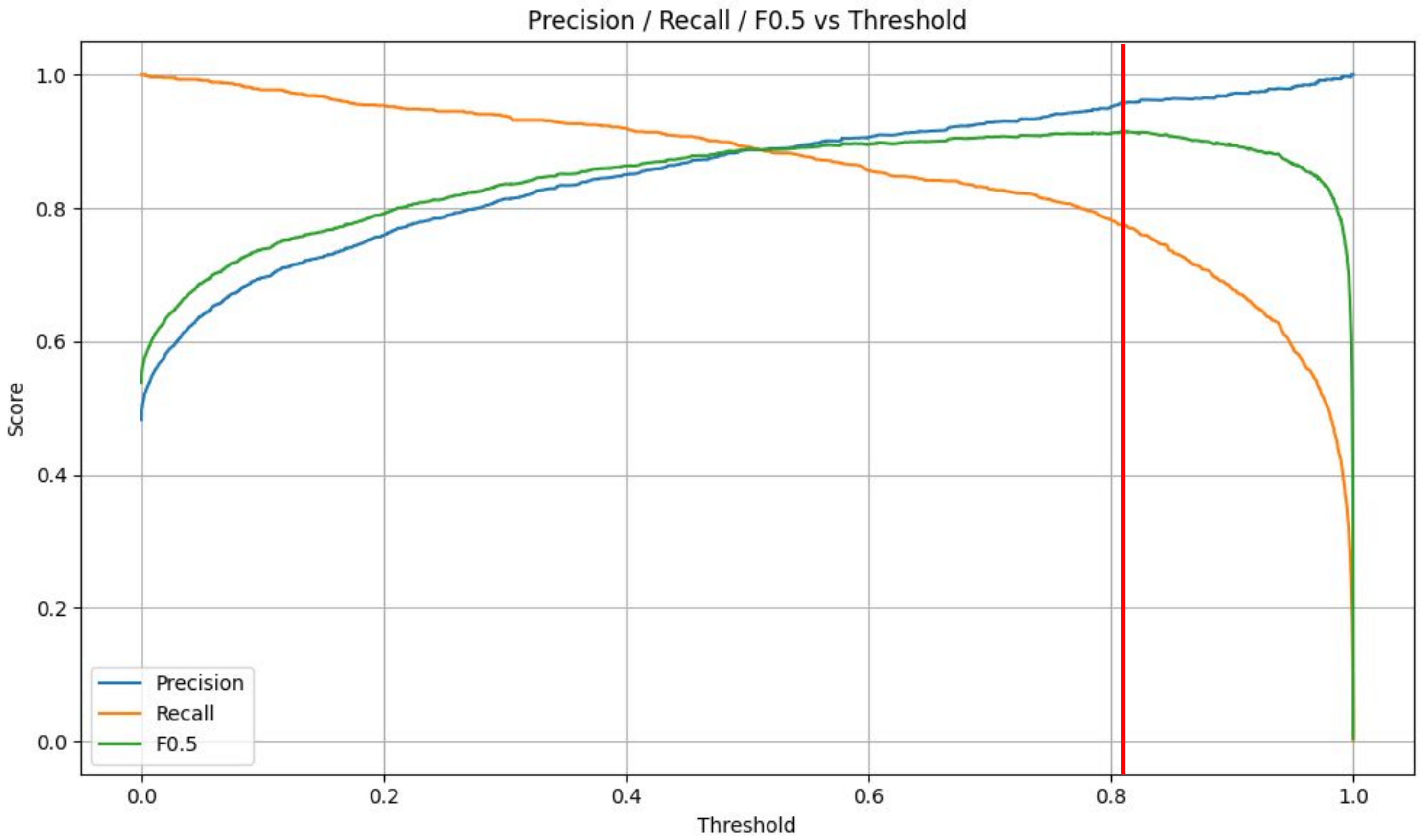
Model 5 - Deeper classifier has best F0.5 and is fairly stable

Final Improvement with Threshold Shifting

- Shifting the final model's threshold can achieve small F0.5 improvement (between 0.006 and 0.026) depending on the specific hyperparameters

- Threshold shifting was attempted with the following hyperparameters
  - Audio shifted by ∓2400 samples (150 ms)
  - Learning rate reduced by a factor of ⅓ after 2 consecutive epochs without F0.5 improvement
  - pos_weight of 0.5, 0.625, and 0.75

- Shifting the decision threshold to a value between 0.5 and 0.8 improves the F0.5 through improved precision at the cost of lower recall



Max F0.5: 0.915 at threshold of 0.809 (not recommended)

Working with Real-Time Audio

Input Data Doesn't Wait

- Robust filler word detection means we can't stop taking in audio to process it
- Real-time feedback requires some form of asynchronous processing
  - Some amount of latency is unavoidable
- Python's asyncio module provides utilities for real-time multiprocessing

```python
async def audiostream(channels=1, **kwargs):
    '''Generator yielding blocks of input audio data'''
    q_in = asyncio.Queue()
    loop = asyncio.get_event_loop()

    def callback(data, frame_count, time_info, status):
        loop.call_soon_threadsafe(q_in.put_nowait, (data.copy(), status))

    sd.default.device = 'Blue Snowball: USB Audio'
    stream = sd.InputStream(callback=callback, channels=channels, **kwargs)
    with stream:
        while True:
            data, status = await q_in.get()
            yield data, status
```

Breaking Input into Segments

```python
async def classify_audiostream(classifier, torch_device, threshold=0.5, **kwargs):
    '''Asynchronous task to sample & classify audiostream data as it arrives'''
    print('Waiting for filler words...')
    filler_count = 0
    async for audio, _ in audiostream(**kwargs):
        data = torch.unsqueeze(torch.flatten(torch.from_numpy(audio)), 0)
        outputs = classifier(data.to(torch_device))
        is_filler = float(outputs) > threshold
        filler_count += is_filler
        print(f'Filler detected! (total: {filler_count})') if is_filler else None
```

Classifying Asynchronous Input

Demonstration

Improvements and Recommendations

- Consider fine-tuning the full transfer model
  - Risks: Overfitting due to comparatively smaller PodcastFillers dataset, much slower training
  - Benefits: may further improve precision and/or recall
- Consider additional data augmentation
  - Pitch shifting, speaker speed adjustments, background noise, etc.
  - Add more datasets of similar data to create a more robust training process
  - Include datasets with different dialects and accents
- Add a model for detecting different speakers
  - Attribute each filler word to the different speakers
  - Generate a final report with the total count of each speaker's fillers
  - A much more challenging and complex task with additional benefits to the user

Questions