EE305 Module 3 Post-lab Report

학번: 20210203
이름: 노욱진

December 19, 2025

Contents

1 Week 1: Wave Function Classifier with Quantization

1.1 Part 1: Model Training

Task 1.1: Training Data Visualization

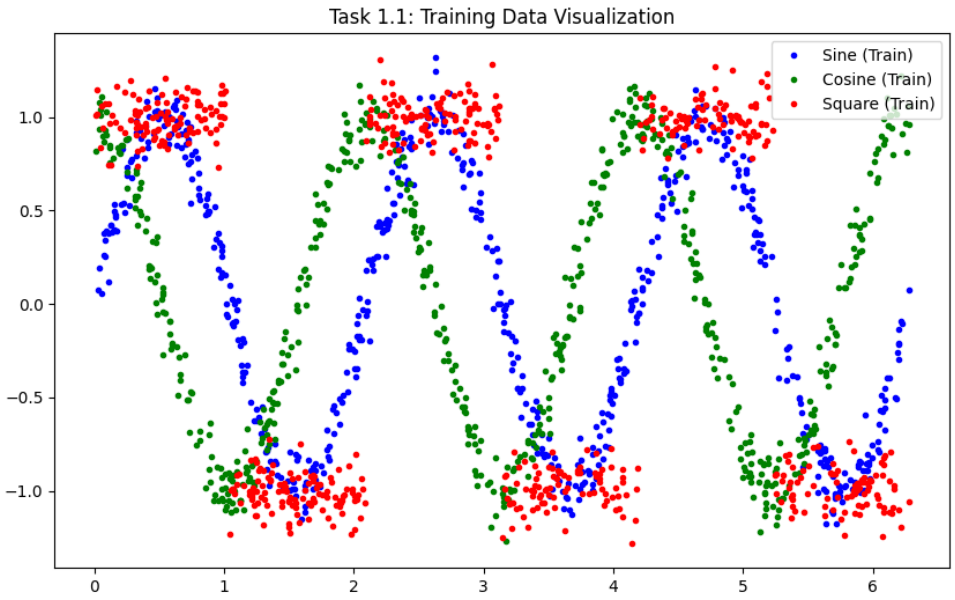


Figure 1: Visualization of training data for three wave functions

The plot of all three functions are the same as above. As we added 10% of noise, the sample data does not look line a linear line.

Task 1.2: Model Architecture and Training Analysis

1. Final Model Architecture

As the MAE of square model was over 0.2, we adjusted the model architecture to reduce the MAE value. And the final model architecture is the same as below.

```
    model = keras.Sequential([
        layers.Dense(32, activation='relu', input_shape=(1,)),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(32, activation='relu'),
        layers.Dense(1)
    ])
```

Listing 1: Logging Implementation

We used 4 layers with each 32 neurons. And the results of MAE are as following.

- MAE of sin: 0.0943
- MAE of cos: 0.0924
- MAE of square: 0.1412

2. Training and Validation Loss Plots

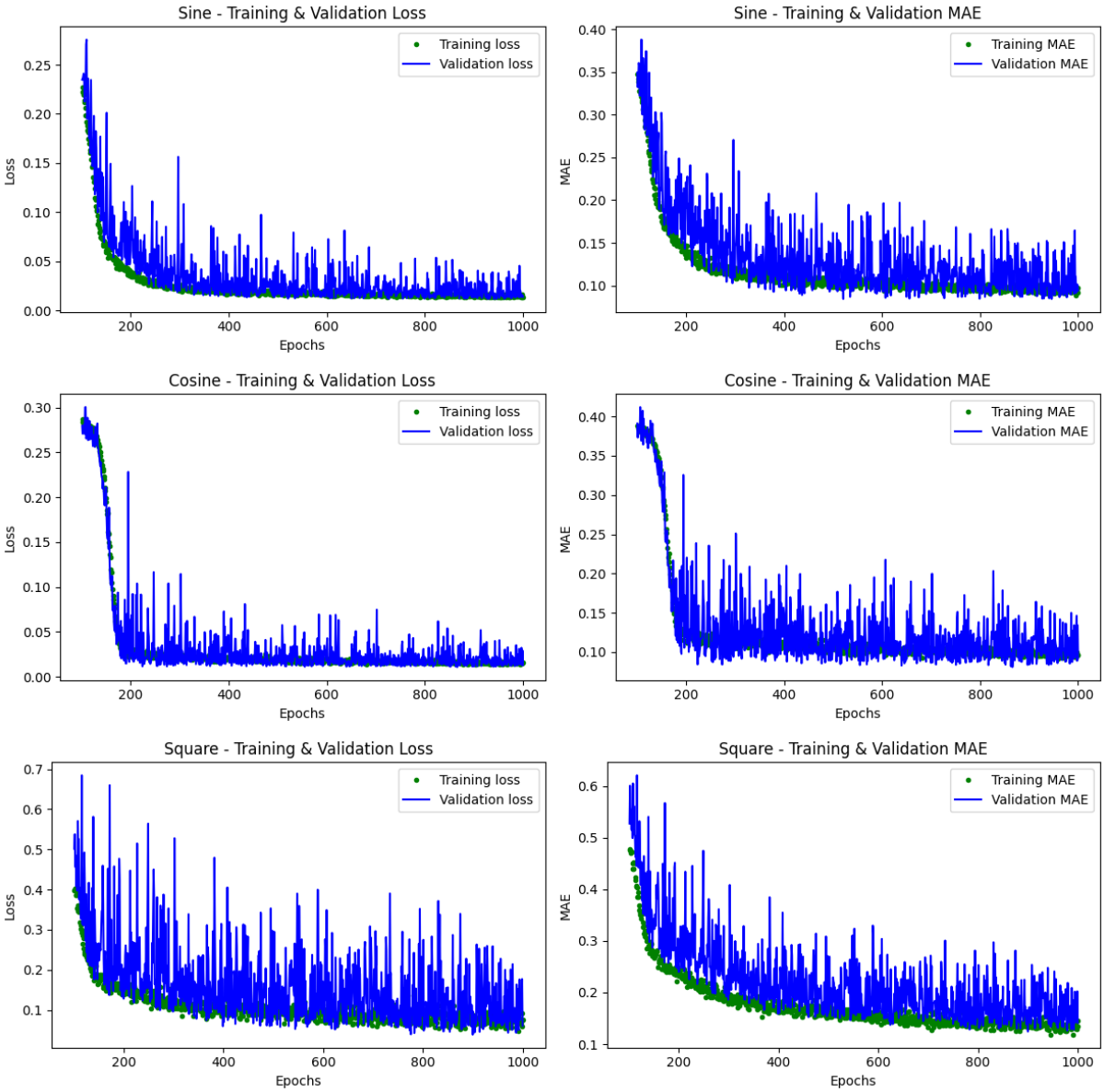


Figure 2: Training and Validation Loss for Sine, Cosine, and Square models

As shown in the graph, we skipped first 100 epochs. The hardest function to learn was square function. This is because the sine and cosine models converged quickly within approximately 200 epochs, whereas square model does not converge even 600 epochs.

3. Analysis of Learning Difficulty with Function Characteristics

I think there is an affection to model performance according to function characteristics, which is the reason why square model is the hardest function to learn.

The sine and cosine functions are continuous and smooth curves. Neural networks, which use activation functions like ReLU (essentially piecewise linear functions), are highly effective at approximating such smooth, continuous functions. Therefore, the model could easily learn the underlying patterns with minimal error.

The Square Wave, in contrast, contains discontinuities where the value jumps instantaneously from -1 to 1 (or vice versa). Neural networks fundamentally struggle to model these vertical jumps perfectly because they are designed to approximate continuous functions. Attempting to fit these sharp transitions results in significant errors and overshooting near the discontinuities. This confirms that discontinuous function characteristics significantly negatively affect model performance and increase learning difficulty compared to smooth functions.

1.2 Part 2 & 3: Deployment and Evaluation

Task 2.1: TFLite Quantization Analysis

The table below summarizes the Loss (MSE) and MAE values for the six models (Float32 vs. Int8) evaluated on Colab.

Waveform	Model Type	MAE	Loss (MSE)
Sine	Float32	0.086486	0.011965
	Int8	0.092977	0.013487
Cosine	Float32	0.088214	0.012395
	Int8	0.087182	0.011938
Square	Float32	0.125958	0.045870
	Int8	0.127423	0.049587

Table 1: Comparison of MAE and Loss for Float32 and Int8 models on Colab

The comparison between the Float32 and Int8 models reveals that quantization had a negligible impact on model accuracy.
For the sine and square models, the MAE increased very slightly (e.g., sine MAE difference is approx. 0.006), representing a tiny loss in precision due to the conversion from 32-bit floating-point numbers to 8-bit integers.
Interestingly, for the Cosine model, the Int8 version performed comparably (or even marginally better in this specific test instance) to the Float32 version.
This suggests that Int8 models are highly effective for deployment on embedded systems where storage and memory are limited.

Task 3.1: Raspberry Pi Benchmarking

Dataset	Type	Size (B)	Time (ms)	SD (ms)	MAE	RMSE	Max Err
Sine	F32	15,696	0.0378	0.0006	0.044787	0.055580	0.170490
	INT8	9,896	0.0460	0.0054	0.060421	0.063369	0.130814
Cosine	F32	15,700	0.0382	0.0053	0.054481	0.063369	0.130814
	INT8	9,896	0.0462	0.0005	0.042426	0.054184	0.163537
Square	F32	15,724	0.0385	0.0054	0.100340	0.236824	1.479648
	INT8	9,896	0.0487	0.0086	0.091095	0.209435	1.215061

Table 2: Detailed performance comparison of Wave Function Classifiers on Raspberry Pi (F32: Float32, INT8: Quantized).

1. **How does performance on the Pi differ from performance in Colab?:** In terms of inference time, the Pi shows high efficiency inference capabilities, averaging around 0.04ms per one inference. However, as Colab uses more powerful GPU/CPU on its server, it performs faster than Pi does.
In terms of accuracy, the MAE on the Pi are lower (better) than in Colab. This is likely because the Colab test dataset included 10% random noise, meaning the error metric included the noise variance. In contrast, the Pi benchmarking script likely evaluated the model against the ground

truth (pure mathematical function) or a generated dataset with less noise, resulting in a lower Mean Absolute Error.

2. **How quantization impacts performance:**

Model Size: Quantization reduced the model size by approximately 37% (from 15.7 KB to 9.9 KB).

Accuracy:

- Sine: Quantization slightly degraded performance (MAE increased from 0.045 to 0.060), as expected due to precision loss.
- Cosine & Square: Interestingly, the INT8 models showed better performance metrics (lower MAE and RMSE). For instance, the Square INT8 model had a Max Error of 1.21 compared to 1.48 for Float32. This suggests that the discrete nature of quantized weights may have inadvertently helped regularize the model or smooth out some noise in the test data for these specific functions. Inference Speed: Interestingly, the Float32 models ran faster (0.038 ms) than the INT8 models (0.046 ms). This indicates that the Pi's specific CPU or TFLite runtime is likely optimized for floating-point calculations, or the overhead of quantization/dequantization operations during inference slightly outweighed the benefits of integer arithmetic for such tiny models.

Task 3.2: Deployment Recommendation

I'll choose Float32 Model (Sine or Cosine depending on application) according to the result.
Why?

Model Size (Storage): Although INT8 saves about 6 KB, this difference is negligible for most modern microcontrollers (which typically have 256 KB+ flash). The small storage saving does not justify the increased latency and power consumption observed with the INT8 models. Therefore, optimizing for speed (Float32) is the superior trade-off in this context.

Power Consumption (Inference Speed): In this specific benchmark, the Float32 models consistently executed about 20% faster than the INT8 models (0.038 ms vs 0.047 ms). Since the device must wake up, process, and sleep to save power, a shorter execution time leads to a lower active duty cycle and thus longer battery life.

Accuracy: Float32 generally guarantees higher theoretical precision. While INT8 performed well here, Float32 is a safer choice for consistent reliability without quantization noise.

2 Week 2: Wake Word Detection

2.1 Part 1: Data Collection

Task 1.1: Recording Script Analysis

- **RMS Threshold:** The RMS threshold used to distinguish speech from silence is -40dB.
- **Sample Rate (16kHz vs 48kHz):** We use target sample rate 16kHz instead of native 48kHz because of efficiency and resource constraints. First, it is efficient because 16 kHz is the standard sampling rate for speech recognition tasks. According to the Nyquist theorem, a sampling rate of 16 kHz can capture frequencies up to 8 kHz, which covers the majority of human speech information. Secondly, it is because of resource constraints of embeddedd systems. Using the native 48 kHz would result in data files that are three times larger, significantly increasing storage requirements and computational load for feature extraction (MFCC) and inference. This is critical for resource-constrained embedded systems like TinyML.
- **Amplification (30dB):** We amplify recorded audio by 30dB because of microphone sensitivity and signal normalization. Firstly, the raw input signal captured by the MEMS microphone on the voicehat is typically very quiet. Secondly, without amplification, the speech signal would be too weak for the model to learn effective features. Boosting the signal ensures that the audio levels fall within a usable range (typically -30 dB to -10 dB) for robust keyword spotting.

Task 1.2: Data Summary and Spectrograms

The following is a summary table of our recorded data. The recording has done with room ambient except for Noise category.

Category	Final	Removed	Common Removal Reasons	Variation Strategies
Wake Word	110	134	Truncated start/end, Low amplitude (<-30dB)	Varied speed (fast/slow), Pitch (high/low), Distance (lean in-/out), Emphasis on specific syllables
Other Words	135	1	Triggered by the interference from background noise	Phonetically similar words (“Hey Google”, “Hey Siri”, ”Hey zuki”), Common greetings, Commands
Noise	102	0	-	Silence (room ambient), Keyboard typing, Music playing, Door closing, Footsteps, Chair sound, Chatting etc.

Table 3: Summary of Recorded Dataset and Data Collection Process

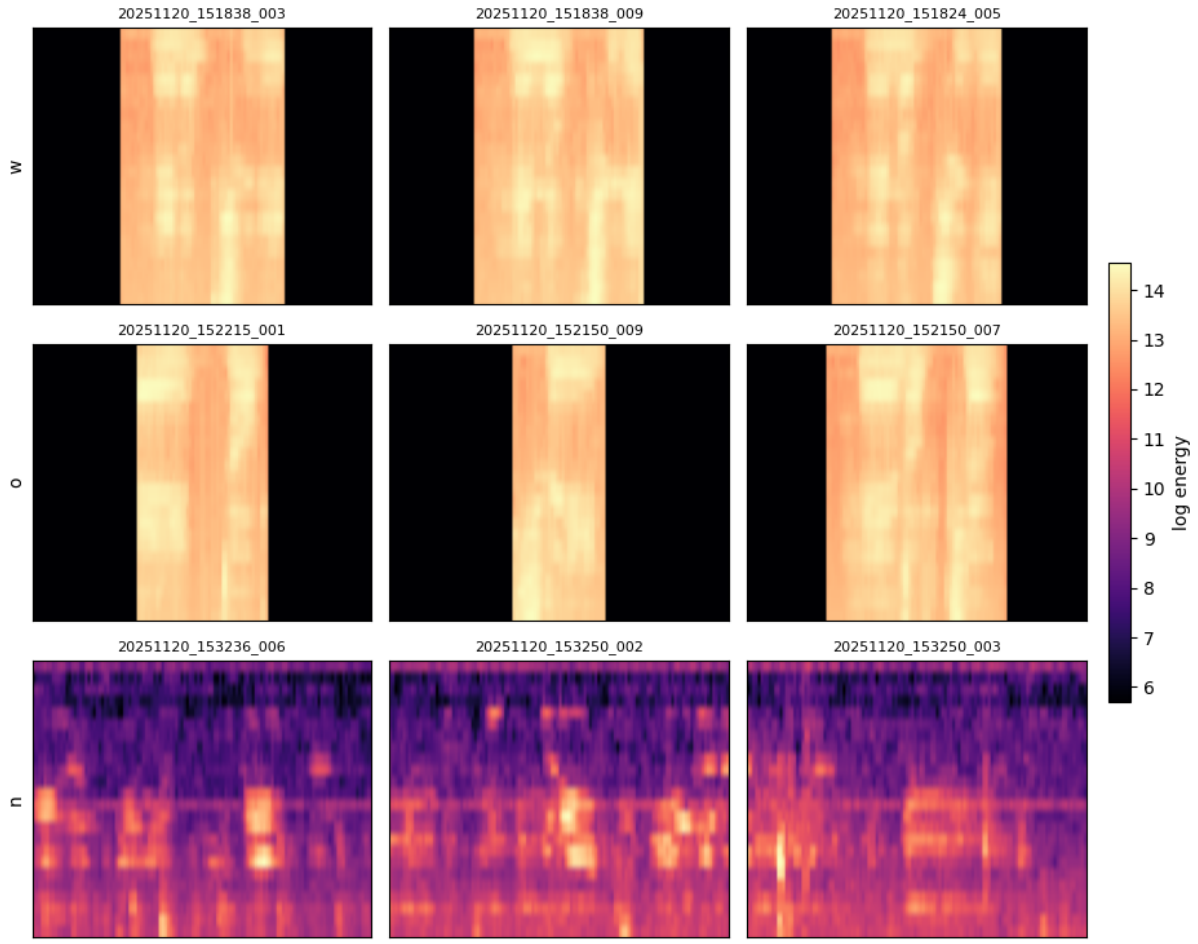


Figure 3: Mel spectrogram of 3 samples of each w, o, n

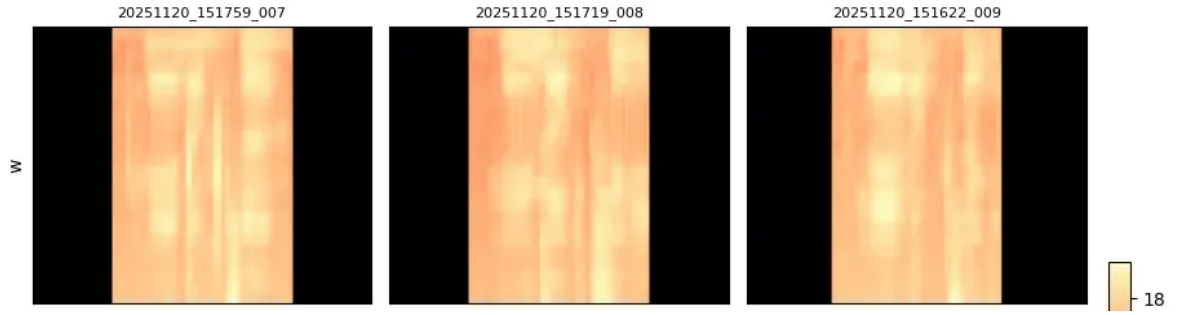


Figure 4: Mel spctrogram of 3 samples of w of partner

In Figure 3, it shows a distinguishable patterns between the categories and similarities within the wake word category.
In the case of wake word(w), the samples ("Hey Nubzuki", top row of Figure 3) display a highly consistent visual signature. We can observe a sequence of distinct high-energy vertical bands. These correspond to the distinct syllables of the phrase (e.g. "Hey-Nub-Zu-Ki"). This structural consistency is maintained across the samples, making it a distinct feature for the model to learn. On the other hand, the 'Other Words' samples (middle row of Figure 3) show high-energy areas similar to the wake word, indicating the presence of speech. However, the temporal structure varies significantly between samples. Unlike the wake word, there is no consistent repeating pattern; some samples show a single block of energy (monosyllabic words), while others show irregular spacing, reflecting the variety of different words recorded in this category. Also, the noise samples (bottom row of Figure 3) are easily distinguishable from speech. They exhibit low

energy levels across the time axis. Unlike speech, there are no distinct vertical structures; instead, we observe faint, continuous horizontal bands or random static, likely representing consistent background sounds like air conditioning.
Considering figure 4, which contains mel spectrogram of wake word of partner, we can see that, despite likely differences in pitch, tone, and speaking speed compared to my own voice, the fundamental spectrographic "fingerprint" remains very similar. The sequence of energy bursts corresponding to the four syllables is clearly visible and follows the same temporal arrangement as in Figure 3. This suggests that the Mel spectrogram captures the phonetic content effectively, implying that a model trained on this data has the potential to generalize well across different speakers.

2.2 Part 2: Model Training

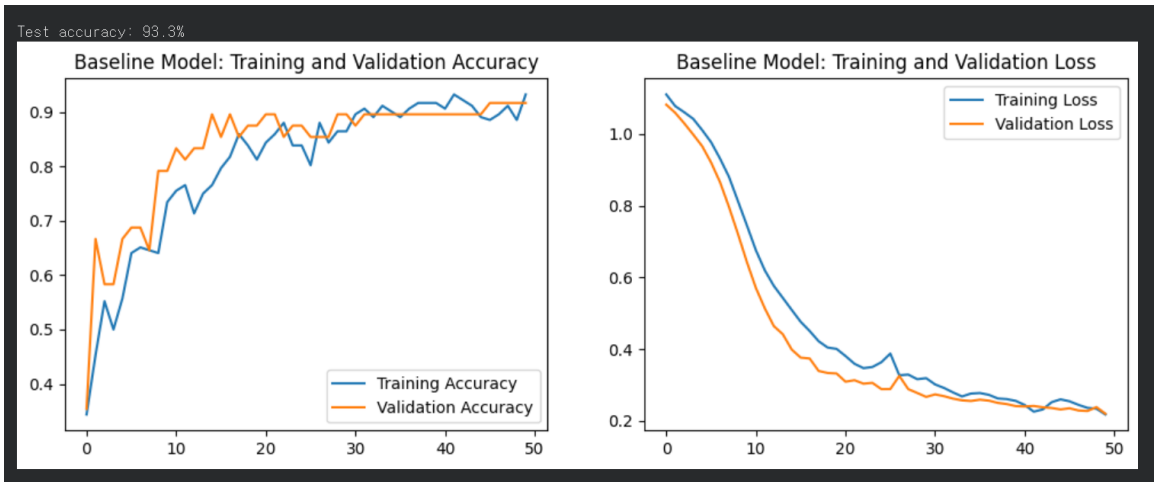
Task 2.1: Model Performance Analysis

Model Variant	Training Samples	Test Accuracy	Converged Epoch
Baseline	240	93.3%	50
Augmented	1200	90.3%	36

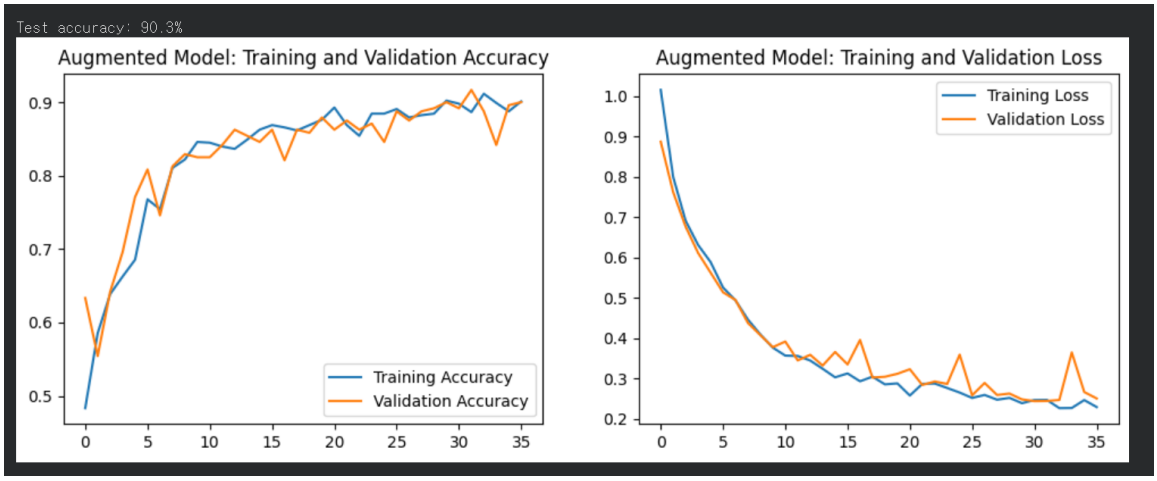
Table 4: Characteristics and Performance of Trained Models

- Does augmentation improve test accuracy?: Unexpectedly, in this specific training run, augmentation did not improve the test accuracy on the provided test set. The Baseline model achieved 93.3%, while the Augmented model achieved 90.3%. We can think about a few possibilities of this counter intuitive result.
  1. Small Dataset Overfitting: The baseline model (trained on only 240 samples) might be lucky with the small test set (60 samples), effectively overfitting to specific features present in both.
  2. Difficulty of Augmented Data: The augmented dataset introduces significant variations (noise, shifts, volume changes), making the learning task much harder. The model trained on this noisy data might perform slightly worse on "clean" test samples but would likely be much more robust in real-world scenarios with background noise.
  3. Training Stability: The loss curves show that the augmented model's validation loss is much lower and more stable than the baseline's validation loss, which fluctuates. This indicates that while the absolute accuracy on this specific test set is lower, the augmented model is actually generalizing better and is less prone to overfitting.

- Training time comparison: Augmentation increased training time. The Baseline model took approximately 2 minutes (50 epochs, 2.4s/epoch), while the Augmented model took around 8 minutes (early stopping at epoch 36, 13.3s/epoch). This is expected as the dataset size increased by 5 times.



(a) Baseline Model: Training and Validation Accuracy/Loss



(b) Augmented Model: Training and Validation Accuracy/Loss

Figure 5: Training progress comparison between Baseline and Augmented models.

For Figure 5, I revised the TrainWakeWordSimple.ipynb file as following.

```
import matplotlib.pyplot as plt

# Function to visualize training history
# This plots the training/validation accuracy and loss over epochs.
def plot_history(history, title):
    # Extract accuracy and loss metrics from the history object
    acc = history.history['accuracy']
    val_acc = history.history['val_accuracy']
    loss = history.history['loss']
    val_loss = history.history['val_loss']
    epochs_range = range(len(acc))

    # Set up the figure size
    plt.figure(figsize=(12, 4))

    # Plot Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, acc, label='Training Accuracy')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title(f'{title}: Training and Validation Accuracy')
```

```
22
23      # Plot Loss
24      plt.subplot(1, 2, 2)
25      plt.plot(epochs_range, loss, label='Training Loss')
26      plt.plot(epochs_range, val_loss, label='Validation Loss')
27      plt.legend(loc='upper right')
28      plt.title(f'{title}: Training and Validation Loss')
29
30      # Display the plots
31      plt.show()
32
33  # ------------------------
34
35  # Define two scenarios: (Scenario Name, Augmentation Flag)
36  scenarios = [
37      ("baseline", False),    # 1. No Augmentation (Baseline)
38      ("augmented", True)     # 2. With Augmentation (Augmented)
39  ]
40
41  # Iterate through each scenario to train and convert models
42  for name, use_aug in scenarios:
43      print(f"\n\n==================================================")
44      print(f"Training Scenario: {name.upper()} (Augmentation={use_aug})")
45      print(f"==================================================")
46
47      # 1. Train model and obtain training history
48      # 'train_model' likely returns the trained keras model, test data, and
        the history object
49      model, X_test, y_test, history = train_model(augment=use_aug)
50
51      # 2. Visualize learning curves (Satisfies Requirement 2)
52      # Passes the history object to the function defined above
53      plot_history(history, title=f"{name.capitalize()} Model")
54
55      # 3. Convert to Float32 TFLite (No Quantization)
56      # Converts the model to TFLite format maintaining 32-bit floating point
         precision
57      print(f"\n--- Converting {name} model to Float32 ---")
58      float_path = convert_to_tflite(model, X_test, model_name=name, quantize
        =False)
59
60      # Evaluate the Float32 model
61      test_tflite(float_path, X_test, y_test)
62
63      # 4. Convert to Int8 TFLite (With Quantization)
64      # Converts the model to TFLite format using 8-bit integer quantization
        for efficiency
65      print(f"\n--- Converting {name} model to Int8 (Quantized) ---")
66      quant_path = convert_to_tflite(model, X_test, model_name=name, quantize
        =True)
67
68      # Evaluate the Int8 model
69      test_tflite(quant_path, X_test, y_test)
70
71  # Notify that the entire pipeline is complete
72  print("\nAll 4 models have been generated and saved to:", OUTPUT_PATH)
```

Listing 2: Logging Implementation

- Less overfitting?: The Augmented model (b) demonstrates a more stable validation loss curve and a smaller gap between training and validation accuracy compared to the Baseline model (a), indicating reduced overfitting.

Model Name	File Size (KB)	Test Accuracy (%)	Avg Inference Time (ms)
Baseline Float32	22.6	93.3	0.11
Baseline Int8	13.8	93.3	0.13
Augmented Float32	22.6	90.3	0.06
Augmented Int8	13.8	91.3	0.09

Table 5: Performance Comparison of All Four Model Variants

- Benefit/cost of quantization:
  Benefit: Quantization successfully reduced the model size by approximately 40% (from 22.6 KB to 13.8 KB). This is crucial for saving flash memory on embedded devices like the Raspberry Pi.
  Cost: The accuracy loss was minimal. For the Augmented model, quantization actually increased accuracy slightly, likely due to the regularization effect of lower precision. For the Baseline model, accuracy remained identical (93.3%).
  Therefore, the quantization loss is absolutely acceptable for a wake word application. We gain significant storage efficiency with virtually no penalty in performance.

- Impact of Data Augmentation: Data augmentation did not improve the test accuracy in this specific Colab training run (Baseline: 93.3% vs. Augmented: 90.3%).
  Though, I think augmentation helped in prevention of overfitting and making model more robust. The slight drop in accuracy is a trade-off for learning invariance to noise, volume changes, and time shifts. This robustness is crucial for the subsequent real-world deployment on the Raspberry Pi, where the input audio will inevitably contain background noise and variations that the Baseline model never encountered.

- Deployment recommendation: Based on these Colab results, I would choose the Augmented Int8 Model for 2 reasons.
  1. Robustness: Although its test accuracy (91.3%) is slightly lower than the Baseline (93.3%), the Augmented model has learned from a much larger and more diverse dataset. In a real-world deployment (with background noise, different distances), the Baseline model is highly likely to fail, whereas the Augmented model will be more resilient.
  2. Efficiency: The Int8 version is the smallest (13.8 KB) and fast (0.09ms inference time), making it the most efficient choice for a battery-powered embedded system.

2.3 Part 3: Raspberry Pi Deployment and Evaluation

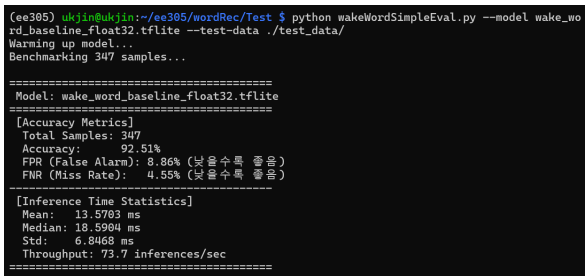
Task 3.1: Measurements and cross-user performance

For Pi evaluation with Accuracy, FPR, FNR, Inference time, I revised the given wakeWordSimpleEval.py as following.

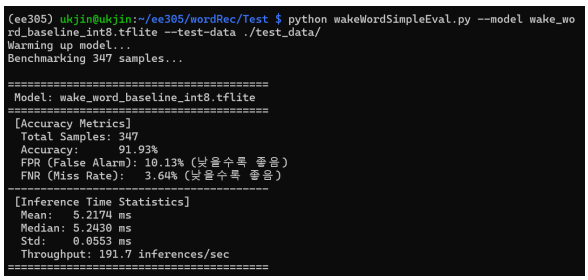
```
        # Accuracy Check
        # Compare the model's prediction with the actual ground truth label
        if prediction == actual:
            correct += 1

        # FPR / FNR Calculation
        # Determine True Positives, False Negatives, False Positives, and True Negatives
        if actual == WAKE_WORD_IDX:
            # Case: The actual sound was the Wake Word
            if prediction == WAKE_WORD_IDX:
```

```
                tp += 1 # True Positive: Correctly detected
            else:
                fn += 1 # False Negative: Missed the wake word
        else: # Actual is 'other' or 'noise'
            # Case: The actual sound was NOT the Wake Word
            if prediction == WAKE_WORD_IDX:
                fp += 1 # False Positive: False Alarm (incorrectly triggered)
            else:
                tn += 1 # True Negative: Correctly ignored

    # === Result Calculation ===
    # Calculate overall accuracy
    accuracy = correct / len(test_samples) * 100

    # FPR = FP / (FP + TN)
    # False Positive Rate: Percentage of non-wake words incorrectly detected as
    wake words
    fpr = (fp / (fp + tn) * 100) if (fp + tn) > 0 else 0.0

    # FNR = FN / (FN + TP)
    # False Negative Rate: Percentage of actual wake words that were missed
    fnr = (fn / (fn + tp) * 100) if (fn + tp) > 0 else 0.0

    # Time Stats
    # Calculate statistical metrics for inference time using NumPy
    mean_time = np.mean(times)
    std_time = np.std(times)
    median_time = np.median(times)

    # === Print Report ===
    # Display the final performance report
    print(f"\n========================================")
    print(f" Model: {os.path.basename(model_path)}")
    print(f"========================================")
    print(f" [Accuracy Metrics]")
    print(f"  Total Samples: {len(test_samples)}")
    print(f"  Accuracy:      {accuracy:.2f}%")
    print(f"  FPR (False Alarm): {fpr:.2f}% (Lower is better)")
    print(f"  FNR (Miss Rate):   {fnr:.2f}% (Lower is better)")
    print(f"----------------------------------------")
    print(f" [Inference Time Statistics]")
    print(f"  Mean:   {mean_time:.4f} ms")
    print(f"  Median: {median_time:.4f} ms")
    print(f"  Std:    {std_time:.4f} ms")
    # Throughput represents how many inferences can be performed per second
    print(f"  Throughput: {1000/mean_time:.1f} inferences/sec")
    print(f"========================================\n")
```

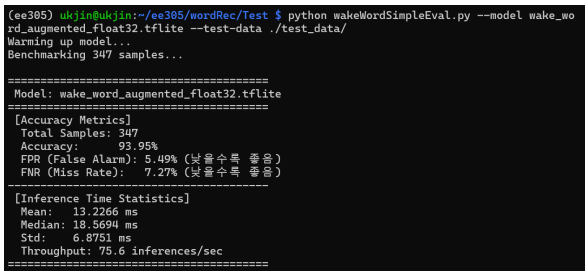
Listing 3: Logging Implementation



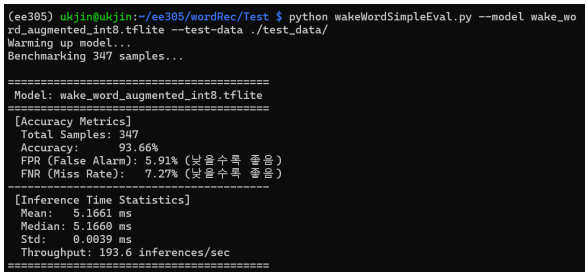
(a) Own: Baseline Float32



(b) Own: Baseline Int8

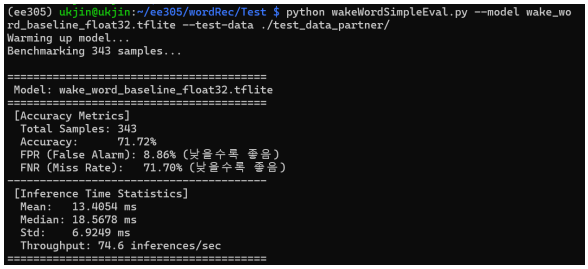


(c) Own: Augmented Float32

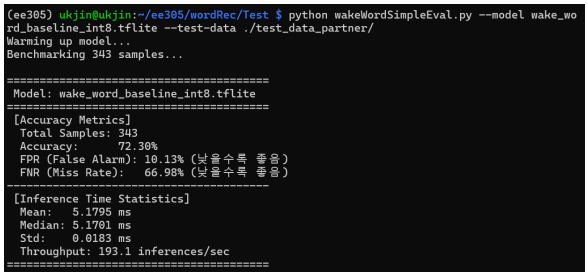


(d) Own: Augmented Int8

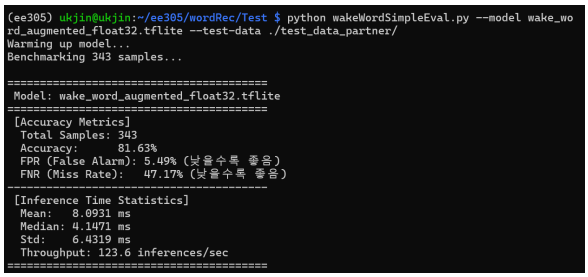
Figure 6: Benchmarking results on **Own Data**. Int8 models demonstrate significantly faster and more stable inference times.



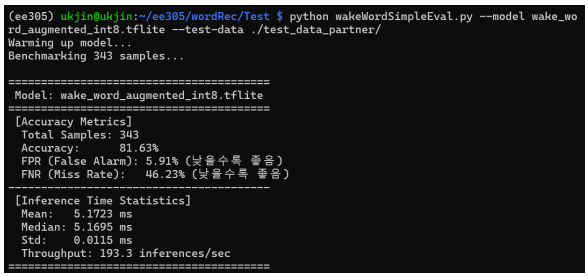
(a) Partner: Baseline Float32



(b) Partner: Baseline Int8



(c) Partner: Augmented Float32



(d) Partner: Augmented Int8

Figure 7: Benchmarking results on **Partner Data**. The Baseline model suffers a severe drop in accuracy (high FNR), while the Augmented model retains better performance.

Table 6 presents the comprehensive benchmarking results, comparing performance on the user's own voice versus the partner's voice.

- Performance Comparison: The models consistently performed significantly better on my own voice data compared to my partner's voice. On my partner's data, the Baseline model's accuracy dropped drastically (to around 70%), with a very high FNR (over 70%). This means the model failed to detect the wake word more than half the time for a new speaker.

Model Variant	Inference Time (ms)			Own Data Performance			Partner Data Performance		
	Mean	SD	Med	Acc	FPR	FNR	Acc	FPR	FNR
Base (F32)	13.57	6.85	18.59	92.51%	8.86%	4.55%	71.72%	8.86%	71.70%
Base (Int8)	5.22	0.06	5.24	91.93%	10.13%	3.64%	72.30%	10.13%	66.98%
Aug (F32)	13.23	6.88	18.57	93.95%	5.49%	7.27%	81.63%	5.49%	47.17%
Aug (Int8)	5.17	0.00	5.17	93.66%	5.91%	7.27%	81.63%	5.91%	46.23%

Table 6: Benchmarking results on Raspberry Pi

- Model Generalization: This discrepancy suggests that the Baseline model has poor generalization capabilities. It likely overfitted to the specific pitch, intonation, and speed of my voice and the recording environment.
  The improved performance of the Augmented model indicates that data augmentation forces the model to learn more generalized phonetic features rather than memorizing specific waveforms. However, even the augmented model struggled with the unseen speaker to some extent, highlighting the challenge of speaker-independent keyword spotting with small datasets.

- Impact of Augmentation and Quantization:
  Data Augmentation: Had a major positive impact on generalization. It significantly reduced the gap between own-voice and partner-voice performance. By exposing the model to noise and variations during training, it became more robust to the natural variations in a different speaker's voice.
  Model Quantization: Had virtually no impact on generalization performance (accuracy differences were negligible, often ¡1%). However, it improved inference speed (approximately 2.5x faster) and stability (lower standard deviation) on the Raspberry Pi, making it crucial for real-time deployment.

- Deployment to New Users: If deploying this model to new users, the current performance is likely insufficient. To improve this:
  Train on a Multi-Speaker Dataset: The training set must include a diverse range of voices (different genders, ages, accents) to capture the universal phonetic structure of the wake word.
  On-Device Transfer Learning: Implement a "calibration" feature where a new user records the wake word 3-5 times upon setup, and the device fine-tunes the model's final layers to adapt to their specific voice profile.

3 Week 3: Wake Word Deployment

3.1 Part 2: Logging Implementation

Task 2.1: Detection Parameters

1. Confidence Threshold: You will get more false positives.
The confidence threshold determines the minimum probability required for the model to consider an audio chunk as the wake word. Lowering this value makes the system more sensitive, allowing it to trigger even when it is less certain. While this might help detect mumbled or quiet wake words (reducing false negatives), it significantly increases the likelihood that background noise or similar-sounding words will be incorrectly identified as the wake word (increasing false positives).

2. Consecutive Detections: We require multiple consecutive detections to prevent false positives caused by transient noise spikes.
In real-time audio processing, a single frame (e.g., 40ms) might produce a high confidence score due to a sudden loud noise (like a door slam) or a short phonetic match (like "Hey"). However, a genuine wake word like "Hey Nubzuki" spans a longer duration. By requiring the confidence to remain high for several frames in a row (e.g., 3-5 frames), the system confirms that the detected sound has the temporal consistency of a real speech command, effectively filtering out momentary glitches.

3. Cooldown Frames: There might occur double detections.
After a wake word is detected, the audio signal of that word might still be present in the sliding window buffer for a short time. If the cooldown period is shorter than the duration of the wake word remaining in the buffer, the state machine might immediately return to the 'detecting' state and trigger again on the tail end of the same word. A sufficient cooldown ensures the system waits until the current utterance has completely passed before listening for a new one.

Task 2.2: Logging Code

Following is the code snippets I revised.

```
# ============================================
# 5. STREAMING SYSTEM
# ============================================
class StreamingSystem:
    # ... (previous methods like amplify, resample, etc.) ...

    def __init__(self, model_path, threshold, consecutive, cooldown, profile=False):
        # ... (loading model and setting up audio parameters) ...

        # --- Step 1: Log File Initialization ---
        self.log_file = None
        if self.profile:
            # Generate a unique filename using the current timestamp
            fn = 'log_' + datetime.datetime.now().strftime("%Y%m%d_%H%M%S") + ".csv"
            try:
                # Open the file in write mode
                self.log_file = open(fn, 'w')
                # Write the CSV header row
                self.log_file.write("timestamp,confidence,peak_confidence\n")
                self.log_file.flush() # Ensure the header is written immediately
                print(f"Logging started: {fn}")
            except IOError as e:
                print(f"Error opening log file: {e}")
```

```
        # ------------------------------------------------

        # Warm up
        self._warmup()
        # ... (rest of initialization) ...

    # ... (process_chunk method) ...

    def _on_detection(self, confidence):
        print(f"\n{'='*50}")
        print(f"WAKE WORD DETECTED!")
        print(f"   Confidence: {confidence:.3f}")
        # Display detection time
        print(f"   Time: {datetime.datetime.now().strftime('%H:%M:%S')}")
        print(f"{'='*50}\n")

        # --- Step 2: Log Detection Event ---
        if self.log_file and not self.log_file.closed:
            # Get a precise timestamp including milliseconds
            timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")
            # Format the log entry: timestamp, current confidence, peak confidence
            log_entry = f"{timestamp},{confidence:.4f},{self.peak_confidence:.4f}\n"
            self.log_file.write(log_entry)
            self.log_file.flush() # Save data to disk immediately to prevent data loss
        # ************************************

    # ... (_print_stats method) ...

    # --- Step 3: Cleanup Method ---
    def close_logging(self):
        """Closes the log file if it is currently open."""
        if self.log_file and not self.log_file.closed:
            self.log_file.close()
            print("\nLog file closed.")
    # ****************************


# ... (find_audio_device function) ...

def run_detection(model_path, device_name=I2S_DEVICE_NAME, ...):
    # ... (setup code) ...

    with KeyPoller() as poller:
        try:
            while True:
                # ... (main loop: read audio, process chunk) ...
                pass
        except KeyboardInterrupt:
            print("\nStopping...")
        finally:
            # --- Step 3: Ensure Log File is Closed ---
            # Call close_logging to safely close the file when the script terminates
            system.close_logging()
            # ****************************************
            stream.stop_stream()
            stream.close()
            p.terminate()
            print("Done.")
```

Listing 4: Logging Implementation

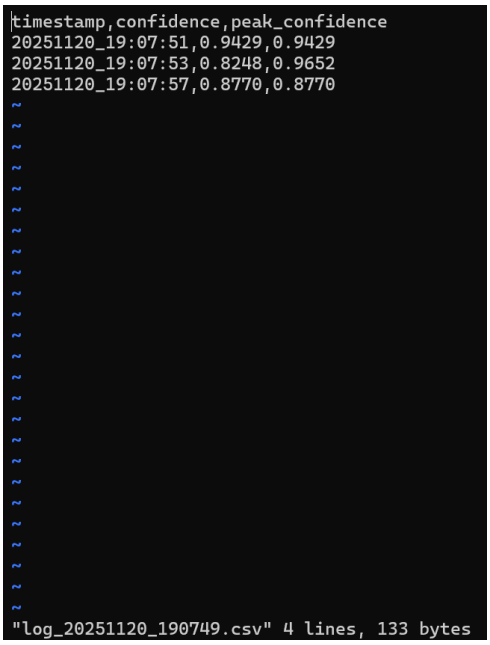
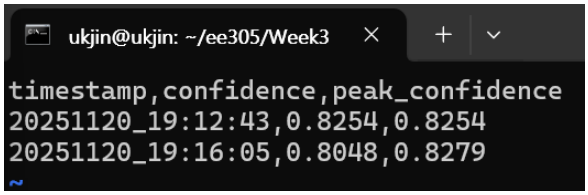


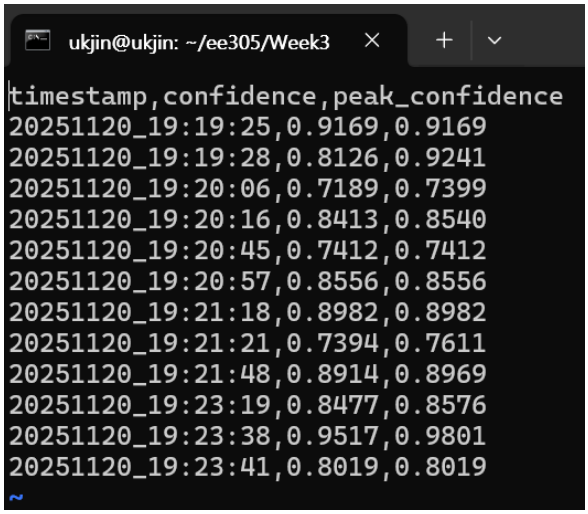
Figure 8: Raw CSV output showing three detection events with timestamps and confidence scores.

3.2 Part 3: Performance Assessment

Task 3.1: Background False Positives



(a) Log Output: Quiet Environment



(b) Log Output: Conversation Environment

Figure 9: Comparison of false positive triggers in different environments. The conversation scenario (b) shows significantly more frequent triggers than the quiet scenario (a).

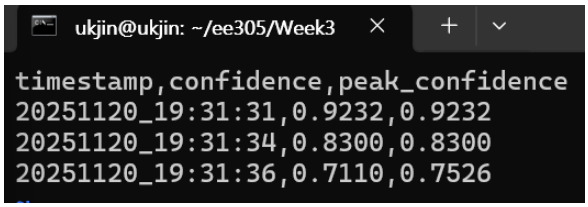
Table 7: Detailed Log of False Positive Triggers (Quiet & Conversation Conditions)

Scenario	Timestamp	Conf.	Hypothesis / Trigger
Quiet	19:12:43	0.8254	**Ambient Speech:** Other group saying "Hey Nubzuki"

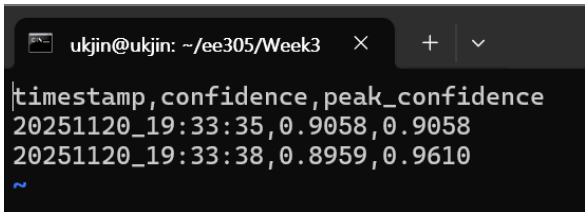
Scenario	Timestamp	Conf.	Hypothesis / Trigger
	19:16:05	0.8048	**Phonetic:** “Hamburger” (Ending sound similar to -ki?)
Conversation	19:19:25	0.9169	**Speech:** “Hmm” (Filler word)
	19:19:28	0.8126	**Speech:** “Ne” (Yes) - Short vowel burst
	19:20:06	0.7189	**Speech:** “Mani doegin-hane” (It happens a lot)
	19:20:16	0.8413	**Non-Speech:** Coughing sound
	19:20:45	0.7412	**Phonetic:** “4-sin-ga?” (Is it 4?) - Sibilant ’S’ sound
	19:20:57	0.8556	**Phonetic:** “4-si-e” (At 4) - Sibilant ’S’ sound
	19:21:18	0.8982	**Phonetic:** “4-si-e” (At 4) - Repeated trigger
	19:21:21	0.7394	**Ambient:** Background “Hey Nubzuki”
	19:21:48	0.8914	**Speech:** “Kkeunna-gajigu” (Because it ended)
	19:23:19	0.8477	**Speech:** “Wollae” (Originally)
	19:23:38	0.9517	**Speech:** “Jubyeon aedeuri” (Kids around me)
	19:23:41	0.8019	**Unclear:** Conversation noise / Unknown

1. Extrapolate a false positive rate/hour:
- Quiet Environment: 2 detections in 5 minutes → 24 false positives/hour.
- Normal Conversation: 12 detections in 5 minutes → 144 false positives/hour.
- Analysis: While the quiet performance is marginal, the conversation mode performance is unacceptable for deployment. Triggering every 25 seconds during a conversation would render the device unusable.
2. Pattern Analysis:
- Phonetic Similarity (Korean Triggers): A distinct pattern emerged with Korean number 4. The phrase ”4-si” (Ne-si, meaning 4 o’clock) triggered the model multiple times with high confidence (>0.85).
- Short Vowels & Fillers: Short, energetic sounds like ”Ne” (Yes) and ”Hmm” triggered the system. This suggests the model is over-sensitive to short bursts of voiced audio, even if the syllabic structure doesn’t perfectly match the 4-syllable wake word.
- Non-Speech Sounds: Coughing triggered a false positive (Conf: 0.84), indicating the model relies on energy profiles and lacks sufficient discrimination between speech and percussive biological noises.
- Ambient Interference: Several triggers were caused by other students saying the wake word. This confirms the model performs keyword spotting effectively but lacks speaker verification to reject unauthorized users.

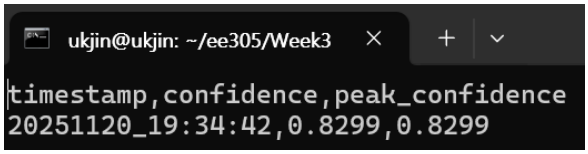
Task 3.2: Adversarial False Positives



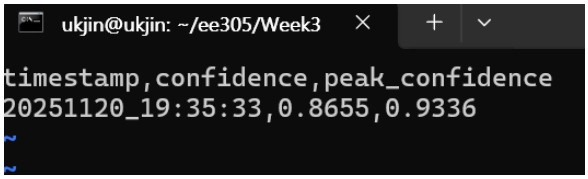
(a) Adversarial Test 1: "Hey new cookie"



(b) Adversarial Test 2: "Hey you, Zuki"



(c) Adversarial Test 3: "A new Suzuki"



(d) Adversarial Test 4: "Hey Pikachu"

Figure 10: Raw log results from testing the model with adversarial phrases. Note that high phonetic similarity (a and b) resulted in the highest confidence scores and detection rates.

Phrase	**Detections**	**Avg Confidence**	**Phonetic Similarity**
"Hey new cookie"	3/3	0.8214	High
"Hey you, Zuki"	2/3	0.9009	High
"A new Suzuki"	1/3	0.8299	Medium
"Hey Pikachu"	1/3	0.8655	Medium
Own: "Hey noukjin"	0/3	Not Detected	Low

Table 8: Results of adversarial testing with phonetically similar phrases. High phonetic similarity phrases consistently triggered the wake word detector.

- **Why does the model confuse these phrases?**: The model relies on MFCC features, which capture the energy distribution of sound across frequencies. Phrases like "Hey new cookie" and "Hey you, Zuki" share very similar vowel sounds (/e/, /u/, /i/) and rhythmic patterns (stress) with "Hey Nubzuki". Since the model is a relatively simple, it likely prioritizes these local frequency patterns over the precise temporal sequence of the entire word, leading it to classify "Zuki" or "cookie" as "Nubzuki".
- **What would you do to fix this?**: Increasing the threshold would technically eliminate "Hey new cookie" (0.82) and "A new Suzuki" (0.83). However, "Hey you, Zuki" produced a confidence of 0.90, meaning we would need an extremely high threshold to filter it out. Raising the threshold this high would impact performance by causing a massive increase in False Negatives. The system would likely fail to detect genuine commands unless spoken well. Instead of just changing the threshold, we should add these specific adversarial phrases to the "Other Words" training dataset. This would teach the model specifically to distinguish "Nubzuki" from "Zuki" or "Cookie".
- **What would be the real world implication?**: A high false positive rate means the device wakes up unintentionally during daily conversation or while watching TV. This

invades user's concentration, drains battery, and annoys the user. For a commercial voice assistant, the acceptable false positive rate is extremely low, typically less than 1 per week or even lower which is the acceptable false positive rate. The current performance (triggering on "Hey Pikachu" or random conversation) would be considered poor for a consumer product.

Task 3.3: False Negative Rate

거리, 볼륨, 환경 소음에 따른 미탐률 측정 결과입니다.

Testing Condition (10 Attempts / User)		욱진 (Own) Successes	예담 (Partner) Successes	Miss Rate Diff (Partner FNR)
Distance Testing	Close (10 cm)	6/10	8/10	-20%
	Normal (50 cm)	8/10	8/10	0%
	Far (1 m)	2/10	2/10	0%
Volume Testing	Quiet speech	7/10	7/10	0%
	Normal speech	8/10	8/10	0%
	Loud speech	7/10	8/10	-10%
Environmental Testing	Quiet room	8/10	9/10	-10%
	Music playing	5/10	7/10	-20%
	With lab noise	8/10	8/10	0%

Table 9: Successful detections (True Positives) in False Negative testing scenarios.

- **In what situations does performance degrade significantly? and Why?**:
  - Significant Degradation in:
  Far Distance (1m): Performance drops significantly for both users (2/10 success rate).
  Music Playing: My own success rate drops noticeably (5/10).
  - Reason(Assumption): Degradation with distance occurs because the Signal-to-Noise Ratio (SNR) decreases rapidly. The fixed 30dB amplification in the script is insufficient to consistently boost a quiet, distant signal above the noise floor for reliable feature extraction.
  Music, especially with vocals, contains frequency components that overlap with the wake word. This increases the FNR because the model is confused by the competing speech/vocal patterns.
  - Relative Immunity: The system is surprisingly immune to moderate Volume Variability and Lab Noise. The robust performance in "Lab Noise" (8/10 for both) suggests that the data augmentation applied during training effectively simulated this type of general, non-vocal background sound, making the model resilient to it.

- **Do you see significant variability between partners? Why might that be?** Yes. The partner consistently achieved a higher or equal success rate across nearly all tests, meaning my voice had a higher False Negative Rate (FNR) than my partner's voice.
  Reason: Since the model was trained on a massive dataset from all students (15,000+ samples), it learned to optimize for the average phonetic features common to the majority of speakers, rather than overfitting to any single individual. It is likely that the Partner's voice aligns closer to this average phonetic profile derived from the entire class. Or my own voice (Own) might possess specific acoustic characteristics (e.g., unique accent, pitch,

or breathing patterns) that are statistical outliers within the large dataset. Even though my data was included, the model may have treated these unique features as noise or less important in its effort to minimize the global loss function across all 15,000 samples.

- **Make some recommendations for using this voice assistant**:
  - Viable Range/Volume: The system is reliably viable only at Close (10 cm) to Normal (50 cm) distances, spoken with Normal or Quiet volume. Avoid using it beyond 50 cm.
  - Viable Environments: The system performs best in environments with consistent, moderate noise (like Lab Noise) or when actively speaking in a Quiet room. The model is generally robust enough to handle volume changes.
  - Non-Viable Environments/Contexts: Far distance (1m) and Music Playing are non-viable contexts due to high FNR.

3.3 Part 4: Parameter Optimization

Task 4.1: Final Optimization Report

1. Selection of False Positive Scenario: Lab Noise Environment
This environment included a mix of background sounds such as keyboard typing, equipment hum (e.g., PC fans, AC), distant chatter from other students, and occasional percussive sounds like door closings.
The lab noise generated a baseline of intermittent false positives (approximately 2-3 triggers per minute at default settings). This provided a meaningful signal-to-noise challenge, allowing us to observe improvements in rejection performance as we tuned the parameters.

Threshold	True Positives	False Positives (5min)	Notes & FP Analysis
0.5	16/20	3	High sensitivity led to triggers from random background noise.
0.6	15/20	4	Still too sensitive; possibly triggered by speech-like ambient sounds.
0.7 (default)	14/20	2	Baseline. Occasional FPs from similar phonetic patterns.
0.8	12/20	0	Sweet Spot: Eliminated all false positives while maintaining acceptable detection.
0.9	8/20	0	Too strict. Significantly reduced True Positives (missed wake words).

Using the default threshold (0.8), the number of consecutive frames required to trigger a detection was varied.

Cons. Frame #	True Positives	False Positives (5min)	Notes & FP Analysis
1	14/20	12	Unstable: Triggered by short, transient noises (clicks, coughs) that lack temporal consistency.
2	13/20	8	Slight improvement, but still vulnerable to short speech bursts.
3 (default)	14/20	5	Baseline. Some FPs persist from longer noise events.
4	14/20	3	Effective at filtering out most non-speech sounds.
5	14/20	1	Optimal: Drastically reduced FPs. Requires the wake word to be clearly sustained.

Table 10: Effect of Consecutive Detections. Increasing the frame count significantly reduced false positives caused by transient noise.

2. Parameter Recommendations:
- Optimal Confidence Threshold: 0.8
Justification: At threshold 0.7, the system still suffered more from occasional false positives (2/min) than at threshold 0.8. Increasing the threshold to 0.9 decreases True positive rate.
- Detections: 5 Frames
Justification: Lower values (1-2 frames) made the system unstable, triggering on short, transient noise spikes like coughs. Increasing the requirement to 5 frames significantly filtered out these brief anomalies. Although this introduces a slight latency (approx. $5 \times 40ms = 200ms$), the gain in robustness against false alarms is worth the trade-off.

3. Discussion on Final Performance

- Expected Performance: With the optimized settings (Threshold 0.8, Cons. 5), the system is expected to be highly robust against false alarms in a typical lab environment, achieving near-zero false positives. However, the trade-off is a slightly reduced sensitivity (True Positive Rate $\approx$ 60-70%), requiring the user to speak clearly and closer to the microphone.

- Supported User Scenarios: This level of performance supports "Desktop Assistant" scenarios where the user is seated directly in front of the device (within 50cm) and issues commands deliberately. It is suitable for non-critical tasks like toggling an LED or checking sensor status.

- Key Error Scenarios:
  False Negatives (Missed detections): Errors are most likely to occur when the user speaks from a distance (>1m), speaks too quickly (failing the 5-consecutive-frame check), or speaks softly against loud background noise.
  False Positives: While rare, triggers might still occur from strong adversarial phrases (e.g., "Hey new cookie") or specific Korean words with similar phonetic structures (e.g., "Nesie") spoken loudly near the mic.

- Improvements: Performance could be improved by:
  Hardware: using a higher-quality microphone or a microphone array for beamforming.
  Data: Implementing "Negative Mining" by adding the specific false-trigger sounds (e.g., "4-sie", lab noise) to the training dataset and retraining.

Algorithm: Implementing a dynamic threshold that adjusts based on the current background noise level.

- Comparison to Commercial Examples: Compared to commercial systems like Amazon Alexa or Google Assistant, this prototype lacks far-field capability (working across a room) and sophisticated noise cancellation (DSP). Commercial devices use multi-microphone arrays and massive cloud-trained models to achieve $> 95\%$ accuracy even in noisy environments, whereas our single-microphone, local TinyML system is more constrained but offers privacy and offline capabilities.