

Speech-to-Text Fine-Tuning Guide - Book 3 - Specialized Topics

Speech-to-Text Fine-Tuning Guide - Book 3 - Specialized Topics

Speech-to-Text Fine-Tuning Guide

Book 3: Specialized Topics

Mobile ASR, File Formats & Vocabulary

Table of Contents

Part VII: Mobile ASR

Mobile and edge device deployment (1 chapters)

Part VIII: File Formats

Audio and model file formats (2 chapters)

Part IX: Vocabulary & Language

Vocabulary recognition and language considerations (2 chapters)

Part VII: Mobile ASR

Mobile and edge device deployment

Phone Vs Desktop Whisper Performance

Question

How can a OnePlus Nord 3 5G smartphone run Whisper at comparable performance to an AMD Radeon 7700 XT desktop GPU, despite the massive difference in physical hardware size?

Answer

Your observation is astute and highlights one of the most interesting developments in mobile AI. Let's break down why your phone performs better than you'd expect.

Hardware Comparison

Your Desktop GPU: AMD Radeon 7700 XT

- **Architecture:** RDNA 3 (Navi 32)
- **Die Size:** ~346 mm²
- **Power Consumption:** 245W TDP
- **VRAM:** 12GB GDDR6
- **Compute:** ~35 TFLOPS FP32, ~70 TFLOPS FP16
- **Manufacturing:** 5nm + 6nm chiplet
- **Physical dimensions:** ~267mm x 111mm x 50mm
- **Weight:** ~1kg

Your Phone: OnePlus Nord 3 5G

- **SoC:** MediaTek Dimensity 9000
- **Architecture:** ARM Mali-G710 MC10 GPU
- **Die Size:** ~120 mm² (entire SoC, including CPU, GPU, NPU, modem)
- **Power Consumption:** ~10W peak (entire device)
- **Memory:** Shared 8-16GB LPDDR5X
- **Compute:** ~5 TFLOPS FP32 (GPU) + dedicated AI accelerator
- **Manufacturing:** 4nm TSMC
- **Physical dimensions (entire phone):** ~163mm x 75mm x 8mm
- **Weight:** ~195g

Your intuition is right: the desktop GPU is physically ~10x larger and uses ~25x more power.

Why the Performance Gap Is Smaller Than Expected

1. Dedicated AI Accelerators on Mobile (NPUs/APUs)

Critical insight: Your phone isn't running Whisper primarily on its GPU.

Modern flagship SoCs like the Dimensity 9000 have **dedicated AI Processing Units (APUs)** optimized for neural network inference:

Dimensity 9000 APU Specs: - 5th-gen APU: 4x faster than previous gen - **6 TOPS (trillion operations per second) INT8 performance** - **Optimized for transformer models** (like Whisper) - **Power efficiency:** 5x more efficient than GPU for AI workloads - **Dedicated memory access paths** (minimizes bandwidth bottlenecks)

Why This Matters:

Desktop GPU: General-purpose compute (graphics, AI, compute)
→ Not optimized specifically for transformer inference
→ Whisper uses a fraction of available compute

Phone APU: Purpose-built for AI inference
→ Every transistor designed for neural network operations
→ Whisper runs on optimized silicon

Analogy: It's like comparing a large dump truck (desktop GPU) to a Formula 1 race car (phone APU) for driving on a highway. The dump truck is bigger and more powerful, but the F1 car is optimized for speed in its specific use case.

2. Quantization and Mobile-Optimized Models

Your phone likely isn't running the same Whisper model as your desktop.

Desktop (typical): - **Precision:** FP32 or FP16 (32-bit or 16-bit floating-point) - **Model:** Full Whisper base/small/medium - **Framework:** PyTorch with ROCm

Phone (typical): - **Precision:** INT8 (8-bit integer quantization) - **Model:** Quantized Whisper variant optimized for mobile - **Framework:** TensorFlow Lite, ONNX Runtime Mobile, or vendor-specific (MediaTek NeuroPilot)

Quantization Impact:

FP32 model: 1.0 GB, 100% accuracy baseline
INT8 model: 0.25 GB (4x smaller), ~98% accuracy

Speed improvement: 2-4x faster inference
Memory bandwidth: 4x reduction
Power efficiency: 5-10x better

Your phone achieves similar perceptual quality with 1/4 the data movement and compute.

3. Memory Bandwidth and Data Movement

Counterintuitive fact: For Whisper inference, memory bandwidth matters more than raw compute.

Why Transformers Are Memory-Bound: Whisper (and all transformer models) spend most time: - Loading weights from memory - Moving activations between layers - Accessing attention matrices

Not performing math operations (those are fast on modern hardware).

Desktop Setup (Naive):

CPU → PCIe bus → GPU VRAM → Compute cores
^slow^

Desktop Setup (Optimized):

All data in GPU VRAM → Compute cores
^fast, but still limited by VRAM bandwidth^

Phone Setup:

APU integrated in SoC → Unified memory → Direct access
^no PCIe bottleneck, low latency^

Key Difference: - **Desktop GPU:** High bandwidth (384 GB/s), but data must traverse PCIe bus from system RAM unless pre-loaded - **Phone APU:** Lower bandwidth (60-100 GB/s), but **integrated in SoC** with direct memory access and lower latency

For Whisper's inference pattern (small batches, streaming audio), low latency often beats high bandwidth.

4. Optimization and Software Stack

Mobile AI Software Is Highly Optimized (Out of Necessity)

Phone Software Stack (Highly Optimized):

- **MediaTek NeuroPilot:** Vendor-specific APU acceleration
- **TensorFlow Lite / ONNX Runtime Mobile:** Optimized for mobile inference
- **Kernel fusion:** Multiple operations combined into single kernels
- **Mixed precision:** Uses INT8 where possible, FP16 where necessary
- **Pruning:** Removes unnecessary model weights
- **Hardware-specific tuning:** Optimized for Dimensity 9000 specifically

Desktop Stack (Less Optimized for Whisper):

- **PyTorch + ROCm:** General-purpose, not Whisper-specific
- **FP16/FP32:** Larger data types (more accurate but slower)
- **Fewer mobile optimizations:** Desktop ecosystem prioritizes flexibility over efficiency

Mobile developers had to squeeze every drop of performance due to power/thermal constraints.
Desktop developers have more headroom, so less aggressive optimization.

5. Thermal and Power Constraints (Paradoxically Helpful)

Your desktop GPU throttles less, but also wastes more.

Desktop (AMD 7700 XT): - Runs at high clock speeds (2.5 GHz+) - High power consumption (200W+) - Large cooling solution allows sustained performance - **But:** Whisper doesn't fully utilize the GPU (low occupancy) - GPU is running at high clocks waiting for memory - Wasting power on idle cores

Phone (Dimensity 9000 APU): - Runs at lower clocks (~1 GHz APU) - Low power consumption (5-10W) - Thermal throttling kicks in quickly - **But:** APU is fully utilized (100% occupancy) - Every core doing useful work - Efficient at its target workload

Efficiency Comparison:

Desktop: 245W to run Whisper → 0.5x realtime (example)
Phone: 5W to run Whisper → 0.4x realtime

Performance: Similar

Efficiency: Phone wins by 20-30x

6. Model Size Sweet Spot

Whisper Base/Small models fit mobile hardware perfectly.

Whisper Model Sizes:

Model	Parameters	Disk Size	VRAM/RAM Needed
Tiny	39M	73 MB	~400 MB
Base	74M	139 MB	~600 MB
Small	244M	461 MB	~1.5 GB
Medium	769M	1.45 GB	~4 GB
Large	1.5B	2.87 GB	~8 GB

Your Phone (8-16GB RAM): - Can comfortably run **Base** or **Small** (INT8 quantized) - Quantized Small: ~350 MB - Leaves plenty of RAM for OS and other apps

Your Desktop GPU (12GB VRAM): - Can run up to **Large** (FP16) - But you're likely testing **Base** or **Small** for fair comparison - Desktop is underutilized (using <5% of VRAM)

When testing equivalent model sizes, desktop advantage shrinks dramatically.

7. Real-World Performance Comparison

Let's estimate actual inference speeds:

Scenario: Whisper Small (244M params), 30-second audio clip

Desktop (AMD 7700 XT, FP16, PyTorch + ROCm): - Inference time: ~2-4 seconds - Preprocessing: 0.5 seconds - **Total: ~2.5-4.5 seconds** - **Realtime factor: 0.08-0.15x** (6-12x faster than realtime)

Phone (Dimensity 9000, INT8, TensorFlow Lite): - Inference time: ~3-5 seconds - Preprocessing: 0.5 seconds - **Total: ~3.5-5.5 seconds** - **Realtime factor: 0.12-0.18x** (5-8x faster than realtime)

Difference: Desktop is ~1.3-1.5x faster

Your observation: “not drastically better” is accurate!

Why Desktop Isn't 10x Faster (Summary)

Factor	Desktop Advantage	Why Gap Is Smaller
Raw compute	7x more TFLOPS	Whisper is memory-bound, not compute-bound
Memory bandwidth	4x higher	Mobile has lower latency, integrated design
Die size	3x larger	Phone has dedicated AI silicon (APU)
Power consumption	25x higher	Wasted on idle cores, not efficiently utilized
Optimization	Less optimized	Mobile stack highly tuned for efficiency
Quantization	Uses FP16/FP32	Phone uses INT8 (4x smaller, faster)
Hardware specialization	General GPU	APU purpose-built for transformers

Bottom line: For Whisper inference specifically, your phone's dedicated AI silicon and optimized software stack nearly closes the gap with your desktop's brute-force GPU power.

When Desktop Wins Big

Desktop advantage grows significantly when:

1. **Batch processing:** Desktop can process 8-16 audio files simultaneously
 - Phone: Limited by RAM (batch size 1-2)
 - Desktop: Can batch 16+ (10x faster throughput)
2. **Larger models:** Whisper Large or custom fine-tuned models
 - Phone: Cannot run Large (insufficient RAM)
 - Desktop: Runs Large-v3 comfortably
3. **Training/fine-tuning:** Desktop crushes phone
 - Phone: Not designed for training (APUs are inference-only)

- Desktop: Can fine-tune models 100x faster

4. Long-form audio: Hours of audio

- Phone: Thermal throttling becomes an issue
- Desktop: Sustained performance over hours

For single-clip, base/small model inference (your use case), the gap is small.

Broader Implications

The Mobile AI Revolution

Your observation reflects a broader trend:

2015-2020: Desktop/Cloud Dominated AI - Models too large for mobile - Mobile = cloud API calls

2020-2025: Mobile AI Catches Up - Dedicated AI accelerators (Apple Neural Engine, Google TPU, MediaTek APU, Qualcomm AI Engine) - Quantization techniques (INT8, INT4) - On-device inference for privacy, latency, offline use

Result: Flagship phones now rival mid-range desktop GPUs for inference.

Efficiency > Raw Power for Inference

For inference (not training): - **Purpose-built silicon** (APU) beats general-purpose (GPU) - **Software optimization** matters as much as hardware - **Memory hierarchy** (latency, bandwidth) matters more than compute - **Quantization** enables massive speedups with minimal quality loss

Your phone is a testament to the power of specialized, efficient design.

Practical Takeaways

When to Use Desktop:

- Fine-tuning models

- Batch processing (dozens of files)
- Large models (Whisper Medium/Large)
- Long recording sessions (hours)
- Experimenting with custom models

When to Use Phone:

- Real-time transcription
- On-the-go recordings
- Single clips (<5 minutes)
- Privacy (offline inference)
- Power efficiency

For your daily use case (speech-to-text input), phone is likely sufficient—and more convenient.

Future Outlook

Mobile AI is getting better, faster:

- **Next-gen SoCs (2024-2025):** 10-15 TOPS APUs
- **Improved quantization:** INT4, mixed INT8/FP16
- **On-device fine-tuning:** Possible within 2-3 years
- **Larger models on-device:** Whisper Medium on flagship phones soon

Desktop advantage will remain for: - Training and fine-tuning - Extremely large models (10B+ parameters) - Batch processing at scale

But for inference, mobile will continue closing the gap.

Conclusion

Your OnePlus Nord 3 5G performs surprisingly well because:

1. **Dedicated AI silicon (APU)** purpose-built for transformers

2. **Aggressive quantization** (INT8 vs. FP16/FP32)
3. **Highly optimized software stack** (TensorFlow Lite, vendor kernels)
4. **Integrated memory architecture** (low latency, no PCIe bottleneck)
5. **Whisper is memory-bound** (not compute-bound), favoring efficient designs

Your desktop GPU has more raw power, but Whisper inference doesn't fully utilize it.

The result: **Phone ~0.6-0.8x the speed of desktop for equivalent models**—much closer than the 10x physical size difference would suggest.

This is modern AI hardware engineering: efficiency through specialization.

Note: This analysis was generated by Claude Code (claude-sonnet-4-5) for Daniel Rosehill's STT Fine-Tuning Notebook. Performance varies by model size, implementation, and specific hardware. For the most accurate comparison, benchmark both devices with identical models (same Whisper variant, same precision) using tools like `faster-whisper` (desktop) and `whisper.cpp` (mobile). Mobile AI capabilities are rapidly evolving—expect continued improvements in coming years.

Part VIII: File Formats

Audio and model file formats

Formats

When you fine-tune Whisper using a standard notebook example, you'll end up with a folder structure like this:

alt text

alt text

The directory contains `runs` for resuming from checkpoints, but the core model file is:

`model.safetensors`

This file is directly usable for STT/inference. However, given the wide variety of Whisper deployment scenarios and the fact that you may want to use your fine-tuned model on devices with different processing capabilities, you'll likely need to convert it to other formats.

Common Model Formats

SafeTensors (Original Format)

- **File extension:** `.safetensors`
- **Use case:** Direct inference, training, PyTorch-based applications
- **Advantages:** Safe serialization format, prevents arbitrary code execution
- **Disadvantages:** Limited compatibility with optimized inference engines

GGML (Legacy)

- **File extension:** `.bin`
- **Use case:** CPU-only inference on edge devices (e.g., FUTO Keyboard)
- **Compatible with:** `whisper.cpp` (older versions)

- **Advantages:** Enables deployment on resource-constrained devices, edge computing
- **Status:** Legacy format, superseded by GGUF
- **Considerations:** Hardware limitations still apply. Consider converting Tiny models to GGML for mobile/edge deployment while keeping Base or larger models in CTranslate2 for desktop applications.

GGUF (Recommended for Edge/CPU)

- **File extension:** `.gguf`
- **Use case:** CPU-only inference on edge devices, local deployment
- **Compatible with:** Modern `whisper.cpp`, llama.cpp ecosystem
- **Advantages:**
 - Improved metadata handling (embedded model info, hyperparameters)
 - Better version control and compatibility checking
 - More efficient storage and loading
 - Standardized format across llama.cpp ecosystem
 - Supports quantization for smaller model sizes
- **Best for:** Modern edge deployments, CPU inference, resource-constrained environments
- **Migration:** GGUF is the successor to GGML and should be preferred for new projects

CTranslate2

- **File extension:** `.bin` (directory with model files)
- **Use case:** Optimized inference for desktop applications
- **Compatible with:** Faster Whisper and many local STT applications
- **Advantages:** Significantly faster inference, reduced memory usage, optimized for CPU and GPU
- **Best for:** Production deployments requiring speed and efficiency

ONNX

- **File extension:** `.onnx`
- **Use case:** Cross-platform deployment, inference optimization
- **Compatible with:** ONNX Runtime, various inference engines
- **Advantages:** Hardware-agnostic, works across different ML frameworks. Long recording durations / less chunking.

- **Best for:** Applications requiring maximum portability across platforms and hardware

Core ML (Apple Devices)

- **File extension:** `.mlmodel` or `.mlpackage`
- **Use case:** iOS, macOS, and Apple Silicon deployment
- **Advantages:** Native Apple Neural Engine acceleration, optimized battery usage
- **Best for:** Native Apple applications

TensorFlow Lite

- **File extension:** `.tflite`
- **Use case:** Mobile deployment (Android/iOS)
- **Advantages:** Lightweight, optimized for mobile inference
- **Best for:** Mobile applications with size and performance constraints

Format Selection Guide

Format	Best Use Case	Performance	Compatibility
SafeTensors	Training, PyTorch apps	Baseline	PyTorch ecosystem
GGML	Legacy edge devices	Optimized for CPU	Old whisper.cpp projects
GGUF	Modern edge devices, CPU-only	Optimized for CPU	Modern whisper.cpp, llama.cpp
CTranslate2	Desktop apps, servers	High (GPU/CPU)	Faster Whisper, production apps
ONNX	Cross-platform deployment	Good	Wide framework support
Core ML	Apple devices	Excellent (on Apple HW)	Apple ecosystem only
TFLite	Mobile apps	Good	Android/iOS

Conversion Considerations

- **Model size:** Larger models (Base, Small, Medium) may not be practical for GGML/GGUF on edge devices
- **Target hardware:** GPU availability, CPU capabilities, RAM constraints
- **Use case:** Real-time vs. batch processing, latency requirements
- **Deployment environment:** Cloud, edge, mobile, desktop

Gguf Vs Ggml

Overview

GGML (Georgi Gerganov Machine Learning) was the original quantized model format created for CPU-based inference in the llama.cpp ecosystem. GGUF (GGML Universal Format) is its successor, designed to address limitations and improve the overall user experience.

What Changed?

GGML (Legacy Format)

File Extension: `.bin`

Characteristics: - Basic binary serialization format - Minimal metadata embedded in the model file - Version information stored externally or not at all - Required manual tracking of model parameters, quantization settings, and architecture details - Prone to compatibility issues when model formats evolved - Used across early whisper.cpp and llama.cpp projects

Limitations: - No standardized way to store metadata - Difficult to validate model compatibility automatically - Version mismatches could cause silent failures or crashes - Required users to manually track model configurations - Limited error messages when loading incompatible models

GGUF (Modern Format)

File Extension: `.gguf`

Improvements: - **Rich Metadata:** Embeds comprehensive model information directly in the file - Model architecture details - Tokenizer information - Quantization parameters - Version information - Custom metadata fields - **Version Control:** Built-in versioning system prevents compatibility issues - **Self-Describing:** Models carry all necessary information for proper loading - **Better Error Handling:** Provides clear error messages for incompatible versions - **Standardization:** Unified format across the entire llama.cpp ecosystem - **Extensibility:** Designed to accommodate future format changes without breaking compatibility

Technical Comparison

Feature	GGML	GGUF
Metadata Storage	Minimal/External	Embedded & Comprehensive
Version Checking	Manual	Automatic
Error Messages	Vague	Detailed & Helpful
Cross-Tool Compatibility	Limited	Excellent
Future-Proofing	Poor	Good
File Size Overhead	Minimal	Slightly larger (negligible)
Loading Speed	Fast	Fast (comparable)

Migration Path

When to Use GGML

- **Legacy Systems:** You're maintaining older whisper.cpp or llama.cpp deployments
- **Existing Tooling:** Your production pipeline is built around GGML and migration isn't feasible
- **Compatibility:** You need to support older versions of tools that don't support GGUF yet

When to Use GGUF (Recommended)

- **New Projects:** All new fine-tuning and deployment projects
- **Modern Tools:** Working with up-to-date whisper.cpp, llama.cpp, or compatible tools

- **Better Maintenance:** Want self-documenting models with clear version information
- **Long-Term Support:** Building applications that need to be maintained over time

Conversion Between Formats

GGML to GGUF

Most modern versions of whisper.cpp and llama.cpp include conversion utilities:

```
python convert-whisper-to-ggml.py --model path/to/model --output-format gguf
```

Hugging Face Hub

Many model repositories now offer both formats:
- Look for files ending in `.gguf` for the modern format
- Older repositories may only have `.bin` files (GGML)
- Prefer GGUF versions when available

Real-World Impact

For Whisper Fine-Tuning

GGML Era Workflow: 1. Fine-tune model 2. Convert to GGML `.bin` 3. Manually document quantization settings 4. Hope the target device's whisper.cpp version is compatible 5. Debug cryptic errors if versions don't align

GGUF Era Workflow: 1. Fine-tune model 2. Convert to GGUF `.gguf` 3. Metadata automatically embedded 4. Target device validates compatibility automatically 5. Clear error messages if there are issues

For Deployment

Benefits in Production: - Easier model versioning and rollback - Better debugging when issues occur - Simplified model management in multi-model systems - More reliable cross-platform deployment

Recommendations

For Fine-Tuning Projects

- ✓ Use GGUF for all new Whisper fine-tuning projects targeting edge/CPU deployment

For Edge Deployment

- ✓ Migrate to GGUF if your whisper.cpp version supports it (most versions since mid-2023)

For Mobile/Embedded

- ✓ GGUF provides better long-term maintainability, even if initial setup seems similar

For Legacy Systems

- ⚠ GGML may still be necessary for very old deployment targets, but plan migration

Key Takeaway

GGUF is not a different inference engine or a performance upgrade—it's a better packaging format for the same underlying quantized model technology. Think of it as upgrading from a ZIP file with a separate README to a self-documenting archive that validates itself when opened.

For Whisper fine-tuning projects targeting CPU/edge deployment, **always prefer GGUF unless you have a specific reason to use the legacy GGML format.**

Part IX: Vocabulary & Language

Vocabulary recognition and language considerations

Target Vocabulary In Training Data

The Question

When recording training data for ASR fine-tuning that includes target foreign vocabulary (e.g., Hebrew words in English sentences), each training sample contains:

- **Known words:** Words the model already handles well (“I’m going to the”)
- **Target vocabulary:** The new word you want to teach (“Mekolet” - Hebrew for convenience store)

Does the model need to “learn” the entire sentence, or just the target vocabulary? Should you minimize surrounding context to increase the signal-to-noise ratio for learning?

Short Answer

No, you should NOT minimize surrounding context. The fine-tuning process naturally handles this, and surrounding context actually *improves* learning of target vocabulary through:

1. Co-articulation effects (how sounds blend between words)
2. Contextual embeddings
3. Statistical regularization

Include natural, varied sentences around your target vocabulary—this helps, not hurts.

How ASR Models Process Training Data

Sequence-to-Sequence Learning

Whisper and similar models use a sequence-to-sequence architecture:

```
Audio sequence → Encoder → Context representation → Decoder → Text sequence
```

During fine-tuning, the model learns:

1. **Acoustic patterns:** What does “Mekolet” *sound* like?
2. **Phonetic structure:** What phonemes compose it?
3. **Contextual usage:** Where does it appear in sentences?
4. **Transitions:** How do surrounding words affect its pronunciation?

Gradient-Based Learning

The loss function compares predicted text to ground truth **across the entire sequence**:

```
Loss = sum of prediction errors for each token in the output
```

However, the **gradient magnitude** (how much the model adjusts) is automatically higher for tokens where the prediction error is larger:

- Words already known well (e.g., “going”, “the”) → Small prediction error → Small gradient → Minimal learning
- Unknown words (e.g., “Mekolet”) → Large prediction error → Large gradient → Significant learning

The model automatically focuses learning where it’s needed most. You don’t need to manually increase the signal-to-noise ratio by removing context.

Why Surrounding Context Helps Learning

1. Co-Articulation Effects

Speech is not discrete—sounds blend between words:

"I'm going to the Mekolet"

↓

Pronunciation of "the" affected by following "M"

Pronunciation of "Me-" affected by preceding "the"

If you trained only on isolated “Mekolet” pronunciations, the model would learn:

- How “Mekolet” sounds in isolation
- But NOT how it sounds after “the”
- Or after “to the”
- Or how native speakers phonetically reduce preceding words

Natural sentence context teaches the model real-world pronunciation patterns.

2. Contextual Embeddings

Modern transformer-based models use contextual embeddings—the representation of “Mekolet” is different depending on surrounding words:

"I'm going to the Mekolet" → Embedding_A for "Mekolet"

"Meet me at Mekolet" → Embedding_B for "Mekolet"

This context helps the model:

- Disambiguate similar-sounding words
- Understand typical usage patterns
- Build more robust representations

Varied contexts create richer, more generalizable learning.

3. Statistical Regularization

When the model sees:

"I'm going to the Mekolet"

"We stopped at the Mekolet"

"The Mekolet sells groceries"

The **consistent presence of known words** acts as an anchor:

- The model is confident about “I’m going to the”

- This confidence constrains the solution space for “Mekolet”
- Prevents overfitting to spurious patterns

Context provides statistical scaffolding that guides learning.

4. Language Model Priors

Whisper includes a language model component that predicts likely next words. During fine-tuning:

- It learns: “after *to the*, *Mekolet* is a plausible next word”
- It learns: *Mekolet* appears in similar contexts as “store”, “market”, “shop”
- This helps during inference with partial/noisy audio

Context teaches the model *when* to predict your target vocabulary.

The “Isolated Vocabulary” Experiment

What if you **only** trained on isolated target words?

Approach A: Isolated words only

Training data:

- "Mekolet" (1 second)
- "Mekolet" (1 second)
- "Mekolet" (1 second)
- × 100 samples

Problems:

1. **Overfitting:** Model memorizes the specific recording conditions
2. **Poor generalization:** Doesn’t learn how “Mekolet” sounds in natural speech
3. **No co-articulation:** Fails when preceded/followed by other words
4. **Catastrophic forgetting:** May *degrade* performance on other words because loss function doesn’t reinforce them

Approach B: Natural sentences (recommended)

Training data:

- "I'm going to the Mekolet"
- "The Mekolet is closed today"
- "She works at the Mekolet"
- × 33 samples (same total audio length)

Benefits:

1. **Natural co-articulation:** Learns real pronunciation patterns
2. **Contextual learning:** Understands typical usage
3. **No catastrophic forgetting:** Reinforces known words too
4. **Better generalization:** More robust to variations

Empirical evidence: Approach B consistently outperforms Approach A in ASR fine-tuning.

Best Practices for Training Data with Target Vocabulary

1. Use Natural Sentences

✓ “I’m going to the Mekolet to buy milk” ✗ “Mekolet” ✗ “The Mekolet Mekolet Mekolet”

2. Vary the Context

Include target word in different sentence positions:

- Beginning: “Mekolet is my favorite store”
- Middle: “I shop at the Mekolet daily”
- End: “Let’s meet at the Mekolet”

Include different preceding/following words:

- “...to the Mekolet”
- “...at the Mekolet”
- “...from the Mekolet”

3. Balance Target Density

Good ratio: 1-3 target words per 10-15 word sentence

✓ “I’m going to the Mekolet to buy milk” (1 target / 9 words = 11%) ✗ “Mekolet Mekolet Mekolet Mekolet” (4 targets / 4 words = 100%) ✗ “I’m going to the store today to buy groceries and then heading home” (0 targets / 14 words = 0%)

4. Include Prosodic Variation

Record with different:

- Speaking speeds (normal, fast, slow)
- Emphasis patterns (“I’m going to the MEKOLET” vs. “I’m **going** to the Mekolet”)
- Emotional tone (neutral, excited, tired)

5. Don’t Artificially Isolate

✗ Don’t insert unnatural pauses: “I’m going to the … MEKOLET” ✓ Speak naturally: “I’m going to the Mekolet”

6. Quantity: Target Words vs. Total Words

For effective learning, you need:

- **Absolute target word instances:** 50-100+ instances of each target word
- **Total training data:** 30-60 minutes typical for few-word fine-tuning

Example for 10 target words:

- $10 \text{ words} \times 70 \text{ instances each} = 700 \text{ target word instances}$
- In natural sentences (10% density) = 7,000 total words
- At $\sim 2 \text{ words/second} = \sim 3,500 \text{ seconds} = \sim 60 \text{ minutes of speech}$

This provides both sufficient target word exposure AND enough surrounding context.

The Fine-Tuning Loss Function in Practice

Here’s conceptually how the model learns from:

```
Ground truth: "I'm going to the Mekolet"  
Prediction:   "I'm going to the [??]"
```

Loss computation (simplified):

```
tokens = ["I'm", "going", "to", "the", "Mekolet"]  
predicted_probs = model.predict(audio)  
  
loss = 0  
for i, token in enumerate(tokens):  
    error = -log(predicted_probs[i][token])  
    loss += error
```

The model's attention automatically focuses on errors. Including known words doesn't dilute learning—it stabilizes it.

Special Case: Very Limited Data

If you have **extremely limited data** (< 20 minutes total), you might consider:

1. **Slightly higher target density:** 15-20% instead of 10%
2. **Shorter surrounding sentences:** “Go to the Mekolet” vs. “I think we should go to the Mekolet tomorrow”
3. **But still include natural context:** Never train on isolated words

Even with limited data, context helps more than it hurts.

What About Data Augmentation?

Rather than removing context, consider **augmenting** your target vocabulary training:

Effective augmentation:

1. **Pitch shifting:** Simulate different speakers
2. **Speed variation:** 0.9x - 1.1x playback speed
3. **Background noise:** Add realistic noise at low levels
4. **Room reverb:** Simulate different recording environments

These help the model generalize without sacrificing contextual learning.

Ineffective augmentation:

✗ Cutting sentences to isolate target words ✗ Repeating target words artificially ✗ Removing surrounding words

Monitoring During Fine-Tuning

You can verify this behavior during training:

```
Known word losses:  
"I'm": 0.01  
"going": 0.02  
"to": 0.01  
"the": 0.02  
  
Target word losses:  
"Mekole": 3.45 ← much higher
```

If you see all losses roughly equal, something is wrong with your training setup.

Conclusion

You should NOT minimize surrounding context in your training data. The fine-tuning process naturally handles the signal-to-noise issue through gradient-based learning that automatically focuses on high-error tokens (your target vocabulary).

Surrounding context actively helps by:

1. Teaching natural co-articulation patterns
2. Providing contextual embeddings
3. Offering statistical regularization
4. Building language model priors
5. Preventing catastrophic forgetting

Best practice: Record natural sentences with 10-15% target vocabulary density, varied contexts, and natural prosody. Let the model’s learning algorithm handle the focus—it’s designed for exactly this scenario.

The intuition that “more signal, less noise” would help is understandable but misapplies concepts from traditional signal processing. In neural network training, context *is* signal, not noise.

Note: This document was generated by Claude Code, an AI assistant. Please validate technical details and test recommendations in your specific environment before implementing.

Vocabulary Recognition Asr Training Data

Question

Why does Whisper recognize some Hebrew words (like “Shabbat”) but not others (like “Macolet”)? Does ASR work at the word level or phonetic level, and how does training data determine which words make it into the model?

Answer

Excellent question that touches on fundamental ASR mechanics. Let’s break down how vocabulary recognition works in models like Whisper.

How ASR Models Handle Vocabulary

The Short Answer:

ASR models work at the subword/phoneme level, not the word level.

Whether a word is recognized depends on: 1. Frequency in training data (how often it appeared) 2. Phonetic similarity to known words 3. Tokenization strategy (how the model breaks down sounds) 4. Language mode (English vs. Hebrew vs. multilingual)

Your observation is spot-on: “Shabbat” is common enough in English-language audio (especially in North America/UK content) to be well-represented, while “Macolet” (מכוֹלֶת) is Israeli-specific and rare in international English audio.

How Whisper (and Modern ASR) Works

Phonetic Level → Subword Tokens → Words

Architecture Overview:

```
Audio → Mel-spectrogram → Encoder → Decoder → Subword tokens → Words
```

Key Insight: Whisper doesn't have a “vocabulary” like a dictionary.

Instead: 1. **Audio encoding:** Convert sound waves → spectral features 2. **Sequence modeling:** Encoder learns phonetic patterns 3. **Token prediction:** Decoder predicts subword tokens (BPE - Byte-Pair Encoding) 4. **Token → Text:** Subword tokens combine into words

Byte-Pair Encoding (BPE) Tokenization

What is BPE? - Breaks words into frequent subword units - Common subwords become single tokens - Rare words are split into smaller pieces

Example:

Common word: "hello" → [hello] (single token)
Rare word: "Macolet" → [Mac][ol][et] (multiple tokens)

Whisper's tokenizer has ~50,000 tokens: - Common English words: Single tokens - Common names/terms: Single tokens - Rare words: Split into subwords

Why This Matters: If "Shabbat" appears frequently in training data, it becomes a **single token** in Whisper's vocabulary. If "Macolet" doesn't, it must be constructed from **phonetic subword tokens**—and this is where errors happen.

Why "Shabbat" Works But "Macolet" Doesn't

Case Study: "Shabbat"

Frequency in Training Data: - Whisper trained on 680,000 hours of audio - Sources include: - YouTube subtitles (religious/cultural content) - Podcasts (Jewish topics, interfaith discussions) - TV shows/movies (Jewish characters, cultural references) - News (stories about Israel, Judaism)

"Shabbat" appears in: - Religious content (sermons, lectures) - Cultural programming (food shows, travel vlogs) - Mainstream media (discussions of Jewish holidays)

Result: - **High frequency** → BPE tokenizer creates a token **[Shabbat]** - Whisper learns acoustic patterns for "Shabbat" - Decoder predicts **[Shabbat]** token confidently

Transcription:  "Shabbat" (correct)

Case Study: "Macolet" (מַכּוֹלֶת)

Frequency in Training Data: - "Macolet" (or "Makolet") is **Israeli-specific slang** - Rarely used in English-language media - Not commonly in international English audio - Whisper's training data skews toward: - North American English - British English - International content (but not hyper-local terms)

Result: - **Low/zero frequency** → No **[Macolet]** token - Whisper must construct from phonetic subwords - Decoder guesses: **[Mac][ol][et]** or similar - Acoustically similar words interfere (e.g., "makeup lot", "mackerel", "macho let")

Transcription: ❌ “Makeup lot” / “Maco late” / gibberish (incorrect)

The Phonetic Level: Why Errors Happen

How Whisper “Hears” Unknown Words

When you say “Macolet” (/ma-ko-let/):

1. **Acoustic encoding:** Whisper converts sound → spectral features
 - Recognizes phonemes: /m/ , /a/ , /k/ , /o/ , /l/ , /e/ , /t/
2. **Decoder prediction:** Tries to match phonemes to known tokens
 - Searches for tokens that match /ma-ko-let/ acoustically
 - Finds partial matches:
 - “Mac” (common prefix: Macintosh, McDonald’s)
 - “lot” (common word)
 - “late” (common word)
3. **Decoder outputs best guess:**
 - “Mac lot” (if it parses as two words)
 - “Macolate” (if it tries to keep as one word)
 - “Macaulay” (if it finds a similar name)

The problem: Without seeing “Macolet” in training, Whisper has no prior to favor the correct spelling.

Training Data Determines Recognition

The Rule:

If a word appears frequently enough in training data, it will be recognized reliably.

“Frequently enough” depends on: - **Raw count:** How many times it appears - **Acoustic variability:** Different speakers, accents, contexts - **Context:** Surrounding words that help disambiguation

Thresholds (Rough Estimates):

>10,000 occurrences: Very likely to be a single token → reliable recognition
1,000-10,000: May be a token or common subword sequence → good recognition
100-1,000: Likely subword split → moderate recognition (context-dependent)
<100: Definitely subword split → poor recognition (often fails)

“**Shabbat**”: Likely 10,000+ occurrences in Whisper’s training data “**Macolet**”: Likely <10 occurrences (if any)

Language Mode and Code-Switching

Your Use Case: English + Hebrew Words

Whisper’s multilingual model has language detection:

Audio → Language detection → Decoder (language-specific mode)

What happens when you speak English with Hebrew words:

Option 1: Whisper detects English - Decoder uses English tokens - Hebrew words must map to English phonetics - Result: Hebrew words often mis-transcribed

Option 2: Whisper detects Hebrew - Decoder uses Hebrew tokens - English words must map to Hebrew phonetics - Result: English words may be transliterated incorrectly

Option 3: Whisper code-switches (rare) - Decoder flips between English and Hebrew tokens - Can work if the model learned this pattern - But Whisper wasn’t explicitly trained for code-switching

Your experience: - When you say “I need to go to the Macolet,” Whisper stays in English mode - “Macolet” has no English token → phonetic guessing → error

Fine-Tuning to Fix This

How Fine-Tuning Helps:

Your fine-tuning data:

Audio: "I'm going to the Macolet to buy milk"
Text: "I'm going to the Macolet to buy milk"

What the model learns: 1. **Phonetic pattern:** /ma-ko-let/ → “Macolet” (consistent mapping)

Context: “Macolet” appears after “the” (like “the store”, “the shop”) 3. **Frequency:** If you provide 50-100 examples, “Macolet” becomes a learned pattern

Post-fine-tuning: - Whisper’s decoder learns to output “Macolet” when it hears /ma-ko-let/ - Even if “Macolet” isn’t a single token, the model learns the subword sequence - Context helps (e.g., “going to the [Macolet]” vs. “Mac” + “lot”)

Result:  Reliable transcription of “Macolet”

Vocabulary Expansion Strategies

1. Fine-Tuning (Your Best Option)

Data collection: - Record yourself using Hebrew words in English sentences - Transcribe with the correct spelling (e.g., “Macolet”) - 2-5 hours of audio with these words

Fine-tuning: - Train Whisper on your data - Model learns your code-switching patterns - Hebrew words become consistently transcribed

Benefit: - Works for ALL your Hebrew words (Macolet, misrad, etc.) - Learns your pronunciation patterns

2. Custom Tokenizer (Advanced, Not Recommended)

Concept: - Retrain Whisper’s BPE tokenizer with your vocabulary - Add “Macolet”, “misrad”, etc. as explicit tokens

Problems: - Requires retraining the entire model (not just fine-tuning) - Extremely compute-intensive - Breaks compatibility with standard Whisper

Not worth it for your use case.

3. Post-Processing (Spelling Correction)

Concept: - Let Whisper transcribe (“Mac lot”) - Apply a spell-checker or LLM to fix known errors

Implementation:

```
from faster_whisper import WhisperModel

model = WhisperModel("medium")
segments, info = model.transcribe("audio.wav")
text = " ".join([seg.text for seg in segments])

corrections = {
    "Mac lot": "Macolet",
    "miss rod": "misrad",
    "to that say hoot": "te'udat zehut",
}

for wrong, right in corrections.items():
    text = text.replace(wrong, right)

print(text)
```

Pros: - ✓ Works immediately (no training) - ✓ Easy to implement

Cons: - ✗ Manual dictionary maintenance - ✗ Fragile (Whisper might transcribe “Mac lot” differently each time) - ✗ Doesn’t generalize (new words need new rules)

Use case: Temporary fix while preparing fine-tuning data.

4. Prompt/Injection (Whisper’s Hidden Feature)

Whisper supports “initial prompt” (hint to the decoder):

```
result = model.transcribe(
    "audio.wav",
    initial_prompt="Common Hebrew words: Macolet, misrad, te'udat zehut,
                    Shabbat"
)
```

How it works: - Decoder sees these words as context - Slightly biases output toward these spellings

Effectiveness: - Modest improvement (not a silver bullet) - Works best for words that are phonetically close to transcription errors - Doesn't add new tokens, just biases existing ones

Worth trying as a quick test!

Linguistic Origin vs. Training Data

Your Question: Does Linguistic Origin Matter?

Short answer: No, training data matters.

Examples:

Word	Origin	Whisper Recognition	Reason
"Shabbat"	Hebrew	✓ Good	High frequency in English audio
"Macolet"	Hebrew	✗ Poor	Rare in English audio
"Schadenfreude"	German	✓ Good	Common in English discourse
"Fernweh"	German	✗ Poor	Rare in English discourse
"Sushi"	Japanese	✓ Excellent	Ubiquitous in English
"Omakase"	Japanese	⚠ Mixed	Growing but not universal

What determines recognition: 1. **Frequency** in English-language audio (not the word's origin) 2. **Cultural integration** (how much the word is used in English contexts) 3. **Media representation** (how often it appears in Whisper's training sources)

Hebrew words in English: - "Shabbat", "kosher", "Hanukkah" → ✓ Well-known, high frequency - "Macolet", "misrad", "te'udat zehut" → ✗ Israeli-specific, low frequency

Summary: Why Variance Exists

Your observation: > "I encounter variance in what I find [Whisper recognizing]"

Explanation:

Factor	“Shabbat” (Works)	“Macolet” (Fails)
Training data frequency	High (10k+ examples)	Low/Zero (<10 examples)
BPE tokenization	Single token [Shabbat]	Subword split [Mac][ol][et]
Phonetic ambiguity	Low (distinct sound)	High (sounds like “Mac lot”)
Cultural integration	International Jewish culture	Israeli-specific slang
Media representation	YouTube, podcasts, TV	Rare outside Israel

The variance is entirely due to training data distribution, not linguistic origin.

Practical Recommendations for You

Option 1: Fine-Tune (Best Long-Term)

Collect 2-5 hours of your speech with Hebrew words, transcribe carefully, fine-tune Whisper.

Result: All your Hebrew words (Macolet, misrad, etc.) recognized correctly.

Option 2: Initial Prompt (Quick Test)

```
result = model.transcribe(  
    "audio.wav",  
    initial_prompt="Hebrew words used: Macolet (convenience store), misrad  
    (office), te'udat zehut (ID card)"  
)
```

Result: Modest improvement (worth trying).

Option 3: Post-Processing (Interim Fix)

Maintain a dictionary of corrections, apply after transcription.

Result: Works but fragile.

Recommended Path:

1. **Now:** Use initial prompt + post-processing
 2. **Short-term:** Collect audio data with Hebrew words
 3. **Long-term:** Fine-tune Whisper (or wait for a Hebrew-English code-switching dataset to fine-tune on)
-

Bottom Line

ASR works at the phonetic/subword level, but vocabulary recognition is driven by training data frequency.

- “**Shabbat**” **works**: High frequency in Whisper’s training data (English-language audio with Jewish cultural content)
- “**Macolelet**” **fails**: Low/zero frequency (Israeli-specific, rare outside Israel)

Fine-tuning is the solution: By providing examples of your Hebrew words in English contexts, you teach Whisper to recognize them reliably.

This is exactly the use case where personal fine-tuning shines.

Note: This explanation was generated by Claude Code (claude-sonnet-4-5) for Daniel Rosehill’s STT Fine-Tuning Notebook. Whisper’s vocabulary recognition is probabilistic and depends on training data distribution. For reliable transcription of code-switched speech (English + Hebrew), fine-tuning is the most effective solution. Consider creating a dataset of 2-5 hours with Hebrew words you use regularly, ensuring diverse contexts and pronunciations. Initial prompts can provide modest improvements as an interim measure.