Day 3 · Speech recognition: a tiny CTC model from scratch, then Whisper and a Khmer ASR judge, scored by CER

Goal in one line: understand how a network turns 100 audio frames per second into a handful of Khmer characters, build the CTC machinery by hand, and learn to *score* a transcript with the character error rate (CER), the way the anonymization project does.

What you will build: a NumPy CTC forward algorithm (checked against PyTorch), a tiny BiGRU-CTC recognizer trained on 36 Khmer clips, a prefix beam search decoder, and an evaluation tool `score_system()` + `show_diff()` that reproduces the project's CER leaderboard with speaker-bootstrap confidence intervals.

Time: ~3.5 h

- **BiGRU** (bidirectional GRU): a recurrent layer that reads the frames in order and carries a memory along. Two run at once, one forwards and one backwards, so every step knows the frames before and after it.

By the end you can ...

- explain the alignment problem (many frames, few characters) and the two standard answers: CTC and attention encoder-decoder;
- treat Khmer text as a sequence of Unicode code points, normalize it, and say why we report CER rather than WER;
- write the CTC collapse rule, enumerate alignments, and run the CTC forward algorithm (the alpha trellis) by hand;
- train a small CTC model, read its spiky per-frame posteriors, decode it greedily and with a beam;
- run Whisper on Khmer, recognize hallucination, and know why the project uses Qwen3-ASR as its judge;
- compute CER per system and per corpus with a 95 % confidence interval that resamples *speakers*, not utterances.

How to read this notebook (for the team)

You see	What it is	Do you need it?
Purple heading	a new section and its explanation	**yes**: read the bold lines and the "Why it matters" line
Visualization Version	pictures (some animated) that explain the section	**yes**: the first image of each section is the fastest way in
Additional reading	what the next code cell does and what its output means, plus extra notes	optional: read just the **In short** lines if you are busy

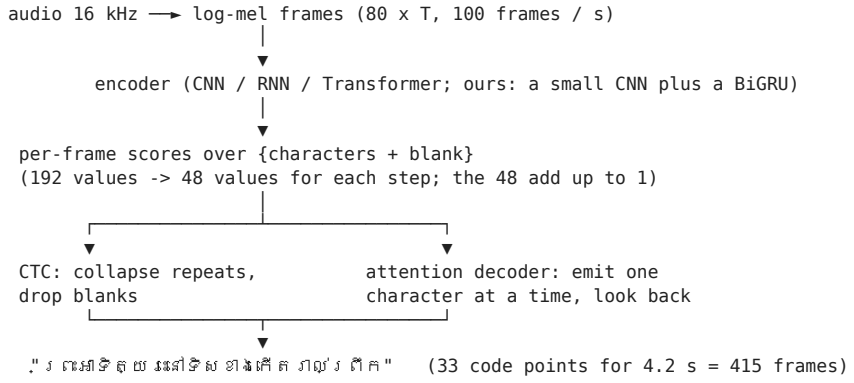
code cell + output	the real computation on our 48 Khmer clips and the 430-clip CER table	run it, look at the result

Fast path (≈ 45 min): purple text → first Visualization image → "In short" lines. **Full path (≈ 3.5 h):** read everything and run every cell top to bottom.

> The numbers in the explanations come from the saved run in this file. A re-run on another machine can move CER by a point or two (the held-out set has only 12 clips) and the training time changes.
>
> **Section 6 (Whisper, Qwen3-ASR):** these models need a download that was blocked where this file was built. Three cells there keep **the output from Vicheese's own run**. They are marked in the text. Every other cell was run here from top to bottom.

1. The big idea

Goal: a network turns about **100 audio frames per second** into a few dozen Khmer characters. Nothing tells it which frame belongs to which character.



A 4-second Khmer sentence is about 400 feature frames but only ~30-50 characters. Nothing tells us which frames belong to which character, and the ratio changes with speaking rate. This is the **alignment problem**. Two families of models solve it:

- **CTC** (Connectionist Temporal Classification): the network outputs one symbol *per frame* from an alphabet that includes a special **blank**. A fixed rule collapses the frame sequence to text. The loss sums over *every* frame alignment that collapses to the reference, so no hand-made alignment is needed. Fast, simple, monotonic. Used by wav2vec 2.0, DeepSpeech, many production systems.
- **Attention encoder-decoder** (Whisper, Qwen3-ASR): a decoder emits characters/tokens one at a time, conditioning on the previous ones and on an attention-weighted view of the encoder. It learns an implicit language model, which makes it fluent, and sometimes *too* fluent (hallucination).

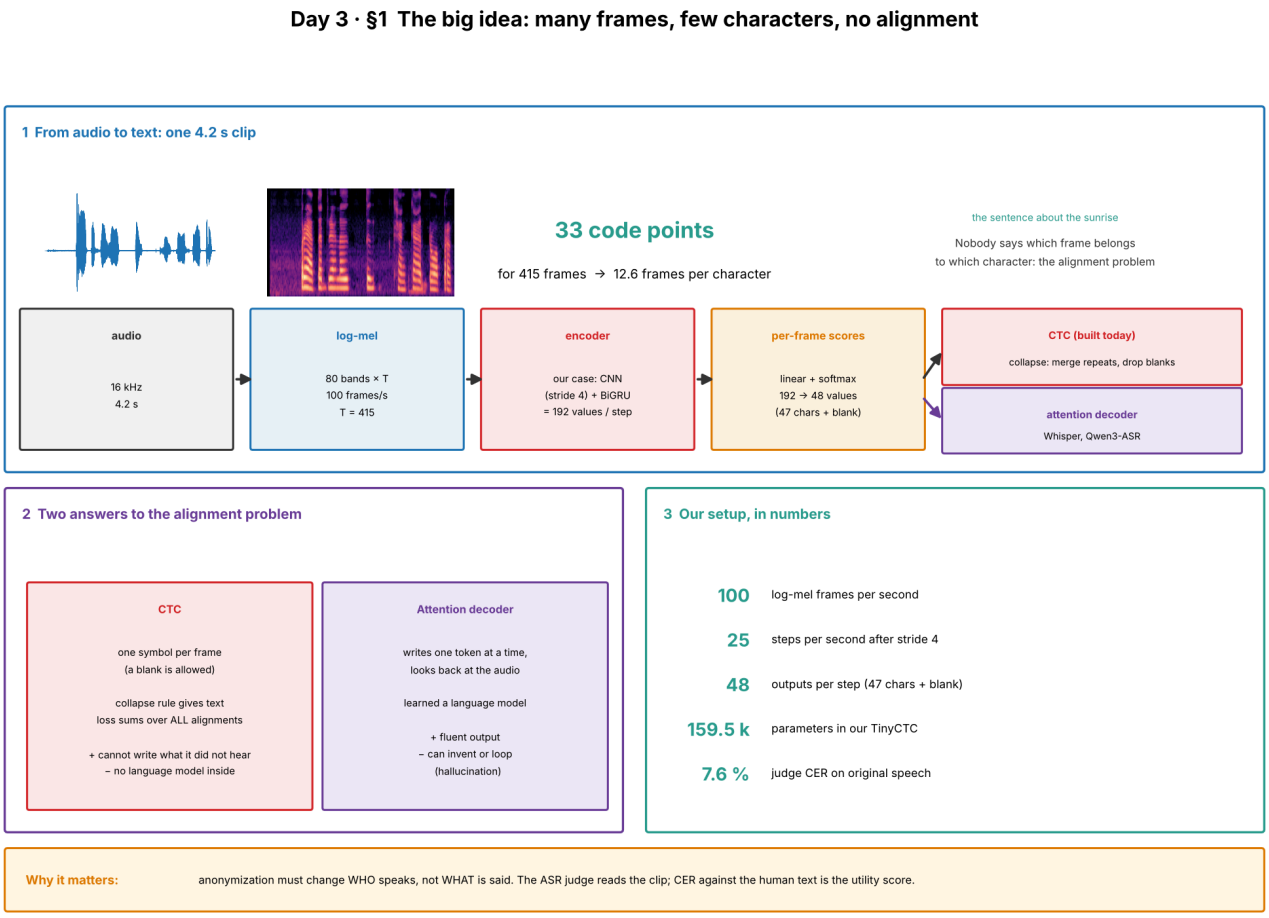
Today we build CTC by hand because it is small enough to see every piece, then use the big attention models as black boxes.

In our research. Anonymization changes *who* is speaking; it must not change *what* is said. The project's judge is a Khmer ASR model (Qwen3-ASR). It transcribes the original clip and the

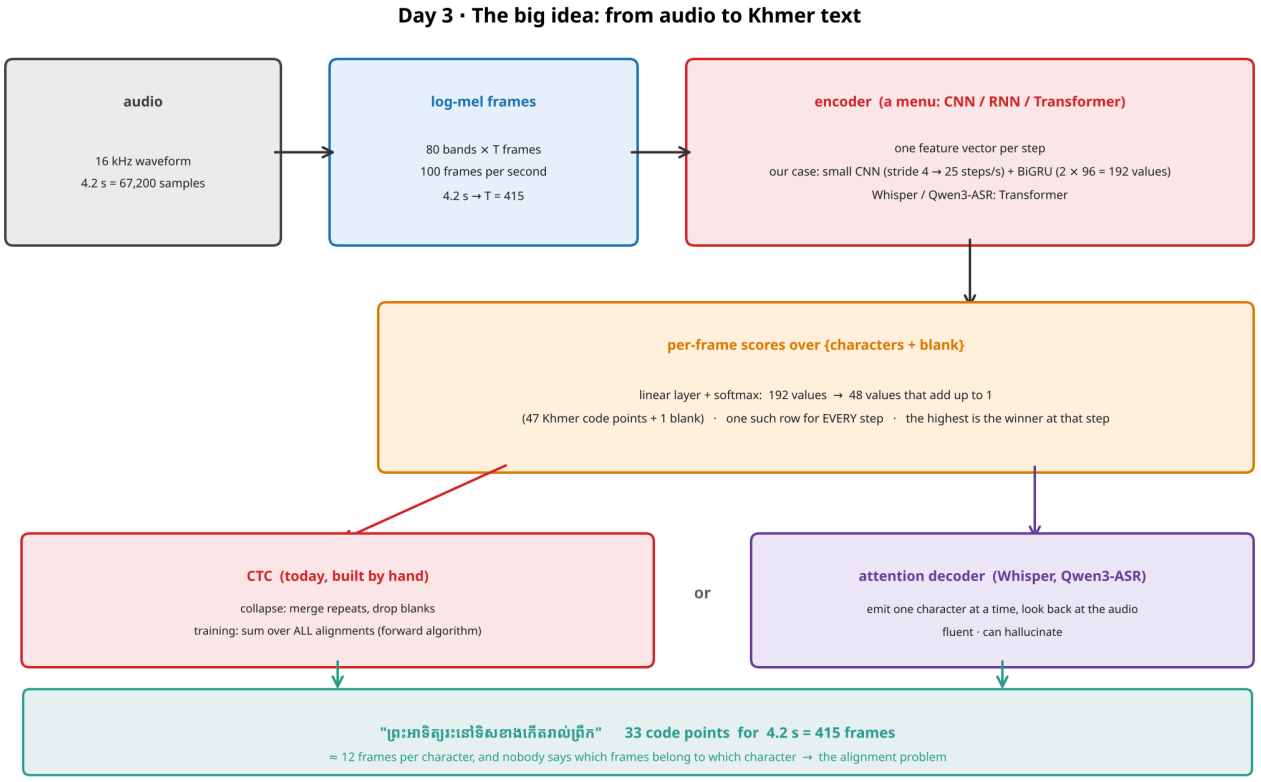
anonymized clip, and the distance between the transcript and the human reference, measured as CER, is the "utility" score on the leaderboard. Original speech scores CER ≈ 7.6 %; a system that pushes that to 40 % has destroyed a lot of content. Everything in sections 2 and 7 is exactly what produces the `cer` column in `leaderboard.csv`.

Visualization Version

1st image is the summary. The 2nd is Vicheese's own diagram of the same pipeline.



Vicheese's own big-idea picture (the same pipeline; 415 frames ÷ 33 code points = 12.6 frames per character)



Additional reading: you can read all of it or just read the "In short"

CTC vs the attention decoder, in plain words

In short CTC writes only what the frames support (typo-like errors); the attention decoder is fluent but can invent text.

- **CTC is the dumb-but-honest one.** Every step votes on its own; no step talks to any other. You read the winner at each step, which gives a long string like "ព ព blank blank◌ blank រ", and then apply two rules: merge repeats, delete blanks. Out comes "រ ព". That is the whole decoder. There is nothing to train in it.
- **The attention decoder is the opposite.** It is a second network that writes one character at a time, like a person. It looks at the whole encoder output and asks "which part of the audio should I look at now?". It picks a character, then asks again for the next one, using the characters it already wrote as context. It stops when it emits an end symbol. It is free to look anywhere, and it blends what it hears with what it just wrote. That is when you get repeated or invented words.
- **That is the trade-off.** CTC can only write what it heard, so its errors look like typos. The attention decoder has a language model inside, so it writes fluent Khmer, but it can keep writing after the audio has stopped. That is the hallucination the judge can produce.

Additional reading: you can read all of it or just

read the "In short"

What this cell does

In short load the libraries (numpy, pandas, librosa, PyTorch), set the paths and seeds, import the Day 1 helpers, and read the four metadata tables.

- `probe` : the 48 clips we train and test on (8 speakers × 6 sentences). `demo` : 20 clips with text. `trans` : 430 clips with the reference and one judge transcript per anonymization system. `lb` : the project leaderboard.
- `from src.utils import ...` : the Day 1 helpers (`load` , `logmel` , `cer` , ...).
- Seeds are fixed (`np.random.seed(0); torch.manual_seed(0)`) so the training result below can be repeated.

What the output means

In short it prints the table sizes: probe (48, 6), demo (20, 7), transcripts (430, 11), and the PyTorch version.

- If this cell fails, nothing below runs. The exact PyTorch version number does not matter.

```
import os, sys, warnings, time, difflib, unicodedata
os.environ.setdefault("OMP_NUM_THREADS", "4")   # before numpy / torch
warnings.filterwarnings("ignore")
sys.path.insert(0, "..")
from pathlib import Path
from collections import Counter, defaultdict

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import librosa
from IPython.display import HTML, display

from src.utils import play, show_wave, show_spec, median_f0, load, logmel, cer, HOP

SR = 16000
DATA = Path("../data")
OUT = Path("../outputs/day3"); OUT.mkdir(parents=True, exist_ok=True)
np.random.seed(0); torch.manual_seed(0); torch.set_num_threads(4)

probe = pd.read_csv(DATA / "meta/probe_clips.csv")      # 48 clips, 8 speakers x 6 sentences
demo  = pd.read_csv(DATA / "meta/demo_clips.csv")       # 20 clips with text
trans = pd.read_csv(DATA / "meta/transcripts.csv")      # 430 clips: reference + hyp_<system>
lb    = pd.read_csv(DATA / "meta/leaderboard.csv").set_index("system")
print("probe", probe.shape, "| demo", demo.shape, "| transcripts", trans.shape, "| torch", torch.__version__)
```

```
probe (48, 6) | demo (20, 7) | transcripts (430, 11) | torch 2.14.1+cu130
```

2. Khmer text as a sequence

To a network, text is a list of integers. For Khmer the natural unit is the **Unicode code point**. The word ខ្ញុំ ("I") looks like one syllable but is **5 code points**: KHA, COENG (the "subscript" sign that turns the next consonant into a subscript), NYO, the vowel sign U, and NIKAHIT (the final -m).

Khmer writes no spaces between words (spaces mark phrases, and writers use them inconsistently). So *word* error rate would depend on a word segmenter that itself makes mistakes. **We score on code points instead:**

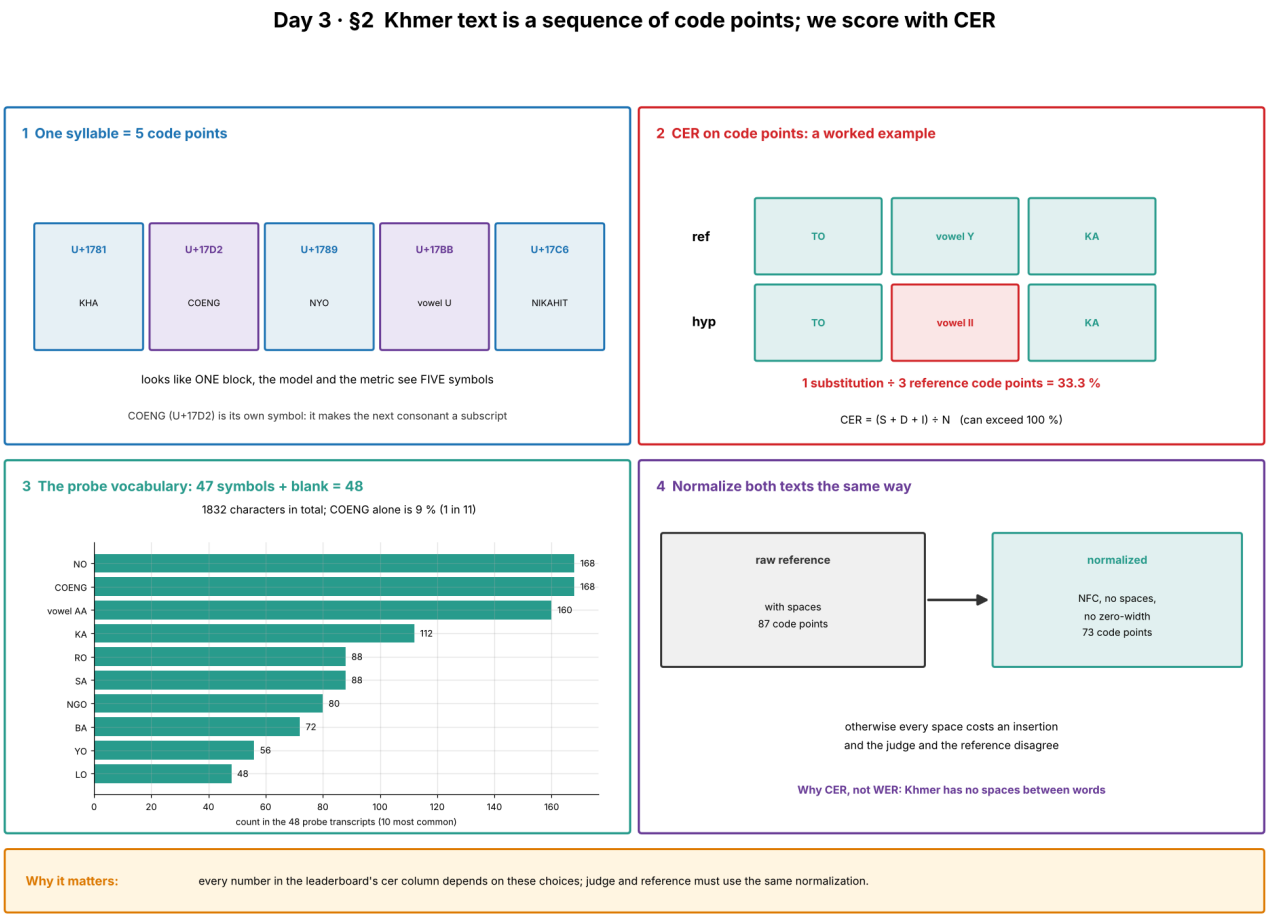
$$\mathrm{CER} = \frac{S + D + I}{N} = \frac{\text{edit distance}(\text{ref}, \text{hyp})}{\text{len}(\text{ref})}$$

S, D, I are substitutions, deletions and insertions on the cheapest path, and N is the reference length. **CER can exceed 100 %** when the hypothesis is much longer than the reference (insertions are unbounded).

Before scoring we **normalize** both strings the same way: Unicode NFC (so the same syllable always has the same byte sequence), then remove all whitespace and zero-width characters (U+200B is common in Khmer web text and invisible). Small worked example: ref = `ក្ត` (3 code points), hyp = `ក្ត` → 1 substitution → CER = 1/3 = 33.3 %.

Why it matters. Every number in the leaderboard's `cer` column depends on these choices. Different normalization gives different numbers, so the judge and the reference must agree on them.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define `normalize()`, `edit_distance()` and `cer_norm()`, then take the word ខ្ញុំ apart, check the worked CER example, and show what normalization does to a real reference.

- `normalize`: NFC, then drop whitespace and zero-width characters (a missing hypothesis, NaN, becomes an empty string).
- `edit_distance`: the classic Levenshtein dynamic programme, on code points.
- `cer_norm`: CER in % after our normalization, using the project's `jiwer`-based `cer` helper.
- The last lines compare our own edit distance with `jiwer`, and show the length of one demo reference before and after `normalize`.

What the output means

In short ខ្ញុំ is **5 code points**; ref `ទឹក` vs hyp `ទឹក` has edit distance **1** → CER **33.3 %**, and `jiwer` agrees; the demo reference shrinks from **87 to 73** characters after normalization.

- The 14 removed characters are the phrase spaces. Without normalization each one would cost an insertion.
- `U+17D2 KHMER SIGN COENG` is a code point of its own.

```
ZW = {"\u200b", "\u200c", "\u200d", "\ufeff"}            # zero-width space/non-joiner/joiner, BOM

def normalize(s):
    # NFC, then drop every whitespace and zero-width char. NaN (missing hypothesis) becomes "".
    if s is None or (isinstance(s, float) and np.isnan(s)):
        return ""
    s = unicodedata.normalize("NFC", str(s))
    return "".join(ch for ch in s if not ch.isspace() and ch not in ZW)

def edit_distance(a, b):
    # Levenshtein on code points, classic O(len a * len b) dynamic programme.
    prev = list(range(len(b) + 1))
    for i, ca in enumerate(a, 1):
        cur = [i]
        for j, cb in enumerate(b, 1):
            cur.append(min(prev[j] + 1,              # deletion
                           cur[j - 1] + 1,           # insertion
                           prev[j - 1] + (ca != cb)))# substitution (0 if equal)
        prev = cur
    return prev[-1]

def cer_norm(ref, hyp):
    # CER in % on normalized code points; uses the project helper (jiwer) after our normalization.
    r, h = normalize(ref), normalize(hyp)
    return 100.0 * cer(r, h) if r else float("nan")

# one word, code point by code point: notice COENG (the subscript marker) is a separate code point
word = "ខ្ញុំ"
for ch in word:
    print(f"U+{ord(ch):04X}  {unicodedata.name(ch)}")
print("visible syllable:", word, "| code points:", len(word))

# worked example and a check that our edit_distance agrees with jiwer (src.utils.cer)
ref, hyp = "ទឹក", "ទឹក"
print("edit distance", edit_distance(ref, hyp), "-> CER %.1f %%" % (100 * edit_distance(ref, hyp) / len(ref)),
      "| jiwer says %.1f %%" % cer_norm(ref, hyp))

# normalization matters: the demo references contain spaces (phrase breaks), the probe ones mostly do not
raw = demo.text.iloc[0]
print("raw :", repr(raw[:40]), "len", len(raw))
print("norm:", repr(normalize(raw)[:40]), "len", len(normalize(raw)))
```

```
U+1781  KHMER LETTER KHA
U+17D2  KHMER SIGN COENG
U+1789  KHMER LETTER NYO
U+17BB  KHMER VOWEL SIGN U
U+17C6  KHMER SIGN NIKAHIT
visible syllable: ខ្ញុំ | code points: 5
edit distance 1 -> CER 33.3 % | jiwer says 33.3 %
raw : 'ឈោក អភិបាលខេត្ត កណ្តាល បញ្ជូន មន្ត្រី ជ' len 87
norm: 'ឈោកអភិបាលខេត្តកណ្តាលបញ្ជូនមន្ត្រីជំនាញច,' len 73
```

Additional reading: you can read all of it or just read the "In short"

The five code points of ខ្ញុំ

In short the five printed names, drawn as the characters they make.

- U+1781 KHMER LETTER KHA → ខ
- U+17D2 KHMER SIGN COENG → ◌្
- U+1789 KHMER LETTER NYO → ញ
- U+17BB KHMER VOWEL SIGN U → ◌ុ
- U+17C6 KHMER SIGN NIKAHIT → ◌ំ

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short build the character vocabulary from the 48 probe transcripts, with id 0 reserved for the CTC blank, and print some statistics.

- `counts` : how often each code point appears. `vocab = ["<blank>"] + sorted characters` ; `char2id` maps a character to its number.
- The last two lines print the sentence lengths and the average frames per sentence at 100 frames/s.

What the output means

In short **vocabulary size 48** (47 code points + blank); the most common symbols are ន (168), ◌្ COENG (168), ◌ា (160), ក (112), រ (88); sentences are **26-50** code points long and **356-450** frames.

- Only **6 sentences**, so only 5 different lengths (two sentences have the same length): 26, 36, 38, 43, 50.
- `frames per sentence: [443, 356, 407, 450, 433, 400]` is the mean clip length × 100.

```
# Character vocabulary from the 48 probe transcripts (+ blank for CTC). Notice how few symbols Khmer needs.
probe["norm"] = probe.text.map(normalize)
counts = Counter("".join(probe.norm))
chars = sorted(counts)
vocab = ["<blank>"] + chars                        # id 0 is reserved for the CTC blank
char2id = {c: i for i, c in enumerate(vocab)}
print("vocab size (incl. blank):", len(vocab))
print("5 most common:", [(c, n, unicodedata.name(c).replace("KHMER ", "")) for c, n in counts.most_common(5)])
print("sentence lengths (code points):", sorted(set(len(s) for s in probe.norm)))
print("frames per sentence (100/s):", [int(100 * d) for d in probe.groupby("sentence_id").duration.mean().round(2)])
```

```
vocab size (incl. blank): 48
5 most common: [('ន', 168, 'LETTER NO'), ('◌្', 168, 'SIGN COENG'), ('◌ា', 160, 'VOWEL SIGN AA'), ('ក', 112, 'LETTER KA'),
('រ', 88, 'LETTER RO')]
```

```
sentence lengths (code points): [26, 36, 38, 43, 50]
frames per sentence (100/s): [443, 356, 407, 450, 433, 400]
```

Additional Reading

In short 47 code points + blank is a tiny alphabet; a healthy CTC model is mostly blank (about 90 % at 100 frames/s, about 2 blank frames per character at our 25 frames/s).

How to read the numbers. The whole probe set uses only **47 distinct code points** (plus blank, so 48). A sentence of 26-50 code points spans 220-590 frames at 100 frames/s. So a CTC model must emit blank on roughly **90 %** of frames. Keep that in mind for the posterior plot in section 4. After our stride-4 conv the model runs at 25 frames/s, which still leaves about 2 blank frames for every character.

The top five mix frequent consonants (NO, KA, RO) with the subscript marker COENG and the vowel sign AA. **COENG alone is 1 symbol in 11** (168 of 1,832 characters). A model that mishandles subscripts (common on noisy phone audio) loses points every few syllables.

3. CTC explained by hand

CTC adds one symbol, the blank (`-`), to the alphabet. The network outputs one symbol per frame. The **collapse rule** $\mathcal{B}$ turns that frame sequence (a *path*) into text:

1. merge consecutive identical symbols,
2. delete blanks.

So `--kk-aa-a--` → `-k-a-a-` → `kaa`. The blank is what lets us write a double letter: `aa` without a blank between collapses to `a`; `a-a` stays `aa`.

Many paths collapse to the same text. The CTC loss is the negative log of the **sum over all those paths**:

$$P(\mathbf{y} \mid \mathbf{x}) = \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{y})} \prod_{t=1}^{T} p_t(\pi_t)$$

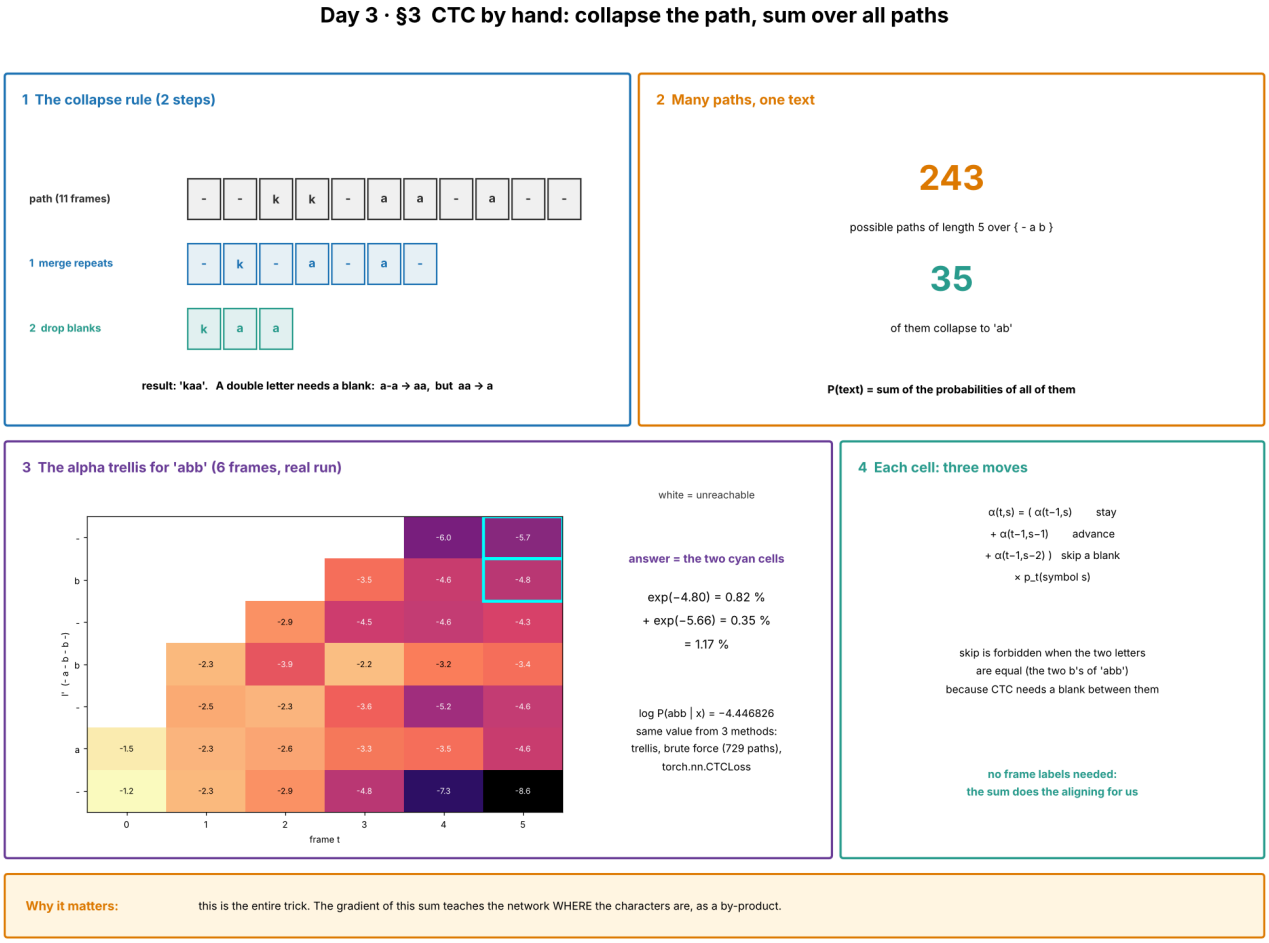
Summing over exponentially many paths is cheap with dynamic programming (the **forward algorithm**). Write the target with blanks around every character, $\ell' = (-, y_1, -, y_2, -, \ldots, -)$. Let $\alpha_t(s)$ be the total probability of all path prefixes of length t that end at position s of ℓ'. Each step can stay (s), advance one ($s-1$), or skip a blank ($s-2$, only allowed when ℓ'_s is a character different from ℓ'_{s-2}):

$$\alpha_t(s) = \big(\alpha_{t-1}(s) + \alpha_{t-1}(s-1) + \alpha_{t-1}(s-2)\big)\, p_t(\ell'_s)$$

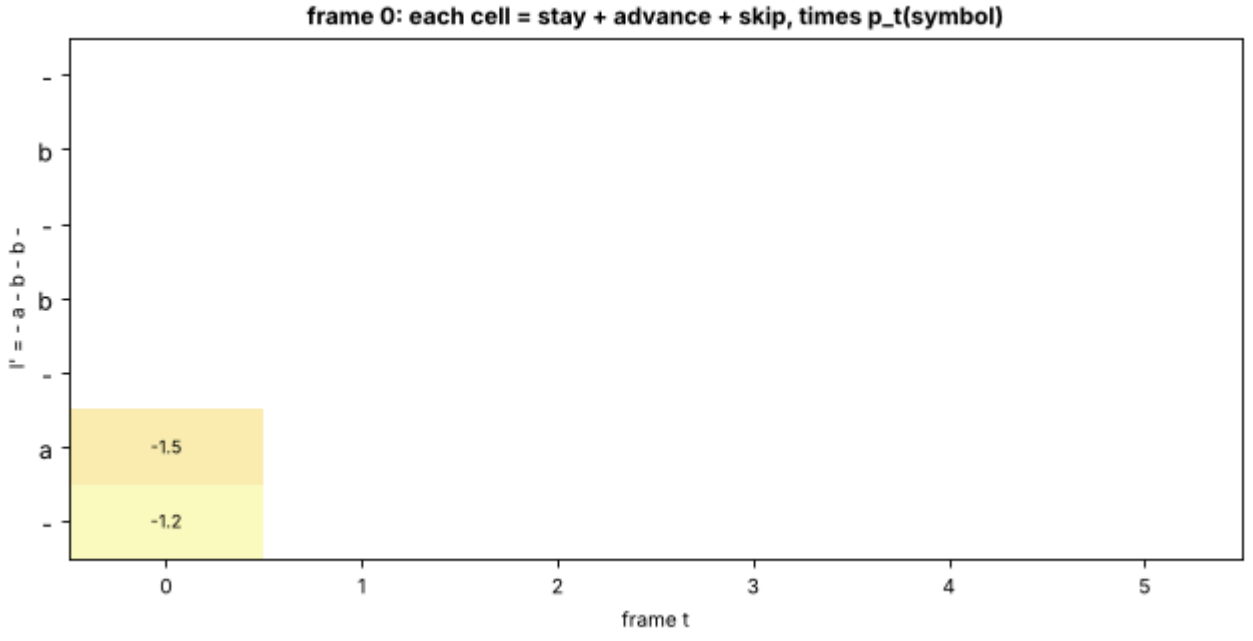
Why it matters. This is the entire trick. No frame labels, no alignment file. The gradient of this sum teaches the network *where* characters are, as a by-product.

Visualization Version

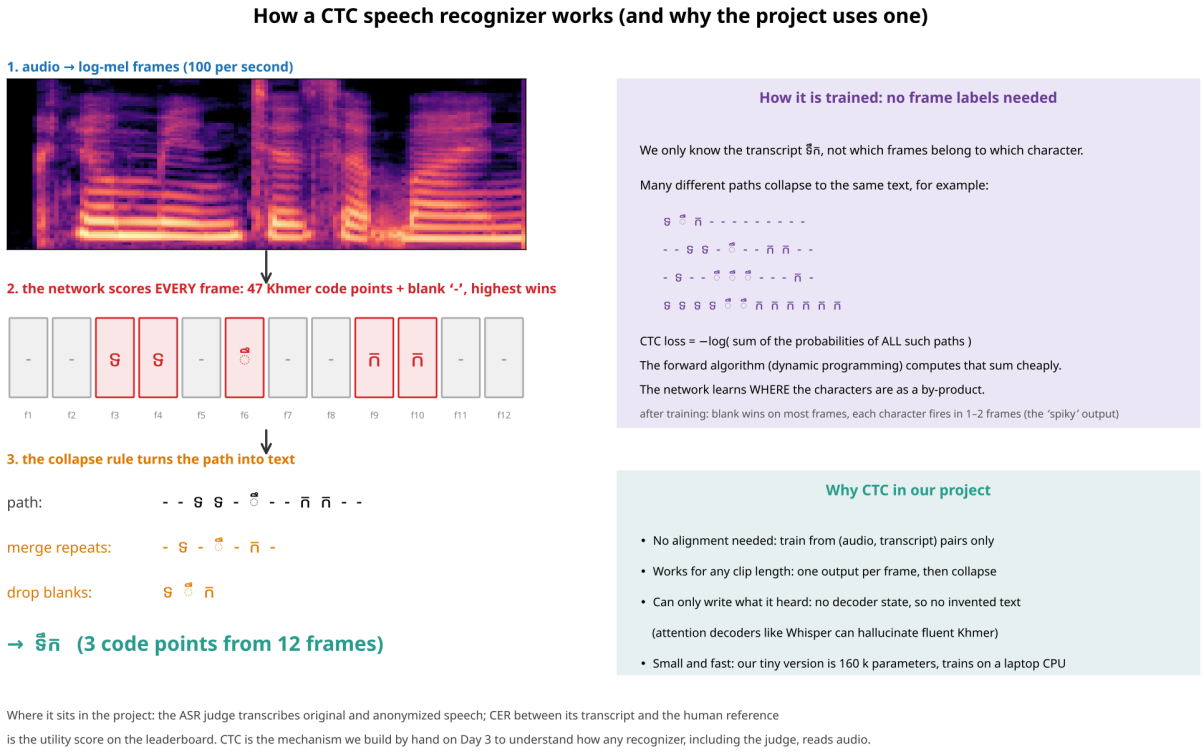
1st image is the summary. Then the trellis animation and Vicheese's CTC picture.



Animated: the alpha trellis fills in frame by frame (the real run of the next cells)



Vicheese's picture: how a CTC recognizer works



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define the collapse rule `collapse()`, test it on three strings, and list every path of length 5 over `{-, a, b}` that collapses to `ab`.

- `collapse`: keep a symbol only if it differs from the previous one and is not a blank.
- `itertools.product("-ab", repeat=5)` makes all $3^5 = 243$ paths; we keep the ones where `collapse(path) == "ab"`.

What the output means

In short `--kk-aa-a--` → **kaa**; `kaa` → **ka** (repeats merge); `ka-a` → **kaa** (a blank keeps the two a's); **35 of 243** paths collapse to `ab`.

- Many paths, one text. That is why the loss has to be a **sum** over paths.
- The 35 paths are printed one by one, e.g. `---ab`, `--a-b`, `aaabb`, `abbbb`.

```
# testing collapse function
def collapse(path, blank="-"):
    # the CTC collapse rule B: merge repeats, then drop blanks
    out, prev = [], None
    for s in path:
        if s != prev and s != blank:
            out.append(s)
        prev = s
```

```
    return "".join(out)

print('collapse("--kk-aa-a--") =', collapse("--kk-aa-a--"))
print('collapse("kaa")         =', collapse("kaa"), "   <- repeats merge without a blank")
print('collapse("ka-a")        =', collapse("ka-a"), "  <- blank separates the two a's")

# Enumerate all paths of length T=5 over {-,a,b} that collapse to "ab": many-to-one
import itertools
T = 5
paths = ["".join(p) for p in itertools.product("-ab", repeat=T)]
good = [p for p in paths if collapse(p) == "ab"]
print(f"\n{len(paths)} possible paths of length {T}, {len(good)} of them collapse to 'ab':")
print(" ".join(good))
```

```
collapse("--kk-aa-a--") = kaa
collapse("kaa")         = ka    <- repeats merge without a blank
collapse("ka-a")        = kaa   <- blank separates the two a's

243 possible paths of length 5, 35 of them collapse to 'ab':
---ab --a-b --aab --ab- --abb -a--b -a-b- -a-bb -aa-b -aaab -aab- -aabb -ab-- -abb- -abbb a---b a--b- a--bb a-b-- a-bb- a-bb
b aa--b aa-b- aa-bb aaa-b aaaab aaab- aaabb aab-- aabb- aabbb ab--- abb-- abbb- abbbb
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short compute "the probability that the model says `abb` " with the forward algorithm, then prove it two other ways.

- **Fake scores:** 6 frames × 3 symbols (blank, a, b) of random numbers, turned into probabilities per frame. No audio, no model; the point is the arithmetic.
- **`ctc_forward` , the trellis:** it writes the target with blanks around every character, `- a - b - b -` , and fills a 6 × 7 table `alpha` . `alpha[t, s]` = total probability of every partial path that has reached position `s` after frame `t` . Each cell is built from three moves from the previous frame: stay on the same position, advance one, or skip over a blank. The skip is forbidden between two identical letters. This is why `abb` was chosen: the two b's must be written `b - b` . The final answer adds the two legal end points: finishing on the last blank or on the last b.
- **Check 1, brute force:** list all 3^6 = 729 paths, keep those that collapse to `abb` , add their probabilities.
- **Check 2, PyTorch:** call `torch.nn.CTCLoss` on the same scores.

What the output means

In short **log P(abb|x) = −4.446826**: with these random scores the model would produce `abb` with probability e^−4.45 ≈ **1.2 %**. All three methods agree to six decimals.

- **loss = 4.446826:** CTC loss is just minus that log-probability. Training pushes it down, which pushes the probability of the right transcript up.
- All three numbers match. That is the real result of the cell: the trellis equals the brute-force sum, and PyTorch's built-in loss is exactly this sum. From here on you can trust `nn.CTCLoss` as a black box, knowing what is inside it.
- The probability is low because the scores are random, as you would expect from an untrained model.

```python
# CTC forward algorithm in NumPy (log domain) on a toy: vocab = [blank, a, b], target "abb", T = 6 frames.
# "abb" is a good test: the two b's force a blank between them, so the s-2 skip is NOT allowed there.
rng = np.random.default_rng(0)
V, T = 3, 6
logits = rng.normal(size=(T, V))
logp = logits - np.log(np.exp(logits).sum(-1, keepdims=True))   # per-frame log-probabilities (T, V)
target = [1, 2, 2]                                              # "abb"

def ctc_forward(logp, target, blank=0):
    # returns log P(target | x) and the full alpha trellis (T, S) in log domain
    ext = [blank]
    for c in target:
        ext += [c, blank]                       # l' = - a - b - b -
    S, T = len(ext), logp.shape[0]
    alpha = np.full((T, S), -np.inf)
    alpha[0, 0] = logp[0, ext[0]]               # start with blank ...
    alpha[0, 1] = logp[0, ext[1]]               # ... or with the first character
    for t in range(1, T):
        for s in range(S):
            cands = [alpha[t - 1, s]]                                   # stay
            if s >= 1:
                cands.append(alpha[t - 1, s - 1])                       # advance
            if s >= 2 and ext[s] != blank and ext[s] != ext[s - 2]:
                cands.append(alpha[t - 1, s - 2])                       # skip the blank between two different chars
            alpha[t, s] = np.logaddexp.reduce(cands) + logp[t, ext[s]]
    ll = np.logaddexp(alpha[T - 1, S - 1], alpha[T - 1, S - 2])         # end on final blank or final char
    return ll, alpha, ext

ll, alpha, ext = ctc_forward(logp, target)
print("NumPy  log P(abb|x) = %.6f   -> loss = %.6f" % (ll, -ll))

# check 1: brute force over all 3^6 = 729 paths
probs = np.exp(logp)
brute = sum(np.prod([probs[t, c] for t, c in enumerate(p)])
            for p in itertools.product(range(V), repeat=T)
            if collapse("".join("-ab"[c] for c in p)) == "abb")
print("brute  log P(abb|x) = %.6f" % np.log(brute))

# check 2: torch.nn.CTCLoss (expects (T, batch, V) log-probs, targets as a flat tensor)
ctc = nn.CTCLoss(blank=0, reduction="sum")
loss_t = ctc(torch.tensor(logp, dtype=torch.float32)[:, None, :], torch.tensor([target]),
             torch.tensor([T]), torch.tensor([len(target)]))
print("torch  loss        = %.6f" % loss_t.item())
```

```
NumPy  log P(abb|x) = -4.446826   -> loss = 4.446826
brute  log P(abb|x) = -4.446826
torch  loss        = 4.446826
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short draw the alpha trellis from the previous cell as a heatmap: rows = positions in `- a - b - b -`, columns = frames, colour and number = log α.

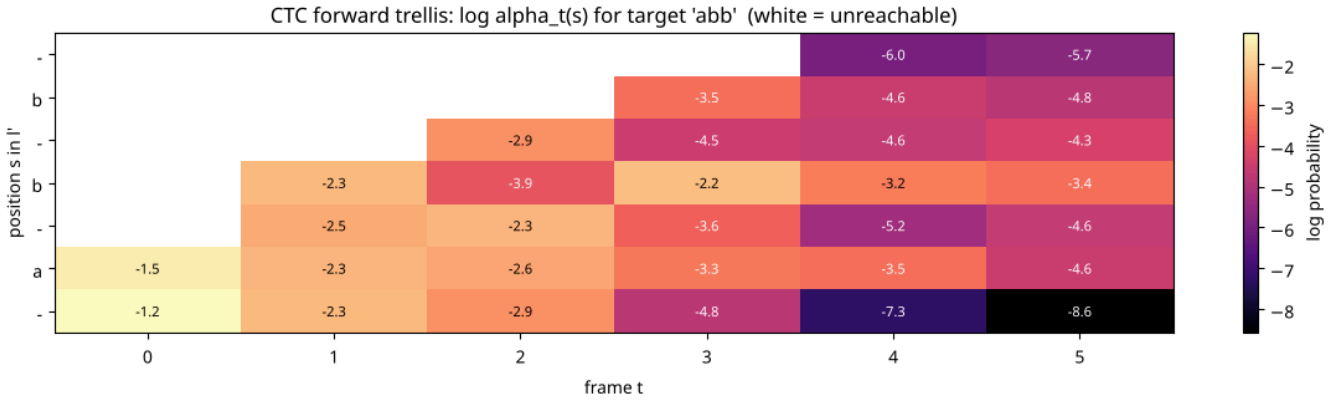
- Cells that cannot be reached are left white (`NaN`). Each reachable cell shows its log-probability.

What the output means

In short the answer is in the **top-right corner**: the last two cells are **−4.8** (end on the final b) and **−5.7** (end on the final blank).

- exp(−4.80) = 0.82 % and exp(−5.66) = 0.35 %. Added: **1.17 % ≈ 1.2 %**, the same number as the previous cell.
- See the How-to-read notes below.

```python
# The alpha trellis: rows = positions in l' (blank, a, blank, b, blank, b, blank), columns = frames.
fig, ax = plt.subplots(figsize=(12, 3.5))
A = np.where(np.isfinite(alpha), alpha, np.nan).T                # (S, T), impossible cells = NaN (white)
im = ax.imshow(A, aspect="auto", cmap="magma", origin="lower")
ax.set_yticks(range(len(ext))); ax.set_yticklabels(["-ab"[c] for c in ext])
ax.set_xticks(range(T)); ax.set_xlabel("frame t"); ax.set_ylabel("position s in l'")
ax.set_title("CTC forward trellis: log alpha_t(s) for target 'abb'  (white = unreachable)")
for s in range(len(ext)):
    for t in range(T):
        if np.isfinite(alpha[t, s]):
            ax.text(t, s, f"{alpha[t, s]:.1f}", ha="center", va="center", fontsize=8,
                    color="white" if alpha[t, s] < np.nanmax(A) - 2 else "black")
plt.colorbar(im, ax=ax, label="log probability"); plt.tight_layout(); plt.show()
```



Additional Reading

In short time runs left to right, `- a - b - b -` bottom to top; white = cannot be reached yet; the answer is the sum of the two top-right cells.

How to read this plot. Time runs left to right. The extended target `- a - b - b -` runs bottom to top. Only two cells are reachable at $t = 0$ (start on a blank or on `a`). The reachable band grows by at most two rows per frame. So the **top-left triangle is white**: you cannot have spelled `a b b` by frame 1. In the bottom right nothing is white: staying on the first blank is still possible at frame 5, but it is a very small number (−8.6, the darkest cell), because that path never wrote any character.

Look at the row of the **second b** (position 5). It can only be reached from the blank below it (position 4) or from itself, never by skipping from the first b (position 3). That is exactly the "no skip between identical characters" rule. The final log-likelihood is the log-sum of the two top-right cells. Darker = less probable. The values are logs, so a difference of 2.3 means 10×.

Everything the network learns comes from $\partial(-\log P)/\partial p_t(k)$, which is just (alpha × beta) renormalized. Frames that lie on many good paths get the strongest push.

Additional reading: you can read all of it or just read the "In short"

Which cell is the answer?

In short only the top-right corner is the answer; the other cells are running totals.

The final answer is just one number, **1.2 %**. It is read off the top-right corner. All the other cells are partial progress on the way there, not answers.

- Each cell answers a smaller question. For example, "position 3 (the first b), frame 2" means: what is the chance that after 3 frames we have written "a b", ending on the first b? Every cell is a different "how far have we got, by when", so the numbers differ.
- Think of a running bank balance. Each day has its own balance, but the only one you care about is the one on the last day. The earlier ones exist so you do not have to redo all the maths from scratch.
- The colours show how big each partial number is. Light yellow = a large probability, dark purple = a small one.

Additional reading: you can read all of it or just read the "In short"

How the two end cells give the answer

In short add the two top-right cells: 0.82 % + 0.35 % = 1.17 % ≈ 1.2 %.

The two cells that give the final probability of **abb** sit in the last frame, in the last two rows. The paths that reach them have already spelled the whole "a b b". One group ends on the final b, the other adds a blank after it. Both collapse to "abb", so you **add** them.

Any other cell in the last column, like −8.6 at the bottom, is a path that ran out of frames before finishing the word. Those do not count.

- −4.8 → 0.82 %
- −5.7 → 0.35 %
- added → 1.17 % ≈ **1.2 %**

4. A tiny CTC model on 36 Khmer clips

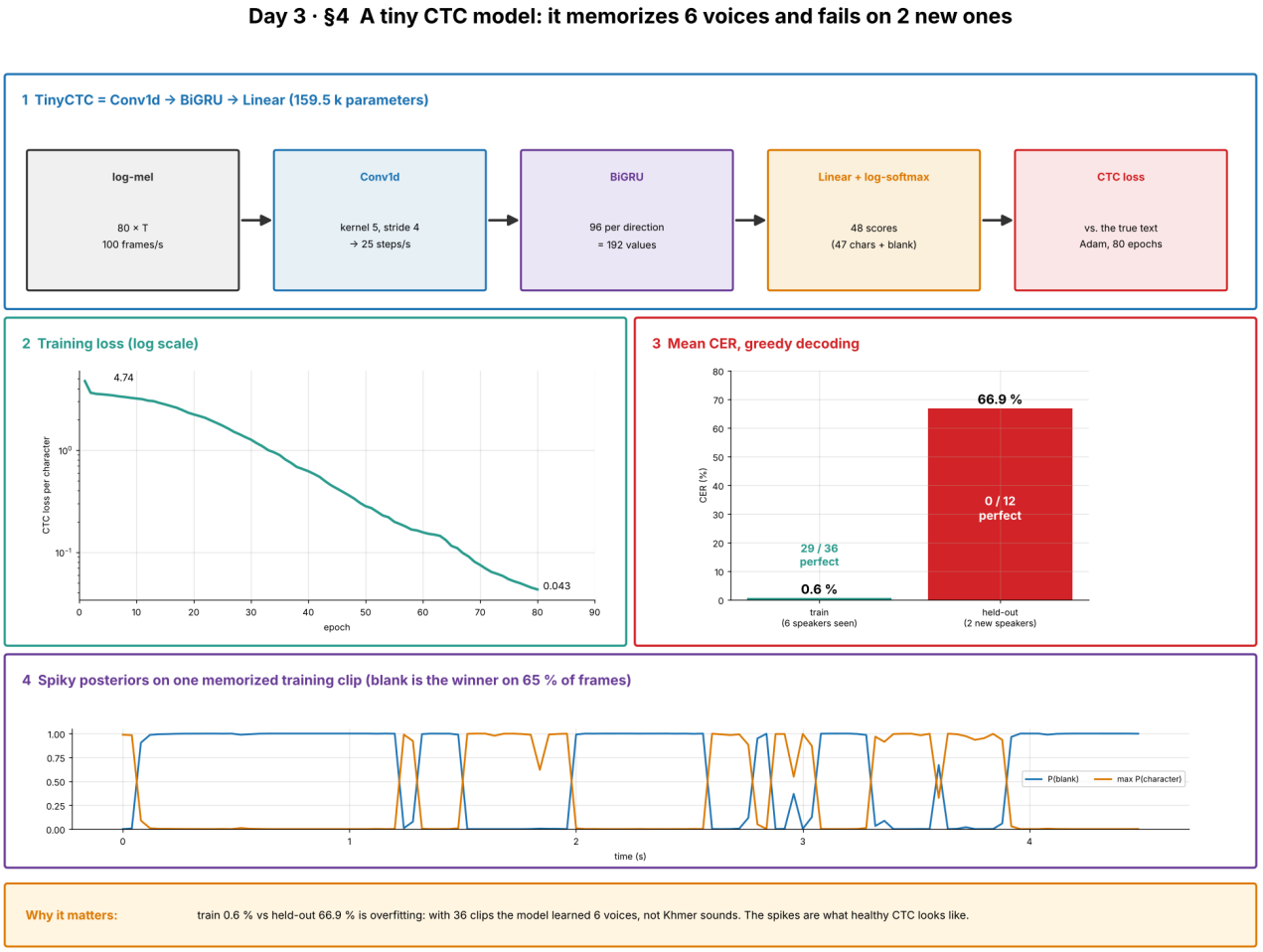
Plan. 80 log-mel bins → 1-D convolution (kernel 5, stride 4: 100 frames/s become 25/s, which is still about 1.5-4.7 frames per character (computed separately, not printed by the cells)) → one bidirectional GRU layer (hidden 96 per direction) → linear layer to `len(vocab)` = characters + blank → log-softmax per frame. Loss = `nn.CTCLoss`, Adam, mini-batches of 4 clips. Training set: 6 speakers × 6 sentences = **36 clips**. Held out: the other **2 speakers saying the same 6 sentences**. (A second GRU layer or stride 2 would work the same way, but Vicheese estimates about 4× the CPU time; not tested here.)

Be honest about what this is. With 6 distinct sentences the model will mostly **memorize** them. It learns "this acoustic shape → this string", not Khmer. That is fine for today. The purpose is to watch CTC do its thing: the loss falling, and the characteristic **spiky** posteriors where the blank dominates and each character fires on one or two frames.

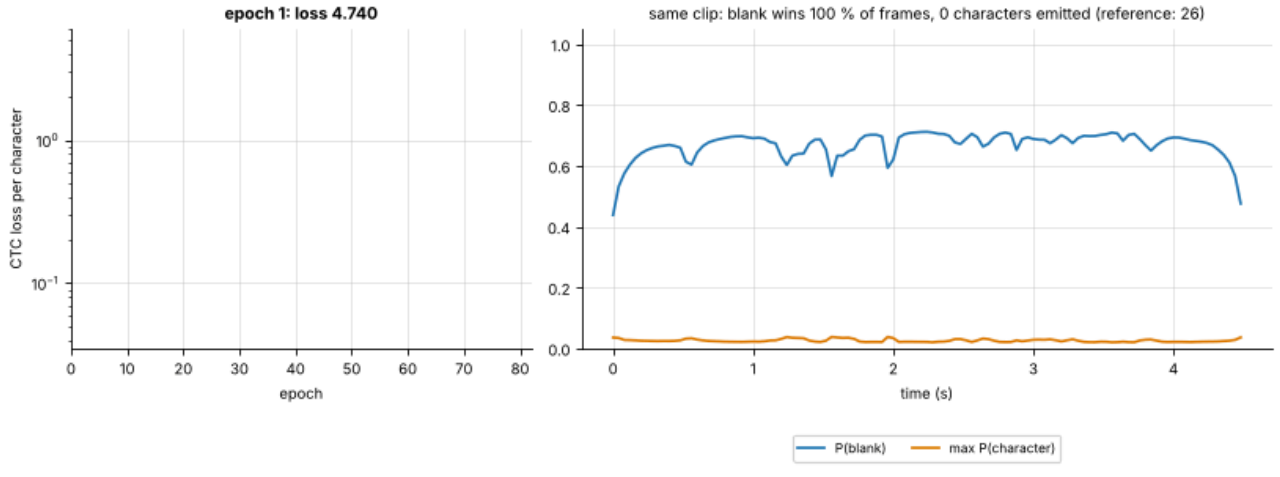
Why it matters. The same loop, with hundreds of hours of audio and a bigger encoder, is a real recognizer. Seeing the spikes once makes every CTC paper readable.

Visualization Version

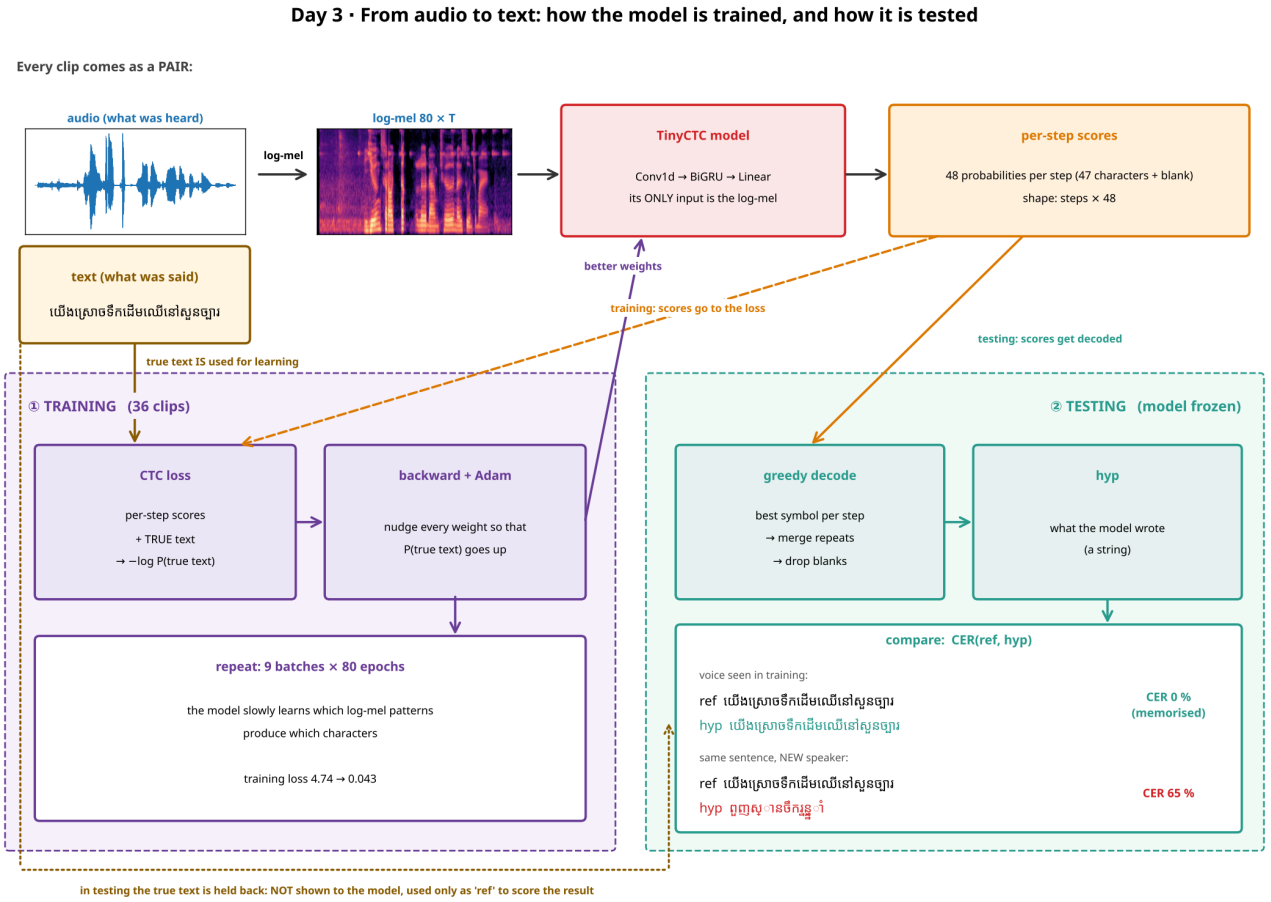
1st image is the summary. Then the training animation and Vicheese's two pictures (train vs test, and the decoding cell).



Animated: the loss curve grows while the posteriors of one clip turn from "all blank" into spikes (real snapshots of this training)



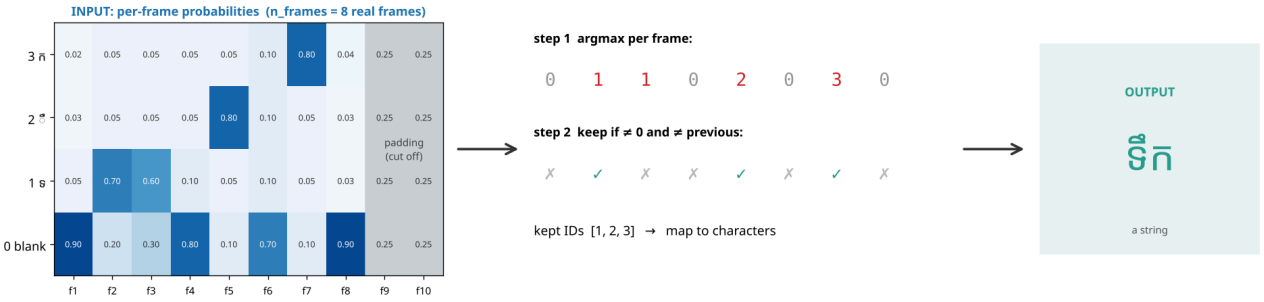
Vicheese's picture: how the model is trained (left) and tested (right)



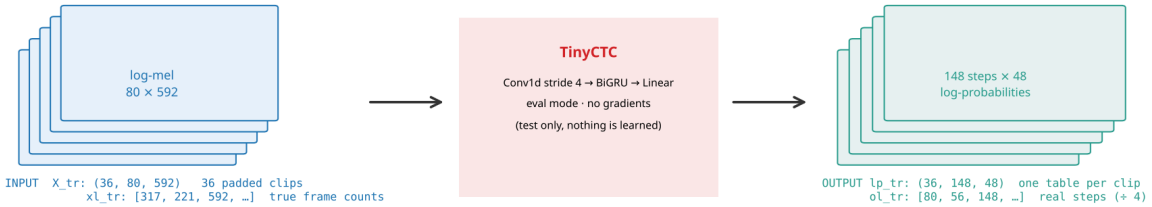
Vicheese's picture: what goes in and what comes out of the decoding cell

Day 3 · Section 4 decoding cell: what goes in, what comes out

1. greedy_decode(logp_row, n_frames) → one clip's scores become text



2. model.eval() + torch.no_grad(): model(X, lengths) → run the trained model on every clip at once




```
INPUT  X_tr: (36, 80, 592)   36 padded clips
       xl_tr: [317, 221, 592, …]  true frame counts
```

```
OUTPUT lp_tr: (36, 148, 48)  one table per clip
       ol_tr: [80, 56, 148, …]  real steps (÷ 4)
```

3. eval_split(df, lp, ol, name) → decode every clip, score with CER, summarise

for every clip: decode, then score

INPUT

```
df   references (ref)
lp   all clips' score tables
ol   real lengths
name  'train' / 'held-out'
```

clip	ref	hyp = greedy_decode(...)	cer_norm(ref, hyp)
1	ธีก	ธีก	0.0 %
2	ธีก	ธีก	33.3 %
3	ธีก	ธก	33.3 %

OUTPUT

prints:
mean CER 22.2 %
perfect clips 1/3

returns:
(hyps, cers)

toy example with 4 symbols (0 = blank, 1 = ธ, 2 = ี, 3 = ก). The real run: 48 symbols, 36 train clips → mean CER 0.6 %, 29/36 perfect; 12 held-out → 66.9 %, 0/12 perfect.

Additional reading: you can read all of it or just read the "In short"

Why a tiny model

In short CTC is the loss, the network is our choice; we use a tiny one so we can see everything.

CTC is not a model, it is a loss. There is no ready-made "CTC model" to plug in. Every CTC recognizer is some network trained with that loss, so you always have to pick a network. We picked a tiny one for three reasons:

- **36 clips is a very small dataset.** A big network would just memorize them. Even the tiny one does: **0.6 % CER on training clips but 66.9 % on new ones.**
- **It runs on your laptop.** 160k parameters train on a CPU in about a minute. A real recognizer has hundreds of millions of parameters and needs GPUs and thousands of hours of speech.
- **The point is to watch CTC learn.** With a small model you can see the loss drop, the spiky outputs and the blank taking over.

A big pretrained CTC model, like a Khmer wav2vec2, would give a much better CER. But that teaches you how to use a model, not how CTC works. It is also the job of the real judge in section 6.

Additional reading: you can read all of it or just read the "In short"

What CTC is for

In short CTC gives one "how wrong are you" number without any frame labels.

CTC is a way to train a speech model when you know **what was said** but not **when each sound happened**.

- **The problem:** a clip has hundreds of frames but only a few dozen characters, and nobody has marked which frame is which character. Normal training needs a correct label for every output, and we do not have that.
- **What CTC does:** the model guesses a symbol (or blank) at every frame. CTC then scores those guesses against the transcript by adding up every frame-by-frame path that collapses to the right text. That is the forward algorithm you just went through. The minus log of that total is the loss.
- **Its job:** give the model one number that says "how far off are you?", without anyone lining up frames and characters by hand. Training pushes that number down, and the model works out the alignment on its own.

So a "CTC model" is shorthand for any network trained with this loss. You read its output by taking the best symbol at each frame and collapsing.

Additional reading: you can read all of it or just read the "In short"

When to use CTC

In short use CTC when the output keeps the order of the input (speech, handwriting); not for translation.

CTC is for one specific situation: the input is a long sequence, the output is a shorter sequence, and you do not know which input part goes with which output part.

- **Speech to text:** frames to characters. This is the classic use.
- **Handwriting recognition:** pen strokes or image columns to letters. CTC is used here too.
- **Image classification**, like your Day 2 speaker ID: one clip, one label, nothing to line up. Plain cross-entropy is enough. No CTC needed.

It also needs the output to come in the **same order** as the input. That is true for speech, but not for translation, where word order changes. That is why translation uses attention decoders instead.

Think of someone saying "cat" slowly: "c c c a a a a t t". The order is already right. CTC's job is to work out where "c" ends and "a" starts, then squash the repeats and blanks into "cat". If the characters were shuffled, like "t a c" when you wanted "cat", CTC could not fix that.

So the short version: CTC is for **in order but stretched**, not for **out of order**.

Additional reading: you can read all of it or just read the "In short"

Our tiny CTC vs the big models

In short our model = Conv1d (shorter) → BiGRU (context) → Linear (48 scores); CTC is only the loss.

The architecture is different too, not just the size. Ours is a small conv layer, then a BiGRU, then a linear layer. That is an older, DeepSpeech-2 style recipe. The big models use a different architecture:

- **wav2vec2** is a CNN plus a Transformer, trained with CTC like ours.
- **Whisper and Qwen3-ASR** have an attention decoder, so they do not use CTC at all.

What we share with wav2vec2 is the overall plan: an encoder makes per-frame scores, CTC trains it, then you collapse. The engine inside the encoder is a different one.

Our three pieces, in order:

- **Conv1d (stride 4):** looks at small windows of the log-mel and squeezes the time axis by 4, so 100 frames a second become 25. It is cheaper, and there are still more steps than characters.
- **BiGRU:** reads those steps in order, carrying a memory along. One GRU runs forwards and one backwards, so every step knows the sound before and after it. That matters because Khmer sounds blend into their neighbours.
- **Linear layer:** at each step, turns the BiGRU's 192 numbers into 48 scores, one for each Khmer character plus blank. A softmax turns those into probabilities.

So the conv shortens the sequence, the BiGRU adds context, and the linear layer picks the symbol. **CTC is not one of the layers; it is the loss that trains all three.**

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short turn every clip into a log-mel "image", turn every transcript into a list of ids, split speakers into train and held-out, and build padded batches.

- **Features:** `feats` makes a log-mel (80 bands × frames) and rescales it to roughly −1…1 so the network trains more smoothly.
- **Labels:** each transcript becomes a list of numbers, one id per Khmer code point.
- **Split:** the last 2 of the 8 speakers are held out → **36 training clips** and **12 test clips** from voices the model never hears.
- **`make_batch`:** clips have different lengths, so shorter ones are padded with "silence" (−1) up to the longest. It also keeps each clip's true length so CTC ignores the padding.

What the output means

In short clips run from **221 to 592 frames** (2.2-5.9 s); the full training batch is **36 clips × 80 bands × 592 frames**; held-out speakers are `iany-s-600bbcad` and `iany-s-707a4c7a`.

- Features take a few seconds on CPU.
- One example clip has shape (80, 317): 3.17 s.

```
# Features: log-mel (80, T) in dB, rescaled to roughly [-1, 1]. Same STFT settings as Day 1.
def feats(path):
    M = logmel(load(path))                      # (80, T), 0 dB = loudest bin, -80 = floor
    return ((M + 40.0) / 40.0).astype(np.float32)

t0 = time.time()
probe["feat"] = [feats(DATA / "audio/probe" / f"{u}.wav") for u in probe.utt_id]
probe["ids"] = [[char2id[c] for c in s] for s in probe.norm]
print("features computed in %.1f s; example shape %s" % (time.time() - t0, probe.feat.iloc[0].shape))

speakers = sorted(probe.speaker.unique())
held_out = speakers[-2:]                        # leave 2 speakers out
train_df = probe[~probe.speaker.isin(held_out)].reset_index(drop=True)
test_df  = probe[probe.speaker.isin(held_out)].reset_index(drop=True)
print("train clips", len(train_df), "| held-out clips", len(test_df), "| held-out speakers", held_out)

def make_batch(df):
    # pad features to the longest clip in the batch; CTC needs the true lengths
    T_max = max(f.shape[1] for f in df.feat)
    X = np.full((len(df), 80, T_max), -1.0, dtype=np.float32)   # pad with "silence" (-1 = floor)
    for i, f in enumerate(df.feat):
```

```
        X[i, :, :f.shape[1]] = f
    x_lens = torch.tensor([f.shape[1] for f in df.feat])
    targets = torch.tensor([c for ids in df.ids for c in ids])    # flat, as CTCLoss wants
    y_lens = torch.tensor([len(ids) for ids in df.ids])
    return torch.tensor(X), x_lens, targets, y_lens

X_tr, xl_tr, y_tr, yl_tr = make_batch(train_df)
X_te, xl_te, y_te, yl_te = make_batch(test_df)
print("train batch", tuple(X_tr.shape), "| frames per clip", xl_tr.min().item(), "..", xl_tr.max().item())
```

```
features computed in 0.9 s; example shape (80, 317)
train clips 36 | held-out clips 12 | held-out speakers ['iany-s-600bbcad', 'iany-s-707a4c7a']
train batch (36, 80, 592) | frames per clip 221 .. 592
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define `TinyCTC`, then train it for 80 epochs with CTC loss, Adam and gradient clipping, and save the weights.

- **`TinyCTC`** is the Conv1d → BiGRU → Linear stack. Two details: it works out each clip's new length after the stride-4 squeeze, and "packing" stops the backward GRU from reading the padding.
- **Training setup:** Adam optimizer, CTC loss with blank as symbol 0. Each of the **80 epochs** is **9 mini-batches of 4 clips**. Gradient clipping stops sudden loss spikes, which RNNs with CTC are prone to.

What the output means

In short **159.5 k parameters**; the loss falls from **4.74 (epoch 1) to 3.20 (epoch 10), 1.26 (30), 0.28 (50), 0.043 (80)**; training takes about a minute on CPU; the weights go to `outputs/day3/tiny_ctc.pt`.

- At the start, the probability given to the right transcript was tiny. At the end it is very high, **but only for the 36 training clips**.
- The real test is the decoding cell below, on the 2 unseen speakers. Expect it to do much worse there, because the model has basically memorized 6 sentences.

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

STRIDE = 4                                          # 100 frames/s -> 25 frames/s

class TinyCTC(nn.Module):
    def __init__(self, n_mels=80, hidden=96, n_out=len(vocab)):
        super().__init__()
        self.conv = nn.Conv1d(n_mels, hidden, kernel_size=5, stride=STRIDE, padding=2)
        self.gru = nn.GRU(hidden, hidden, num_layers=1, bidirectional=True, batch_first=True)
        self.fc = nn.Linear(2 * hidden, n_out)                                  # characters + blank

    def forward(self, x, x_lens):                   # x: (B, 80, T)
        h = torch.relu(self.conv(x)).transpose(1, 2)  # (B, T/4, hidden)
        lens2 = (x_lens + STRIDE - 1) // STRIDE       # output length of the strided conv (ceil)
        packed = pack_padded_sequence(h, lens2.cpu(), batch_first=True, enforce_sorted=False)
        out, _ = self.gru(packed)                   # packing stops the backward GRU from reading padding
        out, _ = pad_packed_sequence(out, batch_first=True)
        return self.fc(out).log_softmax(-1), lens2    # (B, T/4, V) log-probs

model = TinyCTC()
print("parameters: %.1f k" % (sum(p.numel() for p in model.parameters()) / 1e3))
opt = torch.optim.Adam(model.parameters(), lr=5e-3)
ctc_loss = nn.CTCLoss(blank=0, reduction="mean", zero_infinity=True)   # zero_infinity: skip a clip if T < L
```

```
EPOCHS, BS = 80, 4
losses = []
t0 = time.time()
for epoch in range(1, EPOCHS + 1):
    model.train()
    perm = torch.randperm(len(train_df)); ep_loss = []
    for b in range(0, len(train_df), BS):                        # 9 mini-batches of 4 clips per epoch
        Xb, xlb, yb, ylb = make_batch(train_df.iloc[perm[b:b + BS].tolist()].reset_index(drop=True))
        opt.zero_grad()
        logp, out_lens = model(Xb, xlb)
        loss = ctc_loss(logp.transpose(0, 1), yb, out_lens, ylb)   # CTCLoss wants (T, B, V)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 5.0)     # RNN + CTC: clip to avoid loss spikes
        opt.step(); ep_loss.append(loss.item())
    losses.append(float(np.mean(ep_loss)))
    if epoch % 10 == 0 or epoch == 1:
        print(f"epoch {epoch:3d}  ctc loss {losses[-1]:.3f}   ({time.time() - t0:.0f} s)")
torch.save(model.state_dict(), OUT / "tiny_ctc.pt")
print("trained in %.0f s" % (time.time() - t0))
```

```
parameters: 159.5 k
epoch   1  ctc loss 4.740   (1 s)
epoch  10  ctc loss 3.200   (8 s)
epoch  20  ctc loss 2.234   (16 s)
epoch  30  ctc loss 1.265   (25 s)
epoch  40  ctc loss 0.621   (33 s)
epoch  50  ctc loss 0.282   (41 s)
epoch  60  ctc loss 0.157   (49 s)
epoch  70  ctc loss 0.075   (57 s)
epoch  80  ctc loss 0.043   (65 s)
trained in 65 s
```

Additional reading: you can read all of it or just read the "In short"

What CTC loss means

In short CTC loss ≈ −log P(correct text), per character; use it to compare, not as a score.

CTC loss is **minus the log of the probability** the model gives the correct transcript: the same sum over all paths we worked through for `"abb"`. In our code PyTorch also divides by the transcript length, so it is roughly **per character**.

- **0** = the model is completely sure of the right transcript.
- **Higher** = less sure.
- **0.043** per character ≈ **96 % confidence** on each character (e^−0.043 = 0.958).
- **4.74** at the start = the untrained model (a high loss: it is far from sure of the right transcript).

Good or bad? There is **no universal cut-off**: the value depends on transcript length, vocabulary size and how the loss is averaged. Use it **comparatively**:

- it should go **down** during training;
- training loss should stay **close to** test loss.

Our 0.043 looks great, but it is only on clips the model **memorized**. To judge the model, look at **CER on the held-out speakers** in the decoding cell. **Tiny training loss + high held-out CER = overfitting.**

Additional reading: you can read all of it or just

read the "In short"

Adam and gradient clipping

In short Adam = safe default optimizer; clipping caps each update (max norm 5) so one odd batch cannot wreck training.

Adam is the usual default. It is not the most efficient optimizer, but it is the safe go-to. It gives each weight its own step size, so it works well without much tuning. Big Transformers usually use AdamW, a close variant.

Gradient clipping is a speed limit on updates. If the gradient suddenly gets huge, which happens with GRUs and CTC, it is scaled down to a maximum size (here 5) before the weights change.

It matters in our case for two reasons:

- The GRU reuses the same weights at every step, over **56 to 148 steps per clip**, so gradients can multiply and suddenly blow up.
- With only 4 clips per batch, one odd clip can produce a huge gradient.

Without clipping, that one step could throw the weights far off and the loss would spike, sometimes all the way to NaN. With clipping, a huge gradient keeps its direction but gets shrunk, so training stays smooth. Clipping may help keep training smooth; we did not test it (we never trained without it).

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short plot the training loss per epoch on a log axis and print its first and last value.

- `losses` holds the mean loss of each epoch.

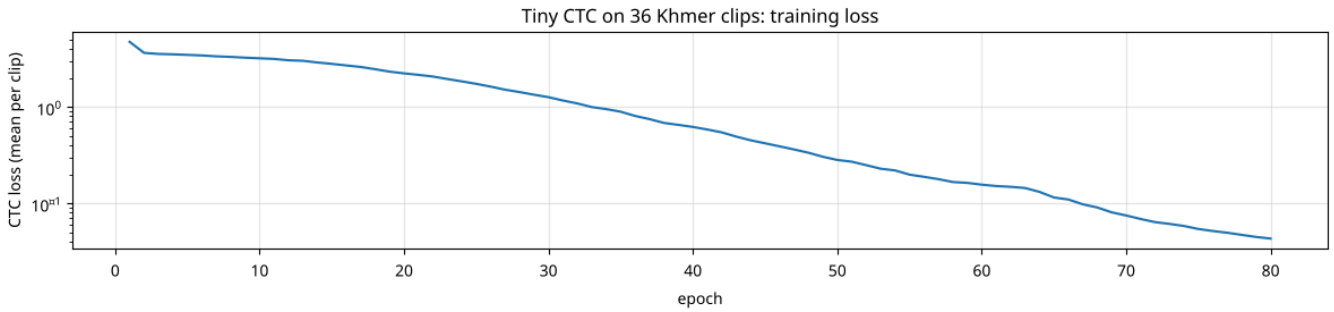
What the output means

In short a smooth, falling curve: **4.74 → 0.043**.

- The y axis is logarithmic, so a straight line means a constant percentage decrease per epoch.
- See the How-to-read notes below.

```
fig, ax = plt.subplots(figsize=(12, 3))
ax.plot(range(1, EPOCHS + 1), losses)
ax.set(title="Tiny CTC on 36 Khmer clips: training loss", xlabel="epoch", ylabel="CTC loss (mean per clip)", yscale="log")
ax.grid(alpha=0.3); plt.tight_layout(); plt.show()
print("loss: first %.2f -> last %.3f" % (losses[0], losses[-1]))
```

```
Font 'default' does not have a glyph for '\u2212' [U+2212], substituting with a dummy symbol.
Font 'default' does not have a glyph for '\u2212' [U+2212], substituting with a dummy symbol.
Font 'default' does not have a glyph for '\u2212' [U+2212], substituting with a dummy symbol.
Font 'default' does not have a glyph for '\u2212' [U+2212], substituting with a dummy symbol.
```




```
loss: first 4.74 -> last 0.043
```

Additional Reading

In short the loss drops fast in epoch 1-2, crawls until about epoch 10, then falls steadily; it never goes up.

How to read this plot. The y axis is logarithmic. The loss drops from 4.74 to 3.64 in the first two epochs and is still 3.20 at epoch 10. Our reading (an interpretation, not something the output shows): in this slow start the model first learns to output *blank almost everywhere*, which is the easiest way to reduce CTC loss. After that the loss falls steadily: 1.26 at epoch 30, 0.28 at epoch 50, 0.043 at epoch 80. This is where characters start to fire at the right places.

A loss well below 0.5 per character on 36 clips of ~35 characters means the training sentences are essentially memorized. In this run the curve has **no bumps**: the epoch mean never went up from one epoch to the next (clipping may help with this; we did not test it).

About the plot cell above: its y-axis label says "per clip", but the loss is **per character** (PyTorch `reduction="mean"` divides by the transcript length). The odd tick symbol on the log axis (10^{-1}) and the four font warnings are a font problem (missing minus glyph), not an error.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define `greedy_decode` (best symbol per frame, then the collapse rule), run the model on all train and held-out clips, and score both sets with CER.

- **`greedy_decode`** cuts the scores to the clip's real length, takes the `argmax` at each frame, keeps a symbol only if it differs from the previous one and is not blank (id 0), and joins the characters.
- **The middle block** is not a function. `model.eval()` switches to test mode and `torch.no_grad()` turns off gradient tracking. It feeds in all train clips and all held-out clips.
- **`eval_split`** runs `greedy_decode` on every clip, scores it with `cer_norm` (the section 2 CER), and prints the average CER and how many clips scored exactly 0.

What the output means

In short **train: mean CER 0.6 %, 29/36 clips perfect. held-out: mean CER 66.9 %, 0/12 perfect.**

- About **2 edits for every 3 reference characters** on the new voices.
- Three held-out examples are printed with their CER: 63.9 %, 65.4 %, 57.9 %. Read them in the notes below.

```
# Greedy decoding = argmax per frame, then the collapse rule from section 3.
def greedy_decode(logp_row, n_frames):
    ids = logp_row[:n_frames].argmax(-1).tolist()
    out, prev = [], None
    for i in ids:
        if i != prev and i != 0:
            out.append(vocab[i])
        prev = i
    return "".join(out)

model.eval()
with torch.no_grad():
    lp_tr, ol_tr = model(X_tr, xl_tr)
    lp_te, ol_te = model(X_te, xl_te)

def eval_split(df, lp, ol, name):
    hyps = [greedy_decode(lp[i], ol[i].item()) for i in range(len(df))]
    cers = [cer_norm(r, h) for r, h in zip(df.norm, hyps)]
    print(f"{name:9s} mean CER {np.mean(cers):5.1f} %   (clips with CER = 0: {sum(c == 0 for c in cers)}/{len(cers)})")
    return hyps, cers

hyp_tr, cer_tr = eval_split(train_df, lp_tr, ol_tr, "train")
hyp_te, cer_te = eval_split(test_df, lp_te, ol_te, "held-out")

print("\nheld-out examples (speaker not seen in training, sentence seen 6 times):")
for i in range(3):
    print(f"  ref: {test_df.norm[i]}\n  hyp: {hyp_te[i]}   CER {cer_te[i]:.1f} %")
```

```
train     mean CER   0.6 %   (clips with CER = 0: 29/36)
held-out  mean CER  66.9 %   (clips with CER = 0: 0/12)

held-out examples (speaker not seen in training, sentence seen 6 times):
  ref: ពួកយើងលង ហាល់ទាត់នៅទីលាន ក្រោយម៉ោងរៀន
  hyp: ពួយើងលាននៅ្រ នាយនារៀ   CER 63.9 %
  ref:យើងស្រមៃទឹកដើមឈើនៅសួនច្បារ
  hyp: ពួញស្ងានចឹកន្ទនទ្ធ   CER 65.4 %
  ref: ចឹកទឹកថ្ងៃបន្ទរលើនៀសវាងការខ្សោតទឹក
  hyp: ចឹកចកមន្ទង វាងគ្រាចខ្សោតិទឹក   CER 57.9 %
```

Additional reading: you can read all of it or just read the "In short"

Reading the decoding result

In short train 0.6 % vs held-out 66.9 % = the model memorized 6 voices; its errors are typos, not invented text.

split	mean CER	perfect clips
train (6 speakers seen)	**0.6 %**	29 / 36
held-out (2 new speakers)	**66.9 %**	0 / 12

- **Train:** almost flawless.
- **Held-out:** about **2 edits for every 3 reference characters**.

That gap is **overfitting**. The held-out sentences are the *same 6* the model trained on, and it still fails, because it learned **those 6 voices**, not Khmer sounds in general. Six speakers are not enough to generalize to a new voice.

Reading the examples

- It gets **some pieces right**: ចំ ការ at the start of the third clip, ភ្ញៀ, តែ, ច.
- **Stray ◌្ (coeng) marks** appear everywhere: the model learned that coeng is common, but not where it belongs.
- Mistakes look like **garbled typos, not invented sentences**. That is typical of CTC: it can drop or swap characters, but it cannot make up fluent text the way an attention decoder can when it hallucinates.
- The held-out hypotheses are too short: on average **70 %** of the reference length (computed separately, not printed by the cells). The model falls back to blank where it cannot decide.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short pick the training clip with the most frames per character, then draw its log-mel, the P(blank) / max P(character) curves, and the per-frame posteriors as a heatmap.

- Cyan `+` marks the frame where greedy decoding emits a new character.
- The last lines print how many characters were emitted and how often blank is the winner, for this clip and on average for the held-out clips.

What the output means

In short this clip (`iany_49b4ab7e_2f5f47ae`) has **4.3 frames per character**; **26 characters on 113 frames; blank wins on 65 %** of frames; on held-out clips blank wins on **64 %** on average.

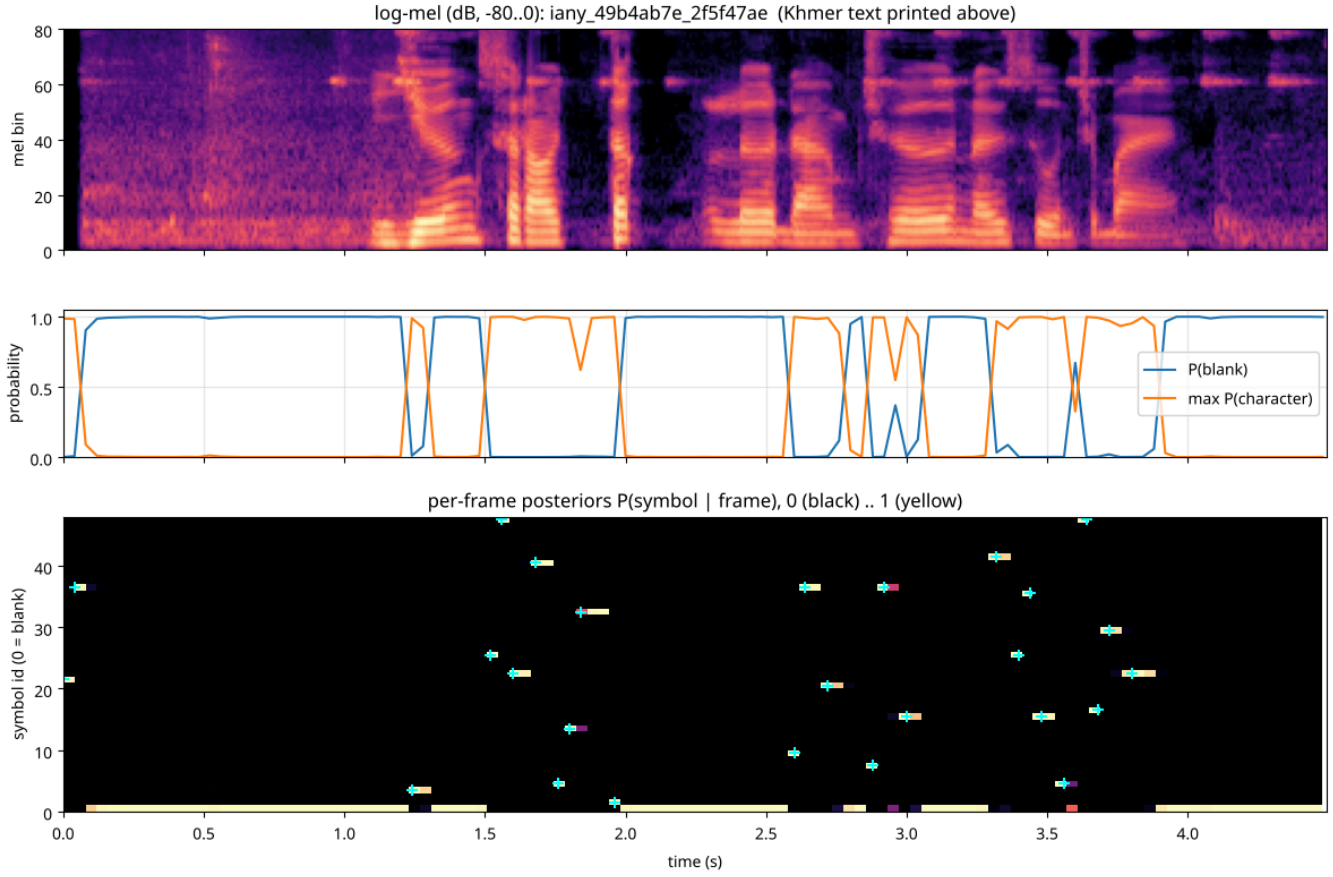
- The greedy text of this training clip equals the reference exactly: it is memorized.
- See the How-to-read notes below.

```
# The classic CTC picture: log-mel on top, blank vs. best-character probability in the middle, posteriors below.
# We plot a TRAINING clip (memorized, so the spikes are clean), the slowest-spoken one (most frames per character).
i = int(np.argmax([ol_tr[j].item() / len(train_df.norm[j]) for j in range(len(train_df))]))
print("frames per character: %.1f (at 25 frames/s)" % (ol_tr[i].item() / len(train_df.norm[i])))
x = load(DATA / "audio/probe" / f"{train_df.utt_id[i]}.wav")
P = lp_tr[i, :ol_tr[i]].exp().numpy()   # (T/4, V) probabilities
print("clip:", train_df.utt_id[i], "| ref:", train_df.norm[i], "| greedy hyp:", hyp_tr[i])
t_axis = np.arange(P.shape[0]) * STRIDE * HOP / SR   # 25 frames/s -> seconds
best = P[:, 1:].max(1)                               # most confident non-blank per frame

fig, axes = plt.subplots(3, 1, figsize=(12, 8), sharex=True, gridspec_kw={"height_ratios": [1.2, 0.8, 1.6]})
M = logmel(x)
axes[0].imshow(M, aspect="auto", origin="lower", cmap="magma", vmin=-80, vmax=0, extent=[0, len(x) / SR, 0, 80])
axes[0].set(ylabel="mel bin", title=f"log-mel (dB, -80..0): {train_df.utt_id[i]}  (Khmer text printed above)")
axes[1].plot(t_axis, P[:, 0], label="P(blank)", lw=1.5)
axes[1].plot(t_axis, best, label="max P(character)", lw=1.5)
axes[1].set(ylabel="probability", ylim=(0, 1.05)); axes[1].legend(loc="center right"); axes[1].grid(alpha=0.3)
im = axes[2].imshow(P.T, aspect="auto", origin="lower", cmap="magma", vmin=0, vmax=1,
                    extent=[t_axis[0], t_axis[-1], 0, len(vocab)])
axes[2].set(xlabel="time (s)", ylabel="symbol id (0 = blank)", title="per-frame posteriors P(symbol | frame), 0 (black) ..
1 (yellow)")
# cyan '+' = the frame where greedy decoding emits a new character
ids = P.argmax(1)
emit = [t_ for t_, k in enumerate(ids) if k != 0 and (t_ == 0 or ids[t_ - 1] != k)]
axes[2].plot(t_axis[emit], ids[emit] + 0.5, "+", color="cyan", ms=7, mew=1.5)
plt.tight_layout(); plt.show()
print("emitted %d characters on %d frames; blank is the argmax on %.0f %% of frames" % (len(emit), len(ids), 100 * (ids ==
0).mean()))
# the same statistic on the held-out clips, where the model is unsure
```

```
blank_te = [(lp_te[j, :ol_te[j]].argmax(-1) == 0).float().mean().item() for j in range(len(test_df))]
print("held-out clips: blank is the argmax on %.0f %% of frames on average" % (100 * np.mean(blank_te)))
```

```
frames per character: 4.3 (at 25 frames/s)
clip: iany_49b4ab7e_2f5f47ae | ref:យើងរស់ចឹកដើមល្វីនៅស្នូបរ | greedy hyp:យើងរស់ចឹកដើមល្វីនៅស្នូបរ
```




```
emitted 26 characters on 113 frames; blank is the argmax on 65 % of frames
held-out clips: blank is the argmax on 64 % of frames on average
```

Additional Reading

In short blank is high almost everywhere; each character is one narrow spike; two characters fire in the opening silence = memorization.

How to read this plot. Top: the log-mel you know from Day 1 (bright horizontal stripes low down = voiced vowels, vertical smears to the top = fricatives, dark = pauses). **Middle:** the blue blank probability sits near 1 for most of the clip and drops to zero in short dips; the orange "best character" curve mirrors it as narrow **spikes**. Each spike is one character emitted in one or two frames (40-80 ms at 25 frames/s), much shorter than the sound it represents. **Bottom:** the same thing as a heatmap, one row per symbol. The bottom row (id 0, blank) is bright nearly everywhere. The characters appear as isolated bright dots scattered across the vocabulary rows, each marked with a cyan `+` where greedy decoding reads it.

Nothing in the loss asked for spikes. They emerge because sharing probability mass among neighbouring frames is a worse way to maximize the path sum than committing to one frame.

Frames per character. This is the slowest training clip, with 4.3 frames per character. Over all training clips the range is **1.5 to 4.3 (mean 2.7)**; for the held-out clips it is **2.0 to 4.7 (mean 3.2)** (bothcomputed separately, not printed by the cells). Blank wins on about two thirds of the frames

here (65 %). That is lower than the ~90 % a 100 frames/s model shows. The mechanism is the same; there is just less room between characters.

The giveaway. Look at the first 1.2 s: the clip is silence, yet two characters fire at t ≈ 0-0.05 s. That is memorization, not recognition. The model learned that this sentence starts with those two symbols and emits them as soon as the clip begins, before any evidence. A model trained on real data would not do this.

Two cautions. The spikes are *not* a segmentation: they tend to sit near the start of a sound, not spread over it. And on a held-out speaker (re-run the cell with `test_df`, `lp_te`, `ol_te`, `hyp_te`) the blue curve dips in many places where no clean spike follows. The printed blank share on held-out clips is about the same (64 %). An unsure model falls back to blank where it cannot commit to a character, which is also why the held-out hypotheses in the previous cell are too short.

Honest interpretation. *(The facts in this paragraph are background facts from Vicheese, not checked here.)* Six sentences is not a language model and not a recognizer. What makes a real CTC system work is scale: wav2vec 2.0 fine-tunes on 10-1000 h; commercial systems use tens of thousands. For Khmer, Vicheese lists these public options (not re-checked here): OpenSLR 42 (Google, ~2.9 k studio sentences, a few hours), FLEURS (Google, ~10 h of read Wikipedia sentences per language, includes `km_kh`), and the small iAny phone corpus used here. Mozilla Common Voice has had little or no released Khmer audio (check the current release before relying on it). Multilingual models (Whisper, MMS, Qwen3-ASR) are trained on web-scale data with modest Khmer shares, which is why Qwen3-ASR is the project's judge and Whisper is not.

Additional reading: you can read all of it or just read the "In short"

The plot, in Vicheese's words

In short blank 65 % (this clip) and 64 % (held-out); one spike per character.

- **Top:** the log-mel, the sound itself.
- **Middle (the key one):** the **blue line is P(blank)**, meaning "nothing new here", and it sits near 1 most of the time. Each time it briefly drops and the **orange line** shoots up, the model writes **one character**. These quick dips are called **spikes**.
- **Bottom:** the same thing in more detail. Each **row is one symbol** (row 0 = blank), and each short dash shows **which character fired at that moment**. The cyan **+** marks where greedy decoding reads it.

clip(s)	blank wins on
this training clip	**65 %** of steps
held-out speakers (average)	**64 %** of steps

- This clip came out **perfect**, but that is **memorization**, not recognition.
- **The giveaway:** two characters fire in the **silence at the very start**, before there is any sound.

Big lesson: CTC does not spread a character across its whole sound. It goes *blank, blank, blank, h, blank*: each character fires in **one quick spike**.

Additional reading: you can read all of it or just read the "In short"

Training vs testing pipeline

In short training uses the true text inside the loss; testing uses it only afterwards to score.

Shared top row (both cases): audio → log-mel → TinyCTC → per-step scores. The pipeline then splits into two lanes.

Lane	What happens	Is the true text used?
① **Training**	per-step scores + **true text** → CTC loss → backward + Adam → better weights → repeat (loss **4.74 → 0.043**)	**Yes**: it goes into the loss
② **Testing**	the model is **frozen** → greedy decode → hypothesis (`hyp`) → compare with the true text using CER	**Only afterwards**, as `ref` to score the result

- **Held-back text:** the dotted line in the picture shows the true text skipping the model and going straight to the comparison box. In testing the model never sees it.
- The picture's two example numbers are for **one sentence** (the clip in the posterior plot): CER 0 % when the voice was seen in training, **65 %** (65.4 %) for the same sentence from a new speaker. The averages over all clips are **0.6 %** and **66.9 %**.

Overall idea: each clip comes as a pair: the audio (what was heard) and the text (what was said). We already have the text. We convert the audio into a log-mel, and the model turns that log-mel back into text. Then we compare the model's text with the real sentence using CER.

- **Training:** the real text is used to teach the model (it goes into the CTC loss).
- **Testing:** the real text is hidden from the model and used only afterwards, to score its output.

5. Beam search vs greedy

Greedy decoding picks the argmax symbol at each frame and collapses. It ignores that *several paths* share one text. Example: if `a` has probability 0.4 at two frames and blank has 0.6 at both, greedy reads blank-blank (no `a`). But the text `a` has total probability 0.4·0.6 + 0.6·0.4 + 0.4·0.4 = **0.64**, and blank-blank only 0.36. So `a` should win.

Prefix beam search keeps the B best *texts* (prefixes), each with two numbers: probability of ending in blank (p_b) and in a non-blank (p_{nb}). At every frame it extends each prefix by blank, by repeating its last character, or by a new character, merging paths that produce the same text.

$$\text{score}(\text{prefix}) = p_b + p_{nb}, \qquad \text{extend by } c: \; p'_{nb} \mathrel{+}= (p_b + p_{nb})\, p_t(c) \;\; (\text{or } p_b\, p_t(c) \text{ if } c = \text{last cl}$$

The place where a **language model** plugs in is the extension step: multiply by $P_{LM}(c \mid \text{prefix})^{\alpha}$ (plus a length bonus). That is how CTC systems get fluent output without a decoder.

Why it matters. For a well-trained model, greedy and beam usually agree within a few tenths of a CER point; the beam matters when you add an LM (background fact from Vicheese, not tested here). For our memorizing model the difference is small and noisy, which is itself a useful lesson.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short implement `prefix_beam_search` (beam 8), decode the 12 held-out clips with it, and compare its CER with greedy clip by clip.

- `beams` maps a prefix to `(p_blank, p_nonblank)`. For each frame every prefix is extended by blank, by a repeat of its last character, or by each character that has some probability (`prune = 1e-3`).
- After each frame only the 8 best prefixes survive. The comment in the code marks where an LM factor would go.

- The result table has the greedy CER, the beam CER, whether the two texts are identical, and `p_beam` : the total probability of the best text.

What the output means

In short mean CER held-out: greedy 66.9 %, beam 65.4 %; identical outputs on 4 of 12 clips.

- Beam is better on 4 clips (e.g. 68.00 → 66.00, 69.23 → 61.54), the same on 7, and **worse on 1** (55.81 → 58.14).
- `p_beam` goes from 4e-14 to 5e-6 across clips: even the best text is a very unlikely sentence for the model.

```
def prefix_beam_search(P, beam=8, blank=0, prune=1e-3):
    # P: (T, V) probabilities (not logs). Returns the best text as a tuple of symbol ids.
    beams = {(): (1.0, 0.0)}                                  # prefix -> (p_blank, p_nonblank)
    for t in range(P.shape[0]):
        nxt = defaultdict(lambda: [0.0, 0.0])
        cands = np.nonzero(P[t] > prune)[0]                   # only symbols with some mass at this frame
        for prefix, (pb, pnb) in beams.items():
            nxt[prefix][0] += (pb + pnb) * P[t, blank]        # extend by blank: text unchanged
            if prefix:
                nxt[prefix][1] += pnb * P[t, prefix[-1]]      # repeat last char without blank: text unchanged
            for c in cands:
                if c == blank:
                    continue
                new = prefix + (int(c),)
                if prefix and c == prefix[-1]:
                    nxt[new][1] += pb * P[t, c]               # 'a' after 'a' needs a blank in between
                else:
                    nxt[new][1] += (pb + pnb) * P[t, c]       # <- an LM factor P_LM(c | prefix)^alpha goes here
        beams = dict(sorted(nxt.items(), key=lambda kv: -(kv[1][0] + kv[1][1]))[:beam])
    best = max(beams.items(), key=lambda kv: kv[1][0] + kv[1][1])
    return best[0], best[1][0] + best[1][1]

def ids_to_text(ids):
    return "".join(vocab[i] for i in ids)

rows = []
for i in range(len(test_df)):
    P = lp_te[i, :ol_te[i]].exp().numpy()
    b_ids, b_p = prefix_beam_search(P, beam=8)
    g = hyp_te[i]; b = ids_to_text(b_ids)
    rows.append(dict(utt=test_df.utt_id[i], cer_greedy=cer_norm(test_df.norm[i], g),
                     cer_beam=cer_norm(test_df.norm[i], b), same=(g == b), p_beam=b_p))
res = pd.DataFrame(rows)
print(res.round(2).assign(p_beam=res.p_beam.map("{:.1e}".format)).to_string(index=False))
print("\nmean CER held-out: greedy %.1f %%  beam %.1f %%  | identical outputs: %d/%d"
      % (res.cer_greedy.mean(), res.cer_beam.mean(), res.same.sum(), len(res)))
```

```
                 utt  cer_greedy  cer_beam  same  p_beam
iany_600bbcad_10f7caf3       63.89     63.89 False 1.6e-09
iany_600bbcad_4aa14a48       65.38     65.38  True 1.7e-07
iany_600bbcad_7b05a074       57.89     57.89 False 2.0e-08
iany_600bbcad_9c1b9049       68.00     66.00 False 4.8e-12
iany_600bbcad_2f2a63e7       62.79     62.79  True 3.5e-09
iany_600bbcad_325d924e       61.11     61.11  True 4.7e-06
iany_707a4c7a_ec6b242a       80.56     77.78 False 1.1e-08
iany_707a4c7a_d81b4554       69.23     61.54 False 2.8e-08
iany_707a4c7a_a39c9c3d       71.05     71.05  True 2.0e-07
iany_707a4c7a_1d8b082e       78.00     78.00 False 2.9e-08
iany_707a4c7a_c2e806bf       55.81     58.14 False 4.3e-14
iany_707a4c7a_f373befd       69.44     61.11 False 2.6e-07

mean CER held-out: greedy 66.9 %  beam 65.4 %  | identical outputs: 4/12
```

Additional reading: you can read all of it or just read the "In short"

Reading the table

In short beam beats greedy by 1.5 points on average (4 clips better, 1 worse, 7 equal); it maximizes P(text), not CER.

Reading the table. On the 36 training clips the two decoders agree almost everywhere (**30 of 36** identical (computed separately, not printed by the cells)). Sharp spikes leave every path but one with negligible mass, so the beam collapses to greedy. On the held-out speakers only **4 of 12** outputs are identical, because the posteriors are soft and ambiguous (the new voice does not fit a memorized shape), so summing over paths changes the winner.

The beam is better on average by about **1.5 CER points** (66.9 → 65.4), but not on every row. The beam maximizes $P(\text{text})$, not CER, and a more probable wrong text can be further from the reference.

The `p_beam` column is the probability of the best text, summed over paths. Do **not** use it as a confidence score here: the clip with the smallest `p_beam` (4e-14) has one of the *lowest* CERs (58 % beam), and the correlation between `log p_beam` and CER is only 0.19 (computed separately, not printed by the cells).

Additional reading: you can read all of it or just read the "In short"

Beam vs greedy, one paragraph

In short same job, but beam sums over paths; usually a bit better, not guaranteed.

They do the same job: turn the model's per-frame scores into text. Beam does it better by adding up every path that spells the same text. But it is not guaranteed to give a lower CER. It picks the most likely text, which usually means a lower CER, but on one clip here it actually got worse.

6. Pretrained models: Whisper and the project's Khmer judge

Whisper (OpenAI) is an attention encoder-decoder trained on 680 k h of web audio, 99 languages, with Khmer as a small slice (background facts from Vicheese, not checked here). Its decoder is a strong language model. That gives it a failure mode CTC does not have: on noisy or unfamiliar audio it can emit **fluent, grammatical, unrelated** text, or repeat a phrase many times. This is called **hallucination**. It shows up as CER > 100 % (hypothesis longer than reference). CTC cannot do this: with no decoder state, it emits a symbol only when a frame supports it.

Qwen3-ASR (Alibaba) is a newer multilingual model with much better Khmer. It is the judge that produced every `hyp_*` column in `transcripts.csv`. The leaderboard's original-speech CER of **7.6 %** is its floor on this data.

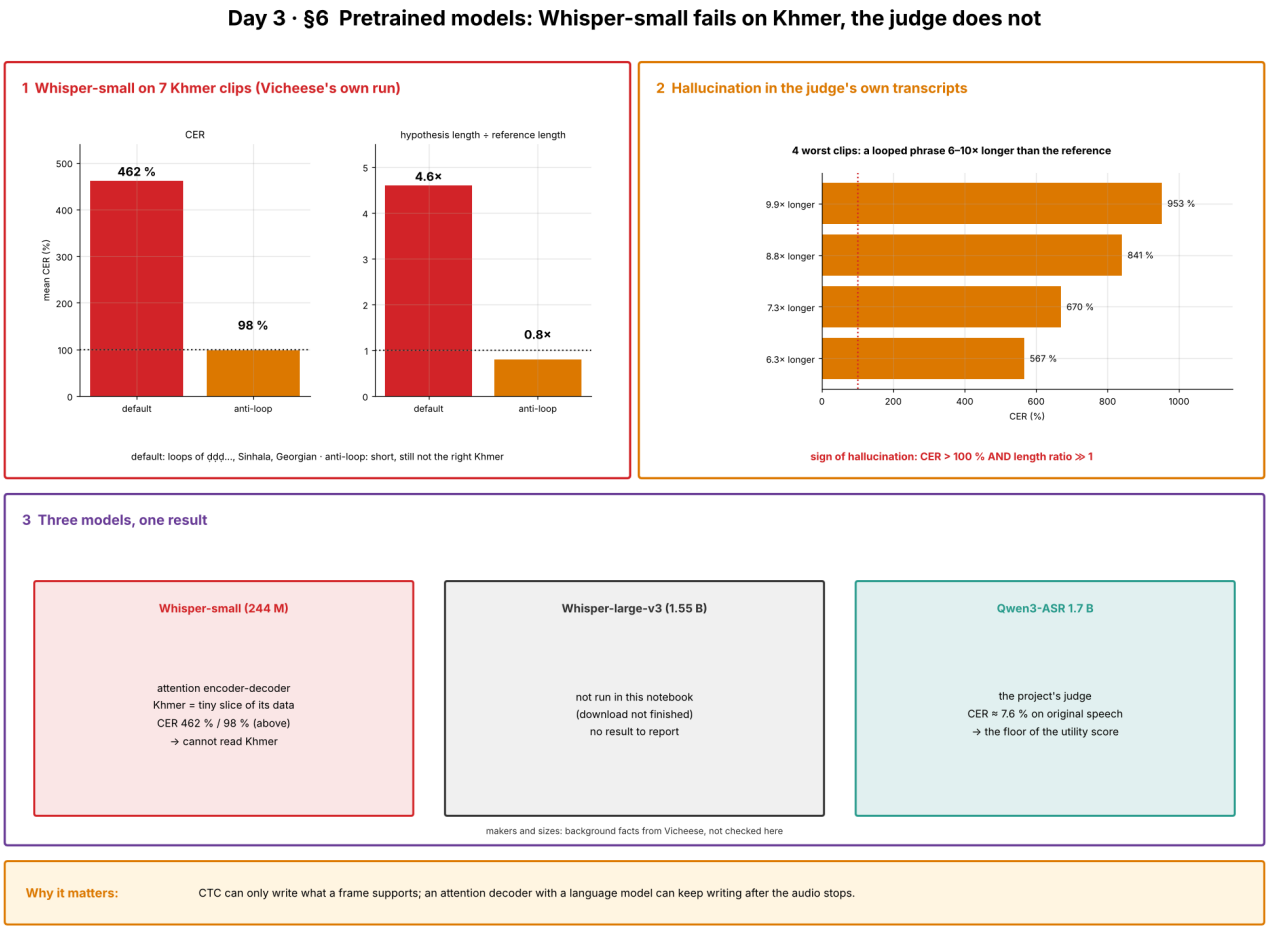
Before scoring any model output, apply the same `normalize()` as the reference. Whisper adds spaces and sometimes punctuation that the references do not have. Without normalization you would pay insertions for every space.

Why it matters. The judge's own error rate is the noise floor of the utility metric. If the judge scores 7.6 % on clean originals, a system at 9 % is "as good as clean", and the interesting range is above that.

> **The first three code cells below show the output from Vicheese's own run.**
> Whisper and Qwen3-ASR cannot be downloaded on the machine that built this file, so these three cells were not re-run. The cells after them were run here.

Visualization Version

The Whisper numbers in the picture are from Vicheese's own run; the hallucination bars are from the judge transcripts in this repo.



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short load `openai/whisper-small`, transcribe 8 Khmer clips and print the CER and the transcripts. **Output from Vicheese's own run:** this cell was not re-run here (the model cannot be downloaded on this machine). The output below is his saved result.

- `eval_items` : the 3 main clips (studio, phone female, phone male) + 5 phone demo clips. One clip is in the list twice (see the next cell).
- `pipeline(...)` runs Whisper with `language="khmer"`. If the download fails, the `except` branch prints a skip message.

What the output means

In short **Output from Vicheese's own run:** mean CER **460.3 %** on 8 clips; every transcript is a long **loop** (`ḍḍḍḍ…` , Sinhala `අපි…` , Georgian `ღღღ…`), about 25 s per clip.

- CER between 400 % and 681 %: the hypothesis is 4-7× longer than the reference.
- The same clip appears twice (`iany_ab586c1d_52008df7`); the next cell removes the duplicate.

```
# (a) Whisper-small via transformers. openai/whisper-small, ~970 MB download, cached in ~/.cache/huggingface
# needs internet (first run only)
clips = [("khmer_studio.wav", "slr42_3154_4104924772"), ("khmer_phone_female.wav", "iany_ab586c1d_52008df7"),
         ("khmer_phone_male.wav", "iany_ad89c739_3dee3a99")]
demo_ref = demo.set_index("utt_id").text
eval_items = [(DATA / "audio" / f, demo_ref[u], u) for f, u in clips]          # the 3 main clips
eval_items += [(DATA / "audio/demo" / f"{u}.wav", demo_ref[u], u) for u in demo.utt_id.iloc[6:11]]  # 5 phone demo clips

whisper_rows = []
try:
    from transformers import pipeline
    t0 = time.time()
    asr = pipeline("automatic-speech-recognition", model="openai/whisper-small", device="cpu")
    print("whisper-small loaded in %.0f s" % (time.time() - t0))
    for path, ref, u in eval_items:
        t0 = time.time()
        hyp = asr(load(path), generate_kwargs={"language": "khmer", "task": "transcribe"})["text"]
        whisper_rows.append(dict(utt_id=u, ref=ref, hyp=hyp, cer=cer_norm(ref, hyp), sec=time.time() - t0))
    wdf = pd.DataFrame(whisper_rows)
    print(wdf[["utt_id", "cer", "sec"]].round(1).to_string(index=False))
    print("mean CER (Whisper-small, 8 clips): %.1f %%" % wdf.cer.mean())
    for _, r in wdf.iterrows():
        print(f"\n{r.utt_id}\n  ref: {normalize(r.ref)}\n  hyp: {normalize(r.hyp)}")
except Exception as e:
    print("skipped: Whisper could not be loaded (no internet or transformers missing):", type(e).__name__, str(e)[:120])
    print("Expected result on these clips: Whisper-small CER is typically 60-100 % on Khmer phone audio and shows "
          "hallucination; see the shipped Qwen3-ASR transcripts below for what a good Khmer judge produces.")
```

```
               utt_id   cer  sec
 slr42_3154_4104924772 504.5 25.3
iany_ab586c1d_52008df7 448.5 26.5
iany_ad89c739_3dee3a99 400.0 25.6
iany_ab586c1d_52008df7 448.5 23.2
iany_ab586c1d_c1396ba8 400.0 23.0
iany_b643351e_2d6423e4 681.1 23.3
iany_b643351e_3f003edd 400.0 23.1
iany_32bb69d3_2c02ab33 400.0 25.0
mean CER (Whisper-small, 8 clips): 460.3 %

slr42_3154_4104924772
  ref: ខ្ញុំមានចិត្តសប្បាយណាស់ដែលបានជួបលោកវ្របាដី
  hyp: අ
පිත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්
ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත්ත

iany_ab586c1d_52008df7
  ref:. ព្រះអាទិត្យរះនៅទិសខាងកើតរាល់ព្រឹក
  hyp: ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ
ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ

iany_ad89c739_3dee3a99
  ref: ព័ត៌មានពីស្រុកខ្មែរថ្ងៃនេះមឃុស្រឡូលអ
  hyp: ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ
ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ

iany_ab586c1d_52008df7
  ref:. ព្រះអាទិត្យរះនៅទិសខាងកើតរាល់ព្រឹក
  hyp: ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ
ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ

iany_ab586c1d_c1396ba8
  ref: ព័ត៌មានពីស្រុកខ្មែរថ្ងៃនេះមឃុស្រឡូលអ
```

```
  hyp: ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ
ḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍḍ

iany_b643351e_2d6423e4
  ref: ព័ត៌មានពីស្រុកខ្មែរថ្ងៃនេះមេឃស្រឡះល្អ
  hyp: අපිඅපිඅපි,අපිඅපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,
අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි,අපි

iany_b643351e_3f003edd
  ref: ទីផ្សារនាំចេញផលិតផលកសិកម្មកំពុងកើនឡើង
  hyp: දදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදද
දදදදදදදදදදදදදදදදදදදදදදදදදදදදදද

iany_32bb69d3_2c02ab33
  ref: ព័ត៌មានពីស្រុកខ្មែរថ្ងៃនេះមេឃស្រឡះល្អ
  hyp: දදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදදද
දදදදදදදදදදදදදදදදදදදදදදදදදදදදදද
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short run Whisper-small again with fixes: unique clips, explicit 16 kHz audio, a length ratio, and an **anti-loop** setting (`no_repeat_ngram_size=3` , `max_new_tokens=100`). **Output from Vicheese's own run:** this cell was not re-run here (the model cannot be downloaded on this machine). The output below is his saved result.

- The length ratio `len(hyp) ÷ len(ref)` is the hallucination detector.
- `MODELS` has only `whisper-small` ; `whisper-large-v3` is listed as a comment and was **not run**.

What the output means

In short **Output from Vicheese's own run** (7 unique clips): default settings **CER 462 %, length ratio 4.6×**; anti-loop settings **CER 98 %, length ratio 0.8×**.

- Anti-loop kills the repetition, but the text is still **not the right Khmer**: five of the seven outputs are in other scripts (Sinhala, Devanagari, Latin); two contain Khmer letters, but they are garbage (CER 91.9 %).
- A length ratio far above 1 with CER ≫ 100 % is the signature of looping hallucination.

```
# (a2) Whisper, fixed: unique clips, explicit 16 kHz, length ratio, anti-loop settings
import librosa
from transformers import pipeline

# 1) remove the duplicate clip
seen, items = set(), []
for path, ref, u in eval_items:
    if u not in seen:
        seen.add(u); items.append((path, ref, u))
print("unique clips:", len(items))

MODELS = ["openai/whisper-small"]          # add "openai/whisper-large-v3" when it has downloaded
SETTINGS = {
    "default":   {"language": "khmer", "task": "transcribe"},
    "anti-loop": {"language": "khmer", "task": "transcribe",
                  "no_repeat_ngram_size": 3, "max_new_tokens": 100},
}

rows = []
for m in MODELS:
    asr = pipeline("automatic-speech-recognition", model=m, device="cpu")
    for sname, kw in SETTINGS.items():
        for path, ref, u in items:
```

```
            y, sr = librosa.load(str(path), sr=16000)          # 2) Whisper needs 16 kHz
            t0 = time.time()
            hyp = asr({"raw": y, "sampling_rate": 16000}, generate_kwargs=kw)["text"]
            rows.append(dict(model=m.split("/")[-1], settings=sname, utt_id=u,
                             cer=cer_norm(ref, hyp),
                             len_ratio=len(normalize(hyp)) / len(normalize(ref)),   # 3)
                             sec=time.time() - t0, ref=ref, hyp=hyp))

res = pd.DataFrame(rows)
print(res[["model", "settings", "utt_id", "cer", "len_ratio", "sec"]].round(2).to_string(index=False))
print("\n", res.groupby(["model", "settings"])[["cer", "len_ratio"]].mean().round(1))
for _, r in res[res.settings == "anti-loop"].iterrows():
    print(f"\n{r.model} | {r.utt_id}\n  ref: {normalize(r.ref)}\n  hyp: {normalize(r.hyp)[:120]}")
```

```
        model  settings                utt_id    cer  len_ratio   sec
whisper-small   default  slr42_3154_4104924772 504.55       5.05 27.35
whisper-small   default iany_ab586c1d_52008df7 448.48       4.48 24.00
whisper-small   default iany_ad89c739_3dee3a99 400.00       4.00 23.34
whisper-small   default iany_ab586c1d_c1396ba8 400.00       4.00 26.42
whisper-small   default iany_b643351e_2d6423e4 681.08       6.81 26.23
whisper-small   default iany_b643351e_3f003edd 400.00       4.00 23.95
whisper-small   default iany_32bb69d3_2c02ab33 400.00       4.00 23.50
whisper-small anti-loop  slr42_3154_4104924772 100.00       0.82  5.00
whisper-small anti-loop iany_ab586c1d_52008df7 100.00       0.76  2.34
whisper-small anti-loop iany_ad89c739_3dee3a99 100.00       1.00  3.08
whisper-small anti-loop iany_ab586c1d_c1396ba8 100.00       0.92  5.20
whisper-small anti-loop iany_b643351e_2d6423e4 100.00       0.49  3.59
whisper-small anti-loop iany_b643351e_3f003edd  91.89       0.92  5.33
whisper-small anti-loop iany_32bb69d3_2c02ab33  91.89       0.59  4.42

                           cer  len_ratio
model         settings
whisper-small anti-loop   97.7        0.8
              default    462.0        4.6

whisper-small | slr42_3154_4104924772
  ref: ខ្ញុំមានចិត្តសប្បាយណាស់ដែលបានជួយលោកវ៉ែរ ហាដឺ
  hyp: අපිසින්තුප්කාපාරුවිමිනැල්හාවසිවකරයකය.

whisper-small | iany_ab586c1d_52008df7
  ref: ព្រះអាទិត្យរះនៅទិសខាងកើតរាល់ព្រឹក
  hyp: प्रातद्रेलिएतुखंकाडोपरक

whisper-small | iany_ad89c739_3dee3a99
  ref: ពាក្យមានពីរស្រៈនិងមួយព្យញ្ជនៈមិនរាប់បញ្ចូលអ
  hyp: बारमेंभीस्त्रोगकमाइतनहींनिमैंचूलगलाओ.

whisper-small | iany_ab586c1d_c1396ba8
  ref: ពាក្យមានពីរស្រៈនិងមួយព្យញ្ជនៈមិនរាប់បញ្ចូលអ
  hyp: d̪ᵗᵃᵃᵉʳᵕᵒᵇᶜᵏɾᵍʃˆd̪ᵈᵉʒᵉeɳˈəkˢᵐɭbɛgʃɲaɸ

whisper-small | iany_b643351e_2d6423e4
  ref: ពាក្យមានពីរស្រៈនិងមួយព្យញ្ជនៈមិនរាប់បញ្ចូលអ
  hyp: අපිස්නින්වැතමරිග්

whisper-small | iany_b643351e_3f003edd
  ref: ទិដ្ឋភាពនៃចញផលិតផលកសិកម្មមកពិព្រងើនឡើង
  hyp: සඳහාඅවාහලපතිනරණතඵලණඥඩිඝපූඛචඨණඵකඛඅාඡඅඝ

whisper-small | iany_32bb69d3_2c02ab33
  ref: ពាក្យមានពីរស្រៈនិងមួយព្យញ្ជនៈមិនរាប់បញ្ចូលអ
  hyp: សំឡេងរាល្បឿនផ្ទាំងមើរ នោ។
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short try the project's judge, Qwen3-ASR, on the same clips; if it is not installed, print the judge's saved transcripts of 5 original clips. **Output from Vicheese's own run:** this cell was not re-run here (the model cannot be downloaded on this machine). The output below is his saved result.

- `qwen_asr` is not installed, so the `except` branch ran: it samples 5 rows of `transcripts.csv` (the judge's transcript of the **original** audio) and prints CER, reference

and hypothesis.

What the output means

In short **Output from Vicheese's own run:** the judge reads original Khmer well: CER **5.0 %, 11.9 %, 25.6 %, 0.0 %, 17.6 %** on the 5 clips.

- Most errors are small (a vowel sign, a swapped word); two clips have 18-26 % CER (17.6 % and 25.6 %: a changed word, inserted words). This is the judge's "noise floor" (7.6 % on the whole set).
- Qwen3-ASR itself was not run by anyone in this notebook.

```
# (b) Qwen3-ASR, the project's judge (pip install qwen-asr; ~ several GB of weights). Optional.
# needs internet (first run only)
qwen_rows = []
try:
    import importlib
    qa = importlib.import_module("qwen_asr")                    # raises ImportError if not installed
    QwenModel = getattr(qa, "Qwen3ASRModel")
    qmodel = QwenModel.from_pretrained("Qwen/Qwen3-ASR-1.7B", device_map="cpu")
    for path, ref, u in eval_items:
        out = qmodel.transcribe(audio=str(path), language="Khmer")
        hyp = out[0].text if isinstance(out, list) else str(out)
        qwen_rows.append(dict(utt_id=u, ref=ref, hyp=hyp, cer=cer_norm(ref, hyp)))
    print(pd.DataFrame(qwen_rows)[["utt_id", "cer"]].round(1).to_string(index=False))
except Exception as e:
    print("skipped: Qwen3-ASR not available here (%s). Falling back to the shipped judge transcripts." % type(e).__name__)
    # Fallback: the judge's transcripts of ORIGINAL (un-anonymized) audio for 5 clips of transcripts.csv
    sub = trans.sample(5, random_state=3)
    for _, r in sub.iterrows():
        print(f"\n{r.utt_id}  [{r.corpus}]  CER {cer_norm(r.reference, r.hyp_original):.1f} %"
              f"\n  ref: {normalize(r.reference)}\n  hyp: {normalize(r.hyp_original)}")
```

```
skipped: Qwen3-ASR not available here (ModuleNotFoundError). Falling back to the shipped judge transcripts.

slr42_0995_0102576285  [studio (OpenSLR 42)]  CER 5.0 %
  ref: ឧ,ចសេរ៉ាហានស្បៀបការ ូ ចលើយនិងមានកូនមួយ៉ំថមភ្លើត
  hyp: ជិ,តសេរ៉ាហានស្បៀបការ ូ ចលើយនិងមានកូនមួយ៉ំថមភ្លើត

slr42_4119_3188713809  [studio (OpenSLR 42)]  CER 11.9 %
  ref: ចាស់'ចាសនៅ ស,កមសាងមើ ណេនិងចិញ្ចឹមសត្វ លក'
  hyp: ចាស់'ឮនៅ ស,កមសាងមើ ណេនិងចិញ្ចឹមសត្វ លក'

iany_74a486db_ad28e57d  [phone (iAny)]  CER 25.6 %
  ref: បីងទូ លសាបណាបីងទឹកសាបធំបំផុ,តក្ន,ងតំបន'
  hyp: បីងទូ លសាបគឺណាបីងទឹកសាបធំនិងគបំពេ'ក្ន,ងតំបន'

iany_427db163_e7238ecd  [phone (iAny)]  CER 0.0 %
  ref:វ័ ពេលីថ្ងៃក្រសាបវិស្ថាននិងខ្យល់អាកាស
  hyp:វ័ ពេលីថ្ងៃក្រសាបវិស្ថាននិងខ្យល់អាកាស

iany_0b801284_2d8f0c4f  [phone (iAny)]  CER 17.6 %
  ref:ថ្ងៃនេះខ្ញុ,ក្រោកពី ព្រលឹមទៅសាលារៀន
  hyp:ថ្ងៃនេះចុំ ក,ងពី ព្រលឹមទៅសាលារៀន
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short look for **hallucination in the wild**: for each system, find the transcripts that are much longer than the reference, and print the 4 worst plus the worst one in full.

- These come from anonymized audio the judge could not understand; the decoder's language model filled the gap.
- `len_ratio` = length of the judge transcript ÷ length of the reference.

What the output means

In short the worst clips have length ratios **9.89, 8.82, 7.33 and 6.33** and CER **953 %, 841 %, 670 %, 567 %**; the worst one is a 19-code-point reference answered with **188 code points** of a repeated phrase.

- The repeated phrase (…និងកូនព្រហ្មនិងកូនព្រហ្ម…) is fluent Khmer that nobody said.
- This ran here (no model needed): it only reads `transcripts.csv`.

```
# Hallucination in the wild: judge transcripts that are much LONGER than the reference.
# These come from anonymized audio the judge could not understand; the decoder's language model filled the gap.
sys_cols = [c for c in trans.columns if c.startswith("hyp_")]
long = []
for c in sys_cols:
    ratio = trans[c].map(normalize).str.len() / trans.reference.map(normalize).str.len()
    for i in np.argsort(-ratio.values)[:2]:
        long.append(dict(system=c[4:], utt_id=trans.utt_id[i], len_ratio=ratio[i],
                         cer=cer_norm(trans.reference[i], trans[c][i])))
long = pd.DataFrame(long).sort_values("len_ratio", ascending=False).drop_duplicates("utt_id").head(4)
print(long.round(2).to_string(index=False))
r = long.iloc[0]; row = trans.set_index("utt_id").loc[r.utt_id]
print(f"\nworst case: system = {r.system}\n  ref ({len(normalize(row.reference))} cp): {normalize(row.reference)}"
      f"\n  hyp ({len(normalize(row['hyp_' + r.system]))} cp): {normalize(row['hyp_' + r.system])[:200]} ...")
```

```
                       system                utt_id  len_ratio    cer
private_knnvc_english_durations iany_ab586c1d_6752a2be       9.89 952.63
                       sslsas iany_18151a80_1760ed0c       8.82 840.91
private_knnvc_english_durations iany_1f8f808c_b0ee21dc       7.33 670.00
              knnvc_khmer_pool iany_b643351e_c1887ada       6.33 566.67

worst case: system = private_knnvc_english_durations
  ref (19 cp): មួយឆ្នាំមានដប់ពីរខែ
  hyp (188 cp):ឱឱធ្វើរឿងនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ្មនិងកូនព្រហ ...
```

Additional Reading

In short length ratio 6-10× with CER far above 100 % = hallucination; log the ratio next to CER.

Reading the output. A length ratio of 6-10 with CER far above 100 % is the signature of hallucination: the hypothesis is many times longer than the reference, often a phrase repeated, and in fluent Khmer. CER cannot tell "wrong but short" from "wrong and long", so the project caps nothing and lets these count. That is why the `private_knnvc_english_durations` row in the leaderboard shows CER above 100 %. If you ever build a judge, log the length ratio next to CER.

Additional reading: you can read all of it or just read the "In short"

Models tested in section 6

In short Whisper-small cannot read Khmer; Qwen3-ASR can, so it is the judge.

Model	Made by	Size	Type	Khmer result
Whisper-small	OpenAI	244M	attention encoder-	Fails (Vicheese's run). Default: CER 462 %, repetition loops in the wrong script. Anti-loop settings: CER 98

			decoder	%, short but still not the right Khmer.
Whisper-large-v3	OpenAI	1.55B	attention encoder-decoder	*Not run:* the download was not finished. No result.
Qwen3-ASR 1.7B	Alibaba	1.7B	audio-language model	Strong. CER ~7.6 % on original (not anonymized) speech (project's saved transcripts). This is the project's judge.

Makers and sizes in the table are background facts from Vicheese, not checked here.

Takeaway: Khmer was a tiny part of Whisper's training, so the small model cannot read it and its decoder hallucinates. Qwen3-ASR handles Khmer well, which is why it is used to score every anonymized system.

7. Reading the project's CER table without any model

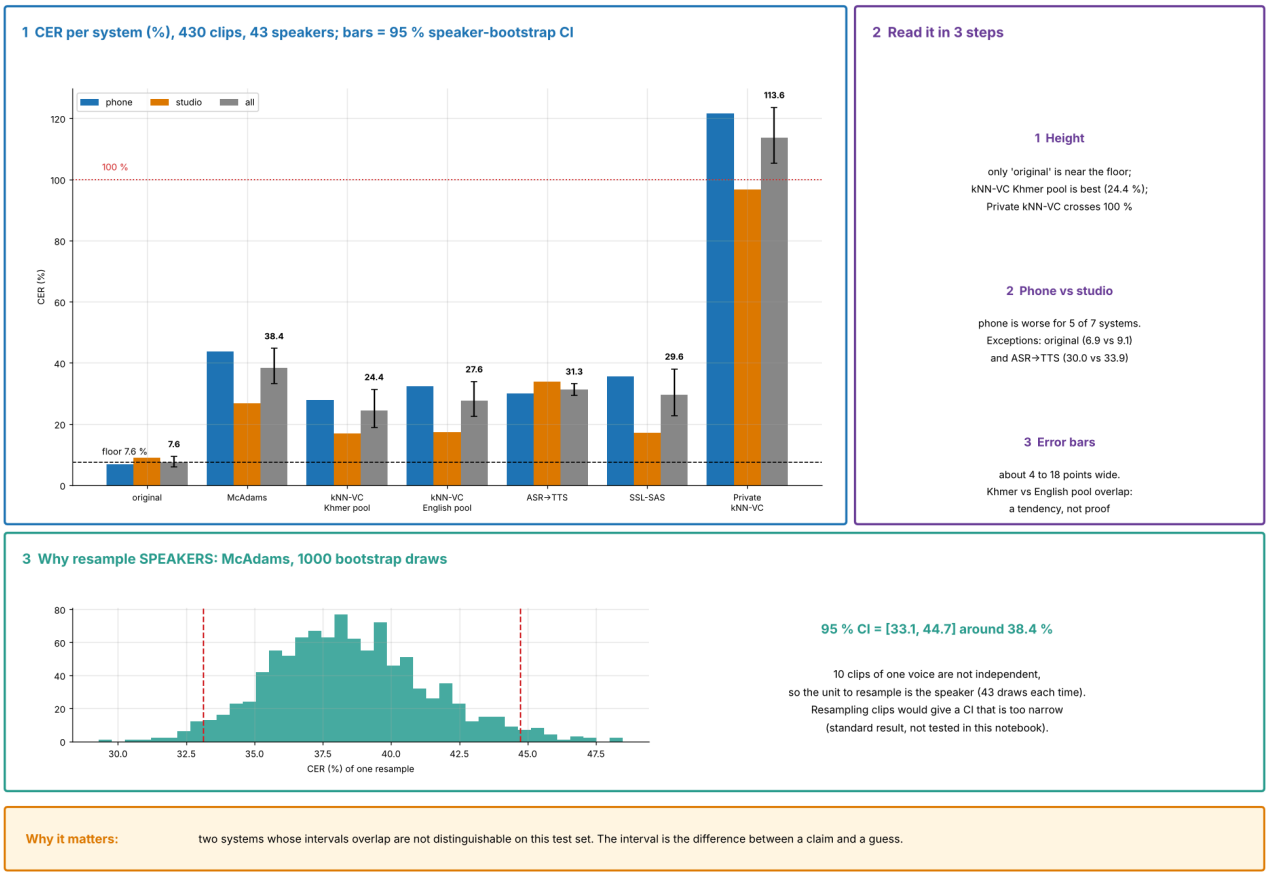
`transcripts.csv` holds the reference and the judge's transcript of the same 430 test clips after each anonymization system. We can recompute the leaderboard's CER with nothing but `normalize()` and edit distance. Two details matter:

- **Corpus-level ("micro") CER** = total edits ÷ total reference characters over the whole set. This is what the leaderboard reports, and it weights long utterances more. The mean of per-utterance CERs ("macro") is a different, usually close, number. We report micro.
- **Uncertainty.** The 430 clips come from **43 speakers**. Clips of one speaker are not independent: a voice the judge finds hard is hard in all 10 of its clips. If we bootstrap *utterances* we pretend we have 430 independent samples, and the interval comes out too narrow (standard result, not tested in this notebook). The honest unit of resampling is the **speaker**: draw 43 speakers with replacement, pool their edits and characters, recompute CER, repeat 1000 times, take the 2.5 and 97.5 percentiles.

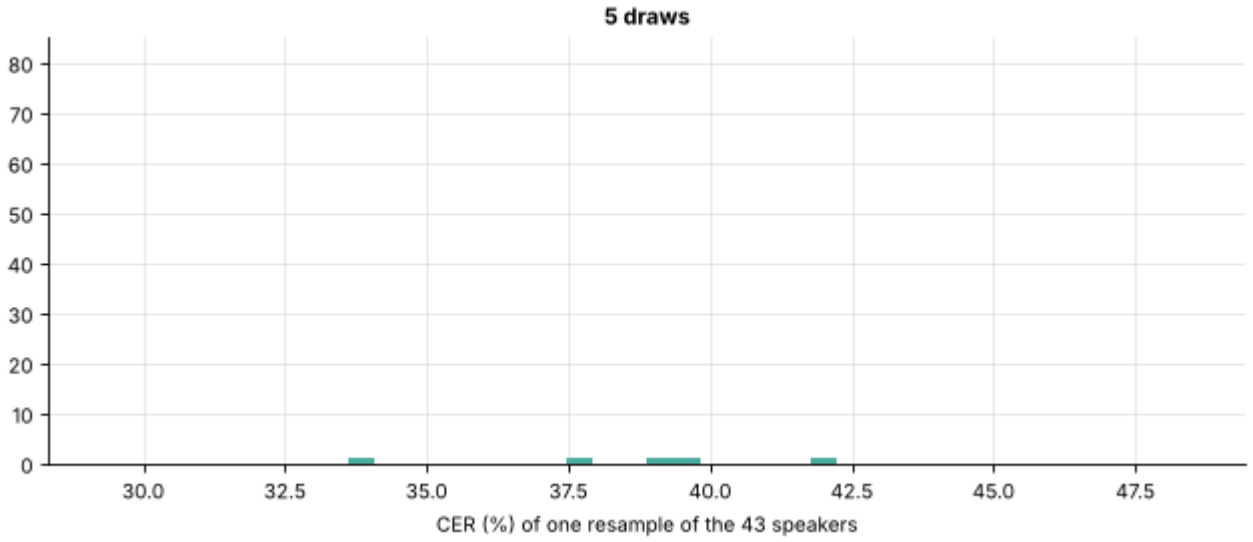
Why it matters. Two systems that differ by 1 CER point with overlapping speaker-bootstrap intervals are not distinguishable on this test set. The interval is the difference between a claim and a guess.

Visualization Version

1st image is the summary. The animation shows where a confidence interval comes from.



Animated: the speaker bootstrap for McAdams fills up to 1000 draws and gives the 95 % CI



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short compute, for every system, the edit count per clip, then the **micro CER** (all clips, phone, studio), the macro CER and the **speaker-bootstrap 95 % CI**, and compare with the leaderboard.

- `E_<system>` : edit distance per clip; `N` : reference length after normalization.
- `micro_cer` : 100 × ΣE ÷ ΣN. `speaker_bootstrap` : add up E and N per speaker, draw 43 speakers with replacement 1000 times, and take the 2.5 / 97.5 percentiles.
- The last line prints the biggest difference between our numbers and `leaderboard.csv` .

What the output means

In short 43 speakers, 320 phone + 110 studio clips; CER: **original 7.59, McAdams 38.36, kNN-VC Khmer pool 24.43, kNN-VC English pool 27.56, ASR→TTS 31.25, SSL-SAS 29.62, Private kNN-VC 113.64**; biggest difference to the leaderboard: **0.06** points.

- 2 hypotheses are missing in the judge output; they are treated as empty text.
- 95 % CI examples: original [5.94, 9.44]; McAdams [33.13, 44.74]; Khmer pool [18.94, 31.35]; English pool [22.53, 33.90].
- Macro CER (e.g. 39.02 for McAdams) is close to micro (38.36) but not the same; for Private kNN-VC they differ most (117.56 vs 113.64).

```
# per-utterance edits and reference lengths for every system, plus speaker ids
spk = pd.read_csv(DATA / "meta/cer_test_utts.csv")[["utt_id", "speaker"]]
tr = trans.merge(spk, on="utt_id", how="left")
tr["ref_n"] = tr.reference.map(normalize)
tr["N"] = tr.ref_n.str.len()
tr["is_phone"] = tr.corpus.str.startswith("phone")
systems = ["original", "mcadams", "knnvc_khmer_pool", "knnvc_english_pool", "asr_tts", "sslsas",
"private_knnvc_english_durations"]
for s in systems:
    tr[f"E_{s}"] = [edit_distance(r, normalize(h)) for r, h in zip(tr.ref_n, tr[f"hyp_{s}"])]
print("speakers:", tr.speaker.nunique(), "| phone clips", tr.is_phone.sum(), "| studio clips", (~tr.is_phone).sum(),
      "| missing hypotheses treated as empty:", int(tr[[f"hyp_{s}" for s in systems]].isna().sum().sum()))

def micro_cer(df, s):
    return 100.0 * df[f"E_{s}"].sum() / df["N"].sum()

def speaker_bootstrap(df, s, B=1000, seed=0):
    # resample SPEAKERS with replacement; each draw pools that speaker's edits and characters
    g = df.groupby("speaker")[[f"E_{s}", "N"]].sum()
    rng = np.random.default_rng(seed)
    idx = rng.integers(0, len(g), size=(B, len(g)))
    E = g[f"E_{s}"].values[idx].sum(1); N = g["N"].values[idx].sum(1)
    return np.percentile(100.0 * E / N, [2.5, 97.5])

rows = []
for s in systems:
    lo, hi = speaker_bootstrap(tr, s)
    rows.append(dict(system=s, cer=micro_cer(tr, s), cer_phone=micro_cer(tr[tr.is_phone], s),
                     cer_studio=micro_cer(tr[~tr.is_phone], s), ci_low=lo, ci_high=hi,
                     cer_macro=100 * (tr[f"E_{s}"] / tr["N"]).mean(),
                     leaderboard_cer=lb.loc[s, "cer"], leaderboard_phone=lb.loc[s, "cer_phone"],
leaderboard_studio=lb.loc[s, "cer_studio"]))
table = pd.DataFrame(rows).set_index("system")
print(table.round(2).to_string())
print("\nmax |ours - leaderboard| over all cells: %.2f CER points"
      % np.abs(table[["cer", "cer_phone", "cer_studio"]].values - table[["leaderboard_cer", "leaderboard_phone",
"leaderboard_studio"]].values).max())
```

```
speakers: 43 | phone clips 320 | studio clips 110 | missing hypotheses treated as empty: 2
                                cer  cer_phone  cer_studio  ci_low  ci_high  cer_macro  leaderboard_cer  leaderboard_pho
ne  leaderboard_studio
system
original                       7.59       6.89        9.09    5.94     9.44       7.41             7.58               6.
88                9.09
mcadams                       38.36      43.77       26.79   33.13    44.74      39.02            38.35              43.
77               26.77
knnvc_khmer_pool              24.43      27.90       17.01   18.94    31.35      25.19            24.42              27.
89               16.99
knnvc_english_pool            27.56      32.33       17.36   22.53    33.90      29.10            27.55              32.
31               17.36
asr_tts                       31.25      30.02       33.89   29.44    33.17      30.96            31.25              30.
02               33.87
sslsas                        29.62      35.50       17.05   22.84    38.06      31.81            29.62              35.
50               17.05
private_knnvc_english_durations  113.64     121.51       96.80  105.32   123.56     117.56           113.60             121.
```

```
45              96.80

max |ours - leaderboard| over all cells: 0.06 CER points
```

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short draw three bars per system (phone, studio, all clips with the 95 % CI), the judge floor (dashed) and the 100 % line (dotted).

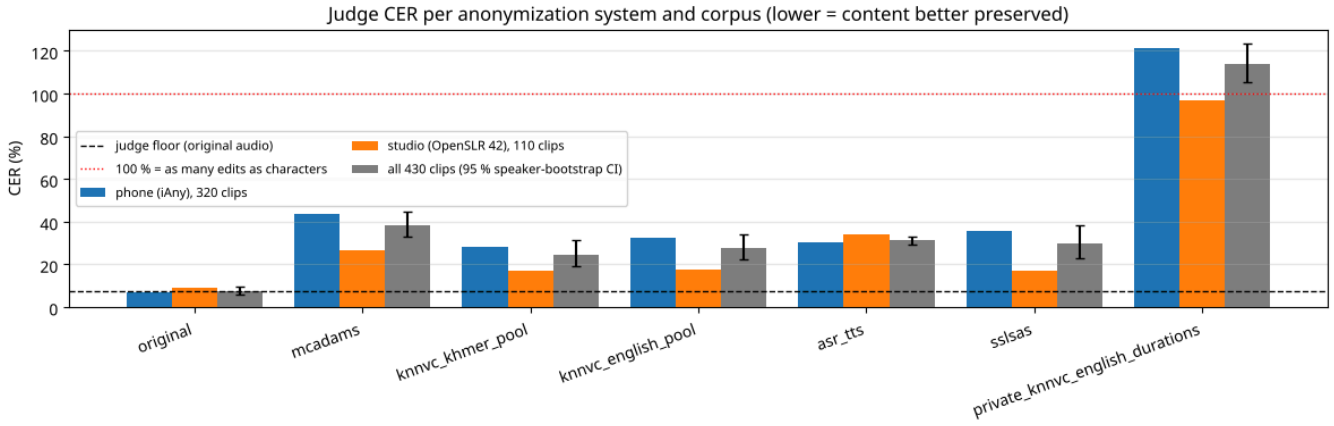
- Error bars use the speaker-bootstrap interval from the previous cell.

What the output means

In short only `original` sits near the floor (7.6 %); **Private kNN-VC crosses 100 %.**

- Read it with the three steps in the notes below.

```
fig, ax = plt.subplots(figsize=(12, 4))
xs = np.arange(len(systems)); w = 0.27
ax.bar(xs - w, table.cer_phone, w, label="phone (iAny), 320 clips")
ax.bar(xs,     table.cer_studio, w, label="studio (OpenSLR 42), 110 clips")
ax.bar(xs + w, table.cer, w, label="all 430 clips (95 % speaker-bootstrap CI)", color="gray",
       yerr=[table.cer - table.ci_low, table.ci_high - table.cer], capsize=3)
ax.axhline(table.loc["original", "cer"], ls="--", color="k", lw=1, label="judge floor (original audio)")
ax.axhline(100, ls=":", color="r", lw=1, label="100 % = as many edits as characters")
ax.set_xticks(xs); ax.set_xticklabels(systems, rotation=20, ha="right")
ax.set(ylabel="CER (%)", title="Judge CER per anonymization system and corpus (lower = content better preserved)")
ax.legend(fontsize=8, ncol=2); ax.grid(axis="y", alpha=0.3); plt.tight_layout(); plt.show()
```



Additional Reading

In short height = how much content is lost; phone is usually worse than studio; overlapping error bars = no proven difference.

How to read this plot. Three bars per system: blue = phone recordings, orange = studio, gray = everything with its 95 % speaker-bootstrap interval. The dashed line is the judge's floor on clean audio (about 7.6 %); no anonymizer can score below it except by luck. Read it in three steps.

1. **Height.** `original` is the only bar near the floor. kNN-VC with the Khmer pool is the best anonymizer (~24 %), the English pool a few points worse (27.6 %). SSL-SAS and the ASR→TTS cascade are around 30 %, McAdams ~38 %. The released Private kNN-VC crosses the red 100 % line: its English phone recognizer plus duration model produces audio the judge cannot read at all (hallucination from section 6).
2. **Phone vs studio.** For 5 of the 7 systems the blue bar is well above the orange one: noisy phone audio degrades more under anonymization. There are **two exceptions**. `original` is the first: phone 6.9 % vs studio 9.1 %. `asr_tts` is the second: phone 30.0 % vs studio 33.9 %. Its output is synthetic speech made from a transcript, so the recording condition is erased and what remains is the ASR error on the input. Vicheese's idea (not tested here): the studio sentences are longer and have rarer vocabulary, so the cascade's own ASR step does worse there.
3. **Error bars.** The intervals on the gray bars are **about 4 to 18 points wide** (43 speakers only). `knnvc_khmer_pool` vs `knnvc_english_pool` overlap, so "Khmer pool is better" is a tendency, not a proven difference. `original` vs everything else, and `private_knnvc` vs everything else, are far outside each other's intervals.

Additional reading: you can read all of it or just read the "In short"

What this cell does

In short print the 3 best and 3 worst clips for McAdams (per-clip CER), with the reference and the judge's transcript side by side.

- `order`: all clips sorted by McAdams CER. `alpha` is the warp strength used for that clip.

What the output means

In short **49 of 430** clips are transcribed perfectly under McAdams; the 3 worst have CER **210 %, 405 % and 572 %**, all with alpha ≈ 0.6 (a strong warp) on phone audio.

- The 3 best are random picks from the many CER = 0 clips (alpha 0.80-0.87), so which three show up can change between runs.
- The worst hypotheses are long looping phrases.

```
# The 3 worst and 3 best utterances for McAdams (per-utterance CER), reference vs hypothesis side by side
s = "mcadams"
tr["cer_utt"] = 100 * tr[f"E_{s}"] / tr["N"]
order = tr.sort_values("cer_utt")
def show_rows(df, title):
    print(title)
    for _, r in df.iterrows():
        print(f"  {r.utt_id}  [{r.corpus[:6]}]  alpha={r.mcadams_alpha:.2f}  CER {r.cer_utt:6.1f} %")
        print(f"    ref: {r.ref_n}\n    hyp: {normalize(r[f'hyp_{s}'])}")
show_rows(order.head(3), "3 BEST (McAdams):")
show_rows(order.tail(3), "\n3 WORST (McAdams):")
print("\nutterances with CER = 0 under McAdams: %d / %d" % ((tr.cer_utt == 0).sum(), len(tr)))
```

```
3 BEST (McAdams):
  iany_046e9e11_ffc16abb  [phone ]  alpha=0.87  CER    0.0 %
    ref:ព្រៃឈើជួយយើងក្នុងការដកដង្ហើមខ្យល់អាកាស
    hyp:ព្រៃឈើជួយយើងក្នុងការដកដង្ហើមខ្យល់អាកាស
  slr42_7548_2931259667  [studio]  alpha=0.87  CER    0.0 %
    ref: អប្សរាជាអត្តសញ្ញាណជាតិខ្មែរ
```

```
    hyp: អបុសរាជាអត្តសញ្ញាណជាតិខ្មែរ
  iany_21c36386_db05c5ab  [phone ]  alpha=0.80  CER    0.0 %
    ref: បឹងទន្លេសាបជាបឹងទឹកសាបធំបំផុតក្នុងតំបន់
    hyp: បឹងទន្លេសាបជាបឹងទឹកសាបធំបំផុតក្នុងតំបន់

3 WORST (McAdams):
  iany_c55ee1da_cf425f75  [phone ]  alpha=0.59  CER  210.3 %
    ref: បឹងទន្លេសាបជាបឹងទឹកសាបធំបំផុតក្នុងតំបន់
    hyp:បើកទាំងអូនកសរ័ចកបើកទាំងអក្សរផងដែរប៉ុន្មានក្នុងបណ្ណយបើកទាំងអូនកសរ័ចកបើកទាំងអក្សរផងដែរប៉ុន្មានក្នុងបណ្ណយ
  iany_914926eb_4ba46ad8  [phone ]  alpha=0.64  CER  405.1 %
    ref: បឹងទន្លេសាបជាបឹងទឹកសាបធំបំផុតក្នុងតំបន់
    hyp:បើកតងឱ្យសរជាកុបិនទឹកសាបធំមួយមួយបើកតងប៉ណ្ណាសរជាកុបិនទឹកសាបធំមួយមួយនិងបើកតង្រសកសរជាកុបិនទឹកសាបធំមួយមួយនិងបើកតង្រសកសរជាកុបិនទឹកសាបធំមួយមួយនិងបើកតង្រសកសរជាកុបិនទឹកសាបធំមួយមួយ
  iany_c55ee1da_09b45dba  [phone ]  alpha=0.61  CER  572.2 %
    ref: មដាយធ្វើនំចំណីផ្អមស្រមាប់ពិធីបុណ្យ
    hyp: ចំណតតលើញ្ជីអងចង្កុះផ្អមស្រមាកពិតបំណងគឺជាអូនកចង្កុះដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអូនកដលមានអារម្មណ៍ថាជាអ

utterances with CER = 0 under McAdams: 49 / 430
```

Additional reading: you can read all of it or just read the "In short"

Reading the examples

In short 49 of 430 clips are perfect; the worst ones are hallucination loops (CER 200-570 %), triggered by a strong warp.

This cell prints McAdams' 3 best and 3 worst clips, with the correct sentence next to what the judge heard.

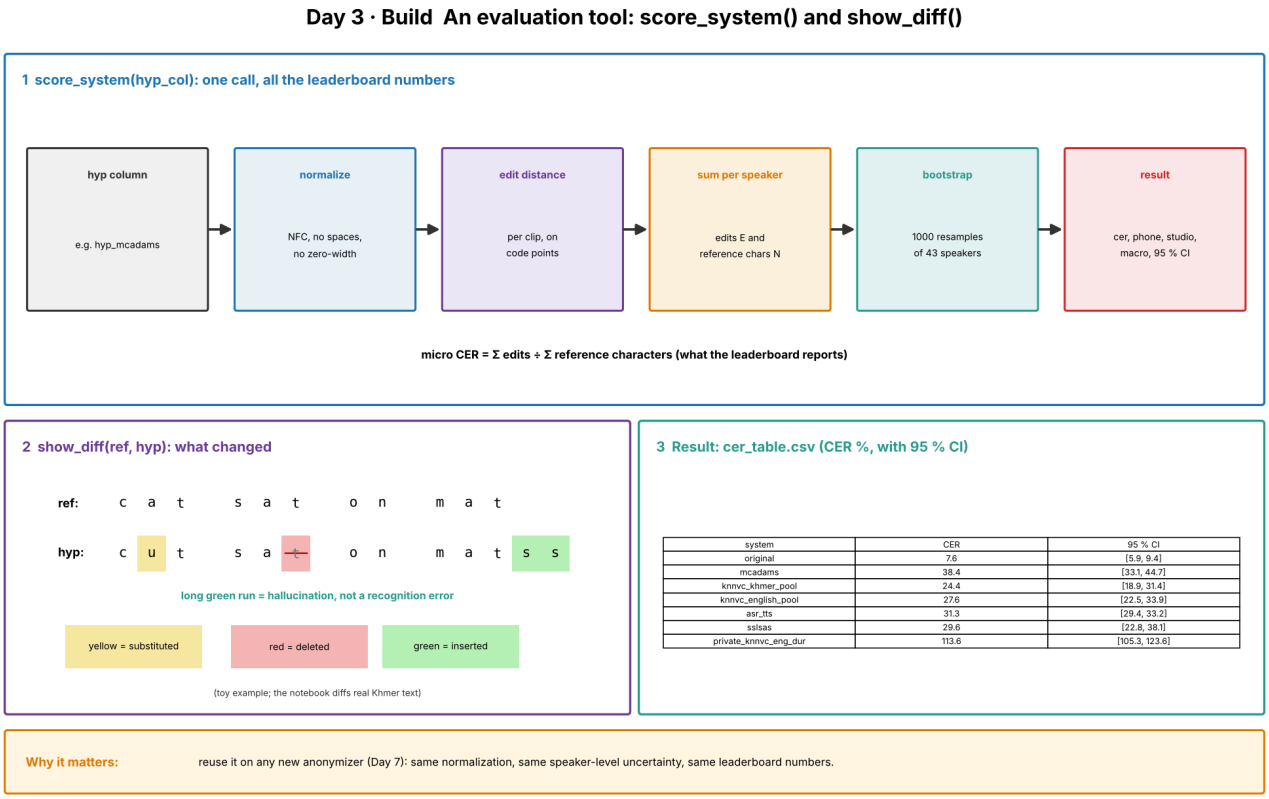
Reading the examples. The best cases (**49 of 430** clips are transcribed perfectly) are common sentences with a mild warp (alpha 0.8-0.9 means little change; the leaderboard draws alpha uniformly in [0.5, 0.9]). The worst cases combine a strong warp (alpha around 0.6) with phone audio. They are *not* ordinary recognition errors: the judge produces a looping phrase several times longer than the reference, CER **210-572 %**. That is the hallucination from section 6 again, this time triggered by a formant shift rather than by a different anonymizer.

The typical McAdams error (see the median-clip diff in the Build section) is milder: sentence length and rhythm survive, and some letters, vowel signs or whole short words get swapped. The 6 McAdams clips with CER above 100 % (1 % of the clips) give about 11 % of all McAdams edits (computed separately, not printed by the cells). Report CER with a confidence interval anyway.

Build: an evaluation tool

Two functions you can reuse on any new system: `score_system(hyp_col)` returns the numbers the leaderboard needs (overall / phone / studio micro CER, macro CER, speaker-bootstrap 95 % CI), and `show_diff(ref, hyp)` highlights what changed using `difflib`. Results go to `outputs/day3/cer_table.csv`.

Visualization Version



Additional reading: you can read all of it or just read the "In short"

What this cell does

In short define `score_system()` and `show_diff()`, score all 7 systems into `cer_table.csv`, and show a diff for the median and the worst McAdams clip.

- `score_system` repeats section 7 in one function, for any hypothesis column.
- `show_diff` compares reference and hypothesis with `difflib`, prints a coloured diff (green = inserted, red = deleted, yellow = substituted) and returns a plain-text version like `[ម>បណ]`.
- The last lines apply it to the **median** McAdams clip and to the **worst** one.

What the output means

In short the table has **the same CER and CI as section 7** (for example McAdams 38.36 [33.13, 44.74], Private kNN-VC 113.64 [105.32, 123.56]) and is saved to `outputs/day3/cer_table.csv`.

- The median clip's plain-text diff has 5 changed spots spread over the sentence: `[ីក>ឹត]`, `[ច>ន]`, `[ុ>ូ]`, `[ាា>ាច]`, `[ៃក>ឹត]`.

- The worst clip's diff has about 8 changed spots: several short substitutions (e.g. `[ម>បណ]`, `[ប់>ក]`, `[ើ>ត]`, `[ុ>ំ]`), one short insertion (`[+ងគឺជាអ]`), and then **one very long replaced run** (`[យ>នកចង្កេះដែល…]`, far longer than the text it replaces): hallucination.

```
def score_system(hyp_col, df=tr, B=1000, seed=0):
    # hyp_col: name of a column in df holding hypotheses (e.g. "hyp_mcadams"). Returns a dict of CER numbers in %.
    ref = df.reference.map(normalize); hyp = df[hyp_col].map(normalize)
    E = np.array([edit_distance(r, h) for r, h in zip(ref, hyp)]); N = ref.str.len().values
    phone = df.corpus.str.startswith("phone").values
    g = pd.DataFrame({"E": E, "N": N, "speaker": df.speaker.values}).groupby("speaker").sum()
    rng = np.random.default_rng(seed)
    idx = rng.integers(0, len(g), size=(B, len(g)))
    boot = 100 * g.E.values[idx].sum(1) / g.N.values[idx].sum(1)
    return dict(system=hyp_col.replace("hyp_", ""), n_utts=len(df), n_speakers=len(g),
                cer=100 * E.sum() / N.sum(), cer_phone=100 * E[phone].sum() / N[phone].sum(),
                cer_studio=100 * E[~phone].sum() / N[~phone].sum(), cer_macro=100 * (E / N).mean(),
                ci_low=np.percentile(boot, 2.5), ci_high=np.percentile(boot, 97.5))

def show_diff(ref, hyp):
    # Character-level diff after normalization. Green = inserted, red = deleted, yellow = substituted.
    r, h = normalize(ref), normalize(hyp)
    sm = difflib.SequenceMatcher(a=r, b=h, autojunk=False)
    html, text = [], []
    for op, i1, i2, j1, j2 in sm.get_opcodes():
        if op == "equal":
            html.append(h[j1:j2]); text.append(h[j1:j2])
        elif op == "insert":
            html.append(f'<span style="background:#b7f0b7">{h[j1:j2]}</span>'); text.append(f"[+{h[j1:j2]}]")
        elif op == "delete":
            html.append(f'<span style="background:#f5b7b7;text-decoration:line-through">{r[i1:i2]}</span>');
text.append(f"[-{r[i1:i2]}]")
        else:
            html.append(f'<span style="background:#f7e7a0"><s>{r[i1:i2]}</s>{h[j1:j2]}</span>'); text.append(f"
[{r[i1:i2]}>{h[j1:j2]}]")
    c = 100 * edit_distance(r, h) / max(len(r), 1)
    display(HTML(f'<div style="font-size:1.4em;line-height:1.8">ref: {r}<br>hyp: {"".join(html)}'
                 f'<br><small>CER {c:.1f} %  (green = inserted, red = deleted, yellow = substituted)</small></div>'))
    return "".join(text)

cer_table = pd.DataFrame([score_system(f"hyp_{s}") for s in systems]).set_index("system")
cer_table.to_csv(OUT / "cer_table.csv")
print(cer_table.round(2).to_string())
print("\nsaved", OUT / "cer_table.csv")

# the diff on the median McAdams utterance and on a worst one
mid = order.iloc[len(order) // 2]
print("\nplain-text diff:", show_diff(mid.reference, mid.hyp_mcadams))
worst = order.iloc[-1]
print("plain-text diff:", show_diff(worst.reference, worst.hyp_mcadams)[:150], "...")
```

system	n_utts	n_speakers	cer	cer_phone	cer_studio	cer_macro	ci_low	ci_high
original	430	43	7.59	6.89	9.09	7.41	5.94	9.44
mcadams	430	43	38.36	43.77	26.79	39.02	33.13	44.74
knnvc_khmer_pool	430	43	24.43	27.90	17.01	25.19	18.94	31.35
knnvc_english_pool	430	43	27.56	32.33	17.36	29.10	22.53	33.90
asr_tts	430	43	31.25	30.02	33.89	30.96	29.44	33.17
sslsas	430	43	29.62	35.50	17.05	31.81	22.84	38.06
private_knnvc_english_durations	430	43	113.64	121.51	96.80	117.56	105.32	123.56

```
saved ../outputs/day3/cer_table.csv
```

ref: ធឹកទឹកឲយបានរចើនស្មៀសវាងការខូសរជាតិទឹក

hyp: ~~ធឹកថតទឹក~~ឲ~~ខ~~យបានរចើនស្មៀនសវាងការខូស~~ណាយចតិ~~ទឹក~~ចិត~~

CER 34.2 % (green = inserted, red = deleted, yellow = substituted)

```
plain-text diff: [ធឹក>ថត]ទឹក[ឲ>ខ]យបានរចើនជស្មៀំនួ]សវាងការខូស[ណា>យច]តិ[ទឹក>ចិត]
```

ref: មដាយធ្វើនិចំណើផ្អមសរមាប់ពីធីបុណ្យ

hyp: ~~មបណ្តាដាយធ្វើតតាយើញ្ជើអងចង្កេះដំចំណើ~~ផ្អមសរមា~~ម់ក~~

ពី~~ធីត~~ បុំណ្យងគឺជាអ្នកចង្កេះដែលមានអាយុមណីថាជាអ្នកតំដែលមាន

អាយុមណីថាជាអ្នកតំដែលមានអាយុមណីថាជាអ្នកតំដែលមានអាយុមណីថា

ជាអ្នកដែលមានអារម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ថាជាអ

CER 572.2 % (green = inserted, red = deleted, yellow = substituted)

```
plain-text diff: [ម>បណ].ុ[ដាយចង់តឈើឆ្នើរអង្គង].ុ[រឿនំចំណីអ្នកឱ្យអមសរមា[ប>ក]ពិ[ជឺ>ត]ប[ុំ]ណ[+ងគឺជាអ].ុ[យ>នកចង្កងដែលមានអារ
ម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ថាជាអ្នកដែលមានអារម្មណ៍ ...
```

Additional reading: you can read all of it or just read the "In short"

What the tool is for

In short one call gives the leaderboard numbers for any new system.

It is a tool for measuring utility: how understandable the speech still is after anonymization. It lets you score any anonymizer, including one you build yourself, and compare it fairly with the others on the leaderboard.

Additional Reading

In short yellow = substituted (old struck through, new after it), red = dropped, green = invented; a long green run = hallucination.

Reading the diff. Yellow spans are substitutions (old struck through, new after it). In the median clip they are a mix: one short word, one letter variant (ត → ន), and a few letters or vowel signs. Red is text the judge dropped, green is text it invented. When green dominates and runs for a whole phrase, you are looking at **hallucination, not recognition error**.

Day 3: what I can now explain

1. The alignment problem, and its two standard answers

My audio gives me about **100 frames per second**, but a sentence has only about 40 characters. Many frames belong to each character, and the dataset never says *which* frames go with *which* character. People also speak at different speeds: one stretches a vowel over 20 frames, another says it in 5. That is the **alignment problem**.

There are two standard answers:

- **CTC:** the model outputs one symbol **or a blank** on every frame. Then I merge repeats and drop blanks to get the text. In training, I sum the probability of **every** frame lineup that spells the correct text, so I never need frame labels. It can only write what the frames support, so it **cannot hallucinate long text**.
- **Attention encoder-decoder** (like Whisper): an encoder reads the whole audio, and a decoder writes the text **one token at a time**, choosing where to "look" in the audio at each step. It is

fluent, because the decoder is a language model. But when the audio is unclear it can **invent text or loop**.

2. Khmer as a sequence of code points; CER, not WER

To the computer, Khmer is not syllables. It is a row of **Unicode code points**. ខ្ញុំ looks like one block, but it is **5 pieces**: ខ + ◌្ (coeng) + ញ + ◌ុ + ◌ំ. My model predicts these pieces, and my metric counts them.

Before scoring, I **normalize** both texts the same way: Unicode **NFC**, then remove spaces and zero-width characters. Otherwise the same word written two ways, or an extra space, would count as an error.

Why CER, not WER: Khmer is written **without spaces between words**, so there is no reliable way to split it into words. WER would depend on which word segmenter I used. CER = (substitutions + deletions + insertions) ÷ reference characters. It is well defined, and it gives partial credit: one wrong vowel sign costs 1 character, not a whole word.

3. CTC collapse, enumeration and the forward algorithm by hand

- **Collapse rule:** first **merge repeats**, then **drop blanks**. `cc-aa-t` → `c-a-t` → `cat`. A double letter needs a blank in between: `a-a` → `aa`, but `aa` → `a`.
- **Enumerate:** list every possible frame path and keep the ones that collapse to the target. For "cat" over 5 frames with 4 symbols, that is 4^5 = **1024 paths, and 28 of them** give "cat". P(text) is the sum of those 28 path probabilities. This only works for tiny examples, because the number of paths explodes.
- **Forward algorithm (alpha trellis):** a grid with **frames across** and the target **with blanks in between** going down (`- c - a - t -`). Each cell α(t, s) = the total probability of every path that reaches that state at that frame. A cell is filled from **the same position in the frame before** (stay), **position s−1** (advance) and **position s−2** (skip a blank, allowed only if the two letters differ). Then multiply by the probability of that symbol at frame t. The answer is the **sum of the last two cells**. Because it works in log space, adding probabilities uses `logaddexp`. My worked example: log P = −4.4468, about 1.2 %, which matches the brute-force enumeration exactly, with far less work.

4. Training a small CTC model and decoding it

- **TinyCTC:** a Conv1d with stride 4 (shrinks the time steps) → a BiGRU with 96 units per direction (reads the sequence in both directions) → a Linear layer to **48 outputs** (47 characters + blank). About **160k parameters** (159.5k). Trained with CTC loss, Adam and gradient clipping.
- **Result:** training loss went from **4.74 to 0.043**. CER was **0.6 % on the training clips but about 67 % (66.9 %) on new speakers**. It **memorized** voices instead of learning Khmer, because 36 clips is far too little data.
- **Spiky posteriors:** about **65 % of frames are blank**, and each character appears as a sharp **spike** on one or two frames. That is normal for CTC: blank fills the gaps between characters.

- **Greedy decoding:** take the best symbol per frame, then collapse. It is fast, but it follows **one path only**.
- **Beam search:** keep the **8 best texts**, merge paths that spell the same text, and score each text by its **total** probability. Example: blank 0.6 vs "a" 0.4 on two frames. Greedy outputs nothing, but "a" really has total probability **0.64**. On my held-out clips: greedy **66.9 %**, beam **65.4 %**. Beam gives the most *probable* text, not always the lowest CER. Its real value is that it is where a **language model** plugs in.

5. Whisper on Khmer, hallucination, and why the judge is Qwen3-ASR

- I ran **Whisper-small** on 7 Khmer clips. With default settings: CER **462 %**, length ratio **4.6×**. It wrote `ḍḍḍḍ…` , Georgian and Sinhala, all in **repetition loops**, never Khmer. With anti-loop settings: CER **98 %**, shorter output, but still not the right Khmer.
- **Hallucination** = a decoder model writing fluent text the speaker never said, often repeating a phrase. The sign is **CER > 100 % together with a length ratio well above 1**. CTC cannot do this, because it emits at most one symbol per frame.
- **Why Qwen3-ASR:** Khmer was a tiny slice of Whisper's training data. Qwen3-ASR gets **about 7.6 % CER on the original (not anonymized) audio**, so it is reliable enough to act as the **judge**. That 7.6 % is also the **floor**: an anonymizer near it keeps the speech as understandable as the original.

6. CER per system and corpus, with a speaker-level 95 % CI

- **Micro CER** = total edits ÷ total characters across all clips. This is what the leaderboard reports. **Macro CER** = the average of each clip's CER. It is usually close.
- I split by **corpus**: phone audio is harder than studio audio for almost every system (the exceptions are the original audio and the ASR→TTS cascade).
- **Confidence interval:** 430 clips but only **43 speakers**, and clips from the same speaker are not independent. So I **resample speakers with replacement**, 1000 times, and recompute CER each time. The 2.5th and 97.5th percentiles give the 95 % CI. Resampling clips would give an interval that is **too narrow** (standard result, not tested in this notebook).
- **Result:** original 7.6 %, kNN-VC Khmer pool **24.4 %** (the best anonymizer for utility), McAdams 38.4 %, Private kNN-VC **113.6 %** (hallucination). The Khmer pool and English pool **overlap**, so I cannot claim one is better.
- I packaged all of this as `score_system()` (CER plus CI for any system) and `show_diff()` (colours substitutions, deletions and insertions), for reuse in Day 7 and in my project.

One-sentence summary: Day 3 taught me how a model turns audio into Khmer text with CTC, how to measure its mistakes fairly with CER, and how to use a strong judge, Qwen3-ASR, to score how understandable each anonymizer leaves the speech.