Pushing the Limits of Efficient Text-to-Speech Training

ZDisket
nika109021@gmail.com

ABSTRACT

This technical report describes **Echolancer**, a neural network for speech synthesis from text. The pipeline is composed of a large language model-like model that predicts discrete audio tokens, followed by a neural audio codec that turns these tokens into audio directly. We train two base models from scratch with just one (1) GPU, of size 177 and 550 million parameters. Furthermore, we demonstrate that with finetuning, one can add speaker conditioning from an external embedder to achieve voice cloning with a clip of just two to ten seconds of audio of a speaker. Our model is tuned for efficiency while maintaining quality, being able to output natural speech while remaining light to deploy.

INTRODUCTION

The field of text-to-speech synthesis with neural networks—thanks to the boom of artificial intelligence—has seen great advancements in the past few years. Modern systems can output natural sounding speech in varying voices, emotions and tones.

However, the vast majority of these aforementioned feats have taken place almost exclusively in big, well-funded corporations with extensive access to experts and compute: as a consequence, most research coming from these places never sees open source, or if it does, only a stripped-down version does, as intellectual property is well-guarded.

Yet innovation should not be confined to the walls of large organizations. The aim of this work is to demonstrate that high-quality speech synthesis can be achieved efficiently, with limited resources and complete transparency, by a single independent researcher. This project, **Echolancer**, aims to explore how far one person, using only one AMD Instinct MI300X GPU, can go to replicate these advances and achieve good quality.

DATA AND MODEL ARCHITECTURE

One crucial characteristic of Transformer-based models is that—compared to RNN (LSTM-GRU)-based systems of old—they are very data hungry. Therefore, a suitable dataset must fit the following criteria:

1. Aligned speech audio and text transcription;
2. Good or acceptable quality;
3. Large scale enough;
4. Easily accessible;

We found the Emilia dataset[1], consisting of 70k+ hours of speech, to fit all of these requirements.

For our model, we went with the tried and tested large language model-based text to speech (LLM-TTS) recipe, used in previous models: a decoder-only model trained to predict concatenated text-audio discrete tokens, both sharing a vocabulary.

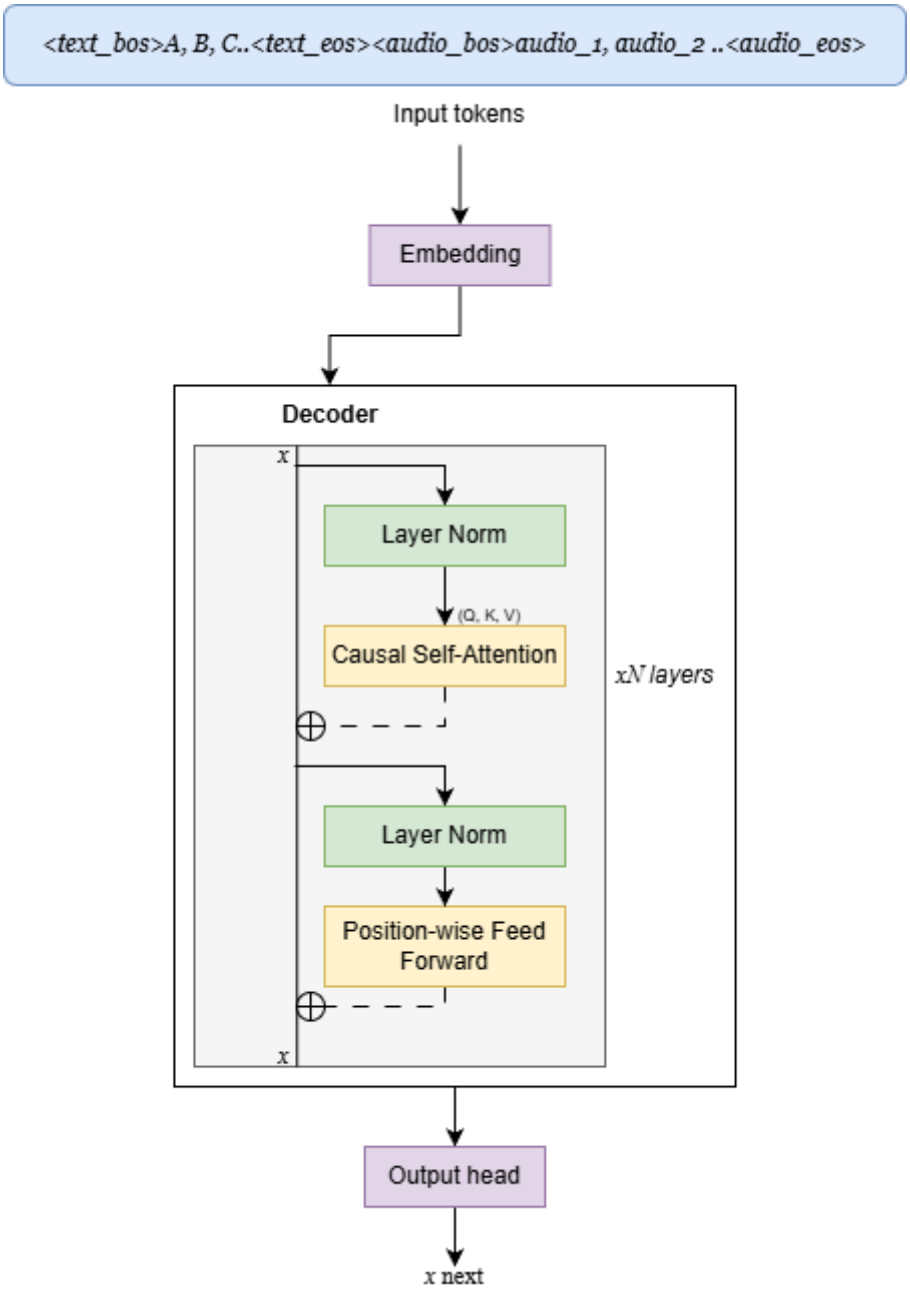


Figure 1: Our model architecture and input modality. Not shown here: we tie the embedding and output head weights to save parameters

Therefore, without explicit cross-attention or alignment mechanisms of any sort, the model learns to predict what audio comes after the appropriate text, making use of the audio and text EOS and BOS tokens as boundary markers, all by minimizing the next token cross-entropy loss, or in other words, the standard autoregressive language modeling objective.

[1] amphion/Emilia-Dataset · Datasets at Hugging Face

Unlike other models, we do not use any phonemizer; instead, our text uses character-level tokenization. This is to see if we can scale our way out of needing a phonemizer, which would be desirable for simplifying later deployment.

We follow best practices in modern transformers and utilize:

1. Pre-normalization
2. No bias in feed-forward
3. GELU activation
4. Grouped Query Attention

Parameter	177M (v0.1)	550M (v0.2)
Layers	10	27
Query heads	16	20
Key/Value heads	4	5
Width	1024	1280
Activation function in FFN	$ReLU^2$	GELU
Vocabulary size	65k	65k

There is one way in which we diverge from standard practice: for positional encoding in our attention, we **utilize Attention with Linear Biases (ALiBi) instead of Rotary Positional Embeddings (RoPE)**. This is because ALiBi is conceptually simple—a *linear* monotonic penalty added directly to attention scores, as opposed to rotary embeddings, which contains complex angular math, while not losing performance.

Due to its complicated math, RoPE is sensitive to differences in kernels, which can cause models to not work properly when exported and implemented in backends with even just minor differences in how these handle its calculations[2]—which has caused headaches[3]. In contrast, ALiBi's simplicity not only **eliminates these concerns**, but also makes it *much easier* to debug were something still to go wrong somehow. This is very important to us since ONNX export for inference in non-Python environments is a key objective.

In addition, places like MosaicML historically abandoned ALiBi not due to performance constraints, but because efficient kernels like Flash Attention didn't support it yet at that time[4]. However, all implementations of Flash Attention 2 support ALiBi, and the newest PyTorch Flex Attention supports *arbitrary* positional encoding, we are taking advantage of a mature optimized kernel stack.

AUDIO TOKENIZATION

Structurally modeling waveform directly is practically **unfeasible**—one second of 16KHz audio in PCM format equals *16 thousand individual floating point or integer values*: therefore, since early on in this field, different methods have been proposed to make audio digestible for deep learning models.

---

[2] vik; X: "all fun and games until you realize you have to implement rope again... https://t.co/VrwOXIdjDs" / X
[3] vik; X: "didn't expect i'd be spending yet another day figuring out why two rope implementations are producing different results... https://t.co/MBqqqrkHHN" / X
[4] Cody Blakeney; X: "@ZDi____ kernels was the big thing. Also our expectation of inference support." / X

One modality that was the de-facto for a while, popularized by the Tacotron series of models (2019), was the mel spectrogram, which simplified the task to predicting about 86 frames per second where each frame was 80 floating point values (mel bins), paired with a neural vocoder that learned to reconstruct audio from said spectrogram, which was a very lossy representation.

Eventually, advances in autoencoders and vector quantization led to the rise of neural audio codecs – models that learned to compress audio into discrete indices at much lower rates than conventional (hand-crafted) codecs, using one or (most of the time) multiple codebooks.

Since this coincides with how language models emit text, as a discrete distribution, suddenly directly generating audio tokens became a reality. As a result, modern text-to-speech and music generation systems began dropping previous modalities and started relying on these audio codecs. However, **not all of them are made the same, as they have differing token rates and number of codebooks.**

Indeed, we want the most efficient way of being able to language model audio. Measuring this is achieved by looking at the *total token rate*. Since neural audio codecs often emit multiple codebooks in parallel, the total rate is the sum of each codebook's token rate.

$$R_{\text{total}} = \sum_{i=1}^{N} R_i$$

Where N is the number of codebooks and R_i is the size of the codebook i. A lower R_{total} indicates fewer steps per second of audio, directly translating to faster autoregressive generation and lower compute cost; having efficient compression thus becomes very important since the cost of full attention in Transformers **scales quadratically with sequence length.**

We did attempt to roll our own neural audio codec, but this was a task too gargantuan to complete in reasonable time considering our resource constraints. For this reason, we decided to go with Neuphonic Lab's NeuCodec, which efficiently compresses audio to a single codebook with a total vocabulary size of 65k, at just 50 tokens/second.

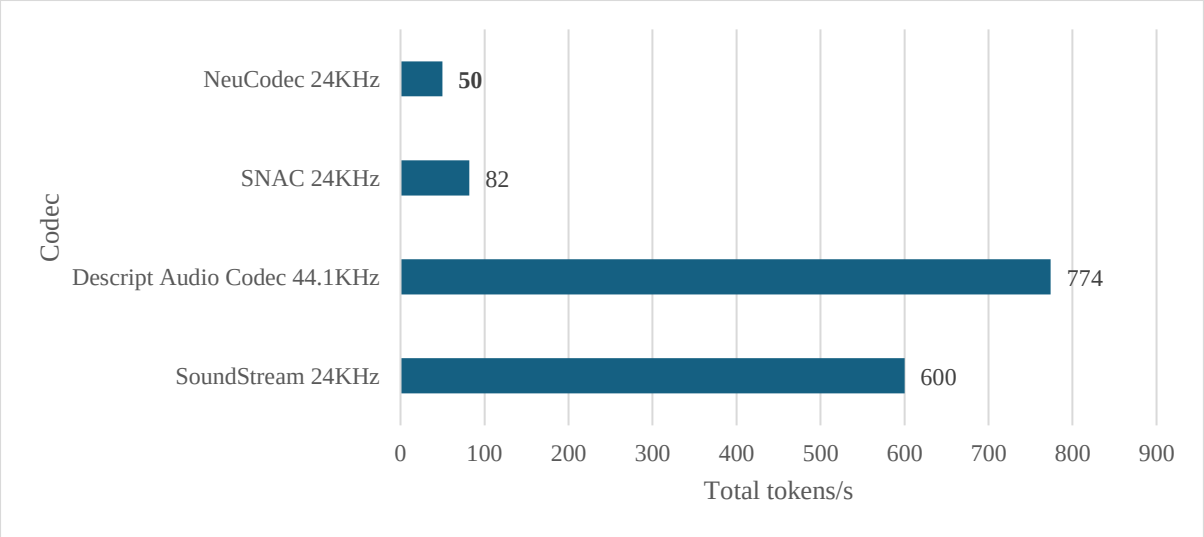


Figure 2: Total token rate of different audio tokens. Lower is better

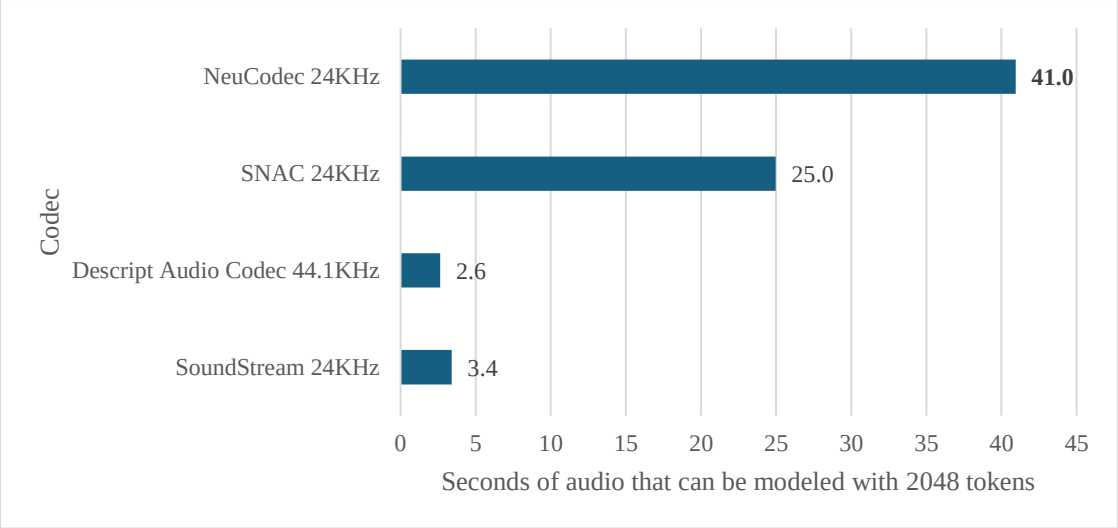


Figure 3: Seconds of audio that can be modeled with a sequence length of 2048. Higher is better

In addition, while the token rates of SNAC and NeuCodec were somewhat competitive (87 vs 50), the latter was trained on a much more diverse corpus of audio, leading it to perform better as well.

For these reasons, we chose NeuCodec as our audio tokenizer. Both the text and audio tokens live in the same vocabulary for our model. We add an index offset to the audio tokens (equivalent to the total text vocab size) during training, then at inference, said offset is subtracted from the output indices to yield a valid token sequence to feed the codec.

PRETRAINING

Our setup consists of a single AMD Instinct MI300X, with 192GB of VRAM, on a virtual machine, using the ROCm PyTorch Training Docker container, which comes with Flash Attention 2, Transformer Engine, and other useful packages preinstalled. No parallelism strategy is used.

For the 177M model, we use a batch size of 80 and a fixed sequence length of 1024 with no gradient accumulation. Our dataset class concatenates several sequences together until the desired length is reached: this is to take advantage of faster training when using static shapes. We use `torch.compile` on `max-autotune` mode, mixed precision with BF16 autocast, and Flash Attention 2 via the composable kernel backend. We did try FP8 autocast with the included Transformer Engine, but combining it with `torch.compile` made it error out, which is also why we didn't use gradient accumulation.

The official PyTorch ROCm documentation says TensorFloat32 is not supported[5], but this is false. We set the environment variable `HIPBLASLT_ALLOW_TF32=1` to enable TensorFloat32 for FP32 matmuls, just in case there were any.

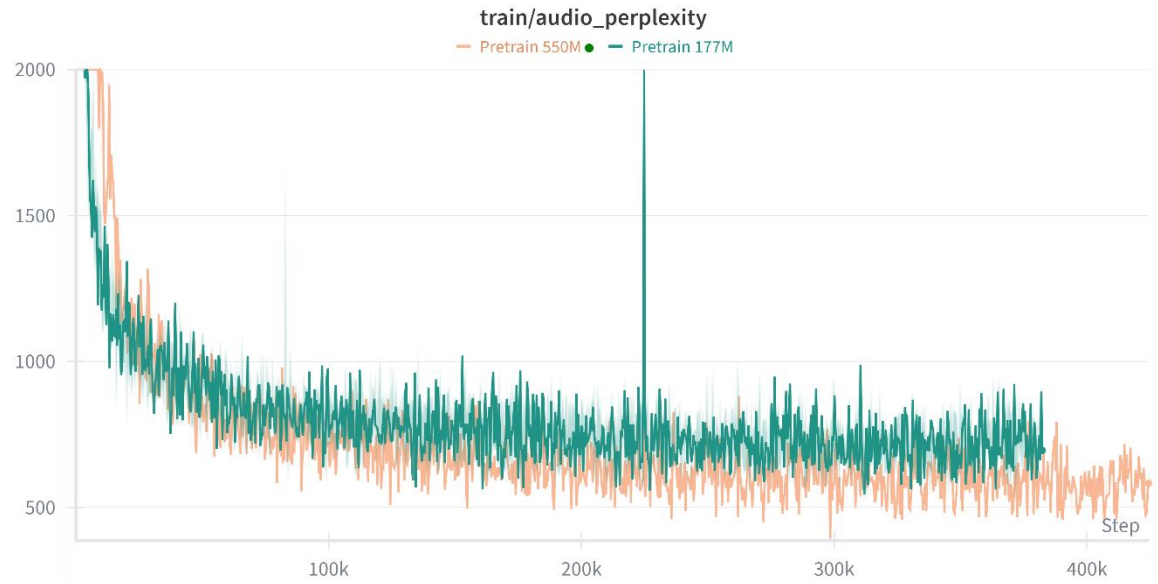
Our optimizer is AdamW; we hope to use Muon in the future as by observation it looks promising but given our constraints we have little room for experimentation: switching optimizers would demand a hyper parameter search. The learning rate is 2×10^{-4} and weight decay 1×10^{-2}

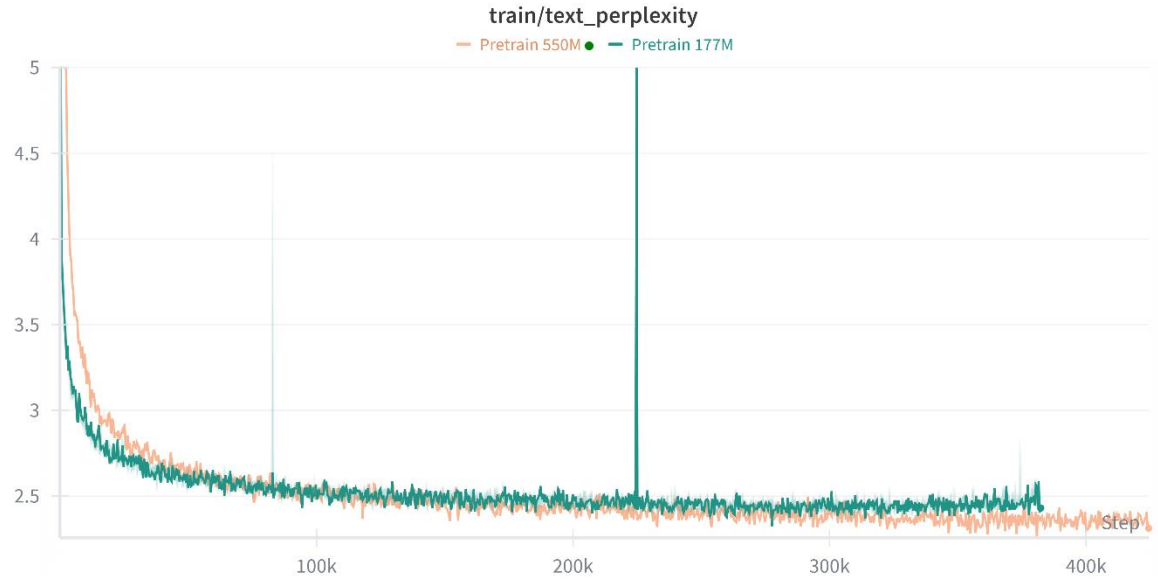
[5] HIP (ROCm) semantics — PyTorch 2.9 documentation

Pretraining the 177M parameter model took about two days uninterrupted; in both runs we do just one pass (epoch) over the whole dataset, as it is huge. The scheduler has a short learning rate warmup followed by a slow decay.

We report the training curves straight from our Weights and Biases dashboard – specifically, the perplexity. Perplexity measures how well the model predicts the next token: lower values indicate that the model is more certain about what comes next, while higher values mean it is more unsure. In other words, perplexity reflects how "surprised" the model is by the true data.

Note that the 550M model is not done training at the time of writing this—hence while we mention it here, we won't dwell on it.





In this case, the MI300X we are using is provided for free by Hot Aisle, but at the lowest available hourly market rate — that is, on-demand pricing *without* bulk discounts — for said hardware at the time of writing this: $1.99/GPU/hour from the same provider, and 60 hours (2.5 days) of training time, we arrive at a total cost of $119.4 for the 177M model run; although that figure only counts the pretraining run itself, and not hparam sweeps/ablations/bug fixes/et al.

Expressed more dramatically, pretraining this model **cost less than a mid-range smartphone**. And while still in progress, the 550M-parameter model is estimated to take 116 hours total, which is $231 (rounded up) of compute using the same formula.

PERPLEXITY ANALYSIS

The earlier section showed that there is a big difference between text and audio token perplexities, which we aim to explain here, in the interest of understanding our modality further.

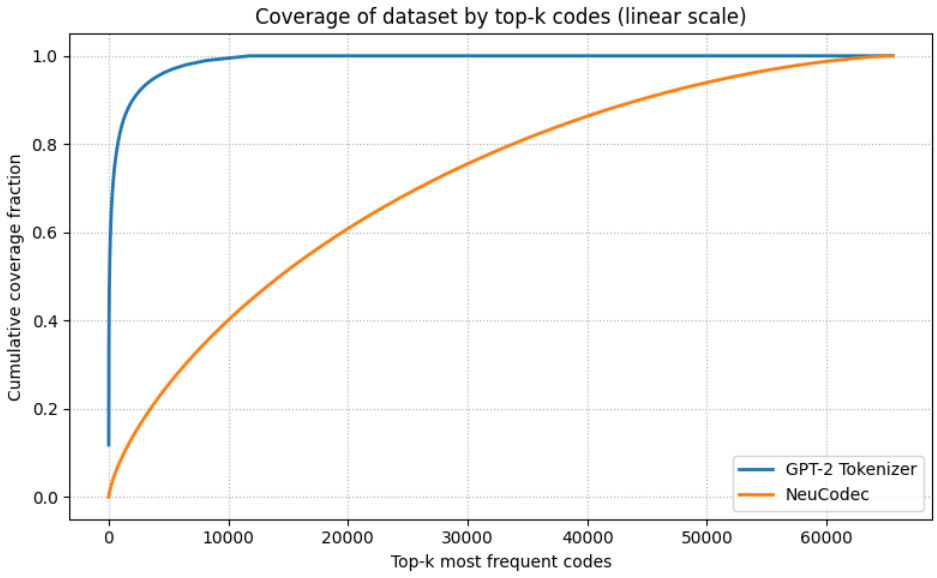
A naïve dismissal might attribute this to the difference between the text (character-level) and audio vocab sizes, of 42 (just forty-two, without a thousand) and 65k respectively, but this explanation would be incomplete at best, as perplexity values of low single digits are considered standard in text language models, whose byte pair encoding (BPE) tokenizers have vocabulary sizes well into the tens of thousands.

If we judged our values by the metrics of usual language models, they would be *disastrous*. Yet, our model can produce perfectly valid speech. Therefore, this begs the question: *why the disparity?*

To answer this question, we can first look at the NeuCodec paper. **Emphasis** is ours. Section 6 states:

> (…) Our analysis indicates that both (1) the encoding has baked-in redundancy and (2) codes that point to different local regions in the space index similar acoustic features. **Since FSQ encourages the encoder to distribute information across all codewords, as long as the codebook is large enough, redundancy becomes a feature of FSQ**, as a redundant representation will be created as information spreads into all codewords regardless of the actual dimensionality of the data. Additionally, **even when intentionally perturbing the code sequences, the reconstruction quality remains high compared to RVQ codecs**. With FSQ, perturbations in the code indices will result in predictable size changes in embedding space. In contrast, other methods of vector quantization impose no such constraints, hence perturbing their code indices can result in arbitrarily-sized changes in the embedding space. These aspects of FSQ result in a robust method of quantization with inherent redundancy and locality in representation space.

Indeed, to validate our theory further, we took a shard of our dataset's audio (8M audio tokens) and a small text corpus, which we then tokenized with the GPT-2 tokenizer using OpenAI's `tiktoken`, whose vocabulary size is of ~50k. Then, we plotted the coverage to see the distribution:



As the previous chart shows, the token distribution of NeuCodec is practically uniform in real-world data, while BPE tokens show a large skew.

Therefore, we conclude that the high perplexity found in audio is not because of poor performance, but **a natural consequence of the distribution of audio codes**, and is normal.

ZERO-SHOT FINETUNING

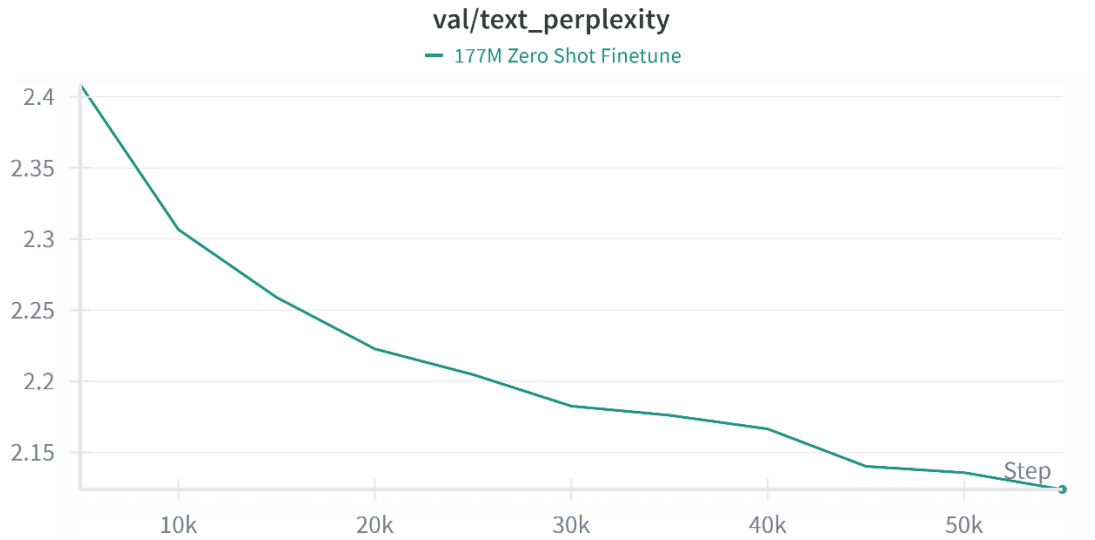
Now, our model was trained on a large-scale multi-speaker dataset, but without using any kind of speaker conditioning, making the voice at inference-time completely random, and a text to speech model that has no way to control the voice is of little to no use generally. Therefore, we had to add some form of speaker conditioning.

Once again considering our situation we went with a tried and proven approach[6]: utilizing a pretrained speaker embedder to provide features for the model to learn. Since said embedding model is trained to tell the difference between one speaker and another, its output contains information about them.

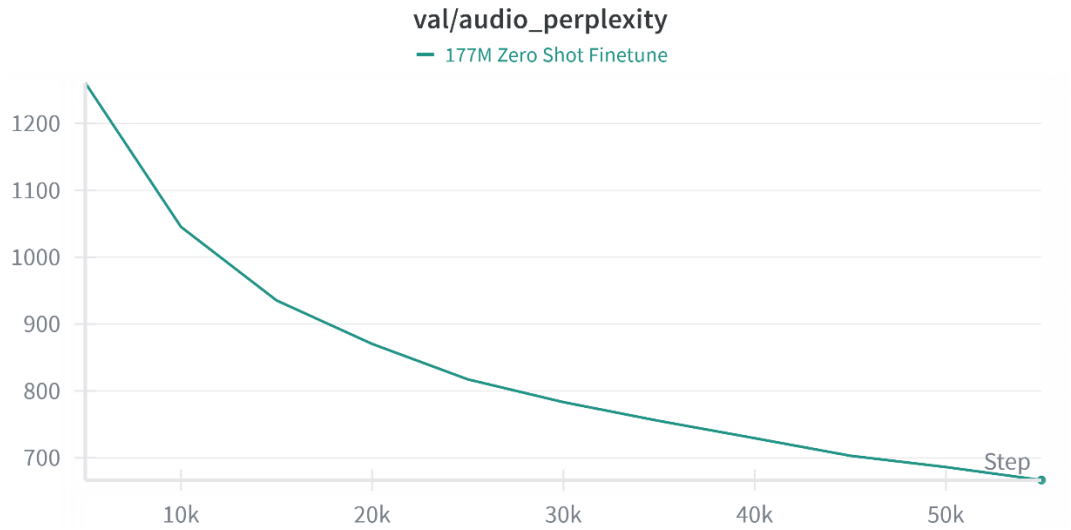
This time, we collected a dataset ourselves through an undisclosed process. About 6500 hours of this dataset were then converted from audio to tokens with NeuCodec-distil (the pretraining dataset was already tokenized by Neuphonic themselves[7]), embedded, and we finetuned the base model on this dataset.

To inject the speaker information, we borrow from vision transformers and replace every layer normalization layer in the transformer with Adaptive Layer Normalization (AdaLN)[8], which takes the normalized speaker embedding and conditions the model at every level.

We initialize all weights from the 177M base model, and unlike pretraining, our dataset yields batches where each item is a single sequence, padded to the maximum length in the batch; this is to use masking and maximize our ability to learn from limited data. Our learning rate is 5×10^{-5}, notably lower than pretraining as we don't want to risk catastrophic forgetting, and batch size 32. Now, we report validation loss.



[6] ZSE-VITS: A Zero-Shot Expressive Voice Cloning Method Based on VITS
[7] neuphonic/emilia-yodas-english-neucodec · Datasets at Hugging Face
[8] AdaLN: A Vision Transformer for Multidomain Learning and Predisaster Building Information Extraction from Images



We only do one pass over the finetuning dataset, as it is big, for a total of 55k steps over 4.5 hours. If we paid for the compute, repeating the earlier formula, that would mean this variant cost **$9** on top of pretraining – *less than a single Chipotle burrito*.[9]

After training, this model can clone the voice of any speaker and speak like them – albeit not quite to our standards, as the ECAPA-TDNN speaker encoder we used to condition the model isn't sensitive enough to capture a lot of characteristics very closely, but that can be iterated on later.

CONCLUSION & TOWARDS THE FUTURE

We have presented our model, which was trained from scratch then finetuned on a minimal budget, by a lone engineer, while having provided explanations for our major design decisions. Our codebase is self-contained and hackable.

However, this is merely the beginning. We hope to achieve ONNX export, scaling up, fine-tuning on individual voices with LoRA, and to explore architectural innovations such as Macaron-Net or state-space models like RWKV, Mamba, *et al.*

For now, we have proven what we set out to prove.

We would like to thank Hot Aisle for the free compute.

[9] Chipotle Mexican Grill