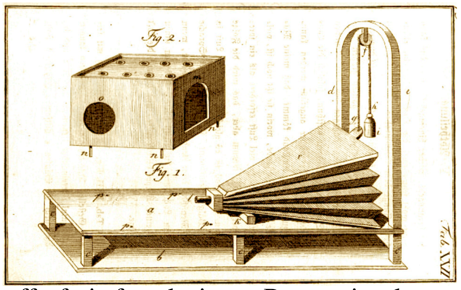
CHAPTER

16 Text-to-Speech

"Words mean more than what is set down on paper. It takes the human voice to infuse them with shades of deeper meaning."
Maya Angelou, *I Know Why the Caged Bird Sings*

The task of mapping from text to speech is a task with an even longer history than speech to text. In Vienna in 1769, Wolfgang von Kempelen built for the Empress Maria Theresa the famous Mechanical Turk, a chess-playing automaton consisting of a wooden box filled with gears, behind which sat a robot mannequin who played chess by moving pieces with his mechanical arm. The Turk toured Europe and the Americas for decades, defeating Napoleon Bonaparte and even playing Charles Babbage. The Mechanical Turk might have been one of the early successes of artificial intelligence were it not for the fact that it was, alas, a hoax, powered by a human chess player hidden inside the box.

What is less well known is that von Kempelen, an extraordinarily prolific inventor, also built between 1769 and 1790 what was definitely not a hoax: the first full-sentence speech synthesizer, shown partially to the right. His device consisted of a bellows to simulate the lungs, a rubber mouthpiece and a nose aperture, a reed to simulate the vocal folds, various whistles for the fricatives, and a small auxiliary bellows to provide the puff of air for plosives. By moving levers with both hands to open and close apertures, and adjusting the flexible leather "vocal tract", an operator could produce different consonants and vowels.



More than two centuries later, we no longer build our synthesizers out of wood and leather, nor do we need human operators. The modern task of **text-to-speech** or **TTS**, also called **speech synthesis**, is exactly the reverse of ASR; to map text:

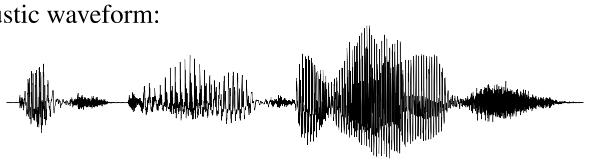
text-to-speech

TTS

speech synthesis

```
It's time for lunch!
```

to an acoustic waveform:



TTS has a wide variety of applications. It is used in spoken language models that interact with people, for reading text out loud, for games, and to produce speech for sufferers of neurological disorders, like the late astrophysicist Steven Hawking after he lost the use of his voice because of ALS.

In this chapter we introduce an algorithm for TTS that, like the ASR algorithms of the prior chapter, are trained on enormous amounts of speech datasets. We'll also briefly touch on other speech applications.

16.1 TTS overview

The task of text-to-speech is to generate a speech waveform that corresponds to a desired text, using in a particular voice specified by the user.

Historically TTS was done by collecting hundreds of hours of speech from a single talker in a lab and training a large system on it. The resulting TTS system only worked in one voice; if you wanted a second voice, you went back and collected data from a second talker.

The modern method is instead to train a speaker-independent synthesizer on tens of thousands of hours of speech from thousands of talkers. To create speech in a new voice unseen in training, we use a very small amount of speech from the desired talker to guide the creation of the voice. So the input to a modern TTS system is a text prompt and perhaps 3 seconds of speech from the voice we'd like to generate the speech in. This TTS task is called **zero-shot TTS** because the desired voice may never have been seen in training.

zero-shot TTS

The way modern TTS systems address this task is to use language modeling, and in particular conditional generation. The intuition is to take an enormous dataset of speech, and use an **audio tokenizer** based on an **audio codec** to induce discrete audio tokens from that speech that represent the speech. Then we can train a language model whose vocabulary includes both speech tokens and text tokens.

We train this language model to take as input two sequences, a text transcript and a small sample of speech from the desired talker, to tokenize both the text and the speech into discrete tokens, and then to conditionally generate discrete samples of the speech corresponding to the text string, in the desired voice.

At inference time we prompt this language model with a tokenized text string and a sample of the desired voice (tokenized by the codec into discrete audio tokens) and conditionally generate to produce the desired audio tokens. Then these tokens can be converted into a waveform.

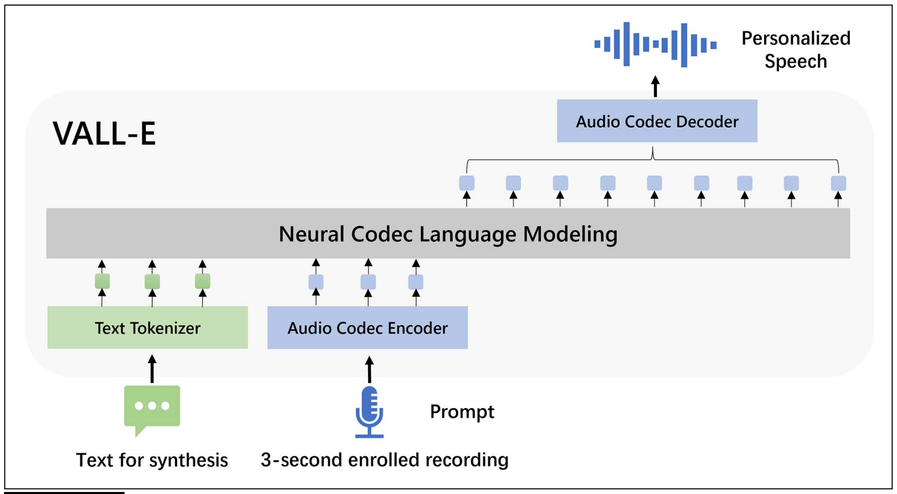


Figure 16.1 VALL-E architecture for personalized TTS (figure from Chen et al. (2025)).

Fig. 16.1 from Chen et al. (2025) shows the intuition for one such TTS system, called **VALL-E**. VALL-E is trained on 60K hours of English speech, from over 7000 unique talkers. Systems like VALL-E have 2 components:

VALL-E

1. The **audio tokenizer**, generally based on an **audio codec**, a system we'll de-

scribe in the next section. Codecs have three parts: an encoder (that turns speech into embedding vectors), a quantizer (that turns the embeddings into discrete tokens) and decoders (that turns the discrete tokens back into speech).

2. The 2-stage **conditional language model** that can generate audio tokens corresponding to the desired text. We'll sketch this in Section 16.3.

16.2 Using a codec to learn discrete audio tokens

Modern TTS systems are based around converting the waveform into a sequence of discrete audio tokens. This idea of manipulating discrete audio tokens is also useful for other speech-enabled systems like **spoken language models**, which take text or speech input and can generate text or speech output to solve tasks like speech-to-speech translation, diarization, or spoken question answering. Having discrete tokens means that we can make use of language model technology, since language models are specialized for sequences of discrete tokens. Audio tokenizers are thus an important component of the modern speech toolkit.

codec The standard way to learn audio tokens is from a neural **audio codec** (the word is formed from **coder/decoder**). Historically a codec was a hardware device that digitized analog symbols. More generally we use the word to mean a mechanism for encoding analog speech signals into a digitized compressed representation that can be efficiently stored and sent. Codecs are still used for compression, but for TTS and also for spoken language models, we employ them for converting speech into discrete tokens.

Of course the digital representation of speech we described in Chapter 14 is already discrete. For example 16 kHz speech stored in 16-bit format could be thought of as a series of 2^{16} = 65,536 symbols, with 16,000 of those symbols per second of speech. But a system that generates 16,000 symbols per second makes the speech signal too long to be feasibly processed by a language model, especially one based on transformers with their inefficient quadratic attention. Instead we want symbols that represent longer chunks of speech, perhaps something on the order of a few hundred tokens a second.

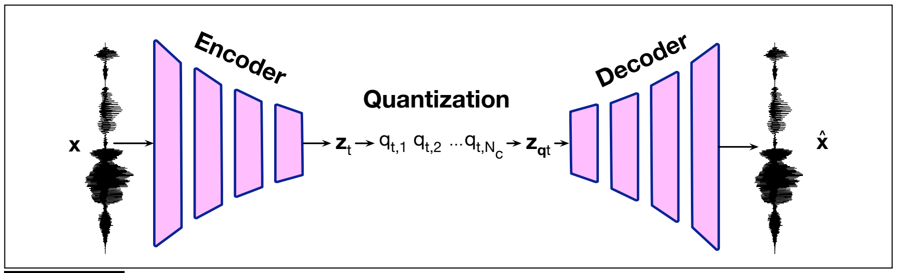


Figure 16.2 Standard architecture of an audio tokenizer performing inference, figure adapted from Mousavi et al. (2025). An input waveform $\mathbf{x}$ is encoded (generally using a series of downsampling convolution networks) into a series of embeddings $\mathbf{z_t}$. Each embedding is then passed through a quantizer to produce a series of quantized tokens q_t. To regenerate the speech signal, the quantized tokens are re-mapped back to a vector $\mathbf{z}_{\mathbf{q}_t}$ and then encoded (usually using a series of upsampling convolution networks) back to a waveform. We'll discuss how the architecture is trained in Section 16.2.4.

Fig. 16.2 adapted from Mousavi et al. (2025). shows the standard architecture of an audio tokenizer. Audio tokenizers take as input an audio waveform, and are trained to recreate the same audio waveform out, via an intermediate representation consisting of discrete tokens created by vector quantization.

Audio tokenizers have three stages:

1. an **encoder** maps the acoustic waveform, a series of T values $\mathbf{x} = x_1, x_2, ..., x_T$, to a sequence of τ embeddings $\mathbf{z} = \mathbf{z_1}, \mathbf{z_2}, ..., \mathbf{z}_\tau$. τ is typically 100-1000 times smaller than T.
2. a **vector quantizer** that takes each embedding $\mathbf{z_t}$ corresponding to part of the waveform, and represents it by a sequence of discrete tokens each taken from one of the N_c codebooks, $q_t = q_{t,1}, q_{t,2}, ..., q_{t,N_c}$. The vector quantizer also sums the vector codewords from each codebook to create a quantizer output vector $\mathbf{z_{q_t}}$.
3. a **decoder** that generates a lossy reconstructed waveform span $\hat{\mathbf{x}}$ from the quantizer output vector $\mathbf{z_{q_t}}$.

Audio tokenizers are generally learned end-to-end, using loss functions that reward a tokenization that allows the system to reconstruct the input waveform.

In the following subsections we'll go through the components of one particular tokenizer, the ENCODEC tokenizer of Défossez et al. (2023).

16.2.1 The Encoder and Decoder for the ENCODEC model

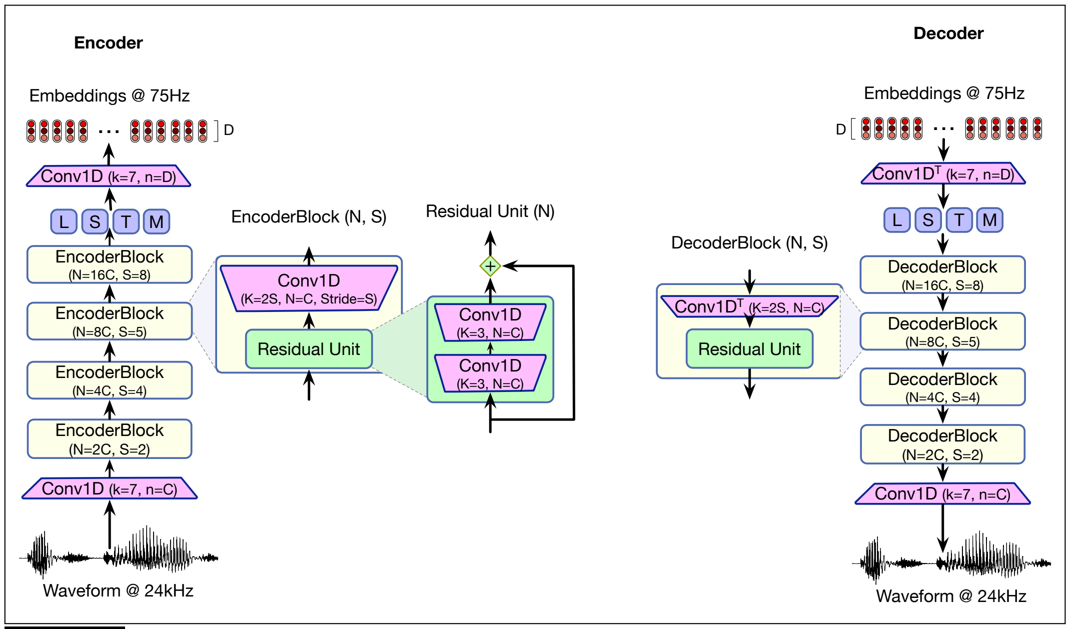


Figure 16.3 The encoder and decoder stages of the ENCODEC model. The goal of the encoder is to downsample an input waveform by encoded it as a series of embeddings $\mathbf{z}_t$ at 75Hz, i.e. 75 embeddings a second. Because the original signal was represented at 24kHz, this is a downsampling of $\frac{24000}{75} = 320$ times. Between the encoder and decoder is a quantization step producing a lossy embedding $\mathbf{z_{q_t}}$. The goal of the decoder is to take the lossy embedding $\mathbf{z}_q t$ and upsample it, converting it back to a waveform.

The encoder and decoder of the ENCODEC model (Défossez et al., 2023) are sketched in Fig. 16.3. The goal of the encoder is to downsample a span of waveform at time t, which is at 24kHz—one second of speech has 24,000 real values—to an embedding representation $\mathbf{z}_t$ at 75Hz—one second of audio is represented by 75 vectors, each of dimensionality D. For the purposes of this explanation, we'll use $D = 256$.

This downsampling is accomplished by having a series of encoder blocks that are made up of convolutional layers with strides larger than 1 that iteratively downsample the audio, as we discussed at the end of Section **??**. The convolution blocks are sketched in Fig. 16.3, and include a long series of convolutions as well as residual units that add a convolution to the prior input.

The output of the encoder is an embedding $\mathbf{z}_t$ at time t, 75 of which are produced per second. This embedding is then quantized (as discussed in the next section), turning each embedding $\mathbf{z}_t$ into a series of N_c discrete symbols $q_t = q_{t,1}, q_{t,2}, \ldots, q_{t,N_c}$, and also turning the series of symbols into a new quantizer output vector $\mathbf{z}_{\mathbf{q}_t}$. Finally, the decoder takes the output embedding from the quantizer $\mathbf{z}_{\mathbf{q}_t}$ and generates a waveform via a symmetric set of convnets that upsample the audio.

In summary, a 24kHz waveform comes through, we encode/downsample it into a vector $\mathbf{z}_t$ of dimensionality $D = 256$, quantize it into discrete symbols q_t, turn it back into a vector $\mathbf{z}_{\mathbf{q}_t}$ of dimensionality $D = 256$, and then decode/upsample that vector back into a waveform at 24kHz.

16.2.2 Vector Quantization

vector quantization
VQ

The goal of the **vector quantization** or **VQ** step is to turn a series of vectors into a series of discrete symbols.

Historically vector quantization (Gray, 1984) was used to compress a speech signal, to reduce the bit rate for transmission or storage. To compress a sequence of vector representations of speech, we turn each vector into an integer, an index representing a class or cluster. Then instead of transmitting a big vector of floating point numbers, we transmit that integer index. At the other end of the transmission, we reconstitute the vector from the index.

For TTS and other modern speech applications we use vector quantization for a different reason: because VQ conveniently creates discrete tokens, and those fit well into the language modeling paradigm, since language models do well at predicting sequences of discrete tokens.

In practice for the ENCODEC model and other audio tokenizers, we use a powerful from of vector quantization called **residual vector quantization** that we'll define in the following section. But it will be helpful to first see the basic VQ algorithm before we extend it.

Vector quantization has a training phase and an inference phase. We already introduced the core of the basic VQ training algorithm when we described k-means clustering of vectors in Section **??**, since k-means clustering is the most common algorithm used to implement VQ. To review, in VQ training, we run a big set of speech wavefiles through an encoder to generate N vectors, each one corresponding to some frame of speech. Then we cluster all these N vectors into k clusters; k is set by the designer as a parameter to the algorithm as the number of discrete symbols we want, generally with $k << N$. In the simplest VQ algorithm, we use the iterative k-means algorithm to learn the clusters. Recall from Section **??** that k-means is a two-step algorithm based on iteratively updating a set of k **centroid** vectors. A **centroid** is the geometric center of a set of a points in n-dimensional space.

centroid

The k-means algorithm for clustering starts by assigning a random vector to each cluster k. Then there are two iterative steps. In the **assignment** step, given a set of k current centroids and the entire dataset of vectors, each vector is assigned to the cluster whose codeword is the closest (by squared Euclidean distance). In the **re-**

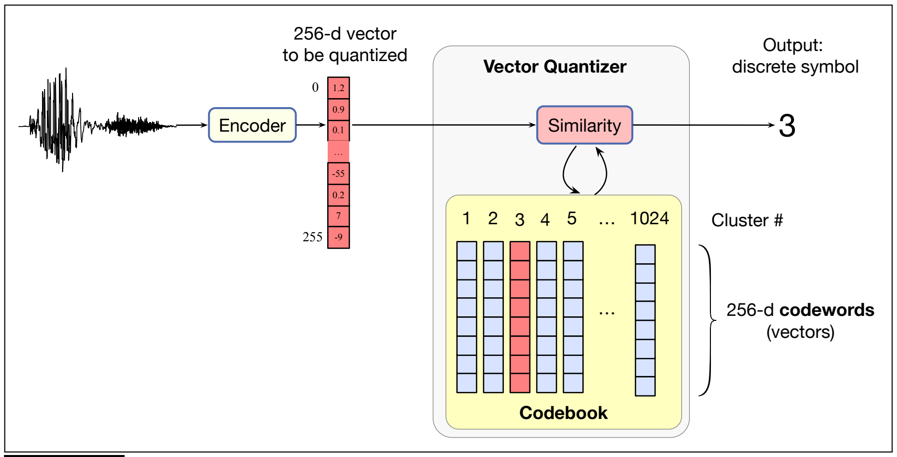


Figure 16.4 The basic VQ algorithm at inference time, after the codebook has been learned. The input is a span of speech encoded by the encoder into a vector of dimensionality $D = 256$. This vector is compared with each codeword (cluster centroid) in the codebook. The codeword for cluster 3 is most similar, so the VQ outputs 3 as the discrete representation of this vector.

estimation step, the codeword for each cluster is recomputed by recalculating a new mean vector. The result is that the clusters and their centroids slowly adjust to the training space. We iterate back and forth between these two steps until the algorithm converges.

VQ can also be used as part of end-to-end training, as we will discuss below, in which case instead of iterative k-means, we instead recompute the means during minibatch training via online algorithms like **exponential moving averages**.

codeword

codebook

code

At the end of clustering, the cluster index can be used as a discrete symbol. Each cluster is also associated with a **codeword**, the vector which is the centroid of all the vectors in the cluster. We call the list of cluster ids (tokens) together with their codeword the **codebook**, and we often call the cluster id the **code**.

In inference, when a new vector comes in, we compare it to each vector in the codebook. Whichever codeword is closest, we assign it to that codeword's associated cluster. Fig. 16.4 shows an intuition of this inference step in the context of speech encoding:

1. an input speech waveform is encoded into a vector **v**,
2. this input vector **v** is compared to each of the 1024 possible codewords in the codebook,
3. **v** is found to be most similar to codeword 3,
4. and so the output of VQ is the discrete symbol 3 as a representation of **v**.

As we will see below, for training the ENCODEC model end-to-end we will need a way to turn this discrete symbol back into a waveform. For simple VQ we do that by directly using the codeword for that cluster, passing that codeword to the decoder for it to reconstruct the waveform. Of course the codeword vector won't exactly match the original vector encoding of the input speech span, especially with only 1024 possible codewords, but the hope is that it's at least close if our codebook is good, and the decoder will still produce reasonable speech. Nonetheless, more powerful methods are usually used, as we'll see in the next section.

16.2.3 Residual Vector Quantization

In practice, simple VQ doesn't produce good enough reconstructions, at least not with codebook sizes of 1024. 1024 codeword vectors just isn't enough to represent the wide variety of embeddings we get from encoding all possible speech waveforms. So what the ENCODEC model (and many other audio tokenization methods) use instead is a more sophisticated variant called **residual vector quantization**, or **RVQ**. In residual vector quantization, we use multiple codebooks arranged in a kind of hierarchy.

residual vector quantization
RVQ

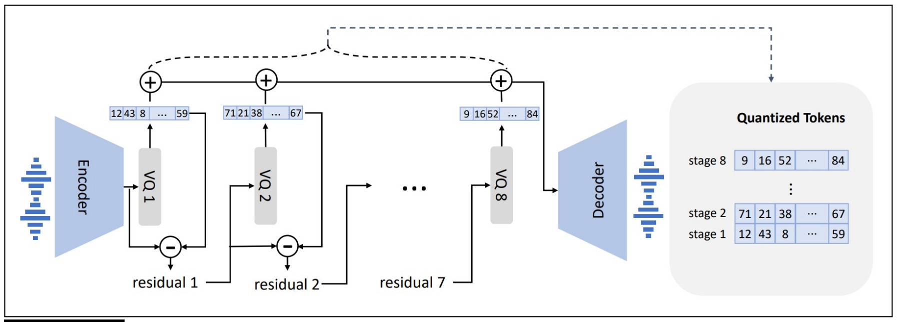


Figure 16.5 Residual VQ (figure from Chen et al. (2025)). We run VQ on the encoder output embedding to produce a discrete symbol and the corresponding codeword. We then look at the **residual**, the difference between the encoder output embedding z_t and the codeword chosen by VQ. We then take a second codebook and run VQ on this residual. We repeat the process until we have 8 tokens.

The idea is very simple. We run standard VQ with a codebook just as in Fig. 16.4 in the prior section. Then for an input embedding $\mathbf{z}_t$ we take the codeword vector that is produced, let's call it $\mathbf{z}_{\mathbf{q1}_t}$ for the $\mathbf{z}_t$ as quantified by codebook 1, and take the difference between the two:

$$\text{residual} = \mathbf{z}_t - \mathbf{z}_{\mathbf{q}\,t}^{(1)}. \tag{16.1}$$

residual

This **residual** is the error in the VQ; the part of the original vector that the VQ didn't capture. The residual is kind of a rounding error; it's as if in VQ we 'round' the vector to the nearest codeword, and that creates some error. So we then take that residual vector and pass it through another vector quantizer! That gives us a second codeword that represents the residual part of the vector. We then take the residual from the second codeword, and do this again. The total result is 8 codewords (the original codeword and the 7 residuals).

That means for RVQ we represent the original speech span by a sequence of 8 discrete symbols (instead of 1 discrete symbol in basic VQ). Fig. 16.5 shows the intuition.

What do we do when we want to reconstruct the speech? The method used in ENCODEC RVQ is again simple: we take the 8 codewords and add them together! The resulting vector $\mathbf{z}_{\mathbf{q}_t}$ is then passed through the decoder to generate a waveform.

.

16.2.4 Training the ENCODEC model of audio tokens

The ENCODEC model (like similar audio tokenizer models) is trained end to end. The input is a waveform, a span of speech of perhaps 1 or 10 seconds extracted from a longer original waveform. The desired output is the same waveform span, since the model is a kind of autoencoder that learns to map to itself. The model is trained to do this reconstruction on large speech datasets like Common Voice (Ardila et al., 2020) (over 30,000 hours of speech in 133 languages) as well as other audio data like Audio Set (Gemmeke et al., 2017) (1.7 million 10 sec excerpts from YouTube videos labeled from a large ontology including natural, animal, and machine sounds, music, and so on).

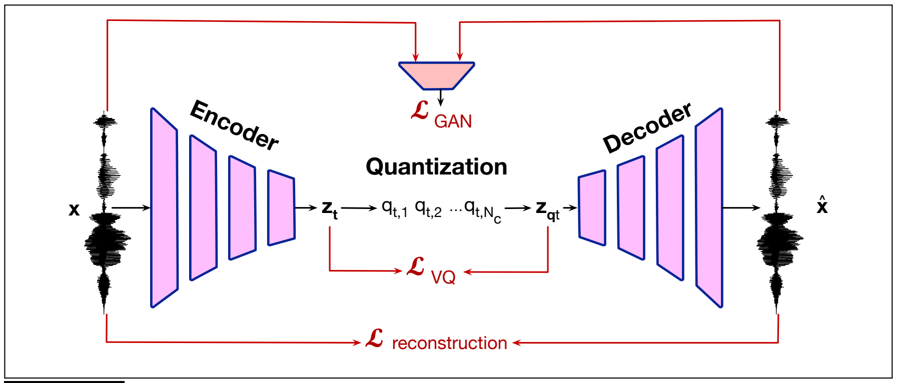

Encoder
Decoder
Quantization
$\mathcal{L}_{\text{GAN}}$
x
$\mathbf{z_t}$
$q_{t,1}$ $q_{t,2}$ ... q_{t,N_c}
$\mathbf{z_{qt}}$
$\hat{\mathbf{x}}$
$\mathcal{L}_{\text{VQ}}$
$\mathcal{L}_{\text{reconstruction}}$

Figure 16.6 Architecture of audio tokenizer training, figure adapted from Mousavi et al. (2025). The audio tokenizer is trained with a weighted combination of various loss functions, summarized in the figure and described below.

reconstruction loss

The ENCODEC model, like most audio tokenizers, is trained with a number of loss functions, as suggested in Fig. 16.6. The **reconstruction loss** $L_{\text{reconstruction}}$ measures how similar the output waveform is to the input waveform, for example by the sum-squared difference between the original and reconstructed audio:

$$L_{\text{reconstruction}}(\mathbf{x}, \hat{\mathbf{x}}) = \sum_{t=1}^{T} ||x_t - \hat{x}_t||^2 \tag{16.2}$$

Similarity can additionally be measured in the frequency domain, by comparing the original and reconstructed mel-spectrogram, again using sum-squared (L2) distance or L1 distance or some combination.

adversarial loss

Another kind of loss is the **adversarial loss** L_{GAN}. For this loss we train a generative adversarial network, a generator and a binary discriminator D, which is a classifier to distinguish between the true wavefile $\mathbf{x}$ and a generated one. We want to train the model to fool this discriminator, so the better the discriminator, the worse our reconstruction must be, and so we use the discriminator's success as a loss function, We can also incorporate various features from the generator.

Finally, we need a loss for the quantizer. This is because having a quantizer in the middle of end-to-end training causes problems in propagation of the gradient in the backward pass of training, because the quantization step is not differentiable. We deal with this problem in two ways. First, we ignore the quantization step in the backward pass. Instead we copy the gradients from the output of the quantizer ($\mathbf{z}_{\mathbf{q}_t}$) back to the input of the quantizer ($\mathbf{z}_t$), a method called the **straight-through estimator** (Van Den Oord et al., 2017).

But then we need a method to make sure the code words in the vector quantizer step get updated during training. One method is to start these off using k-means clustering of the vectors $\mathbf{z_t}$ to get an initial clustering. Then we can add to a loss component, L_{VQ}, which will be a function of the difference between the encoder output vector $\mathbf{z}_t$ and the reconstructed vector after the quantization $\mathbf{z}_{\mathbf{q}_t}$, i.e. the codeword, summed over all the N_c codebooks and residuals.

$$L_{\text{VQ}}(\mathbf{x},\hat{\mathbf{x}}) = \sum_{t=1}^{T}\sum_{c=1}^{N_c} ||\mathbf{z}^{(\mathbf{c})}{}_t - \mathbf{z}_{\mathbf{q}}^{(\mathbf{c})}{}_t|| \tag{16.3}$$

The total loss function can then just be a weighted sum of these losses:

$$L(\mathbf{x},\hat{\mathbf{x}}) = \lambda_1 L_{\text{reconstruction}}(\mathbf{x},\hat{\mathbf{x}}) + \lambda_2 L_{\text{GAN}}(\mathbf{x},\hat{\mathbf{x}}) + \lambda_3 L_{\text{VQ}}(\mathbf{x},\hat{\mathbf{x}}) \tag{16.4}$$

16.3 VALL-E: Generating audio with 2-stage LM

As we summarized in the introduction, the structure of TTS systems like VALL-E is to take as input a text to be synthesized and a sample of the voice to be used, and tokenize both, using BPE for the text and an audio codec for the speech. We then use a language model to conditionally generate discrete audio tokens corresponding to the text prompt, in the voice of the speech sample.

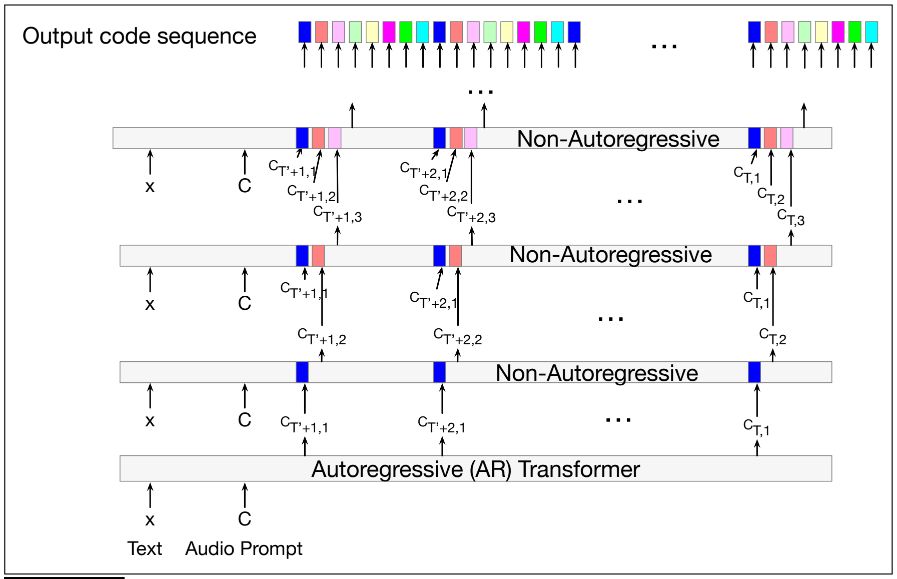


Figure 16.7 The 2-stage language modelling approach for VALL-E, showing the inference stage for the autoregressive transformer and the first 3 of the 7 non-autoregressive transformers. The output sequence of discrete audio codes is generated in two stages. First the autoregressive LM generates all the codes for the first quantizer from left to right. Then the non-autoregressive model is called 7 times to generate the remaining codes conditioned on all the codes from the preceding quantizer, including conditioning on the codes to the right.

Instead of doing this conditional generation with a single autoregressive language model, VALL-E does the conditional generation in a 2-stage process, using two distinct language models. This architectural choice is influenced by the hierarchical nature of the RVQ quantizer that generates the audio tokens. The output of the first RVQ quantizer is the most important token to the final speech, while the subsequent quantizers contribute less and less residual information to the final signal. So the language model generates the acoustic codes in two stages. First, an autoregressive LM generates the first-quantizer codes for the entire output sequence, given the input text and enrolled audio. Then given those codes, a non-autoregressive LM is run 7 times, each time taking as input the output of the initial autoregressive codes and the prior non-autoregressive quantizer and thus generating the codes from the remaining quantizers one by one. Fig. 16.7 shows the intuition for the inference step.

Now let's see the architecture in a bit more detail. For **training**, we are given an audio sample **y** and its tokenized text transcription $\mathbf{x} = [x_0, x_1, \ldots, x_L]$. We use a pretrained ENCODEC to convert **y** into a code matrix **C**. Let T be the number of downsampled vectors output by ENCODEC, with 8 codes per vector. Then we can represent the encoder output as

$$\mathbf{C}^{T\times 8} = \text{ENCODEC}(\mathbf{y}) \tag{16.5}$$

Here **C** is a two-dimensional acoustic code matrix that has $T \times 8$ entries, where the columns represent time and the rows represent different quantizers. That is, the row vector $\mathbf{c}_{t,:}$ of the matrix contains the 8 codes for the t-th frame, and the column vector $\mathbf{c}_{:,j}$ contains the code sequence from the j-th vector quantizer where $j \in [1, ..., 8]$.

Given the text **x** and audio **C**, we train the TTS as a conditional code language model to maximize the likelihood of **C** conditioned on **x**:

$$\begin{aligned} L &= -\log p(\mathbf{C}|\mathbf{x}) \\ &= -\log \prod_{t=0}^{T} p(\mathbf{c}_{<t,:}, \mathbf{x}) \end{aligned} \tag{16.6}$$

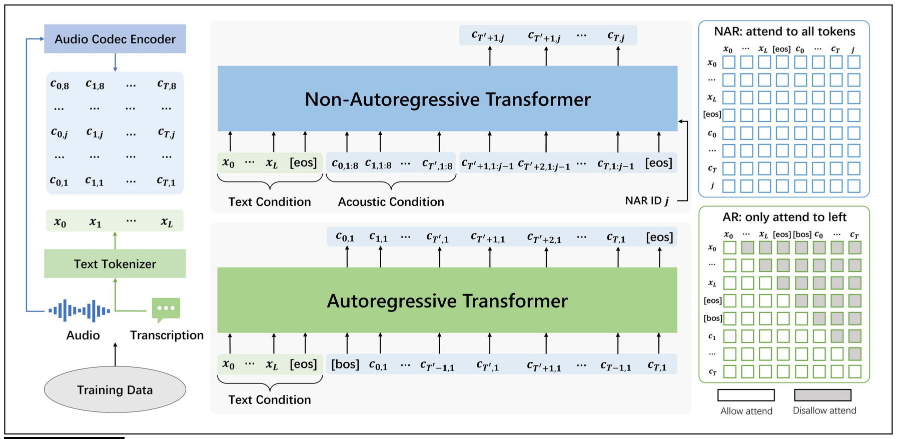

Audio Codec Encoder
$c_{0,8}$ $c_{1,8}$ ... $c_{T,8}$
$c_{0,j}$ $c_{1,j}$ ... $c_{T,j}$
$c_{0,1}$ $c_{1,1}$ ... $c_{T,1}$
x_0 x_1 ... x_L
Text Tokenizer
Audio
Transcription
Training Data
$c_{T'+1,j}$ $c_{T'+1,j}$ ... $c_{T,j}$
Non-Autoregressive Transformer
x_0 ... x_L [eos] $c_{0,1:8}$ $c_{1,1:8}$ ... $c_{T',1:8}$ $c_{T'+1,1:j-1}$ $c_{T'+2,1:j-1}$ ... $c_{T,1:j-1}$ [eos]
Text Condition
Acoustic Condition
NAR ID j
$c_{0,1}$ $c_{1,1}$ ... $c_{T',1}$ $c_{T'+1,1}$ $c_{T'+2,1}$ ... $c_{T,1}$ [eos]
Autoregressive Transformer
x_0 ... x_L [eos] [bos] $c_{0,1}$ ... $c_{T'-1,1}$ $c_{T',1}$ $c_{T'+1,1}$ ... $c_{T-1,1}$ $c_{T,1}$
Text Condition
NAR: attend to all tokens
AR: only attend to left
Allow attend
Disallow attend

Figure 16.8 Training procedure for VALL-E. Given the text prompt, the autoregressive transformer is first trained to generate each code of the first-quantizer code sequence, autoregressively The the non-autoregressive transformer generates the rest of the codes. Figure from Chen et al. (2025).

Fig. 16.8 shows the intuition. On the left, we have an audio sample and its transcription, and both are tokenized. Then we append an [EOS] and [BOS] token to **x** and an [EOS] token to the end of **C** and train the autoregressive transformer to predict the acoustic tokens, starting with $\mathbf{c}_{0,1}$, until [EOS], and then the non-autoregressive transformers to fill in the other tokens.

During **inference**, we are given a text sequence to be spoken as well as $\mathbf{y}'$, an enrolled speech sample from some unseen speaker, for which we have the transcription transcript($\mathbf{y}'$). We first run the codec to get an acoustic code matrix for $\mathbf{y}'$, which will be $\mathbf{C}^P = \mathbf{C}_{:T',:} = [\mathbf{c}_{0,:}, \mathbf{c}_{1,:}, \ldots \mathbf{c}_{T',:} :]$. Next we concatenate the transcription of $\mathbf{y}'$ to the text sequence to be spoken to create the total input text $\mathbf{x}$, which we pass through a text tokenizer. At this stage we thus have a tokenized text $\mathbf{x}$ and a tokenized audio prompt $\mathbf{C}^P$.

Then we generate $\mathbf{C}^T = \mathbf{C}_{>T',:} = [\mathbf{c}_{T'+1,:}, \ldots \mathbf{c}_{T,:}]$ conditioned on the text sequence $\mathbf{x}$ and the prompt $\mathbf{C}^P$:

$$\begin{aligned}\mathbf{C}^T &= \underset{\mathbf{C}^T}{\operatorname{argmax}}\, p(\mathbf{C}^T|\mathbf{C}^P, \mathbf{x}) \\ &= \underset{\mathbf{C}^T}{\operatorname{argmax}} \prod_{t=T'+1}^{T} p(\mathbf{c}_{t,:}|\mathbf{c}_{<t,:}, \mathbf{x}) \qquad (16.7)\end{aligned}$$

Then the generated tokens $\mathbf{C}^T$ can be converted by the ENCODEC decoder into a waveform. Fig. 16.9 shows the intuition.

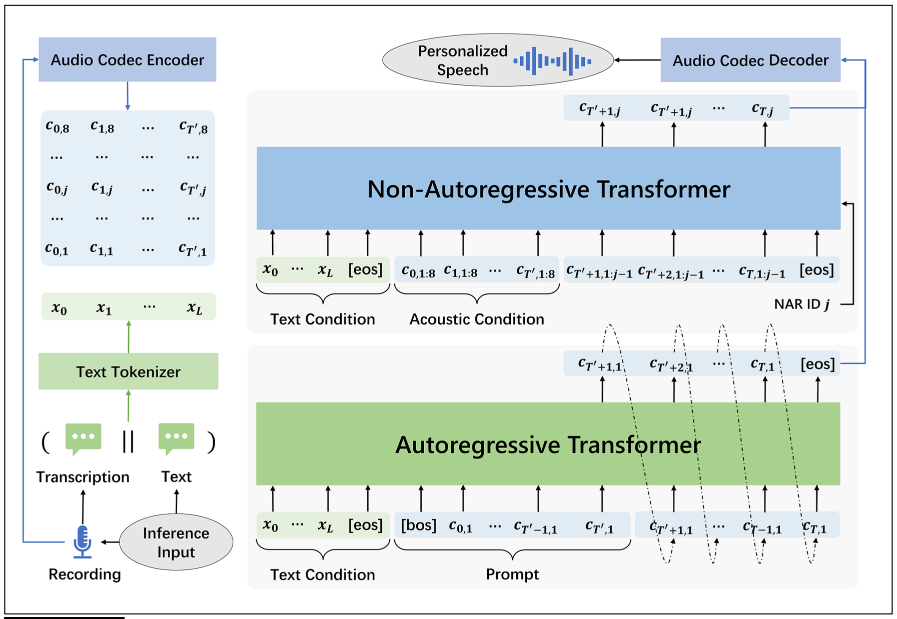

Audio Codec Encoder
Personalized Speech
Audio Codec Decoder
$c_{0,8}$ $c_{1,8}$ ··· $c_{T',8}$
$c_{0,j}$ $c_{1,j}$ ··· $c_{T',j}$
$c_{0,1}$ $c_{1,1}$ ··· $c_{T',1}$
$c_{T'+1,j}$ $c_{T'+1,j}$ ··· $c_{T,j}$
Non-Autoregressive Transformer
x_0 ··· x_L [eos] $c_{0,1:8}$ $c_{1,1:8}$ ··· $c_{T',1:8}$ $c_{T'+1,1:j-1}$ $c_{T'+2,1:j-1}$ ··· $c_{T,1:j-1}$ [eos]
Text Condition
Acoustic Condition
NAR ID j
x_0 x_1 ··· x_L
Text Tokenizer
(||)
Transcription
Text
$c_{T'+1,1}$ $c_{T'+2,1}$ ··· $c_{T,1}$ [eos]
Autoregressive Transformer
x_0 ··· x_L [eos] [bos] $c_{0,1}$ ··· $c_{T'-1,1}$ $c_{T',1}$ $c_{T'+1,1}$ ··· $c_{T-1,1}$ $c_{T,1}$
Inference Input
Recording
Text Condition
Prompt

Figure 16.9 Inference procedure for VALL-E. Figure from Chen et al. (2025). The transcript for the 3 seconds of enrolled speech is first prepended to the text to be generated, and both the speech and text are tokenized. Next the autoregressive transformer starts generating the first codes $\mathbf{c}_{t'+1,1}$ conditioned on the transcript and acoustic prompt.

See Chen et al. (2025) for more details on the transformer components and other details of training.

16.4 TTS Evaluation

MOS TTS systems are evaluated by humans, by playing an utterance to listeners and asking them to give a **mean opinion score** (**MOS**), a rating of how good the synthesized utterances are, usually on a scale from 1–5. We can then compare systems by comparing their MOS scores on the same sentences (using, e.g., paired t-tests to test for significant differences).

If we are comparing exactly two systems (perhaps to see if a particular change actually improved the system), we can also compare using **CMOS** (Comparative MOS). where users give their preference on which of the two utterances is better. CMOS scores range from -3 (the system is much worse than the reference) to 3 (the system is better than the reference) Here we play the same sentence synthesized by two different systems. The human listeners choose which of the two utterances they like better. We do this for say 50 sentences (presented in random order) and compare the number of sentences preferred for each system.

CMOS

Although speech synthesis systems are best evaluated by human listeners, some automatic metrics can be used to add more information. For example we can run the output through an ASR system and compute the word error rate (WER) to see how robust the synthesized output is. Or for measuring how well the voice output of the TTS system matches the enrolled voice, we can treat the task as if it were speaker verification, passing the two voices to a speaker verification system and using the resulting score as a similarity score.

16.5 Other speech tasks

There are a wide variety of other speech-related tasks.

speaker diarization

Speaker diarization is the task of determining 'who spoke when' in a long multi-speaker audio recording, marking the start and end of each speaker's turns in the interaction. This can be useful for transcribing meetings, classroom speech, or medical interactions. Often diarization systems use voice activity detection (VAD) to find segments of continuous speech, extract speaker embedding vectors, and cluster the vectors to group together segments likely from the same speaker. More recent work is investigating end-to-end algorithms to map directly from input speech to a sequence of speaker labels for each frame.

speaker recognition
speaker verification

Speaker recognition, is the task of identifying a speaker. We generally distinguish the subtasks of **speaker verification**, where we make a binary decision (is this speaker X or not?), such as for security when accessing personal information over the telephone, and **speaker identification**, where we make a one of N decision trying to match a speaker's voice against a database of many speakers.

language identification

In the task of **language identification**, we are given a wavefile and must identify which language is being spoken; this is an important part of building multilingual models, creating datasets, and even plays a role in online systems.

wake word

The task of **wake word** detection is to detect a word or short phrase, usually in order to wake up a voice-enable assistant like Alexa, Siri, or the Google Assistant. The goal with wake words is build the detection into small devices at the computing edge, to maintain privacy by transmitting the least amount of user speech to a cloud-based server. Thus wake word detectors need to be fast, small footprint software that can fit into embedded devices. Wake word detectors usually use the same frontend feature extraction we saw for ASR, often followed by a whole-word classifier.

16.6 Spoken Language Models

TBD

16.7 Summary

This chapter introduced the fundamental algorithms of **text-to-speech (TTS)**.

- A common modern algorithm for TTS is to use conditional generation with a language model over audio tokens learned by a codec model.
- A neural audio **codec**, short for **coder/decoder**, is a system that encodes analog speech signals into a digitized, discrete compressed representation for compression.
- The discrete symbols that a codec produces as its compressed representation can be used as discrete codes for language modeling.
- A codec includes an **encoder** that uses convnets to downsample speech into a downsampled embedding, a **quantizer** that converts the embedding into a series of discrete tokens, and a decoder that uses convnets to upsample the tokens/embedding back into a lossy reconstructed waveform.
- **Vector Quantization** (**VQ**) is a method for turning a series of vectors into a series of discrete symbols. This can be done by using k-means clustering, and then creating a **codebook** in which each code is represented by a vector at the centroid of each cluster, called a **codeword**. Input vector can be assigned the nearest codeword cluster.
- **Residual Vector Quantization** (**RVQ**) is a hierarchical version of vector quantization that produces multiple codes for an input vector by first quantizing a vector into a codebook, and then quantizing the residual (the difference between the codeword and the input vector) and then iterating.
- TTS systems like **VALL-E** take a text to be synthesized and a sample of the voice to be used, tokenize with BPE (text) and an audio codec (speech) and then use an LM to conditionally generate discrete audio tokens corresponding to the text prompt, in the voice of the speech sample.
- TTS is evaluated by playing a sentence to human listeners and having them give a **mean opinion score (MOS)**.

Historical Notes

As we noted at the beginning of the chapter, speech synthesis is one of the earliest fields of speech and language processing. The 18th century saw a number of physical models of the articulation process, including the von Kempelen model mentioned above, as well as the 1773 vowel model of Kratzenstein in Copenhagen using organ pipes.

The early 1950s saw the development of three early paradigms of waveform synthesis: formant synthesis, articulatory synthesis, and concatenative synthesis.

Formant synthesizers originally were inspired by attempts to mimic human speech by generating artificial spectrograms. The Haskins Laboratories Pattern Playback Machine generated a sound wave by painting spectrogram patterns on a moving transparent belt and using reflectance to filter the harmonics of a waveform (Cooper et al., 1951); other very early formant synthesizers include those of Lawrence (1953) and Fant (1951). Perhaps the most well-known of the formant synthesizers were the **Klatt formant synthesizer** and its successor systems, including the MITalk system (Allen et al., 1987) and the Klattalk software used in Digital Equipment Corporation's DECtalk (Klatt, 1982). See Klatt (1975) for details.

A second early paradigm, concatenative synthesis, seems to have been first proposed by Harris (1953) at Bell Laboratories; he literally spliced together pieces of magnetic tape corresponding to phones. Soon afterwards, Peterson et al. (1958) proposed a theoretical model based on diphones, including a database with multiple copies of each diphone with differing prosody, each labeled with prosodic features including F0, stress, and duration, and the use of join costs based on F0 and formant distance between neighboring units. But such **diphone synthesis** models were not actually implemented until decades later (Dixon and Maxey 1968, Olive 1977). The 1980s and 1990s saw the invention of **unit selection synthesis**, based on larger units of non-uniform length and the use of a target cost, (Sagisaka 1988, Sagisaka et al. 1992, Hunt and Black 1996, Black and Taylor 1994, Syrdal et al. 2000).

A third paradigm, **articulatory synthesizers** attempt to synthesize speech by modeling the physics of the vocal tract as an open tube. Representative models include Stevens et al. (1953), Flanagan et al. (1975), and Fant (1986). See Klatt (1975) and Flanagan (1972) for more details.

Most early TTS systems used phonemes as input; development of the text analysis components of TTS came somewhat later, drawing on NLP. Indeed the first true text-to-speech system seems to have been the system of Umeda and Teranishi (Umeda et al. 1968, Teranishi and Umeda 1968, Umeda 1976), which included a parser that assigned prosodic boundaries, as well as accent and stress.

History of codecs and modern history of neural TTS TBD.

Exercises

Allen, J., M. S. Hunnicut, and D. H. Klatt. 1987. *From Text to Speech: The MITalk system*. Cambridge University Press.

Ardila, R., M. Branson, K. Davis, M. Kohler, J. Meyer, M. Henretty, R. Morais, L. Saunders, F. Tyers, and G. Weber. 2020. Common voice: A massively-multilingual speech corpus. *LREC*.

Black, A. W. and P. Taylor. 1994. CHATR: A generic speech synthesis system. *COLING*.

Chen, S., C. Wang, Y. Wu, Z. Zhang, L. Zhou, S. Liu, Z. Chen, Y. Liu, H. Wang, J. Li, L. He, S. Zhao, and F. Wei. 2025. Neural codec language models are zero-shot text to speech synthesizers. *IEEE Trans. on TASLP*, 33:705–718.

Cooper, F. S., A. M. Liberman, and J. M. Borst. 1951. The interconversion of audible and visible patterns as a basis for research in the perception of speech. *Proceedings of the National Academy of Sciences*, 37(5):318–325.

Défossez, A., J. Copet, G. Synnaeve, and Y. Adi. 2023. High fidelity neural audio compression. *TMLR*.

Dixon, N. and H. Maxey. 1968. Terminal analog synthesis of continuous speech using the diphone method of segment assembly. *IEEE Transactions on Audio and Electroacoustics*, 16(1):40–50.

Fant, G. M. 1951. Speech communication research. *Ing. Vetenskaps Akad. Stockholm, Sweden*, 24:331–337.

Fant, G. M. 1986. Glottal flow: Models and interaction. *Journal of Phonetics*, 14:393–399.

Flanagan, J. L. 1972. *Speech Analysis, Synthesis, and Perception*. Springer.

Flanagan, J. L., K. Ishizaka, and K. L. Shipley. 1975. Synthesis of speech from a dynamic model of the vocal cords and vocal tract. *The Bell System Technical Journal*, 54(3):485–506.

Gemmeke, J. F., D. P. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter. 2017. Audio Set: An ontology and human-labeled dataset for audio events. *ICASSP*.

Gray, R. M. 1984. Vector quantization. *IEEE Transactions on ASSP*, ASSP-1(2):4–29.

Harris, C. M. 1953. A study of the building blocks in speech. *JASA*, 25(5):962–969.

Hunt, A. J. and A. W. Black. 1996. Unit selection in a concatenative speech synthesis system using a large speech database. *ICASSP*.

Klatt, D. H. 1975. Voice onset time, friction, and aspiration in word-initial consonant clusters. *Journal of Speech and Hearing Research*, 18:686–706.

Klatt, D. H. 1982. The Klattalk text-to-speech conversion system. *ICASSP*.

Lawrence, W. 1953. The synthesis of speech from signals which have a low information rate. In W. Jackson, ed., *Communication Theory*, 460–469. Butterworth.

Mousavi, P., G. Maimon, A. Moumen, D. Petermann, J. Shi, H. Wu, H. Yang, A. Kuznetsova, A. Ploujnikov, R. Marxer, B. Ramabhadran, B. Elizalde, L. Lugosch, J. Li, C. Subakan, P. Woodland, M. Kim, H. yi Lee, S. Watanabe, Y. Adi, and M. Ravanelli. 2025. Discrete audio tokens: More than a survey! ArXiv preprint.

Olive, J. P. 1977. Rule synthesis of speech from dyadic units. *ICASSP77*.

Peterson, G. E., W. S.-Y. Wang, and E. Sivertsen. 1958. Segmentation techniques in speech synthesis. *JASA*, 30(8):739–742.

Sagisaka, Y. 1988. Speech synthesis by rule using an optimal selection of non-uniform synthesis units. *ICASSP*.

Sagisaka, Y., N. Kaiki, N. Iwahashi, and K. Mimura. 1992. Atr – ν-talk speech synthesis system. *ICSLP*.

Stevens, K. N., S. Kasowski, and G. M. Fant. 1953. An electrical analog of the vocal tract. *JASA*, 25(4):734–742.

Syrdal, A. K., C. W. Wightman, A. Conkie, Y. Stylianou, M. Beutnagel, J. Schroeter, V. Strom, and K.-S. Lee. 2000. Corpus-based techniques in the AT&T NEXTGEN synthesis system. *ICSLP*.

Teranishi, R. and N. Umeda. 1968. Use of pronouncing dictionary in speech synthesis experiments. *6th International Congress on Acoustics*.

Umeda, N. 1976. Linguistic rules for text-to-speech synthesis. *Proceedings of the IEEE*, 64(4):443–451.

Umeda, N., E. Matui, T. Suzuki, and H. Omura. 1968. Synthesis of fairy tale using an analog vocal tract. *6th International Congress on Acoustics*.

Van Den Oord, A., O. Vinyals, and K. Kavukcuoglu. 2017. Neural discrete representation learning. *NeurIPS*.