CHAPTER

16 Automatic Speech Recognition and Text-to-Speech

I KNOW not whether
I see your meaning: if I do, it lies
Upon the wordy wavelets of your voice,
Dim as an evening shadow in a brook,
Thomas Lovell Beddoes, 1851

Understanding spoken language, or at least transcribing the words into writing, is one of the earliest goals of computer language processing. In fact, speech processing predates the computer by many decades! The first machine that recognized speech was a toy from the 1920s. "Radio Rex", shown to the right, was a celluloid dog that moved (by means of a spring) when the spring was released by 500 Hz acoustic energy. Since 500 Hz is roughly the first formant of the vowel [eh] in "Rex", Rex seemed to come when he was called (David, Jr. and Selfridge, 1962).

In modern times, we expect more of our automatic systems. The task of **automatic speech recognition** (**ASR**) is to map any waveform like this:

ASR

to the appropriate string of words:

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

Automatic transcription of speech by any speaker in any environment is still far from solved, but ASR technology has matured to the point where it is now viable for many practical tasks. Speech is a natural interface for communicating with smart home appliances, personal assistants, or cellphones, where keyboards are less convenient, in telephony applications like call-routing ("Accounting, please") or in sophisticated dialogue applications ("I'd like to change the return date of my flight"). ASR is also useful for general transcription, for example for automatically generating captions for audio or video text (transcribing movies or videos or live discussions). Transcription is important in fields like law where dictation plays an important role. Finally, ASR is important as part of augmentative communication (interaction between computers and humans with some disability resulting in difficulties or inabilities in typing or audition). The blind Milton famously dictated *Paradise Lost* to his daughters, and Henry James dictated his later novels after a repetitive stress injury.

What about the opposite problem, going from text to speech? This is a problem with an even longer history. 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.

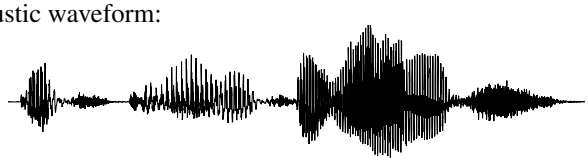
What is less well known is that von Kempelen, an extraordinarily prolific inventor, also built between 1769 and 1790 what was definitely not a hoax: the first full-sentence speech synthesizer, shown partially to the right. His device consisted of a bellows to simulate the lungs, a rubber mouthpiece and 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 **speech synthesis**, also called **text-to-speech** or **TTS**, is exactly the reverse of ASR; to map text:

speech synthesis
text-to-speech
TTS

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

to an acoustic waveform:



Modern speech synthesis has a wide variety of applications. TTS is used in conversational agents that conduct dialogues with people, plays a role in devices that read out loud for the blind or in games, and can be used to speak for sufferers of neurological disorders, such as the late astrophysicist Steven Hawking who, after he lost the use of his voice because of ALS, spoke by manipulating a TTS system.

In the next sections we'll show how to do ASR with encoder-decoders, introduce the **CTC** loss functions, the standard **word error rate** evaluation metric, and describe how acoustic features are extracted. We'll then see how TTS can be modeled with almost the same algorithm in reverse, and conclude with a brief mention of other speech tasks.

16.1 The Automatic Speech Recognition Task

Before describing algorithms for ASR, let's talk about how the task itself varies. One dimension of variation is vocabulary size. Some ASR tasks can be solved with extremely high accuracy, like those with a 2-word vocabulary (*yes* versus *no*) or an 11 word vocabulary like **digit recognition** (recognizing sequences of digits including *zero* to *nine* plus *oh*). Open-ended tasks like transcribing videos or human conversations, with large vocabularies of up to 60,000 words, are much harder.

digit recognition

A second dimension of variation is who the speaker is talking to. Humans speaking to machines (either dictating or talking to a dialogue system) are easier to recognize than humans speaking to humans. **Read speech**, in which humans are reading out loud, for example in audio books, is also relatively easy to recognize. Recognizing the speech of two humans talking to each other in **conversational speech**, for example, for transcribing a business meeting, is the hardest. It seems that when humans talk to machines, or read without an audience present, they simplify their speech quite a bit, talking more slowly and more clearly.

read speech

conversational speech

A third dimension of variation is channel and noise. Speech is easier to recognize if it's recorded in a quiet room with head-mounted microphones than if it's recorded by a distant microphone on a noisy city street, or in a car with the window open.

A final dimension of variation is accent or speaker-class characteristics. Speech is easier to recognize if the speaker is speaking the same dialect or variety that the system was trained on. Speech by speakers of regional or ethnic dialects, or speech by children can be quite difficult to recognize if the system is only trained on speakers of standard dialects, or only adult speakers.

A number of publicly available corpora with human-created transcripts are used to create ASR test and training sets to explore this variation; we mention a few of them here since you will encounter them in the literature. **LibriSpeech** is a large open-source read-speech 16 kHz dataset with over 1000 hours of audio books from the LibriVox project, with transcripts aligned at the sentence level (Panayotov et al., 2015). It is divided into an easier ("clean") and a more difficult portion ("other") with the clean portion of higher recording quality and with accents closer to US English. This was done by running a speech recognizer (trained on read speech from the Wall Street Journal) on all the audio, computing the WER for each speaker based on the gold transcripts, and dividing the speakers roughly in half, with recordings from lower-WER speakers called "clean" and recordings from higher-WER speakers "other".

LibriSpeech

The **Switchboard** corpus of prompted telephone conversations between strangers was collected in the early 1990s; it contains 2430 conversations averaging 6 minutes each, totaling 240 hours of 8 kHz speech and about 3 million words (Godfrey et al., 1992). Switchboard has the singular advantage of an enormous amount of auxiliary hand-done linguistic labeling, including parses, dialogue act tags, phonetic and prosodic labeling, and discourse and information structure. The **CALLHOME** corpus was collected in the late 1990s and consists of 120 unscripted 30-minute telephone conversations between native speakers of English who were usually close friends or family (Canavan et al., 1997).

Switchboard

CALLHOME

The Santa Barbara Corpus of Spoken American English (Du Bois et al., 2005) is a large corpus of naturally occurring everyday spoken interactions from all over the United States, mostly face-to-face conversation, but also town-hall meetings, food preparation, on-the-job talk, and classroom lectures. The corpus was anonymized by removing personal names and other identifying information (replaced by pseudonyms in the transcripts, and masked in the audio).

CORAAL is a collection of over 150 sociolinguistic interviews with African American speakers, with the goal of studying African American Language (**AAL**), the many variations of language used in African American communities (Kendall and Farrington, 2020). The interviews are anonymized with transcripts aligned at the utterance level. The **CHiME** Challenge is a series of difficult shared tasks with corpora that deal with robustness in ASR. The CHiME 5 task, for example, is ASR of conversational speech in real home environments (specifically dinner parties). The

CORAAL

CHiME

corpus contains recordings of twenty different dinner parties in real homes, each with four participants, and in three locations (kitchen, dining area, living room), recorded both with distant room microphones and with body-worn mikes. The

HKUST

HKUST Mandarin Telephone Speech corpus has 1206 ten-minute telephone conversations between speakers of Mandarin across China, including transcripts of the conversations, which are between either friends or strangers (Liu et al., 2006). The

AISHELL-1

AISHELL-1 corpus contains 170 hours of Mandarin read speech of sentences taken from various domains, read by different speakers mainly from northern China (Bu et al., 2017).

Figure 16.1 shows the rough percentage of incorrect words (the **word error rate**, or WER, defined on page 343) from state-of-the-art systems on some of these tasks. Note that the error rate on read speech (like the LibriSpeech audiobook corpus) is around 2%; this is a solved task, although these numbers come from systems that require enormous computational resources. By contrast, the error rate for transcribing conversations between humans is much higher; 5.8 to 11% for the Switchboard and CALLHOME corpora. The error rate is higher yet again for speakers of varieties like African American Vernacular English, and yet again for difficult conversational tasks like transcription of 4-speaker dinner party speech, which can have error rates as high as 81.3%. Character error rates (CER) are also much lower for read Mandarin speech than for natural conversation.

English Tasks	**WER%**
LibriSpeech audiobooks 960hour clean	1.4
LibriSpeech audiobooks 960hour other	2.6
Switchboard telephone conversations between strangers	5.8
CALLHOME telephone conversations between family	11.0
Sociolinguistic interviews, CORAAL (AAL)	27.0
CHiMe5 dinner parties with body-worn microphones	47.9
CHiMe5 dinner parties with distant microphones	81.3
Chinese (Mandarin) Tasks	**CER%**
AISHELL-1 Mandarin read speech corpus	6.7
HKUST Mandarin Chinese telephone conversations	23.5

Figure 16.1 Rough Word Error Rates (WER = % of words misrecognized) reported around 2020 for ASR on various American English recognition tasks, and character error rates (CER) for two Chinese recognition tasks.

16.2 Feature Extraction for ASR: Log Mel Spectrum

feature vector

The first step in ASR is to transform the input waveform into a sequence of acoustic **feature vectors**, each vector representing the information in a small time window of the signal. Let's see how to convert a raw wavefile to the most commonly used features, sequences of **log mel spectrum** vectors. A speech signal processing course is recommended for more details.

16.2.1 Sampling and Quantization

Recall from Section 28.4.2 that the first step is to convert the analog representations (first air pressure and then analog electric signals in a microphone) into a digital sig-

sampling

nal. This **analog-to-digital conversion** has two steps: **sampling** and **quantization**. A signal is sampled by measuring its amplitude at a particular time; the **sampling rate** is the number of samples taken per second. To accurately measure a wave, we must have at least two samples in each cycle: one measuring the positive part of the wave and one measuring the negative part. More than two samples per cycle increases the amplitude accuracy, but less than two samples will cause the frequency of the wave to be completely missed. Thus, the maximum frequency wave that can be measured is one whose frequency is half the sample rate (since every cycle needs two samples). This maximum frequency for a given sampling rate is called the **Nyquist frequency**. Most information in human speech is in frequencies below 10,000 Hz, so a 20,000 Hz sampling rate would be necessary for complete accuracy. But telephone speech is filtered by the switching network, and only frequencies less than 4,000 Hz are transmitted by telephones. Thus, an 8,000 Hz sampling rate is sufficient for **telephone-bandwidth** speech, and 16,000 Hz for microphone speech.

sampling rate

Nyquist frequency

telephone-bandwidth

Although using higher sampling rates produces higher ASR accuracy, we can't combine different sampling rates for training and testing ASR systems. Thus if we are testing on a telephone corpus like Switchboard (8 KHz sampling), we must downsample our training corpus to 8 KHz. Similarly, if we are training on multiple corpora and one of them includes telephone speech, we downsample all the wideband corpora to 8Khz.

Amplitude measurements are stored as integers, either 8 bit (values from -128–127) or 16 bit (values from -32768–32767). This process of representing real-valued numbers as integers is called **quantization**; all values that are closer together than the minimum granularity (the quantum size) are represented identically. We refer to each sample at time index n in the digitized, quantized waveform as $x[n]$.

quantization

16.2.2 Windowing

From the digitized, quantized representation of the waveform, we need to extract spectral features from a small **window** of speech that characterizes part of a particular phoneme. Inside this small window, we can roughly think of the signal as **stationary** (that is, its statistical properties are constant within this region). (By contrast, in general, speech is a **non-stationary** signal, meaning that its statistical properties are not constant over time). We extract this roughly stationary portion of speech by using a window which is non-zero inside a region and zero elsewhere, running this window across the speech signal and multiplying it by the input waveform to produce a windowed waveform.

stationary

non-stationary

The speech extracted from each window is called a **frame**. The windowing is characterized by three parameters: the **window size** or **frame size** of the window (its width in milliseconds), the **frame stride**, (also called **shift** or **offset**) between successive windows, and the **shape** of the window.

frame

stride

To extract the signal we multiply the value of the signal at time n, $s[n]$ by the value of the window at time n, $w[n]$:

$$y[n] = w[n]s[n] \tag{16.1}$$

The window shape sketched in Fig. 16.2 is **rectangular**; you can see the extracted windowed signal looks just like the original signal. The rectangular window, however, abruptly cuts off the signal at its boundaries, which creates problems when we do Fourier analysis. For this reason, for acoustic feature creation we more commonly use the **Hamming** window, which shrinks the values of the signal toward

rectangular

Hamming

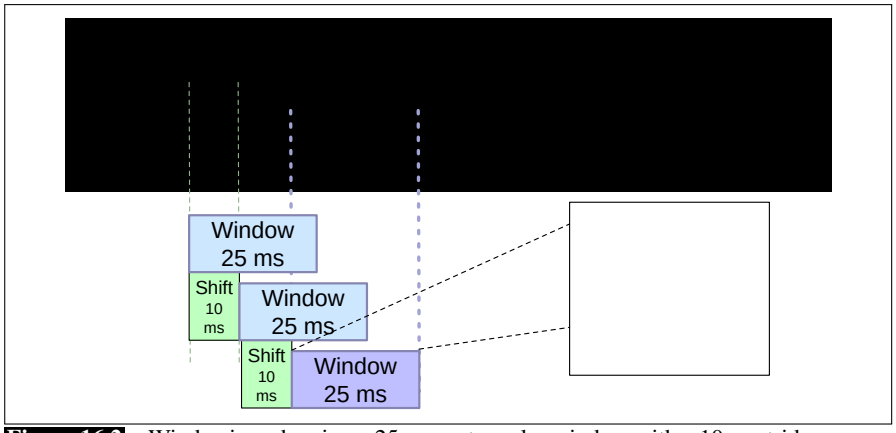


Figure 16.2 Windowing, showing a 25 ms rectangular window with a 10ms stride.

zero at the window boundaries, avoiding discontinuities. Figure 16.3 shows both; the equations are as follows (assuming a window that is L frames long):

$$rectangular \quad w[n] = \begin{cases} 1 & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases} \tag{16.2}$$

$$Hamming \quad w[n] = \begin{cases} 0.54 - 0.46\cos(\frac{2\pi n}{L}) & 0 \le n \le L-1 \\ 0 & \text{otherwise} \end{cases} \tag{16.3}$$

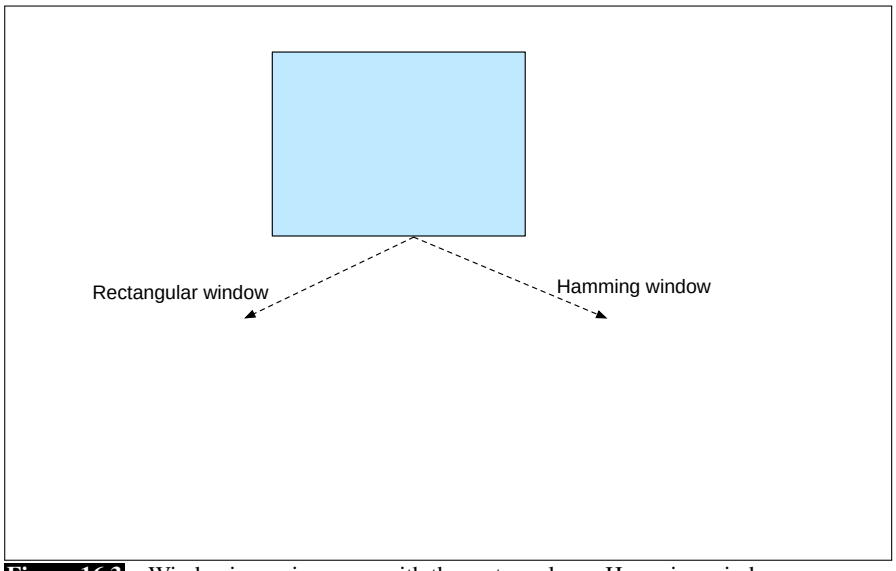


Figure 16.3 Windowing a sine wave with the rectangular or Hamming windows.

16.2.3 Discrete Fourier Transform

The next step is to extract spectral information for our windowed signal; we need to know how much energy the signal contains at different frequency bands. The tool for extracting spectral information for discrete frequency bands for a discrete-time (sampled) signal is the **discrete Fourier transform** or **DFT**.

Discrete Fourier transform DFT

The input to the DFT is a windowed signal $x[n]...x[m]$, and the output, for each of N discrete frequency bands, is a complex number $X[k]$ representing the magnitude and phase of that frequency component in the original signal. If we plot the magnitude against the frequency, we can visualize the **spectrum** that we introduced in Chapter 28. For example, Fig. 16.4 shows a 25 ms Hamming-windowed portion of a signal and its spectrum as computed by a DFT (with some additional smoothing).

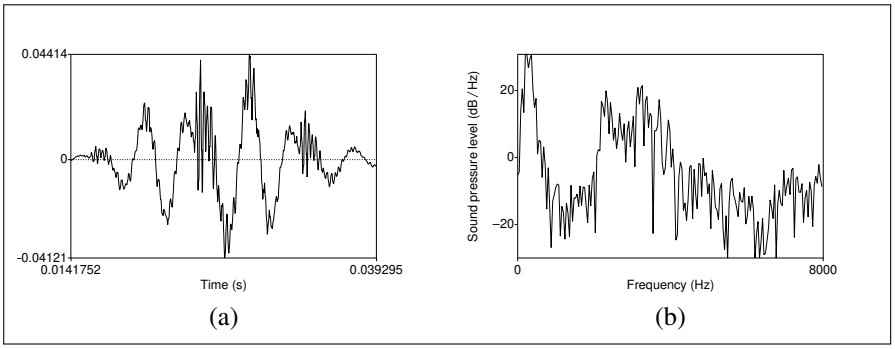


Figure 16.4 (a) A 25 ms Hamming-windowed portion of a signal from the vowel [iy] and (b) its spectrum computed by a DFT.

We do not introduce the mathematical details of the DFT here, except to note that Fourier analysis relies on **Euler's formula**, with j as the imaginary unit:

Euler's formula

$$e^{j\theta} = \cos\theta + j\sin\theta \tag{16.4}$$

As a brief reminder for those students who have already studied signal processing, the DFT is defined as follows:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{16.5}$$

fast Fourier transform
FFT

A commonly used algorithm for computing the DFT is the **fast Fourier transform** or **FFT**. This implementation of the DFT is very efficient but only works for values of N that are powers of 2.

16.2.4 Mel Filter Bank and Log

The results of the FFT tell us the energy at each frequency band. Human hearing, however, is not equally sensitive at all frequency bands; it is less sensitive at higher frequencies. This bias toward low frequencies helps human recognition, since information in low frequencies like formants is crucial for distinguishing values or nasals, while information in high frequencies like stop bursts or fricative noise is less crucial for successful recognition. Modeling this human perceptual property improves speech recognition performance in the same way.

We implement this intuition by collecting energies, not equally at each frequency band, but according to the **mel** scale, an auditory frequency scale (Chapter 28). A **mel** (Stevens et al. 1937, Stevens and Volkmann 1940) is a unit of pitch. Pairs of sounds that are perceptually equidistant in pitch are separated by an equal number of mels. The mel frequency m can be computed from the raw acoustic frequency by a log transformation:

mel

$$mel(f) = 1127\ln(1+\frac{f}{700}) \tag{16.6}$$

We implement this intuition by creating a bank of filters that collect energy from each frequency band, spread logarithmically so that we have very fine resolution at low frequencies, and less resolution at high frequencies. Figure 16.5 shows a sample bank of triangular filters that implement this idea, that can be multiplied by the spectrum to get a mel spectrum.

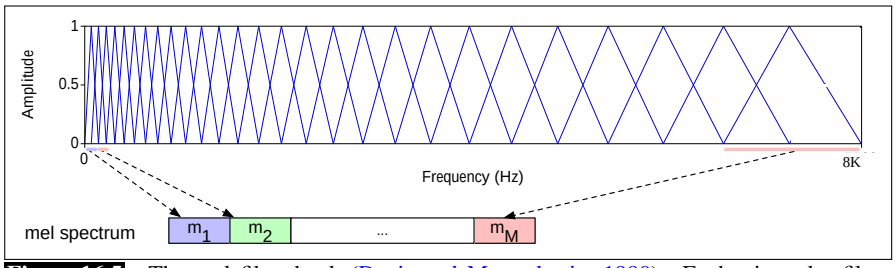


Figure 16.5 The mel filter bank (Davis and Mermelstein, 1980). Each triangular filter, spaced logarithmically along the mel scale, collects energy from a given frequency range.

Finally, we take the log of each of the mel spectrum values. The human response to signal level is logarithmic (like the human response to frequency). Humans are less sensitive to slight differences in amplitude at high amplitudes than at low amplitudes. In addition, using a log makes the feature estimates less sensitive to variations in input such as power variations due to the speaker's mouth moving closer or further from the microphone.

16.3 Speech Recognition Architecture

The basic architecture for ASR is the encoder-decoder (implemented with either RNNs or Transformers), exactly the same architecture introduced for MT in Chapter 13. Generally we start from the log mel spectral features described in the previous section, and map to letters, although it's also possible to map to induced morpheme-like chunks like wordpieces or BPE.

AED
listen attend and spell

Fig. 16.6 sketches the standard encoder-decoder architecture, which is commonly referred to as the **attention-based encoder decoder** or **AED**, or **listen attend and spell** (**LAS**) after the two papers which first applied it to speech (Chorowski et al. 2014, Chan et al. 2016). The input is a sequence of t acoustic feature vectors $F = f_1, f_2, ..., f_t$, one vector per 10 ms frame. The output can be letters or wordpieces; we'll assume letters here. Thus the output sequence $Y = (\langle \text{SOS} \rangle, y_1, ..., y_m \langle \text{EOS} \rangle)$, assuming special start of sequence and end of sequence tokens $\langle \text{sos} \rangle$ and $\langle \text{eos} \rangle$ and each y_i is a character; for English we might choose the set:

$$y_i \in \{a, b, c, ..., z, 0, ..., 9, \langle \text{space} \rangle, \langle \text{comma} \rangle, \langle \text{period} \rangle, \langle \text{apostrophe} \rangle, \langle \text{unk} \rangle\}$$

Of course the encoder-decoder architecture is particularly appropriate when input and output sequences have stark length differences, as they do for speech, with very long acoustic feature sequences mapping to much shorter sequences of letters or words. A single word might be 5 letters long but, supposing it lasts about 2 seconds, would take 200 acoustic frames (of 10ms each).

Because this length difference is so extreme for speech, encoder-decoder architectures for speech need to have a special compression stage that shortens the acoustic feature sequence before the encoder stage. (Alternatively, we can use a loss

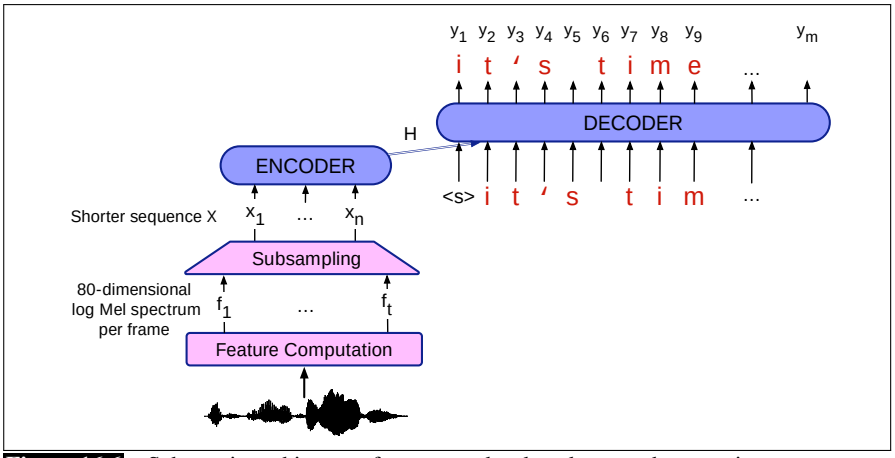


Figure 16.6 Schematic architecture for an encoder-decoder speech recognizer.

function that is designed to deal well with compression, like the CTC loss function we'll introduce in the next section.)

The goal of the subsampling is to produce a shorter sequence $X = x_1, ..., x_n$ that will be the input to the encoder. The simplest algorithm is a method sometimes called **low frame rate** (Pundak and Sainath, 2016): for time i we stack (concatenate) the acoustic feature vector f_i with the prior two vectors f_{i-1} and f_{i-2} to make a new vector three times longer. Then we simply delete f_{i-1} and f_{i-2}. Thus instead of (say) a 40-dimensional acoustic feature vector every 10 ms, we have a longer vector (say 120-dimensional) every 30 ms, with a shorter sequence length $n = \frac{t}{3}$.[1]

low frame rate

After this compression stage, encoder-decoders for speech use the same architecture as for MT or other text, composed of either RNNs (LSTMs) or Transformers.

For inference, the probability of the output string Y is decomposed as:

$$p(y_1, \ldots, y_n) = \prod_{i=1}^{n} p(y_i | y_1, \ldots, y_{i-1}, X) \tag{16.7}$$

We can produce each letter of the output via greedy decoding:

$$\hat{y}_i = \text{argmax}_{\text{char} \in \text{ Alphabet}} P(\text{char} | y_1 \ldots y_{i-1}, X) \tag{16.8}$$

Alternatively we can use beam search as described in the next section. This is particularly relevant when we are adding a language model.

Adding a language model Since an encoder-decoder model is essentially a conditional language model, encoder-decoders implicitly learn a language model for the output domain of letters from their training data. However, the training data (speech paired with text transcriptions) may not include sufficient text to train a good language model. After all, it's easier to find enormous amounts of pure text training data than it is to find text paired with speech. Thus we can can usually improve a model at least slightly by incorporating a very large language model.

The simplest way to do this is to use beam search to get a final beam of hypothesized sentences; this beam is sometimes called an **n-best list**. We then use a language model to **rescore** each hypothesis on the beam. The scoring is done by in-

n-best list

rescore

[1] There are also more complex alternatives for subsampling, like using a convolutional net that downsamples with max pooling, or layers of **pyramidal RNNs**, RNNs where each successive layer has half the number of RNNs as the previous layer.

terpolating the score assigned by the language model with the encoder-decoder score used to create the beam, with a weight λ tuned on a held-out set. Also, since most models prefer shorter sentences, ASR systems normally have some way of adding a length factor. One way to do this is to normalize the probability by the number of characters in the hypothesis $|Y|_c$. The following is thus a typical scoring function (Chan et al., 2016):

$$score(Y|X) = \frac{1}{|Y|_c} \log P(Y|X) + \lambda \log P_{LM}(Y) \tag{16.9}$$

16.3.1 Learning

Encoder-decoders for speech are trained with the normal cross-entropy loss generally used for conditional language models. At timestep i of decoding, the loss is the log probability of the correct token (letter) y_i:

$$L_{CE} = -\log p(y_i|y_1,\ldots,y_{i-1},X) \tag{16.10}$$

The loss for the entire sentence is the sum of these losses:

$$L_{CE} = -\sum_{i=1}^{m} \log p(y_i|y_1,\ldots,y_{i-1},X) \tag{16.11}$$

This loss is then backpropagated through the entire end-to-end model to train the entire encoder-decoder.

As we described in Chapter 13, we normally use teacher forcing, in which the decoder history is forced to be the correct gold y_i rather than the predicted $\hat{y}_i$. It's also possible to use a mixture of the gold and decoder output, for example using the gold output 90% of the time, but with probability .1 taking the decoder output instead:

$$L_{CE} = -\log p(y_i|y_1,\ldots,\hat{y}_{i-1},X) \tag{16.12}$$

16.4 CTC

We pointed out in the previous section that speech recognition has two particular properties that make it very appropriate for the encoder-decoder architecture, where the encoder produces an encoding of the input that the decoder uses attention to explore. First, in speech we have a very long acoustic input sequence X mapping to a much shorter sequence of letters Y, and second, it's hard to know exactly which part of X maps to which part of Y.

CTC

In this section we briefly introduce an alternative to encoder-decoder: an algorithm and loss function called **CTC**, short for **Connectionist Temporal Classification** (Graves et al., 2006), that deals with these problems in a very different way. The intuition of CTC is to output a single character for every frame of the input, so that the output is the same length as the input, and then to apply a collapsing function that combines sequences of identical letters, resulting in a shorter sequence.

Let's imagine inference on someone saying the word *dinner*, and let's suppose we had a function that chooses the most probable letter for each input spectral frame representation x_i. We'll call the sequence of letters corresponding to each input

alignment

frame an **alignment**, because it tells us where in the acoustic signal each letter aligns to. Fig. 16.7 shows one such alignment, and what happens if we use a collapsing function that just removes consecutive duplicate letters.

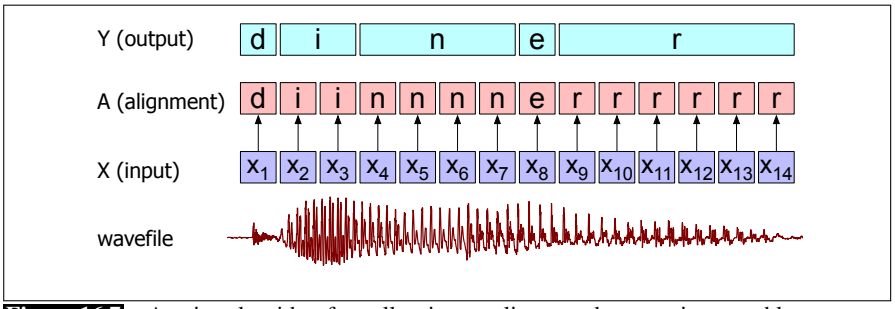


Figure 16.7 A naive algorithm for collapsing an alignment between input and letters.

Well, that doesn't work; our naive algorithm has transcribed the speech as *diner*, not *dinner*! Collapsing doesn't handle double letters. There's also another problem with our naive function; it doesn't tell us what symbol to align with silence in the input. We don't want to be transcribing silence as random letters!

blank

The CTC algorithm solves both problems by adding to the transcription alphabet a special symbol for a **blank**, which we'll represent as ␣. The blank can be used in the alignment whenever we don't want to transcribe a letter. Blank can also be used between letters; since our collapsing function collapses only consecutive duplicate letters, it won't collapse across ␣. More formally, let's define the mapping $B : a \rightarrow y$ between an alignment a and an output y, which collapses all repeated letters and then removes all blanks. Fig. 16.8 sketches this collapsing function B.

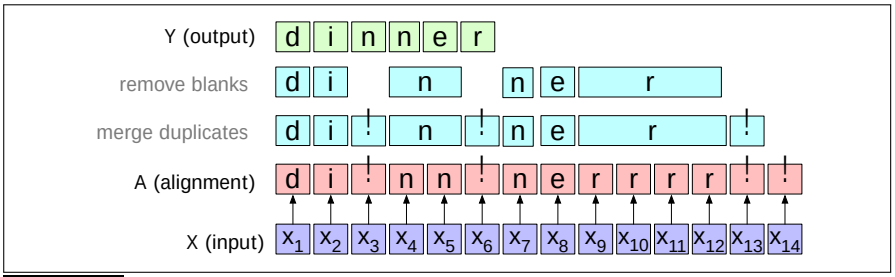


Figure 16.8 The CTC collapsing function B, showing the space blank character ␣; repeated (consecutive) characters in an alignment A are removed to form the output Y.

The CTC collapsing function is many-to-one; lots of different alignments map to the same output string. For example, the alignment shown in Fig. 16.8 is not the only alignment that results in the string *dinner*. Fig. 16.9 shows some other alignments that would produce the same output.

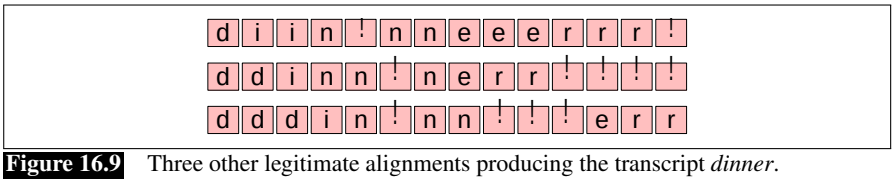


Figure 16.9 Three other legitimate alignments producing the transcript *dinner*.

It's useful to think of the set of all alignments that might produce the same output

Y. We'll use the inverse of our B function, called B^{-1}, and represent that set as $B^{-1}(Y)$.

16.4.1 CTC Inference

Before we see how to compute $P_{\text{CTC}}(Y|X)$ let's first see how CTC assigns a probability to one particular alignment $\hat{A} = \{\hat{a}_1, \ldots, \hat{a}_n\}$. CTC makes a strong conditional independence assumption: it assumes that, given the input X, the CTC model output a_t at time t is independent of the output labels at any other time a_i. Thus:

$$P_{\text{CTC}}(A|X) = \prod_{t=1}^{T} p(a_t|X) \tag{16.13}$$

Thus to find the best alignment $\hat{A} = \{\hat{a}_1, \ldots, \hat{a}_T\}$ we can greedily choose the character with the max probability at each time step t:

$$\hat{a}_t = \underset{c \in C}{\text{argmax}}\, p_t(c|X) \tag{16.14}$$

We then pass the resulting sequence A to the CTC collapsing function B to get the output sequence Y.

Let's talk about how this simple inference algorithm for finding the best alignment A would be implemented. Because we are making a decision at each time point, we can treat CTC as a sequence-modeling task, where we output one letter $\hat{y}_t$ at time t corresponding to each input token x_t, eliminating the need for a full decoder. Fig. 16.10 sketches this architecture, where we take an encoder, produce a hidden state h_t at each timestep, and decode by taking a softmax over the character vocabulary at each time step.

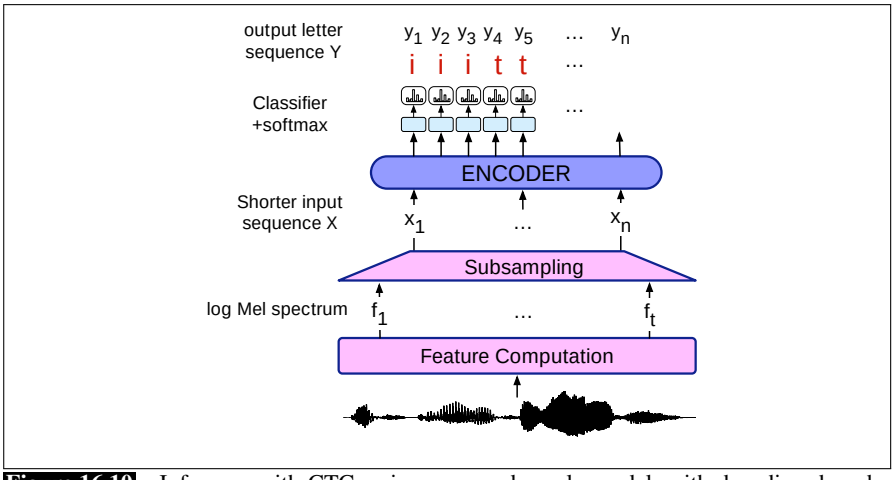


Figure 16.10 Inference with CTC: using an encoder-only model, with decoding done by simple softmaxes over the hidden state h_t at each output step.

Alas, there is a potential flaw with the inference algorithm sketched in (Eq. 16.14) and Fig. 16.9. The problem is that we chose the most likely alignment A, but the most likely alignment may not correspond to the most likely final collapsed output string Y. That's because there are many possible alignments that lead to the same output string, and hence the most likely output string might not correspond to the

most probable alignment. For example, imagine the most probable alignment A for an input $X = [x_1 x_2 x_3]$ is the string [a b ϵ] but the next two most probable alignments are [b ϵ b] and [ϵ b b]. The output Y =[b b], summing over those two alignments, might be more probable than Y =[a b].

For this reason, the most probable output sequence Y is the one that has, not the single best CTC alignment, but the highest sum over the probability of all its possible alignments:

$$\begin{aligned} P_{CTC}(Y|X) &= \sum_{A \in B^{-1}(Y)} P(A|X) \\ &= \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \\ \hat{Y} &= \operatorname*{argmax}_{Y} P_{CTC}(Y|X) \end{aligned} \tag{16.15}$$

Alas, summing over all alignments is very expensive (there are a lot of alignments), so we approximate this sum by using a version of Viterbi beam search that cleverly keeps in the beam the high-probability alignments that map to the same output string, and sums those as an approximation of (Eq. 16.15). See Hannun (2017) for a clear explanation of this extension of beam search for CTC.

Because of the strong conditional independence assumption mentioned earlier (that the output at time t is independent of the output at time $t-1$, given the input), CTC does not implicitly learn a language model over the data (unlike the attention-based encoder-decoder architectures). It is therefore essential when using CTC to interpolate a language model (and some sort of length factor $L(Y)$) using interpolation weights that are trained on a dev set:

$$score_{\text{CTC}}(Y|X) = \log P_{\text{CTC}}(Y|X) + \lambda_1 \log P_{LM}(Y) \lambda_2 L(Y) \tag{16.16}$$

16.4.2 CTC Training

To train a CTC-based ASR system, we use negative log-likelihood loss with a special CTC loss function. Thus the loss for an entire dataset D is the sum of the negative log-likelihoods of the correct output Y for each input X:

$$L_{CTC} = \sum_{(X,Y) \in D} -\log P_{\text{CTC}}(Y|X) \tag{16.17}$$

To compute CTC loss function for a single input pair (X,Y), we need the probability of the output Y given the input X. As we saw in Eq. 16.15, to compute the probability of a given output Y we need to sum over all the possible alignments that would collapse to Y. In other words:

$$P_{\text{CTC}}(Y|X) = \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \tag{16.18}$$

Naively summing over all possible alignments is not feasible (there are too many alignments). However, we can efficiently compute the sum by using dynamic programming to merge alignments, with a version of the **forward-backward algorithm** also used to train HMMs (Appendix A) and CRFs. The original dynamic programming algorithms for both training and inference are laid out in (Graves et al., 2006); see (Hannun, 2017) for a detailed explanation of both.

16.4.3 Combining CTC and Encoder-Decoder

It's also possible to combine the two architectures/loss functions we've described, the cross-entropy loss from the encoder-decoder architecture, and the CTC loss. Fig. 16.11 shows a sketch. For training, we can simply weight the two losses with a λ tuned on a dev set:

$$L = -\lambda \log P_{encdec}(Y|X) - (1-\lambda) \log P_{ctc}(Y|X) \tag{16.19}$$

For inference, we can combine the two with the language model (or the length penalty), again with learned weights:

$$\hat{Y} = \underset{Y}{\operatorname{argmax}} \left[\lambda \log P_{encdec}(Y|X) - (1-\lambda) \log P_{CTC}(Y|X) + \gamma \log P_{LM}(Y) \right] \tag{16.20}$$

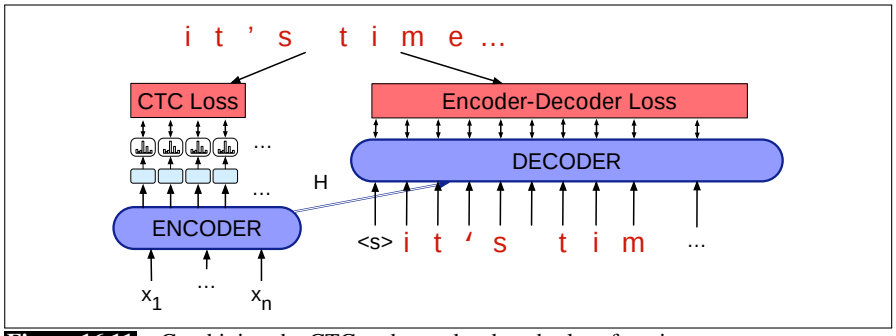


Figure 16.11 Combining the CTC and encoder-decoder loss functions.

16.4.4 Streaming Models: RNN-T for improving CTC

Because of the strong independence assumption in CTC (assuming that the output at time t is independent of the output at time $t-1$), recognizers based on CTC don't achieve as high an accuracy as the attention-based encoder-decoder recognizers. CTC recognizers have the advantage, however, that they can be used for **streaming**. Streaming means recognizing words on-line rather than waiting until the end of the sentence to recognize them. Streaming is crucial for many applications, from commands to dictation, where we want to start recognition while the user is still talking. Algorithms that use attention need to compute the hidden state sequence over the entire input first in order to provide the attention distribution context, before the decoder can start decoding. By contrast, a CTC algorithm can input letters from left to right immediately.

streaming

If we want to do streaming, we need a way to improve CTC recognition to remove the conditional independent assumption, enabling it to know about output history. The RNN-Transducer (**RNN-T**), shown in Fig. 16.12, is just such a model (Graves 2012, Graves et al. 2013). The RNN-T has two main components: a CTC acoustic model, and a separate language model component called the **predictor** that conditions on the output token history. At each time step t, the CTC encoder outputs a hidden state h_t^{enc} given the input $x_1 \ldots x_t$. The language model predictor takes as input the previous output token (not counting blanks), outputting a hidden state h_u^{pred}. The two are passed through another network whose output is then passed through a

RNN-T

softmax to predict the next character.

$$P_{RNN-T}(Y|X) = \sum_{A \in B^{-1}(Y)} P(A|X)$$
$$= \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t, y_{<u_t})$$

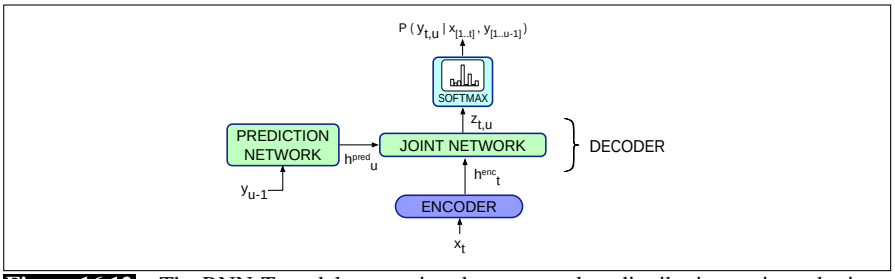


Figure 16.12 The RNN-T model computing the output token distribution at time t by integrating the output of a CTC acoustic encoder and a separate 'predictor' language model.

16.5 ASR Evaluation: Word Error Rate

word error

The standard evaluation metric for speech recognition systems is the **word error** rate. The word error rate is based on how much the word string returned by the recognizer (the **hypothesized** word string) differs from a **reference** transcription. The first step in computing word error is to compute the **minimum edit distance** in words between the hypothesized and correct strings, giving us the minimum number of word **substitutions**, word **insertions**, and word **deletions** necessary to map between the correct and hypothesized strings. The word error rate (WER) is then defined as follows (note that because the equation includes insertions, the error rate can be greater than 100%):

$$\text{Word Error Rate} = 100 \times \frac{\text{Insertions} + \text{Substitutions} + \text{Deletions}}{\text{Total Words in Correct Transcript}}$$

alignment

Here is a sample **alignment** between a reference and a hypothesis utterance from the CallHome corpus, showing the counts used to compute the error rate:

REF:	i	***	**	UM	the	PHONE	IS		i	LEFT	THE	portable	****	PHONE	UPSTAIRS	last night
HYP:	i	GOT	IT	TO	the	*****	FULLEST		i	LOVE	TO	portable	FORM	OF	STORES	last night
Eval:		I	I	S		D	S			S	S		I	S	S	

This utterance has six substitutions, three insertions, and one deletion:

$$\text{Word Error Rate} = 100\frac{6+3+1}{13} = 76.9\%$$

The standard method for computing word error rates is a free script called **sclite**, available from the National Institute of Standards and Technologies (NIST) (NIST,

2005). Sclite is given a series of reference (hand-transcribed, gold-standard) sentences and a matching set of hypothesis sentences. Besides performing alignments, and computing word error rate, sclite performs a number of other useful tasks. For example, for **error analysis** it gives useful information such as confusion matrices showing which words are often misrecognized for others, and summarizes statistics of words that are often inserted or deleted. `sclite` also gives error rates by speaker (if sentences are labeled for speaker ID), as well as useful statistics like the **sentence error rate**, the percentage of sentences with at least one word error.

Sentence error rate

Statistical significance for ASR: MAPSSWE or MacNemar

As with other language processing algorithms, we need to know whether a particular improvement in word error rate is significant or not.

The standard statistical tests for determining if two word error rates are different is the Matched-Pair Sentence Segment Word Error (MAPSSWE) test, introduced in Gillick and Cox (1989).

The MAPSSWE test is a parametric test that looks at the difference between the number of word errors the two systems produce, averaged across a number of segments. The segments may be quite short or as long as an entire utterance; in general, we want to have the largest number of (short) segments in order to justify the normality assumption and to maximize power. The test requires that the errors in one segment be statistically independent of the errors in another segment. Since ASR systems tend to use trigram LMs, we can approximate this requirement by defining a segment as a region bounded on both sides by words that both recognizers get correct (or by turn/utterance boundaries). Here's an example from NIST (2007) with four regions:

```
         I             II             III                IV
REF:  |it was|the best|of|times it|was the worst|of times|  |it was
      |      |        |  |        |             |        |  |
SYS A:|ITS   |the best|of|times it|IS the worst |of times|OR|it was
      |      |        |  |        |             |        |  |
SYS B:|it was|the best|  |times it|WON the TEST |of times|  |it was
```

In region I, system A has two errors (a deletion and an insertion) and system B has zero; in region III, system A has one error (a substitution) and system B has two. Let's define a sequence of variables Z representing the difference between the errors in the two systems as follows:

N_A^i the number of errors made on segment i by system A
N_B^i the number of errors made on segment i by system B
Z $N_A^i - N_B^i, i = 1, 2, \cdots, n$ where n is the number of segments

In the example above, the sequence of Z values is $\{2, -1, -1, 1\}$. Intuitively, if the two systems are identical, we would expect the average difference, that is, the average of the Z values, to be zero. If we call the true average of the differences mu_z, we would thus like to know whether $mu_z = 0$. Following closely the original proposal and notation of Gillick and Cox (1989), we can estimate the true average from our limited sample as $\hat{\mu}_z = \sum_{i=1}^{n} Z_i/n$. The estimate of the variance of the Z_i's is

$$\sigma_z^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Z_i - \mu_z)^2 \tag{16.21}$$

Let

$$W = \frac{\hat{\mu}_z}{\sigma_z/\sqrt{n}} \tag{16.22}$$

For a large enough n (> 50), W will approximately have a normal distribution with unit variance. The null hypothesis is $H_0 : \mu_z = 0$, and it can thus be rejected if $2 * P(Z \geq |w|) \leq 0.05$ (two-tailed) or $P(Z \geq |w|) \leq 0.05$ (one-tailed), where Z is standard normal and w is the realized value W; these probabilities can be looked up in the standard tables of the normal distribution.

McNemar's test

Earlier work sometimes used **McNemar's test** for significance, but McNemar's is only applicable when the errors made by the system are independent, which is not true in continuous speech recognition, where errors made on a word are extremely dependent on errors made on neighboring words.

Could we improve on word error rate as a metric? It would be nice, for example, to have something that didn't give equal weight to every word, perhaps valuing content words like *Tuesday* more than function words like *a* or *of*. While researchers generally agree that this would be a good idea, it has proved difficult to agree on a metric that works in every application of ASR. For dialogue systems, however, where the desired semantic output is more clear, a metric called *slot error rate* or *concept error rate* has proved extremely useful; it is discussed in Chapter 15 on page 323.

16.6 TTS

The goal of text-to-speech (TTS) systems is to map from strings of letters to waveforms, a technology that's important for a variety of applications from dialogue systems to games to education.

Like ASR systems, TTS systems are generally based on the encoder-decoder architecture, either using LSTMs or Transformers. There is a general difference in training. The default condition for ASR systems is to be speaker-independent: they are trained on large corpora with thousands of hours of speech from many speakers because they must generalize well to an unseen test speaker. By contrast, in TTS, it's less crucial to use multiple voices, and so basic TTS systems are speaker-dependent: trained to have a consistent voice, on much less data, but all from one speaker. For example, one commonly used public domain dataset, the LJ speech corpus, consists of 24 hours of one speaker, Linda Johnson, reading audio books in the LibriVox project (Ito and Johnson, 2017), much smaller than standard ASR corpora which are hundreds or thousands of hours.[2]

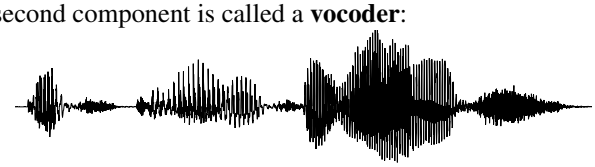
We generally break up the TTS task into two components. The first component is an encoder-decoder model for **spectrogram prediction**: it maps from strings of letters to mel spectrographs: sequences of mel spectral values over time. Thus we

[2] There is also recent TTS research on the task of **multi-speaker** TTS, in which a system is trained on speech from many speakers, and can switch between different voices.

might map from this string:

`It's time for lunch!`

to the following mel spectrogram:



The second component maps from mel spectrograms to waveforms. Generating waveforms from intermediate representations like spectrograms is called **vocoding** and this second component is called a **vocoder**:

vocoding
vocoder

These standard encoder-decoder algorithms for TTS are still quite computationally intensive, so a significant focus of modern research is on ways to speed them up.

16.6.1 TTS Preprocessing: Text normalization

Before either of these two steps, however, TTS systems require text normalization preprocessing for handling **non-standard words**: numbers, monetary amounts, dates, and other concepts that are verbalized differently than they are spelled. A TTS system seeing a number like *151* needs to know to verbalize it as *one hundred fifty one* if it occurs as *$151* but as *one fifty one* if it occurs in the context *151 Chapultepec Ave.*. The number *1750* can be spoken in at least four different ways, depending on the context:

non-standard words

`seventeen fifty:` (in *"The European economy in 1750"*)
`one seven five zero:` (in *"The password is 1750"*)
`seventeen hundred and fifty:` (in *"1750 dollars"*)
`one thousand, seven hundred, and fifty:` (in *"1750 dollars"*)

Often the verbalization of a non-standard word depends on its meaning (what Taylor (2009) calls its **semiotic class**). Fig. 16.13 lays out some English non-standard word types.

Many classes have preferred realizations. A year is generally read as paired digits (e.g., `seventeen fifty` for 1750). *$3.2 billion* must be read out with the word `dollars` at the end, as `three point two billion dollars`. Some abbreviations like *N.Y.* are expanded (to `New York`), while other acronyms like *GPU* are pronounced as letter sequences. In languages with grammatical gender, normalization may depend on morphological properties. In French, the phrase *1 mangue* ('one mangue') is normalized to `une mangue`, but *1 ananas* ('one pineapple') is normalized to `un ananas`. In German, *Heinrich IV* ('Henry IV') can be normalized to `Heinrich der Vierte`, `Heinrich des Vierten`, `Heinrich dem Vierten`, or `Heinrich den Vierten` depending on the grammatical case of the noun (Demberg, 2006).

semiotic class	examples	verbalization
abbreviations	**gov't**, *N.Y., mph*	government
acronyms read as letters	**GPU**, *D.C., PC, UN, IBM*	G P U
cardinal numbers	**12**, *45, 1/2, 0.6*	twelve
ordinal numbers	*May* **7**, *3rd, Bill Gates III*	seventh
numbers read as digits	*Room* **101**	one oh one
times	*3.20,* **11:45**	eleven forty five
dates	**28/02** *(or in US, 2/28)*	February twenty eighth
years	**1999**, *80s, 1900s, 2045*	nineteen ninety nine
money	**$3.45**, *€250, $200K*	three dollars forty five
money in tr/m/billions	**$3.45 billion**	three point four five billion dollars
percentage	**75%** *3.4%*	seventy five percent

Figure 16.13 Some types of non-standard words in text normalization; see Sproat et al. (2001) and (van Esch and Sproat, 2018) for many more.

Modern end-to-end TTS systems can learn to do some normalization themselves, but TTS systems are only trained on a limited amount of data (like the 220,000 words we mentioned above for the LJ corpus (Ito and Johnson, 2017)), and so a separate normalization step is important.

Normalization can be done by rule or by an encoder-decoder model. Rule-based normalization is done in two stages: tokenization and verbalization. In the tokenization stage we hand-write write rules to detect non-standard words. These can be regular expressions, like the following for detecting years:

/(1[89][0-9][0-9])|(20[0-9][0-9]/

A second pass of rules express how to verbalize each semiotic class. Larger TTS systems instead use more complex rule-systems, like the Kestral system of (Ebden and Sproat, 2015), which first classifies and parses each input into a normal form and then produces text using a verbalization grammar. Rules have the advantage that they don't require training data, and they can be designed for high precision, but can be brittle, and require expert rule-writers so are hard to maintain.

The alternative model is to use encoder-decoder models, which have been shown to work better than rules for such transduction tasks, but do require expert-labeled training sets in which non-standard words have been replaced with the appropriate verbalization; such training sets for some languages are available (Sproat and Gorman 2018, Zhang et al. 2019).

In the simplest encoder-decoder setting, we simply treat the problem like machine translation, training a system to map from:

```
They live at 224 Mission St.
```

to

```
They live at two twenty four Mission Street
```

While encoder-decoder algorithms are highly accurate, they occasionally produce errors that are egregious; for example normalizing *45 minutes* as *forty five millimeters*. To address this, more complex systems use mechanisms like lightweight **covering grammars**, which enumerate a large set of possible verbalizations but don't try to disambiguate, to constrain the decoding to avoid such outputs (Zhang et al., 2019).

16.6.2 TTS: Spectrogram prediction

The exact same architecture we described for ASR—the encoder-decoder with attention– can be used for the first component of TTS. Here we'll give a simplified overview

Tacotron2
Wavenet

of the **Tacotron2** architecture (Shen et al., 2018), which extends the earlier Tacotron (Wang et al., 2017) architecture and the **Wavenet** vocoder (van den Oord et al., 2016). Fig. 16.14 sketches out the entire architecture.

The encoder's job is to take a sequence of letters and produce a hidden representation representing the letter sequence, which is then used by the attention mechanism in the decoder. The Tacotron2 encoder first maps every input grapheme to a 512-dimensional character embedding. These are then passed through a stack of 3 convolutional layers, each containing 512 filters with shape 5×1, i.e. each filter spanning 5 characters, to model the larger letter context. The output of the final convolutional layer is passed through a biLSTM to produce the final encoding. It's common to use a slightly higher quality (but slower) version of attention called **location-based attention**, in which the computation of the α values (Eq. 9.37 in Chapter 13) makes use of the α values from the prior time-state.

location-based attention

In the decoder, the predicted mel spectrum from the prior time slot is passed through a small pre-net as a bottleneck. This prior output is then concatenated with the encoder's attention vector context and passed through 2 LSTM layers. The output of this LSTM is used in two ways. First, it is passed through a linear layer, and some output processing, to autoregressively predict one 80-dimensional log-mel filterbank vector frame (50 ms, with a 12.5 ms stride) at each step. Second, it is passed through another linear layer to a sigmoid to make a "stop token prediction" decision about whether to stop producing output.

Figure 16.14 The Tacotron2 architecture: An encoder-decoder maps from graphemes to mel spectrograms, followed by a vocoder that maps to wavefiles. Figure modified from Shen et al. (2018).

The system is trained on gold log-mel filterbank features, using teacher forcing, that is the decoder is fed the correct log-model spectral feature at each decoder step instead of the predicted decoder output from the prior step.

16.6.3 TTS: Vocoding

WaveNet

The vocoder for Tacotron 2 is an adaptation of the **WaveNet** vocoder (van den Oord et al., 2016). Here we'll give a somewhat simplified description of vocoding using WaveNet.

Recall that the goal of the vocoding process here will be to invert a log mel spectrum representations back into a time-domain waveform representation. WaveNet is an autoregressive network, like the language models we introduced in Chapter 9. It

takes spectrograms as input and produces audio output represented as sequences of 8-bit mu-law (page 575). The probability of a waveform , a sequence of 8-bit mu-law values $Y = y_1, ..., y_t$, given an intermediate input mel spectrogram h is computed as:

$$p(Y) = \prod_{t=1}^{t} P(y_t | y_1, ..., y_{t-1}, h_1, ..., h_t) \tag{16.23}$$

This probability distribution is modeled by a stack of special convolution layers, which include a specific convolutional structure called **dilated convolutions**, and a specific non-linearity function.

dilated convolutions

A dilated convolution is a subtype of **causal** convolutional layer. Causal or masked convolutions look only at the past input, rather than the future; the prediction of y_{t+1} can only depend on $y_1, ..., y_t$, useful for autoregressive left-to-right processing. In **dilated convolutions**, at each successive layer we apply the convolutional filter over a span longer than its length by skipping input values. Thus at time t with a dilation value of 1, a convolutional filter of length 2 would see input values x_t and x_{t-1}. But a filter with a distillation value of 2 would skip an input, so would see input values x_t and x_{t-1}. Fig. 16.15 shows the computation of the output at time t with 4 dilated convolution layers with dilation values, 1, 2, 4, and 8.

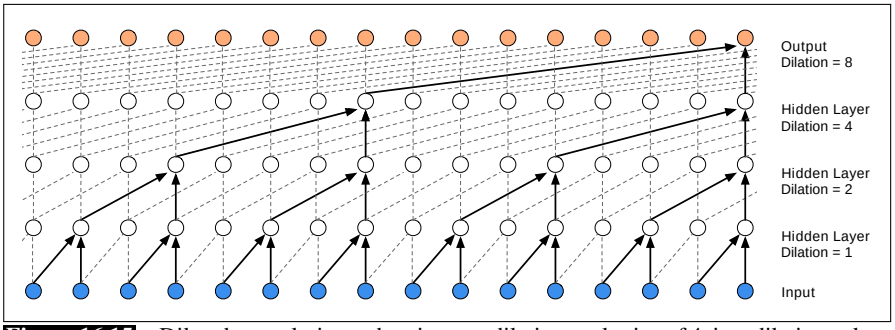


Figure 16.15 Dilated convolutions, showing one dilation cycle size of 4, i.e., dilation values of 1, 2, 4, 8. Figure from van den Oord et al. (2016).

The Tacotron 2 synthesizer uses 12 convolutional layers in two cycles with a dilation cycle size of 6, meaning that the first 6 layers have dilations of 1, 2, 4, 8, 16, and 32. and the next 6 layers again have dilations of 1, 2, 4, 8, 16, and 32. Dilated convolutions allow the vocoder to grow the receptive field exponentially with depth.

WaveNet predicts mu-law audio samples. Recall from page 575 that this is a standard compression for audio in which the values at each sampling timestep are compressed into 8-bits. This means that we can predict the value of each sample with a simple 256-way categorical classifier. The output of the dilated convolutions is thus passed through a softmax which makes this 256-way decision.

The spectrogram prediction encoder-decoder and the WaveNet vocoder are trained separately. After the spectrogram predictor is trained, the spectrogram prediction network is run in teacher-forcing mode, with each predicted spectral frame conditioned on the encoded text input and the previous frame from the ground truth spectrogram. This sequence of ground truth-aligned spectral features and gold audio output is then used to train the vocoder.

This has been only a high-level sketch of the TTS process. There are numerous important details that the reader interested in going further with TTS may want

to look into. For example WaveNet uses a special kind of a gated activation function as its non-linearity, and contains residual and skip connections. In practice, predicting 8-bit audio values doesn't as work as well as 16-bit, for which a simple softmax is insufficient, so decoders use fancier ways as the last step of predicting audio sample values, like mixtures of distributions. Finally, the WaveNet vocoder as we have described it would be so slow as to be useless; many different kinds of efficiency improvements are necessary in practice, for example by finding ways to do non-autoregressive generation, avoiding the latency of having to wait to generate each frame until the prior frame has been generated, and instead making predictions in parallel. We encourage the interested reader to consult the original papers and various version of the code.

16.6.4 TTS Evaluation

Speech synthesis systems are evaluated by human listeners. (The development of a good automatic metric for synthesis evaluation, one that would eliminate the need for expensive and time-consuming human listening experiments, remains an open and exciting research topic.)

MOS

We evaluate the quality of synthesized utterances by playing a sentence to listeners and ask 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).

AB tests

If we are comparing exactly two systems (perhaps to see if a particular change actually improved the system), we can use **AB tests**. In AB tests, we play the same sentence synthesized by two different systems (an A and a B system). 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.

16.7 Other Speech Tasks

While we have focused on speech recognition and TTS in this chapter, there are a wide variety of speech-related tasks.

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.

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 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 . These tasks are related to **language identification**, in which we are given a wavefile and must identify which language is being spoken; this is useful for example for automatically directing callers to human operators that speak appropriate languages.

language identification

16.8 Summary

This chapter introduced the fundamental algorithms of automatic speech recognition (ASR) and text-to-speech (TTS).

- The task of **speech recognition** (or speech-to-text) is to map acoustic waveforms to sequences of graphemes.
- The input to a speech recognizer is a series of acoustic waves. that are **sampled**, **quantized**, and converted to a **spectral representation** like the **log mel spectrum**.
- Two common paradigms for speech recognition are the **encoder-decoder with attention** model, and models based on the **CTC loss function**. Attention-based models have higher accuracies, but models based on CTC more easily adapt to **streaming**: outputting graphemes online instead of waiting until the acoustic input is complete.
- ASR is evaluated using the Word Error Rate; the edit distance between the hypothesis and the gold transcription.
- **TTS** systems are also based on the **encoder-decoder** architecture. The encoder maps letters to an encoding, which is consumed by the decoder which generates **mel spectrogram** output. A neural **vocoder** then reads the spectrogram and generates waveforms.
- TTS systems require a first pass of **text normalization** to deal with numbers and abbreviations and other non-standard words.
- TTS is evaluated by playing a sentence to human listeners and having them give a **mean opinion score (MOS)** or by doing AB tests.

Bibliographical and Historical Notes

ASR A number of speech recognition systems were developed by the late 1940s and early 1950s. An early Bell Labs system could recognize any of the 10 digits from a single speaker (Davis et al., 1952). This system had 10 speaker-dependent stored patterns, one for each digit, each of which roughly represented the first two vowel formants in the digit. They achieved 97%–99% accuracy by choosing the pattern that had the highest relative correlation coefficient with the input. Fry (1959) and Denes (1959) built a phoneme recognizer at University College, London, that recognized four vowels and nine consonants based on a similar pattern-recognition principle. Fry and Denes's system was the first to use phoneme transition probabilities to constrain the recognizer.

The late 1960s and early 1970s produced a number of important paradigm shifts. First were a number of feature-extraction algorithms, including the efficient fast Fourier transform (FFT) (Cooley and Tukey, 1965), the application of cepstral processing to speech (Oppenheim et al., 1968), and the development of LPC for speech coding (Atal and Hanauer, 1971). Second were a number of ways of handling **warping**; stretching or shrinking the input signal to handle differences in speaking rate and segment length when matching against stored patterns. The natural algorithm for solving this problem was dynamic programming, and, as we saw in Appendix A, the algorithm was reinvented multiple times to address this problem. The first application to speech processing was by Vintsyuk (1968), although his result was not picked up by other researchers, and was reinvented by Velichko and Zagoruyko (1970) and Sakoe and Chiba (1971) (and 1984). Soon afterward, Itakura (1975) combined this dynamic programming idea with the LPC coefficients that had previously been used only for speech coding. The resulting system extracted LPC features from incoming words and used dynamic programming to match them against stored LPC templates. The non-probabilistic use of dynamic programming to match a template against incoming speech is called **dynamic time warping**.

warping

dynamic time warping

The third innovation of this period was the rise of the HMM. Hidden Markov models seem to have been applied to speech independently at two laboratories around 1972. One application arose from the work of statisticians, in particular Baum and colleagues at the Institute for Defense Analyses in Princeton who applied HMMs to various prediction problems (Baum and Petrie 1966, Baum and Eagon 1967). James Baker learned of this work and applied the algorithm to speech processing (Baker, 1975a) during his graduate work at CMU. Independently, Frederick Jelinek and collaborators (drawing from their research in information-theoretical models influenced by the work of Shannon (1948)) applied HMMs to speech at the IBM Thomas J. Watson Research Center (Jelinek et al., 1975). One early difference was the decoding algorithm; Baker's DRAGON system used Viterbi (dynamic programming) decoding, while the IBM system applied Jelinek's stack decoding algorithm (Jelinek, 1969). Baker then joined the IBM group for a brief time before founding the speech-recognition company Dragon Systems.

The use of the HMM, with Gaussian Mixture Models (GMMs) as the phonetic component, slowly spread through the speech community, becoming the dominant paradigm by the 1990s. One cause was encouragement by ARPA, the Advanced Research Projects Agency of the U.S. Department of Defense. ARPA started a five-year program in 1971 to build 1000-word, constrained grammar, few speaker speech understanding (Klatt, 1977), and funded four competing systems of which Carnegie-Mellon University's Harpy system (Lowerre, 1968), which used a simplified version of Baker's HMM-based DRAGON system was the best of the tested systems. ARPA (and then DARPA) funded a number of new speech research programs, beginning with 1000-word speaker-independent read-speech tasks like "Resource Management" (Price et al., 1988), recognition of sentences read from the *Wall Street Journal* (WSJ), Broadcast News domain (LDC 1998, Graff 1997) (transcription of actual news broadcasts, including quite difficult passages such as on-the-street interviews) and the Switchboard, CallHome, CallFriend, and Fisher domains (Godfrey et al. 1992, Cieri et al. 2004) (natural telephone conversations between friends or strangers). Each of the ARPA tasks involved an approximately annual **bakeoff** at which systems were evaluated against each other. The ARPA competitions resulted in wide-scale borrowing of techniques among labs since it was easy to see which ideas reduced errors the previous year, and the competitions were probably an im-

bakeoff

portant factor in the eventual spread of the HMM paradigm.

By around 1990 neural alternatives to the HMM/GMM architecture for ASR arose, based on a number of earlier experiments with neural networks for phoneme recognition and other speech tasks. Architectures included the time-delay neural network (**TDNN**)—the first use of convolutional networks for speech— (Waibel et al. 1989, Lang et al. 1990), RNNs (Robinson and Fallside, 1991), and the **hybrid** HMM/MLP architecture in which a feedforward neural network is trained as a phonetic classifier whose outputs are used as probability estimates for an HMM-based architecture (Morgan and Bourlard 1990, Bourlard and Morgan 1994, Morgan and Bourlard 1995).

hybrid

While the hybrid systems showed performance close to the standard HMM/GMM models, the problem was speed: large hybrid models were too slow to train on the CPUs of that era. For example, the largest hybrid system, a feedforward network, was limited to a hidden layer of 4000 units, producing probabilities over only a few dozen monophones. Yet training this model still required the research group to design special hardware boards to do vector processing (Morgan and Bourlard, 1995). A later analytic study showed the performance of such simple feedforward MLPs for ASR increases sharply with more than 1 hidden layer, even controlling for the total number of parameters (Maas et al., 2017). But the computational resources of the time were insufficient for more layers.

Over the next two decades a combination of Moore's law and the rise of GPUs allowed deep neural networks with many layers. Performance was getting close to traditional systems on smaller tasks like TIMIT phone recognition by 2009 (Mohamed et al., 2009), and by 2012, the performance of hybrid systems had surpassed traditional HMM/GMM systems (Jaitly et al. 2012, Dahl et al. 2012, inter alia). Originally it seemed that unsupervised pretraining of the networks using a technique like deep belief networks was important, but by 2013, it was clear that for hybrid HMM/GMM feedforward networks, all that mattered was to use a lot of data and enough layers, although a few other components did improve performance: using log mel features instead of MFCCs, using dropout, and using rectified linear units (Deng et al. 2013, Maas et al. 2013, Dahl et al. 2013).

Meanwhile early work had proposed the CTC loss function by 2006 (Graves et al., 2006), and by 2012 the RNN-Transducer was defined and applied to phone recognition (Graves 2012, Graves et al. 2013), and then to end-to-end speech recognition rescoring (Graves and Jaitly, 2014), and then recognition (Maas et al., 2015), with advances such as specialized beam search (Hannun et al., 2014). (Our description of CTC in the chapter draws on Hannun (2017), which we encourage the interested reader to follow).

The encoder-decoder architecture was applied to speech at about the same time by two different groups, in the Listen Attend and Spell system of Chan et al. (2016) and the attention-based encoder decoder architecture of Chorowski et al. (2014) and Bahdanau et al. (2016). By 2018 Transformers were included in this encoder-decoder architecture. Karita et al. (2019) is a nice comparison of RNNs vs Transformers in encoder-architectures for ASR, TTS, and speech-to-speech translation.

Kaldi
ESPnet

Popular toolkits for speech processing include **Kaldi** (Povey et al., 2011) and **ESPnet** (Watanabe et al. 2018, Hayashi et al. 2020).

TTS 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.

Modern encoder-decoder systems are distant descendants of formant synthesizers. **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.

Exercises

16.1 Analyze each of the errors in the incorrectly recognized transcription of "um the phone is I left the. . . " on page 343. For each one, give your best guess as to whether you think it is caused by a problem in signal processing, pronunciation modeling, lexicon size, language model, or pruning in the decoding search.