Speech and Language Processing. Daniel Jurafsky & James H. Martin. Copyright © 2026. All rights reserved. Draft of August 19, 2026.

CHAPTER

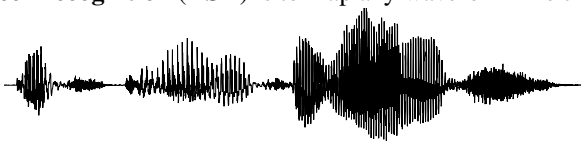
16 Automatic Speech Recognition

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 appliances, or with digital assistants or chatbots, especially on cellphones, where keyboards are less convenient. 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.

In the next sections we'll introduce the various goals of the ASR task, describe how acoustic features are extracted, and introduce the **convolutional neural net** architecture which is commonly used as an initial layer in speech recognition tasks.

We'll then introduce two families of methods for ASR. The first is the **encoder-decoder** paradigm, and we'll introduce the baseline attention-based encoder decoder algorithm, sometimes called Listen Attend and Spell after an early implementation. We'll also introduce a more advanced encoder-decoder system, OpenAI's Whisper system (Radford et al., 2023) as well an open system based on the same architecture, OWSM (the Open Whisper-style Speech Model) (Peng et al., 2023). (These models have additional capabilities including translation, as we'll discuss later). The second is the use of self-supervised speech models (sometimes called SSL for self-supervised learning) like Wav2Vec2.0 or HuBERT, which are encoders that learn abstract representations of speech that can be used for ASR by pairing them with the **CTC** loss function for decoding.

We'll conclude with the standard **word error rate** metric used to evaluate ASR.

16.1 The Automatic Speech Recognition Task

Before describing algorithms for ASR, let's talk about how the ASR task itself varies. One dimension of variation is vocabulary size. Some ASR tasks have long been 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 accurately transcribing videos or human conversations, with large vocabularies of 60,000 or more 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, which has volunteers read and record copyright-free books (Panayotov et al., 2015). It has transcripts aligned at the sentence level. 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. The division was done when the corpus was first released by running a speech recognizer (trained on read speech from the Wall Street Journal) on all the audio, computing the word error rate (WER, formally defined below) for each speaker based on the gold transcripts,

LibriSpeech

and dividing the speakers roughly in half, with recordings from lower-WER speakers called "clean" and recordings from higher-WER speakers "other".

Switchboard 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).

CALLHOME

CHiME A variety of corpora try to include input that is more natural. The **CHiME** Challenge is a series of difficult shared tasks with corpora that deal with robustness in ASR. The CHiME 6 task, for example, is ASR of conversational speech in real home environments (specifically dinner parties). The 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 with distant microphones.

AMI The **AMI** Meeting Corpus contains 100 hours of recorded group meetings (some natural meetings, some specially organized), with manual transcriptions and some additional hand-labels (Renals et al., 2007).

CORAAL **CORAAL** is a collection of over 150 sociolinguistic interviews with African American speakers, with the goal of studying African American English (**AAE**), the many variations of language used in African American communities and others (Kendall and Farrington, 2020). The interviews are anonymized with transcripts aligned at the utterance level.

There are a wide variety of corpora available in other languages. In Chinese, for example, the **HKUST** Mandarin Telephone Speech corpus has 1206 transcribed ten-minute telephone conversations between speakers of Mandarin across China including conversations between friends and between strangers (Liu et al., 2006). The **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).

HKUST

AISHELL-1

Finally, there are many multilingual corpora. Common Voice (Ardila et al., 2020) is a freely available crowd-sourced corpus of transcribed read speech, stored in MPEG-3 format and designed for ASR. Crowd-working volunteers record themselves reading scripted speech, with scripts often extracted from Wikipedia articles. The recordings are then verified by other contributors. As of the writing of this chapter, Common Voice includes 33,150 hours of speech from 133 languages. FLEURS (Conneau et al., 2023) is a parallel speech dataset, built on the MT benchmark FLoRes-101 (Goyal et al., 2022), which has 3001 sentences extracted from English Wikipedia and translated into 101 other languages by human translators. For a subset of 2009 of the sentences in each of the 102 languages, FLEURS has recordings of 3 different native speakers reading the sentence, in total about 12 hours of speech per language.

Figure 16.1 shows the rough percentage of incorrect words (the **word error rate**, or WER, defined on page 22) from roughly state-of-the-art systems as of the time of this writing on some of these tasks. Note that the error rate on English read speech (like the LibriSpeech clean audiobook corpus) is around 2% ; transcription of speech read in English is highly accurate. By contrast, the error rate for transcribing conversations between humans is higher; 5.8 to 11% for the Switchboard and CALLHOME corpora or AMI meetings. The error rate is higher yet again for speakers of varieties

like African American English, and yet again for difficult conversational tasks like transcription of 4-speaker dinner party speech, which can have error rates as high as 25.5%. Character error rates (CER) are also higher for Mandarin natural conversation than for Mandarin read speech. Error rates are even higher for lower resource languages; we've shown a handful of examples.

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
AMI meetings	11
Sociolinguistic interviews, CORAAL (AAE)	16.2
CHiME6 dinner parties with distant microphones	25.5
Sample tasks in other languages	**WER%**
Common Voice 15 Vietnamese	39.8
Common Voice 15 Swahili	51.2
FLEURS Bengali	50
Chinese (Mandarin) Tasks	**CER%**
AISHELL-1 Mandarin read speech corpus	3.9
HKUST Mandarin Chinese telephone conversations	18.5

Figure 16.1 Rough Word Error Rates (WER = % of words misrecognized) reported around 2023-4 for ASR on various American English and other language recognition tasks, and character error rates (CER) for two Chinese recognition tasks.

16.2 Convolutional Neural Networks

CNN The convolutional neural network, or **CNN** (and sometimes shortened as **convnet**), is a network architecture that is particularly useful for extracting features in speech and vision applications. A convolutional layer for speech takes as input a representation of the audio input (either as the raw audio or as Mel spectra) and produces as output a sequence of latent representations of the input speech. In ASR systems like Whisper, wav2vec2.0, or HuBERT, convolutional layers are stacked as an initial set of layers producing speech representations that are then passed to transformer layers.

A standard feedforward layer is fully connected; every input is connected to every output. By contrast, a convolutional network makes use of the idea of a kernel, a kind of smaller network that we pass over the input. For example in image classification tasks, we pass the kernel horizontally and vertically over the image to recognize visual features, and so we describe a visual as a 2d (for 2 dimensional) convolutional network. For speech, we will slide our kernel over the signal in the time dimension to extract speech features, so CNNs for speech are 1d convolutional networks.

Let's flesh out this intuition a bit more. We'll start with a very schematic version of a convolutional layer that takes as input a single sequence of vectors $\mathbf{x}_1 \dots \mathbf{x}_t$ and produces as output a single sequence of vectors $\mathbf{z}_1 \dots \mathbf{z}_t$, of the same length t. Afterwards we'll see how to deal with more complex inputs and outputs.

kernel **convolving** A CNN uses a **kernel**, a small vector of weights $\mathbf{w}_1 \dots \mathbf{w}_k$, to extract features. It does this by **convolving** this kernel with the input. The **convolution** of a kernel with

a signal has 3 steps:

1. Flip the kernel left-to-right
2. Pass the kernel frame by frame (temporally) across the input
   - At each frame computing the dot product of the kernel with the local input values
3. The output is the resulting sequence of dot products

We can think of the convolution process as finding regions in the signal that are similar to the kernel, since the dot product is high when two vectors are similar. The convolution operation is represented by the * operator (an unfortunate overloading of this symbol that also refers to simple multiplication). Let's see how to compute $\mathbf{x} * \mathbf{w}$, the convolution of a single vector $\mathbf{x}$ with a kernel vector $\mathbf{w}$. Let's first think about the simple case of a kernel width of 1. We compute each output element $\mathbf{z}_j$ as the product of the kernel with $\mathbf{x}_j$:

$$\textit{convolution with width-1 kernel:} \quad \mathbf{z}_j = \mathbf{x}_j \mathbf{w}_0 \quad \forall j : 1 \leq j \leq t \tag{16.1}$$

Fig. 16.2 shows an intuition of this computation.

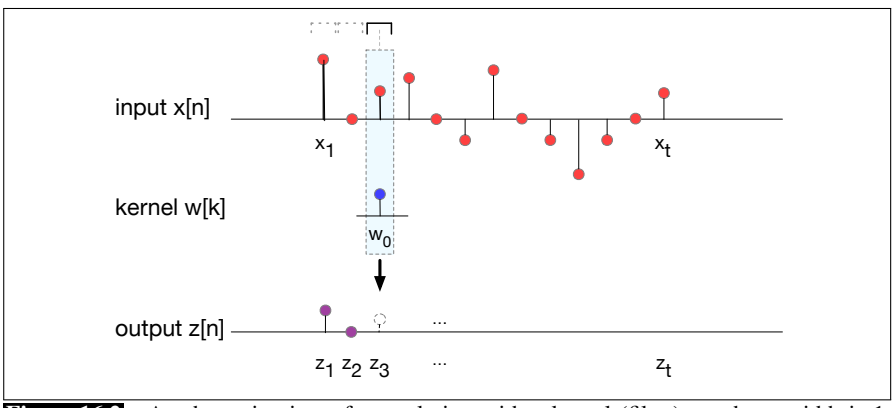


Figure 16.2 A schematic view of convolution with a kernel (filter) $\mathbf{w}$ whose width is 1. The kernel is walked across the input, and the output at each frame $\mathbf{z}_i$ is the dot product of the kernel with the input frame. With a kernel of length 1 we don't have to worry about flipping the kernel, and the dot product is just the scalar product. The figure shows the computation of $\mathbf{z}_3$ as $\mathbf{x}_3 \times \mathbf{w}_1$.

Let's now turn to longer kernels. Although we've described the first step of the convolution as flipping the kernel, in fact in ASR systems (or in component libraries like pytorch) we skip this step. Technically this means that the algorithm we are using is not in fact convolution, it's instead **cross-correlation**, which is the name for an algorithm of walking a kernel across a signal, computing its dot product frame by frame, without flipping it first. The difference doesn't matter, since the parameters of the kernel will be learned during training, and so the model could easily learn a kernel with the parameters in either order. Still, for historical reasons we still call this process a 1d convolution rather than cross-correlation.

cross-correlation

Let's see a more general equation for these longer kernels. To avoid the convolution being undefined at the left and right edges of the signal, we can **pad** the input by adding a small number p of zeros at the beginning and end of the signal, so that we can start the center of the kernel at the first element $\mathbf{x}_1$, and there will be a defined value to the left of $\mathbf{x}_1$. This also turns out to make it simple to have the

pad

output length as the same as the input length. To do this, it's convenient to define the kernel vector as having an odd number of elements of length $k = 2p + 1$, thus with the center element having p elements on either side. Each element z_j of the output vector $\mathbf{z}$ is then computed as the following dot product:

$$\mathbf{z}_j = \sum_{i=-p}^{p} \mathbf{x}_{j+i}\mathbf{w}_{i+p} \tag{16.2}$$

Fig. 16.3 shows the computation of the convolution $\mathbf{x} * \mathbf{w}$ with a kernel whose width is 3, and with padding of 1 frame at the beginning and end of $\mathbf{x}$ with a value of zero.

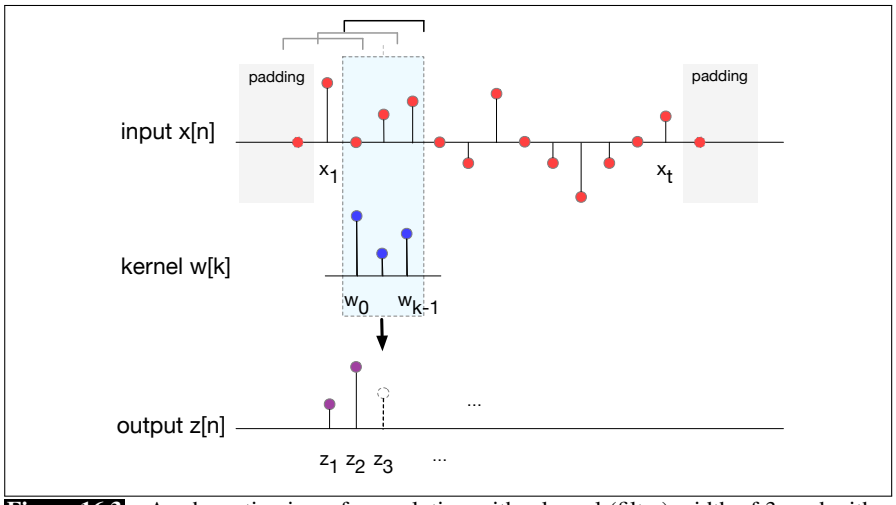


Figure 16.3 A schematic view of convolution with a kernel (filter) width of 3, and with a padding of 1, showing a zero value added at the start and end of the signal. The (already flipped) kernel is walked across the input, and the output at each frame $\mathbf{z}_i$ is the dot product of the kernel with the input in the window. The figure shows the computation of $\mathbf{z}_3$.

Note that the size k (the **receptive field**) of the kernel is designed to be small compared to the signal. For example for the convolutional layers in Whisper, the kernel width is 3 frames, meaning the kernel is a vector of length 3 (we say that the kernel has a **receptive field** of 3). That means that the kernel is being compared to 3 frames of speech. In Whisper there is a frame every 10 ms and each frame represent a window of 25ms of speech information. That means each kernel is extracting information from about 45 ms of speech (10 + 10 + 12.5 + 12.5). That's long enough to extract various phonetic features like formant transitions or stop closures or aspiration.

receptive field

We've now described a simplified view of convolution in which the input is a single vector $\mathbf{x}$ and the output is a single vector $\mathbf{z}$, both corresponding to a signal over time. In practice, the input to a convolutional layer is commonly the output from a log mel spectrum, which means it has many (say 128) channels, one for each log mel filters output. The kernel will have separate vectors for each of these input channels. We say that the kernel has a **depth** of 128, meaning that the kernel is of shape [128,3].

depth

To get the output of the kernel, we sum over all the input channels. That is, we get a single output $\mathbf{z}^c$ for each of the input channels $\mathbf{x}^c$ by convolving the kernel $\mathbf{w}$

with it, and then we sum up all the resulting outputs:

$$\mathbf{z} = \sum_{c=1}^{C_i} \mathbf{x}^c * \mathbf{w} \tag{16.3}$$

The output at frame j, $\mathbf{z}_j$, thus integrates information from all of the input channels.

Finally, the output from a convolution layer is also more complex than just a vector consisting of a single scalar value to represent each frame. Instead, the output of the convolution layer for a given input frame needs to be an embedding, a latent representation of that frame. As with all neural models, latent representations should have the model dimensionality, whatever that is. For example the model dimensionality of Whisper is 1280, and so the convolutional layer needs to have one output channel for each of these 1280 dimensions of the model. In order to do this, we'll actually learn one separate kernel for each of the model dimensions. That is, we'll learn 1280 separate kernels, each kernel having the depth of the number of input channels (for example 128), and a filter-width (say of 3). That way, the embedding representation of each frame will have 1280 independently computed features of the input signal. We show a schematic in Fig. 16.4

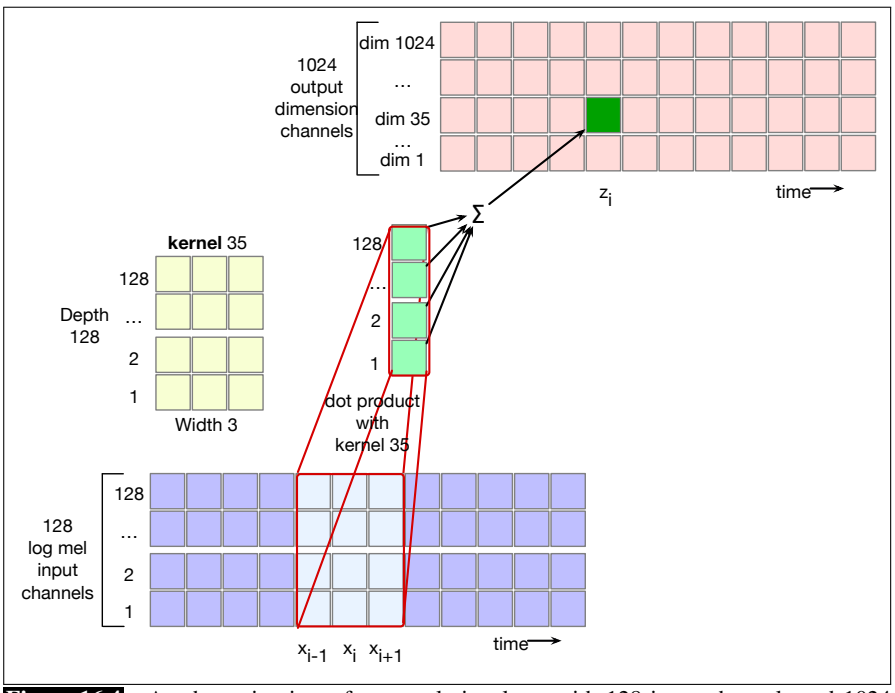


Figure 16.4 A schematic view of a convolutional net with 128 input channels and 1024 output channels. We see how at time point i one of the 1024 kernels ("kernel 35", each of depth 128 and width 3) is dot-product-ed with (each of) the 128 log mel spectrum input vectors, and then summed to produce a single value for one dimension of the output embedding at time i.

stride A 1d convolution layer can also have a **stride**. Stride is the amount that we move the kernel over the input between each step. The figures above show a stride of 1, meaning that we first position the kernel over $\mathbf{x}_1$, then $\mathbf{x}_2$, then $\mathbf{x}_3$, and so on. For a stride of 2, we would first position the kernel over $\mathbf{x}_1$, then $\mathbf{x}_3$, then $\mathbf{x}_5$, and so on. A longer stride means a shorter output sequence; a stride of two means the output

sequence $\mathbf{z}$ will be half the length of the input sequence $\mathbf{x}$. Convolutional layers with strides greater than 1 are commonly used to shorten an input sequence. This is useful partly because a shorter signal takes less memory and computational bandwidth, but also, as we'll see in the next section, because it helps address the mismatch between the length of acoustic frame embeddings (10 ms) and letters or words, which cover much more of the signal.

Finally, in practice a convolutional layer can be followed by an output nonlinearity, like a ReLU layer.

16.3 The Encoder-Decoder Architecture for ASR

AED
listen attend and spell

The first ASR architecture we introduce is the **encoder-decoder** architecture, the same architecture introduced for MT in Chapter 13. Fig. 16.5 sketches this architecture, called **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 to the architecture $\mathbf{x}$ is a sequence of t acoustic feature vectors $\mathbf{X} = \mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_t$, one vector per 10 ms frame. We often start from the log mel spectral features described in the previous section, although it's also possible to start from a raw wavefile. The output sequence Y can be either letters or tokens (BPE or sentencepiece); we'll assume letters just to simplify the explanation 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:

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

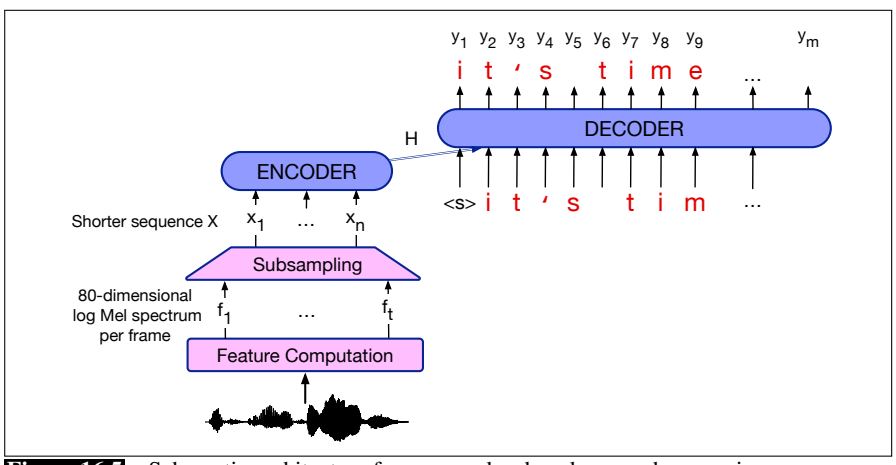


Figure 16.5 Schematic architecture for an encoder-decoder speech recognizer.

This architecture is also used in the Whisper model from OpenAI (Radford et al., 2023). Fig. 16.6 shows a subpart of the Whisper architecture (Whisper also does other speech tasks like speech translation and voice activity detection, which we'll discuss in the next chapter). Whisper models and inference code are publicly released, but the training code and training data are not. However, there are open-source projects that use a Whisper-style architecture, like the Open Whisper-style

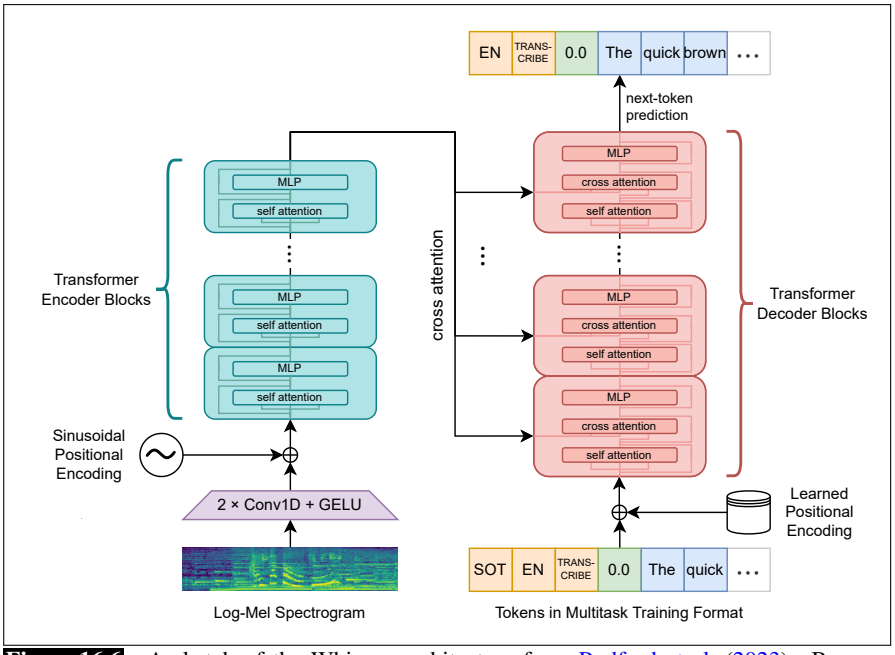


Figure 16.6 A sketch of the Whisper architecture from Radford et al. (2023). Because Whisper is a multitask system that also does translation, Whisper's transcription format has a Start of Transcript (SOT) token, a language tag, and then an instruction token for whether to transcribe or translate.

Speech Model (OWSM), which reproduces Whisper-style training but offers a fully open-source toolkit and publicly available data (Peng et al., 2023).

16.3.1 Input and Convolutional Layers

The encoder-decoder architecture is particularly appropriate when input and output sequences have stark length differences, as they do for speech, with long acoustic feature sequences mapping to much shorter sequences of letters or words. For example English, words are on average 5 letters or 1.3 BPE tokens long (Bostrom and Durrett, 2020) and, in natural conversation, the average word lasts about 250 milliseconds (Yuan et al., 2006), or 25 frames of 10ms. So the speech signal in 10ms frames is about 5 (25/5) to 19 (25/1.3) times longer than the text signal in words or tokens.

Because this length difference is so extreme for speech, encoder-decoder architectures for speech usually have a compression stage that shortens the acoustic feature sequence before the encoder stage. (We can additionally make use of a loss function that is designed to deal well with compression, like the CTC loss function we'll introduce later.)

The goal of the subsampling is to produce a shorter sequence $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ that will be the input to the transformer encoder. A very simple baseline 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 $n = \frac{t}{3}$.

low frame rate

But the most common way of creating a shorter input sequence is to use the convolutional layers we introduced in the previous section. When a convolutional layer has a stride greater than 1, the output sequence becomes shorter than the input sequence. Let's see this in two commonly used ASR systems.

The Whisper system (Radford et al., 2023) has an audio context window of 30 seconds. It extracts 128 channel log mel features for each frame, with a 25ms window and a stride of 10ms. These are then normalized to 0 mean and a range of -1 to 1. A stride of 10 ms (100 frames per second) means there are 3000 input frames in a 30 second context window. These 3000 frames are passed to two convolutional layers, each one followed by a nonlinearity (Whisper uses GELU (Gaussian Error Linear Unit), which is a smoother version of ReLU). The first convolutional layer has 128 input channels and uses a stride of 1, with number of output channels being the model dimensionality, and the window length is 3000. For the second convolutional layer the number of input and output channels is the model dimensionality, and there is a stride of 2. The stride of 2 in the second convolutional layer makes the output sequence half the length of the input sequence, bringing the output window length down to 1500 and producing an audio token every 20 ms. Sinusoidal position embeddings are added to these audio encodings before the output of this front end is passed to the transformer encoder.

HuBERT (Hsu et al., 2021) uses an alternative front end architecture, in which convolutional layers are used to completely replace the computation of the spectrum. So the input is raw 16kHz sampled audio, and it is passed through seven 512-channel layers with strides [5,2,2,2,2,2,2] and kernel widths [10,3,3,3,3,2,2] which learn both to extract spectral information, and to shorten the input sequence by 320x, from 16kHz (= one representation per .0625 ms) down to a 20 ms framerate. Positional encodings are added to the input, and then a GELU and layer norm are applied before the output is passed to the transformer encoder.

16.3.2 Inference

After the convolutional stage, encoder-decoders for speech use the same architecture (transformer with cross-attention) as for MT.

Let's remind ourselves of the encoder-decoder architecture that we introduced in Chapter 13. It uses two transformers: an **encoder**, which is the same as the basic transformer from Chapter 7, and a **decoder**, which has one addition: a new layer called the **cross-attention** layer. The encoder takes the acoustic input $\mathbf{X} = \mathbf{x}_1, ..., \mathbf{x}_n$ and maps them to an output representation $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$; via a stack of encoder blocks.

The decoder is essentially a conditional language model that attends to the encoder representation and generates the target text (letters or tokens) one by one, at each timestep conditioning on the audio representations from the encoder and the previously generated text to generate a new letter or token.

cross-attention

The transformer blocks in the decoder have an extra layer with a special kind of attention, **cross-attention**. Cross-attention has the same form as the multi-head attention in a normal transformer block, except that while the queries as usual come from the previous layer of the decoder, the keys and values come from the output of the *encoder*.

That is, where in standard multi-head attention the input to each attention layer is $\mathbf{X}$, in cross attention the input is the final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$. $\mathbf{H}^{enc}$ is of shape $[n \times d]$, each row representing one acoustic input token. To link the keys and values from the encoder with the query from the prior layer of the decoder,

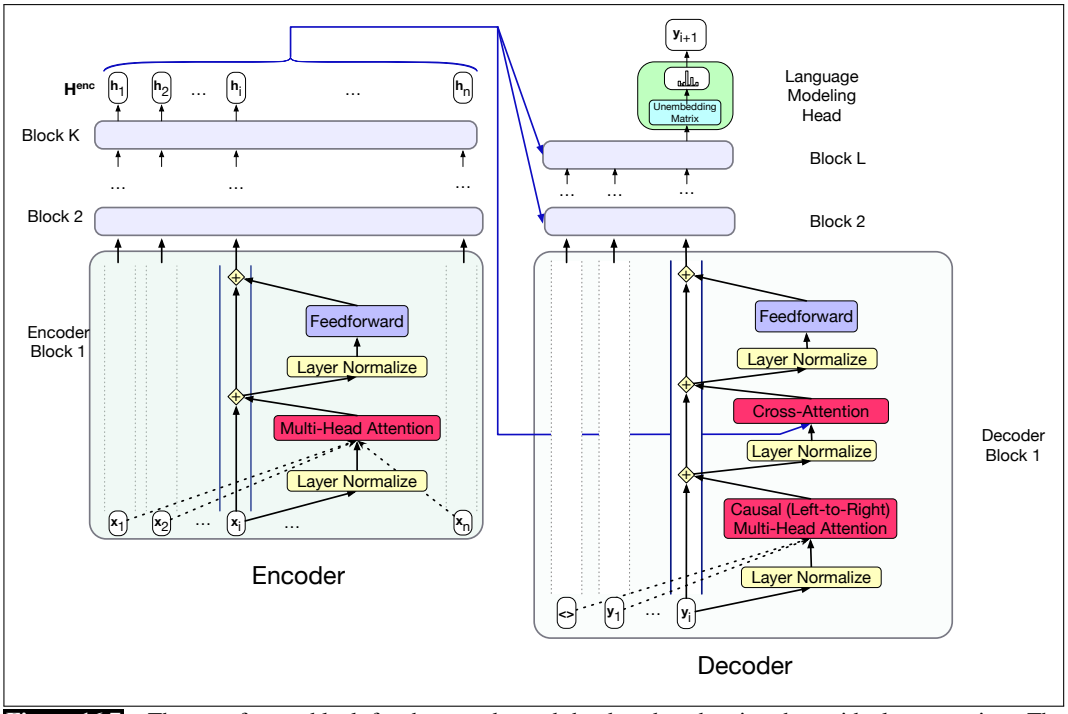


Figure 16.7 The transformer block for the encoder and the decoder, showing the residual stream view. The final output of the encoder $\mathbf{H}^{enc} = \mathbf{h}_1, ..., \mathbf{h}_n$ is the context used in the decoder. The decoder is a standard transformer except with one extra layer, the **cross-attention** layer, which takes that encoder output $\mathbf{H}^{enc}$ and uses it to form its $\mathbf{K}$ and $\mathbf{V}$ inputs.

we multiply the encoder output $\mathbf{H}^{enc}$ by the cross-attention layer's key weights $\mathbf{W^K}$ and value weights $\mathbf{W^V}$. The query comes from the output from the prior decoder layer $\mathbf{H}^{dec[\ell-1]}$, which is multiplied by the cross-attention layer's query weights $\mathbf{W^Q}$:

$$\mathbf{Q} = \mathbf{H}^{dec[\ell-1]}\mathbf{W^Q}; \;\; \mathbf{K} = \mathbf{H}^{enc}\mathbf{W^K}; \;\; \mathbf{V} = \mathbf{H}^{enc}\mathbf{W^V} \tag{16.4}$$

$$\text{CrossAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \;=\; \text{softmax}\left(\frac{\mathbf{QK}^\intercal}{\sqrt{d_k}}\right)\mathbf{V} \tag{16.5}$$

The cross attention thus allows the decoder to attend to the acoustic input as projected into the entire encoder final output representations. The other attention layer in each decoder block, the multi-head attention layer, is the same causal (left-to-right) attention that we saw in Chapter 7. But the multi-head attention in the encoder, however, is allowed to look ahead at the entire source audio, so it is not masked.

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

$$p(y_1, \ldots, y_n) = \prod_{i=1}^{n} p(y_i | y_1, \ldots, y_{i-1}, \mathbf{X}) \tag{16.6}$$

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

$$\hat{y}_i \;=\; \text{argmax}_{\text{char} \in \text{ Alphabet}} P(\text{char} | y_1 ... y_{i-1}, \mathbf{X}) \tag{16.7}$$

Alternatively encoder-decoders like Whisper or OWSM also 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.

n-best list
rescore

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 interpolating 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 the scoring function for Listen, Attend, and Spell (Chan et al., 2016):

$$\text{score}(Y|\mathbf{X}) = \frac{1}{|Y|_c} \log P(Y|\mathbf{X}) + \lambda \log P_{LM}(Y) \tag{16.8}$$

16.3.3 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:

$$L_{CE} = -\log p(\mathbf{y}_i|y_1,\ldots,\mathbf{y}_{i-1},\mathbf{X}) \tag{16.9}$$

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

$$L_{CE} = -\sum_{i=1}^{m} \log p(\mathbf{y}_i|\mathbf{y}_1,\ldots,\mathbf{y}_{i-1},\mathbf{X}) \tag{16.10}$$

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:

$$L_{CE} = -\log p(\mathbf{y}_i|\mathbf{y}_1,\ldots,\hat{\mathbf{y}}_{i-1},\mathbf{X}) \tag{16.11}$$

Modern data sizes are quite large. For example Whisper-v2 is trained on a corpus of 680,000 hours of speech, mostly from English, but also including 118,000 hours from 96 other languages. Data quality is important, so systems that scrape web data for training implement methods to remove ASR-generated transcripts from their training corpora, such as filtering data that is all uppercase or all lowercase. The open OWSM system is trained on 180k hours, mainly hand-transcribed publicly available data, including such datasets as LibriSpeech and Multilingual LibriSpeech, Common Voice, FLEURS, Switchboard, AMI, and others; see (Peng et al., 2023) for details.

16.4 Self-supervised models: HuBERT

self-supervised

An alternative to the encoder-decoder architecture are the class of **self-supervised** speech models. These models don't directly learn to map an acoustic input to a string of letters and tokens. Instead, they first bootstrap a set of discrete phonetic units from the acoustic input, learning to map from waveforms to these induced units. This pretraining phase doesn't require transcripts; just unlabeled speech files. After they are pretrained, these models can then be fine-tuned to do ASR on a smaller set of labeled data, audio paired with transcripts. These models have the advantage that they can take advantage of large amounts of untranscribed audio for most of their training.

HuBERT
wav2vec 2.0

Here we'll introduce one self-supervised model called **HuBERT** (Hsu et al., 2021). HuBERT and similar models like **wav2vec 2.0** (Baevski et al., 2020) use the same intuition as the masked language models like BERT introduced in Chapter 9: we mask out some part of the input and train the model to guess what was hidden by the mask.

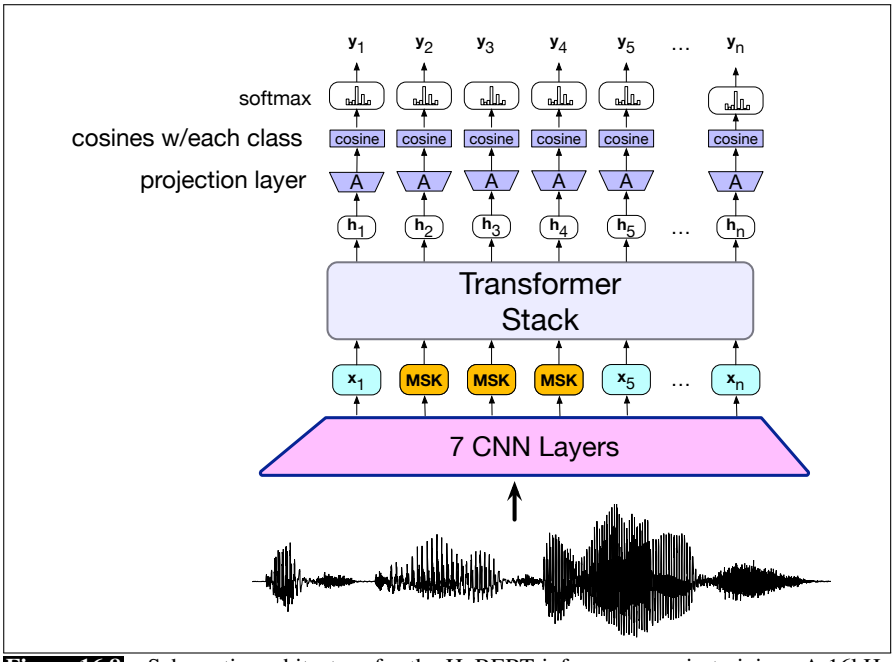


Figure 16.8 Schematic architecture for the HuBERT inference pass in training. A 16kHz wavfile is passed through a series of convolutional layers, some frames are replaced with a MASK token, and then the sequence is passed though a transformer stack, and then a linear layer that projects the transformer output to an output embedding. This embedding is compared via cosine with the embeddings for each of the 100/500 phonetic classes to produce a logit which is passed through softmax to get a probability distribution over the classes at each frame.

16.4.1 HuBERT forward pass

Let's first show just the forward pass for HuBERT used during training, and then we'll see this in its full training context with the backwards pass. As discussed earlier, the input to the HuBERT forward pass is raw 16kHz wavefiles as input,

and the output at each 20ms time frame will be a probability distribution over a set of induced phonetic classes C (100 classes or 500 classes, depending on the stage). Fig. 16.8 shows a sketch of the components. The wavefile is passed through 7 512-channel convolutional layers which learn both to extract spectral information, and to shorten the input sequence down to a 20 ms frame, after which positional encodings are added, and then GELU and layer norm. Selected tokens are then replaced with a mask token, a trained embedding that is shared by all masked frames. The whole sequence is passed through a transformer stack, and the output is passed through a linear projection layer $\mathbf{A}$. The output embedding at each 20ms frame is then compared via cosine with each of the embeddings for the 100 (/500) phonetic classes, resulting in a set of 100 logits representing the similarity of the current 20ms audio timestep to each class. These are then passed through a softmax to get a probability distribution over the classes.

16.4.2 Learning for HuBERT

Let's first discuss how we induce the 100 or 500 phonetic classes that are the target of training. To bootstrap these units, HuBERT starts with **mel frequency cepstral coefficients**, or **MFCC** vectors, a 39-dimensional feature vector that emphasizes aspects of the signal that are relevant for detection of phonetic units. These vectors can be extracted from the acoustic signal as summarized in Section **??**. We extract MFCC vectors for the entire acoustic training dataset (the original HuBERT implementation used 960 hours of LibriSpeech data resulting in 172 million vectors). Next we **cluster** the MFCC vectors using the **k-means** clustering algorithm described below in Section 16.4.3. Clustering means to group the vectors into k classes. The output of clustering is a **codebook** of k vectors, called **codewords** or **templates** or **prototypes**, each representing a cluster. Each of these k clusters is an acoustic unit that we can use as the gold targets for training.

Now let's consider the entire training process. After the acoustic input is run through the CNN layers, a span of tokens in the context window is chosen to be masked, and for those tokens the CNN output is replaced by a MASK embedding. The entire context window is passed through the transformer layers, and the transformer output h_t^L at each timestep t is multiplied by the projection layer matrix $\mathbf{A}$ to project it into the class embedding space. The resulting representation is then compared to the embedding for each of the classes in C (using cosine), and a softmax (with temperature parameter τ=0.1) is used to turn the similarity into a probability:

$$p(c|\mathbf{X},t) = \frac{\exp(\text{sim}(\mathbf{A}\mathbf{h}_t, \mathbf{e_c})/\tau)}{\sum_{c'=1}^{C} \exp(\text{sim}(\mathbf{A}\mathbf{h_t}, \mathbf{e_{c'}})/\tau)} \tag{16.12}$$

As Fig. 16.9 shows, in parallel with this forward pass, the input waveform is passed through an MFCC to create a 39-dimensional vector which is then mapped to one of the 100 classes by choosing the most similar centroid in the codebook. The loss function is then the sum, over the set of masked tokens M, of the probability that the model assigns to these correct units:

$$L = \sum_{t \in M} \log p(z_t|\mathbf{X}) \tag{16.13}$$

Thus, as in masked language modeling, the model is being trained to predict the

units associated with the masked frames. This loss is then backpropagated through the model

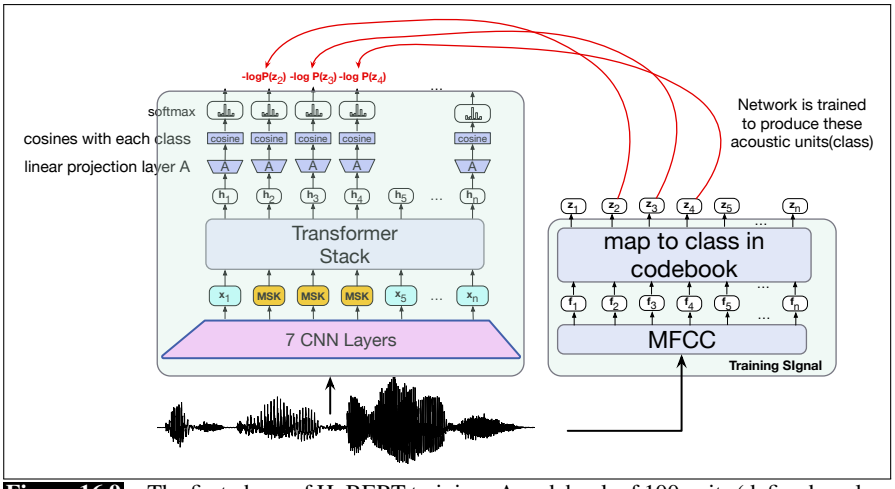


Figure 16.9 The first phase of HuBERT training. A codebook of 100 units (defined as clusters of 39-dimensional MFCC vectors) is used as the targets for training. For each timestep t, computes the probability of that class, and uses the logprob as the loss.

Once the model has been initially trained to map to MFCC vector centroids, a second stage of training occurs, where we take the representations produced by the model, cluster them into 500 clusters, and use those instead as the target for training. The intuition is that the initial MFCC clusters will bias the model toward phonetic representations, but after enough training the model will learn more accurate and fine-grained representations. Fig. 16.10 shows the intuition.

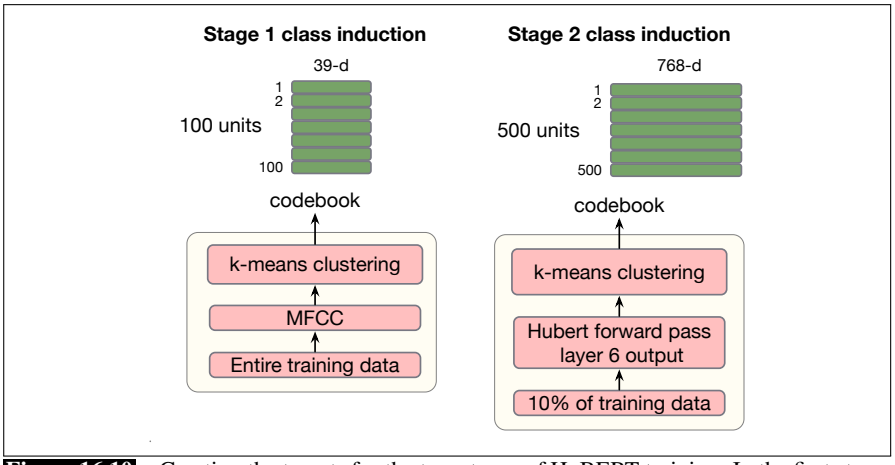


Figure 16.10 Creating the targets for the two stages of HuBERT training. In the first stage, 100 acoustic units are created by computing 39-dimensional MFCC vectors for the entire training data and then clustering them with k-means. In the second stage, 500 units are created by passing a subsample of the training data through the HuBERT model after the first stage training, taking the output of an intermediate transformer layer (layer 6) and clustering them with k-means.

After HuBERT has been pretrained, the projection and cosine layers are removed and a randomly initialized linear + softmax layer is added, mapping into 29 classes

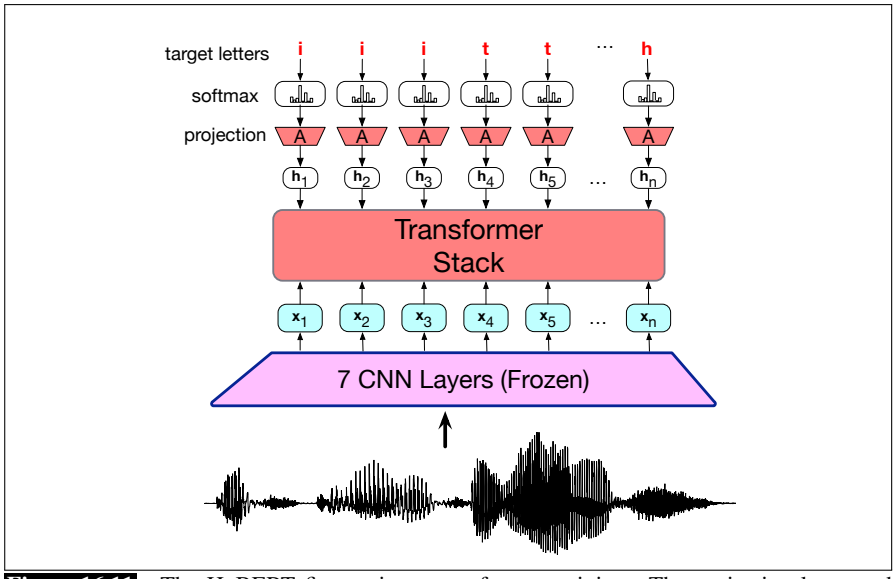


Figure 16.11 The HuBERT fine-tuning pass after pretraining. The projection layer and cosine steps are removed, leaving only a randomly initialized projection/softmax layer. The CNN layers are frozen, and the rest of the model is fine-tuned on a dataset of audio with transcripts, trained with the CTC loss (Section 16.5) to produce letters as output. The parameters that are updated in fine-tuning are shown in red (the projection layer and the transformer stack).

(corresponding to the 26 English letters and a few extra characters) for the ASR task. The CNNs are frozen and the rest of the model is fine-tuned for ASR using the CTC loss function to be described in Section 16.5.

16.4.3 K-means clustering

k-means

In this section we give the **k-means** clustering algorithm more formally. K-means is a family of algorithms for grouping a set of vector data into k clusters. Clustering is useful whenever we want to treat a group of elements in the same way. In speech processing it is very commonly used whenever we need to convert a set of vectors over real values into a set of discrete symbols. Besides its use here in HuBERT, we'll return to it in Chapter 18 as an algorithm for creating discrete acoustic tokens for TTS.

We generally use the name k-means to mean a simple version of the family: a two-step iterative algorithm that is given a set of N vectors $\mathbf{v}^{(1)}..\mathbf{v}^{(N)}$ each of d dimensions, i.e. $\forall i, \mathbf{v}^{(i)} \in \mathbb{R}^d$, and a constant k, where usually $N >> k$.

The two-step algorithm is 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 algorithm has two steps. In the assignment step, given a set of k current centroids and a dataset of vectors, it assigns each vector to the cluster whose codeword is the closest (by squared Euclidean distance). In the re-estimation step, it recomputes the codeword for each cluster by recomputing the mean vector. Note that the resulting mean vector need not be an actual point from the dataset. We iterate back and forth between these two steps.

Here's the algorithm:

Initialization: For each cluster k choose a random vector $\mu_k \in \mathbb{R}^d$ to be the

codeword
template
prototype
codebook

codeword (also called **template** or **prototype**) for the cluster. The result is a **codebook** that has one codeword for each of the k clusters.

Then repeat iteratively until convergence:

1. **Assignment**: For each vector $\mathbf{v}^{(\mathbf{i})}$ in the dataset assign it to one of the k clusters by choosing the one with the nearest codeword μ. Most simply we can define 'nearest' as the cluster whose codeword has the smallest squared Euclidean distance to $\mathbf{v}^{(\mathbf{i})}$.

$$\text{cluster}^{(i)} = \underset{1<j<k}{\text{argmin}} ||\mathbf{v}^{(i)} - \mu_j||^2 \tag{16.14}$$

   where $||\mathbf{v}||$ is the L2 norm of the vector $\sum_{j=1}^{d} \mathbf{v}_j^2$

2. **Re-estimation**: Re-estimate the codeword for each cluster by recomputing the mean (centroid) of all the vectors in the cluster. If S_i is the set of vectors in cluster i, then

$$\begin{aligned} &\forall i: \\ &\mu_i = \frac{1}{|S_i|} \sum_{\mathbf{v} \in S_i} \mathbf{v} \end{aligned} \tag{16.15}$$

16.5 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.

alignment

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 frame an **alignment**, because it tells us where in the acoustic signal each letter aligns to. Fig. 16.12 shows one such alignment, and what happens if we use a collapsing function that just removes consecutive duplicate 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 $B : a \rightarrow y$

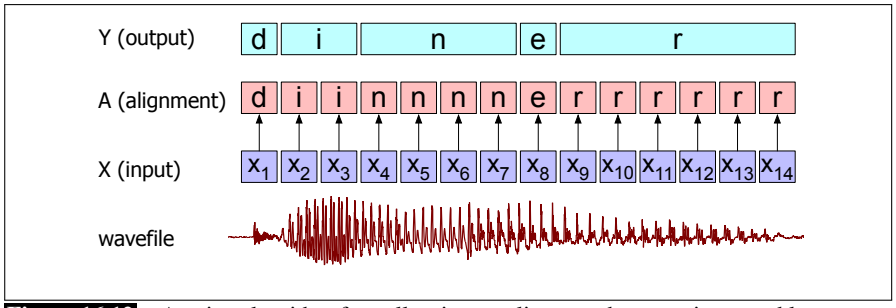


Figure 16.12 A naive algorithm for collapsing an alignment between input and letters.

between an alignment a and an output y, which collapses all repeated letters and then removes all blanks. Fig. 16.13 sketches this collapsing function B.

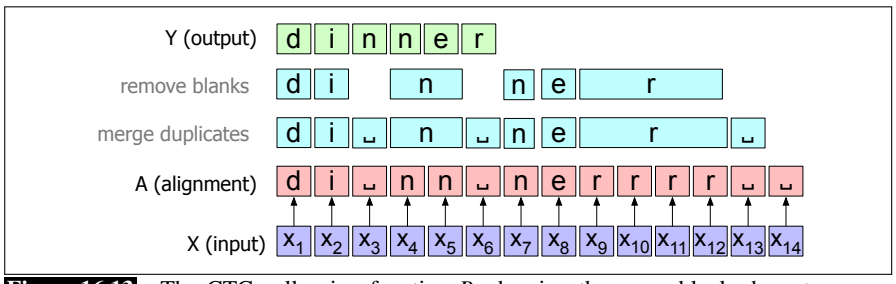


Figure 16.13 The CTC collapsing function B, showing the space blank character ␣; repeated (consecutive) characters in an alignment A are removed to form the output Y.

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. 16.13 is not the only alignment that results in the string *dinner*. Fig. 16.14 shows some other alignments that would produce the same output.

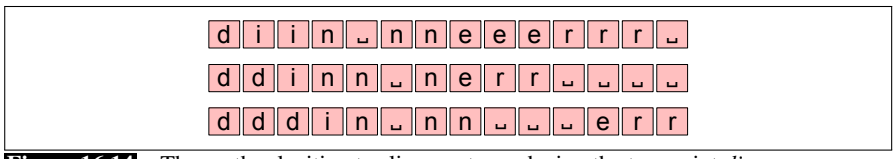


Figure 16.14 Three other legitimate alignments producing the transcript *dinner*.

It's useful to think of the set of all alignments that might produce the same output Y. We'll use the inverse image of our B function, called B^{-1}, and represent that set as $B^{-1}(Y)$.

16.5.1 CTC Inference

Before we see how to compute $P_{\text{CTC}}(Y|\mathbf{X})$ 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 $\mathbf{X}$, the CTC model output a_t at time t is independent of the output labels at any other time a_i. Thus:

$$P_{\text{CTC}}(\mathbf{A}|\mathbf{X}) = \prod_{t=1}^{T} p(a_t|\mathbf{X}) \tag{16.16}$$

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:

$$\hat{a}_t = \underset{c \in C}{\operatorname{argmax}}\, p_t(c|\mathbf{X}) \tag{16.17}$$

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 $\hat{\mathbf{y}}_t$ at time t corresponding to each input token $\mathbf{x}_t$, eliminating the need for a full decoder. Fig. 16.15 sketches this architecture, where we take an encoder, produce a hidden state $\mathbf{h}_t$ at each timestep, and decode by taking a softmax over the character vocabulary at each time step.

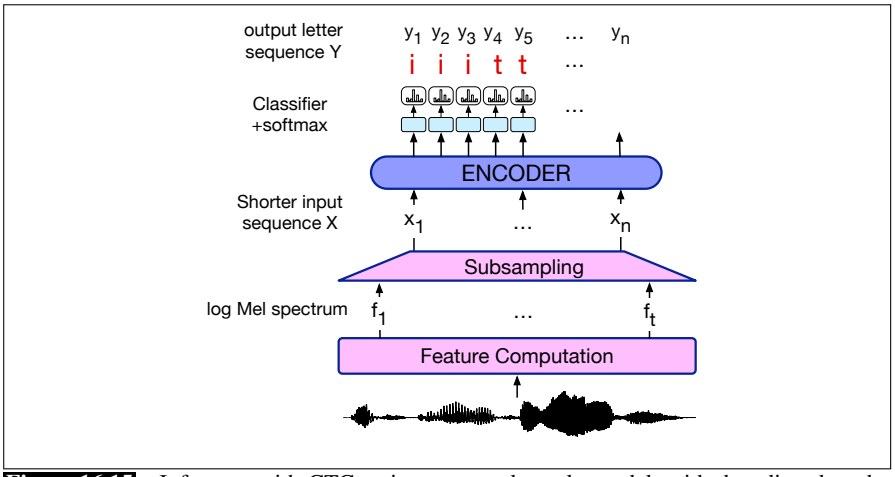


Figure 16.15 Inference with CTC: using an encoder-only model, with decoding done by simple softmaxes over the hidden state $\mathbf{h}_t$ at each output step.

Alas, there is a potential flaw with the inference algorithm sketched in (Eq. 16.17) and Fig. 16.14. 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 $\mathbf{X} = [x_1 x_2 x_3]$ 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:

$$\begin{aligned} P_{CTC}(Y|\mathbf{X}) &= \sum_{A \in B^{-1}(Y)} P(A|\mathbf{X}) \\ &= \sum_{A \in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \\ \hat{Y} &= \underset{Y}{\operatorname{argmax}}\, P_{CTC}(Y|\mathbf{X}) \end{aligned} \tag{16.18}$$

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. 16.18). 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 devset:

$$score_{\text{CTC}}(Y|\mathbf{X}) = \log P_{\text{CTC}}(Y|\mathbf{X}) + \lambda_1 \log P_{LM}(Y)\lambda_2 L(Y) \tag{16.19}$$

16.5.2 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 $\mathbf{X}$:

$$L_{CTC} = \sum_{(X,Y)\in D} -\log P_{\text{CTC}}(Y|\mathbf{X}) \tag{16.20}$$

To compute CTC loss function for a single input pair $(\mathbf{X},Y)$, we need the probability of the output Y given the input $\mathbf{X}$. As we saw in Eq. 16.18, 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:

$$P_{\text{CTC}}(Y|\mathbf{X}) \;=\; \sum_{A\in B^{-1}(Y)} \prod_{t=1}^{T} p(a_t|h_t) \tag{16.21}$$

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.

16.5.3 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. 16.16 shows a sketch. For training, we can simply weight the two losses with a λ tuned on a devset:

$$L = -\lambda \log P_{encdec}(Y|\mathbf{X}) - (1-\lambda)\log P_{ctc}(Y|\mathbf{X}) \tag{16.22}$$

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

$$\hat{Y} = \underset{Y}{\operatorname{argmax}}\,[\lambda \log P_{encdec}(Y|\mathbf{X}) + (1-\lambda)\log P_{CTC}(Y|\mathbf{X}) + \gamma \log P_{LM}(Y)] \tag{16.23}$$

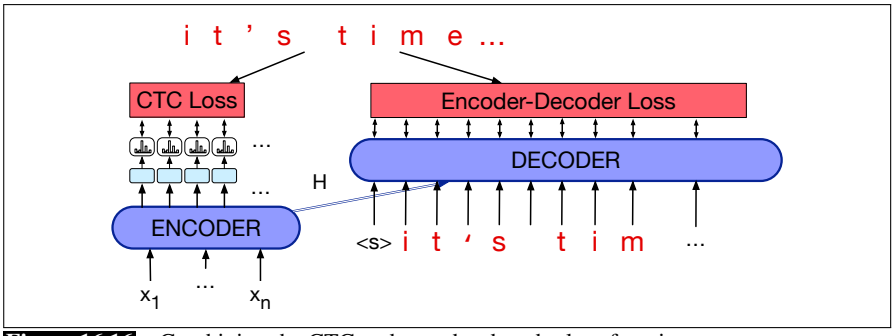


Figure 16.16 Combining the CTC and encoder-decoder loss functions.

16.5.4 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. 16.17, 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 $x_1 ... x_t$. 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 softmax to predict the next character.

RNN-T

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

16.6 ASR Evaluation: Word Error Rate

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

word error

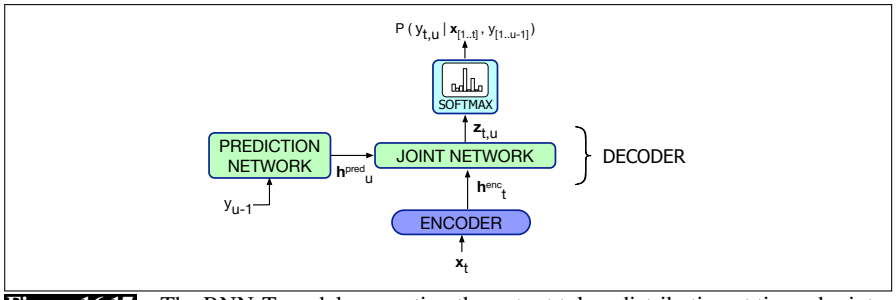


Figure 16.17 The RNN-T model computing the output token distribution at time t by integrating the output of a CTC acoustic encoder and a separate 'predictor' language model.

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

Text normalization before evaluation

It's normal for systems to normalize text before computing word error rate. There are a variety of packages for implementing normalization rules. For example some standard English normalization rules include:

1. Removing metalanguage [non-language, notes, transcription comments] that occur between matching brackets ([,])
2. Remove or standardize interjections or filled pauses ("uh", "um", "err")
3. Standardize contracted and non-contracted forms of English ("I'm"/"I am")

4. Normalize non-standard-words (number, quantities, dates, times) [e.g., "$100 → "One hundred dollars"]
5. Unify US and UK spelling conventions

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

$$\sigma_z^2 = \frac{1}{n-1} \sum_{i=1}^{n} (Z_i - \mu_z)^2 \tag{16.24}$$

Let

$$W = \frac{\hat{\mu}_z}{\sigma_z/\sqrt{n}} \tag{16.25}$$

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.

16.7 Summary

This chapter introduced the fundamental algorithms of automatic speech recognition (ASR).

- 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.

Historical Notes

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 **warp-**

warping

ing; 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**.

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, 1975) 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, 1976), 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 important factor in the eventual spread of the HMM paradigm.

bakeoff

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

hybrid

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).

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).

Exercises

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*.

Atal, B. S. and S. Hanauer. 1971. Speech analysis and synthesis by prediction of the speech wave. *JASA*, 50:637–655.

Baevski, A., Y. Zhou, A. Mohamed, and M. Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *NeurIPS*, volume 33.

Bahdanau, D., J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio. 2016. End-to-end attention-based large vocabulary speech recognition. *ICASSP*.

Baker, J. K. 1975. The DRAGON system – An overview. *IEEE Transactions on ASSP*, ASSP-23(1):24–29.

Baum, L. E. and J. A. Eagon. 1967. An inequality with applications to statistical estimation for probabilistic functions of Markov processes and to a model for ecology. *Bulletin of the American Mathematical Society*, 73(3):360–363.

Baum, L. E. and T. Petrie. 1966. Statistical inference for probabilistic functions of finite-state Markov chains. *Annals of Mathematical Statistics*, 37(6):1554–1563.

Bostrom, K. and G. Durrett. 2020. Byte pair encoding is suboptimal for language model pretraining. *EMNLP*.

Bourlard, H. and N. Morgan. 1994. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer.

Bu, H., J. Du, X. Na, B. Wu, and H. Zheng. 2017. AISHELL-1: An open-source Mandarin speech corpus and a speech recognition baseline. *O-COCOSDA Proceedings*.

Canavan, A., D. Graff, and G. Zipperlen. 1997. CALLHOME American English speech LDC97S42. Linguistic Data Consortium.

Chan, W., N. Jaitly, Q. Le, and O. Vinyals. 2016. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. *ICASSP*.

Chorowski, J., D. Bahdanau, K. Cho, and Y. Bengio. 2014. End-to-end continuous speech recognition using attention-based recurrent NN: First results. *NeurIPS Deep Learning and Representation Learning Workshop*.

Cieri, C., D. Miller, and K. Walker. 2004. The Fisher corpus: A resource for the next generations of speech-to-text. *LREC*.

Conneau, A., M. Ma, S. Khanuja, Y. Zhang, V. Axelrod, S. Dalmia, J. Riesa, C. Rivera, and A. Bapna. 2023. Fleurs: Few-shot learning evaluation of universal representations of speech. *IEEE SLT*.

Cooley, J. W. and J. W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301.

Dahl, G. E., T. N. Sainath, and G. E. Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. *ICASSP*.

Dahl, G. E., D. Yu, L. Deng, and A. Acero. 2012. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Transactions on audio, speech, and language processing*, 20(1):30–42.

David, Jr., E. E. and O. G. Selfridge. 1962. Eyes and ears for computers. *Proceedings of the IRE (Institute of Radio Engineers)*, 50:1093–1101.

Davis, K. H., R. Biddulph, and S. Balashek. 1952. Automatic recognition of spoken digits. *JASA*, 24(6):637–642.

Denes, P. 1959. The design and operation of the mechanical speech recognizer at University College London. *Journal of the British Institution of Radio Engineers*, 19(4):219–234. Appears together with companion paper (Fry 1959).

Deng, L., G. Hinton, and B. Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: An overview. *ICASSP*.

Fry, D. B. 1959. Theoretical aspects of mechanical speech recognition. *Journal of the British Institution of Radio Engineers*, 19(4):211–218. Appears together with companion paper (Denes 1959).

Gillick, L. and S. J. Cox. 1989. Some statistical issues in the comparison of speech recognition algorithms. *ICASSP*.

Godfrey, J., E. Holliman, and J. McDaniel. 1992. SWITCHBOARD: Telephone speech corpus for research and development. *ICASSP*.

Goyal, N., C. Gao, V. Chaudhary, P.-J. Chen, G. Wenzek, D. Ju, S. Krishnan, M. Ranzato, F. Guzmán, and A. Fan. 2022. The flores-101 evaluation benchmark for low-resource and multilingual machine translation. *TACL*, 10:522–538.

Graff, D. 1997. The 1996 Broadcast News speech and language-model corpus. *Proceedings DARPA Speech Recognition Workshop*.

Graves, A. 2012. Sequence transduction with recurrent neural networks. *ICASSP*.

Graves, A., S. Fernández, F. Gomez, and J. Schmidhuber. 2006. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. *ICML*.

Graves, A. and N. Jaitly. 2014. Towards end-to-end speech recognition with recurrent neural networks. *ICML*.

Graves, A., A.-r. Mohamed, and G. Hinton. 2013. Speech recognition with deep recurrent neural networks. *ICASSP*.

Hannun, A. 2017. Sequence modeling with CTC. *Distill*, 2(11).

Hannun, A. Y., A. L. Maas, D. Jurafsky, and A. Y. Ng. 2014. First-pass large vocabulary continuous speech recognition using bi-directional recurrent DNNs. ArXiv preprint arXiv:1408.2873.

Hayashi, T., R. Yamamoto, K. Inoue, T. Yoshimura, S. Watanabe, T. Toda, K. Takeda, Y. Zhang, and X. Tan. 2020. ESPnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit. *ICASSP*.

Hsu, W.-N., B. Bolte, Y.-H. H. Tsai, K. Lakhotia, R. Salakhutdinov, and A. Mohamed. 2021. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM TASLP*, 29:3451–3460.

Itakura, F. 1975. Minimum prediction residual principle applied to speech recognition. *IEEE Transactions on ASSP*, ASSP-32:67–72.

Jaitly, N., P. Nguyen, A. Senior, and V. Vanhoucke. 2012. Application of pretrained deep neural networks to large vocabulary speech recognition. *INTERSPEECH*.

Jelinek, F. 1969. A fast sequential decoding algorithm using a stack. *IBM Journal of Research and Development*, 13:675–685.

Jelinek, F., R. L. Mercer, and L. R. Bahl. 1975. Design of a linguistic statistical decoder for the recognition of continuous speech. *IEEE Transactions on Information Theory*, IT-21(3):250–256.

Karita, S., N. Chen, T. Hayashi, T. Hori, H. Inaguma, Z. Jiang, M. Someki, N. E. Y. Soplin, R. Yamamoto, X. Wang, S. Watanabe, T. Yoshimura, and W. Zhang. 2019. A comparative study on transformer vs RNN in speech applications. *IEEE ASRU-19*.

Kendall, T. and C. Farrington. 2020. The Corpus of Regional African American Language. Version 2020.05. Eugene, OR: The Online Resources for African American Language Project. http://oraal.uoregon.edu/coraal.

Klatt, D. H. 1977. Review of the ARPA speech understanding project. *JASA*, 62(6):1345–1366.

Lang, K. J., A. H. Waibel, and G. E. Hinton. 1990. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43.

LDC. 1998. *LDC Catalog: Hub4 project*. University of Pennsylvania. www.ldc.upenn.edu/Catalog/LDC98S71.html.

Liu, Y., P. Fung, Y. Yang, C. Cieri, S. Huang, and D. Graff. 2006. HKUST/MTS: A very large scale Mandarin telephone speech corpus. *International Conference on Chinese Spoken Language Processing*.

Lowerre, B. T. 1976. *The Harpy Speech Recognition System*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Maas, A., Z. Xie, D. Jurafsky, and A. Y. Ng. 2015. Lexicon-free conversational speech recognition with neural networks. *NAACL HLT*.

Maas, A. L., A. Y. Hannun, and A. Y. Ng. 2013. Rectifier nonlinearities improve neural network acoustic models. *ICML*.

Maas, A. L., P. Qi, Z. Xie, A. Y. Hannun, C. T. Lengerich, D. Jurafsky, and A. Y. Ng. 2017. Building dnn acoustic models for large vocabulary speech recognition. *Computer Speech & Language*, 41:195–213.

Mohamed, A., G. E. Dahl, and G. E. Hinton. 2009. Deep Belief Networks for phone recognition. *NIPS Workshop on Deep Learning for Speech Recognition and Related Applications*.

Morgan, N. and H. Bourlard. 1990. Continuous speech recognition using multilayer perceptrons with hidden markov models. *ICASSP*.

Morgan, N. and H. A. Bourlard. 1995. Neural networks for statistical recognition of continuous speech. *Proceedings of the IEEE*, 83(5):742–772.

NIST. 2005. Speech recognition scoring toolkit (sctk) version 2.1. http://www.nist.gov/speech/tools/.

NIST. 2007. Matched Pairs Sentence-Segment Word Error (MAPSSWE) Test.

Oppenheim, A. V., R. W. Schafer, and T. G. J. Stockham. 1968. Nonlinear filtering of multiplied and convolved signals. *Proceedings of the IEEE*, 56(8):1264–1291.

Panayotov, V., G. Chen, D. Povey, and S. Khudanpur. 2015. Librispeech: an ASR corpus based on public domain audio books. *ICASSP*.

Peng, Y., J. Tian, B. Yan, D. Berrebbi, X. Chang, X. Li, J. Shi, S. Arora, W. Chen, R. Sharma, W. Zhang, Y. Sudo, M. Shakee, J. weon Jung, S. Maiti, and S. Watanabe. 2023. Reproducing whisper-style training using an open-source toolkit and publicly available data. *ASRU*.

Povey, D., A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovský, G. Stemmer, and K. Veselý. 2011. The Kaldi speech recognition toolkit. *ASRU*.

Price, P. J., W. Fisher, J. Bernstein, and D. Pallet. 1988. The DARPA 1000-word resource management database for continuous speech recognition. *ICASSP*.

Pundak, G. and T. N. Sainath. 2016. Lower frame rate neural network acoustic models. *INTERSPEECH*.

Radford, A., J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever. 2023. Robust speech recognition via large-scale weak supervision. *ICML*.

Renals, S., T. Hain, and H. Bourlard. 2007. Recognition and understanding of meetings: The AMI and AMIDA projects. *ASRU*.

Robinson, T. and F. Fallside. 1991. A recurrent error propagation network speech recognition system. *Computer Speech & Language*, 5(3):259–274.

Sakoe, H. and S. Chiba. 1971. A dynamic programming approach to continuous speech recognition. *Proceedings of the Seventh International Congress on Acoustics*, volume 3. Akadémiai Kiadó.

Sakoe, H. and S. Chiba. 1984. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on ASSP*, ASSP-26(1):43–49.

Shannon, C. E. 1948. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423. Continued in the following volume.

Velichko, V. M. and N. G. Zagoruyko. 1970. Automatic recognition of 200 words. *International Journal of Man-Machine Studies*, 2:223–234.

Vintsyuk, T. K. 1968. Speech discrimination by dynamic programming. *Cybernetics*, 4(1):52–57. Original Russian: Kibernetika 4(1):81-88. 1968.

Waibel, A., T. Hanazawa, G. Hinton, K. Shikano, and K. J. Lang. 1989. Phoneme recognition using time-delay neural networks. *IEEE Transactions on ASSP*, 37(3):328–339.

Watanabe, S., T. Hori, S. Karita, T. Hayashi, J. Nishitoba, Y. Unno, N. E. Y. Soplin, J. Heymann, M. Wiesner, N. Chen, A. Renduchintala, and T. Ochiai. 2018. ESPnet: End-to-end speech processing toolkit. *INTERSPEECH*.

Yuan, J., M. Liberman, and C. Cieri. 2006. Towards an integrated understanding of speaking rate in conversation. *Interspeech*.