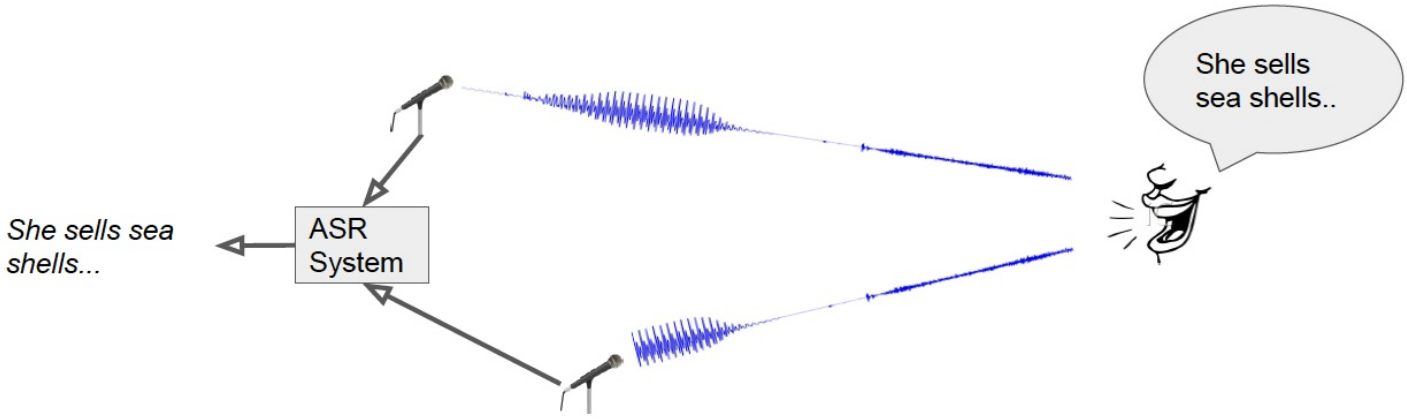
Speech Recognition

Speech recognition is invading our lives. It's built into our phones (Siri), our game consoles (Kinect), our smartwatches (Apple Watch), and even our homes (Amazon Echo). But speech recognition has been around for decades, so why is it just now hitting the mainstream?

The reason is that deep learning finally made speech recognition accurate enough to be useful outside of carefully-controlled environments. In this blog post, we'll learn how to perform speech recognition with 3 different implementations of popular deep learning frameworks.

Speech Recognition — The Classic Way

In the era of *OK Google*, I might not really need to define ASR, but here's a basic description: Say you have a person or an audio source saying something textual, and you have a bunch of microphones that are receiving the audio signals. You can get these signals from one or many devices, and then pass them into an ASR system — whose job it is to infer the original source transcript that the person spoke or that the device played.



So why is ASR important?

Firstly, it's a very natural interface for human communication. You don't need a mouse or a keyboard, so it's obviously a good way to interact with machines. You don't even really need to learn new techniques because most people learn to speak as a function of natural development. It's a very natural interface for talking with simple devices such as cars, handheld phones, and chatbots.

Challenges of speech recognition

Noise

Voice recording machines detect sound waves that are generated through speech. Background noises in rooms make it hard for systems to understand and distinguish between the specific sound waves from the host voice. This blurs the sound picked-up by the devices, confusing, and limiting its processing ability.

Echo

Echoes are basically sound-waves reflected across various surfaces, such as walls, tables, or other furniture. This leads to a disorganised return of sound waves back to the receptors, thus reducing clarity.

Accents

A wide range of accents in every language is another factor that leads to difficulties in speech recognition. If the same word can be pronounced in a number of different ways, the syllables and phonetics of the same word tend to vary, making it harder for the machine to process.

Similar Sounds

Similar sounding words and phrases can prevent proper encoding and decoding of the voice message. For example "Let's wreck a nice beach" and "Let's recognise speech" are phonetically very similar and can easily confuse the device.

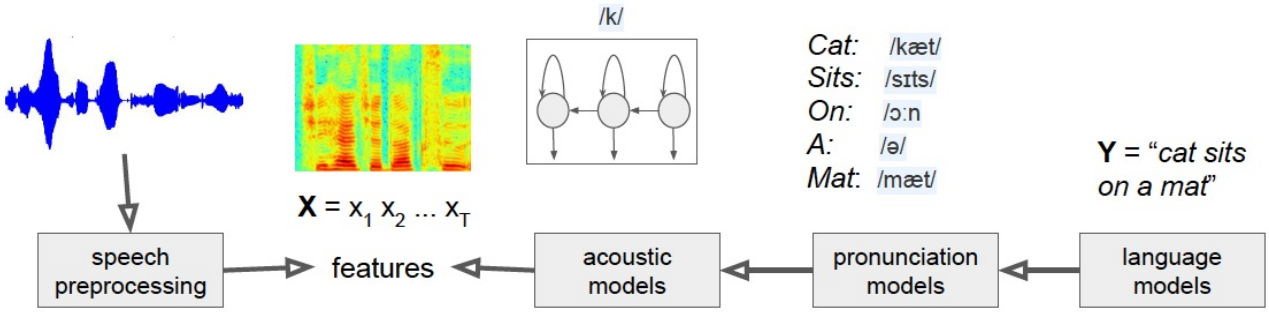
So how is this done classically?

ASR is not easy since there are lots of variabilities:

- acoustics:
  - variability between speakers (inter-speaker)
  - variability for the same speaker (intra-speaker)
  - noise, reverberation in the room, environment...
- phonetics:
  - articulation
  - elisions (grouping some words, not pronouncing them)
  - words with similar pronunciation
- linguistics:
  - size of vocabulary
  - word variations

From a Machine Learning perspective, ASR is also really hard:

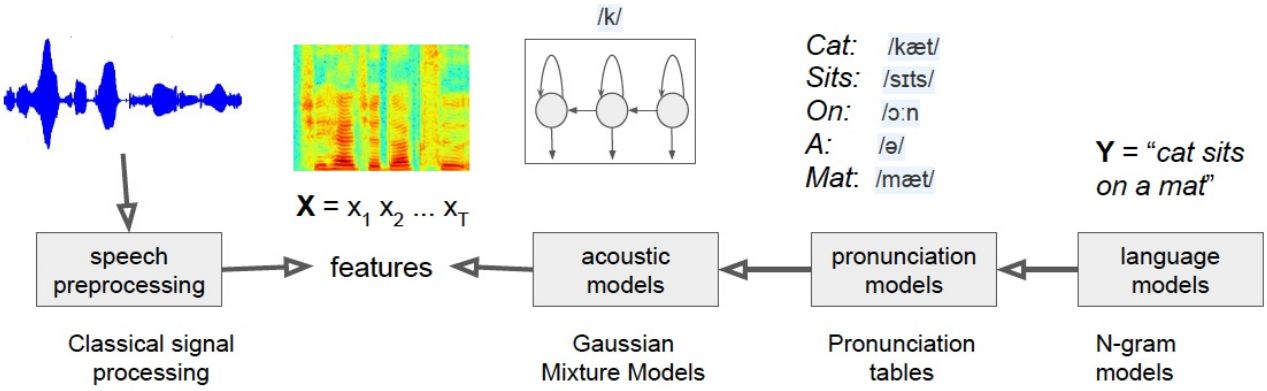
- very high dimensional output space, and a complex sequence to sequence problem
- few annotated training data
- data is noisy



The classic way of building a speech recognition system is to build a generative model of language. On the rightmost side, you produce a certain sequence of words from language models. And then for each word, you have a pronunciation model that says how this particular word is spoken. Typically it's written out as the sequence of phonemes — which are basic units of sound, but for our vocabulary, we'll just say a sequence of tokens — which represent a cluster of things that have been defined by linguistics experts.

A pronunciation model can use tables to convert words to phones, or a corpus is already transcribed with phonemes already. The acoustic model is about modelling a sequence of feature vectors given a sequence of phones instead of words. But we will continue the use of the notation $p(X|W)$ for the acoustic model. Just be aware.

The language model is about the likelihood of the word sequence. For example, "I watch a movie" will be more likely than "I you movie watch" or "I watch an apple". It predicts the next word given the previous words.



Each of these different components in this pipeline uses a different statistical model:

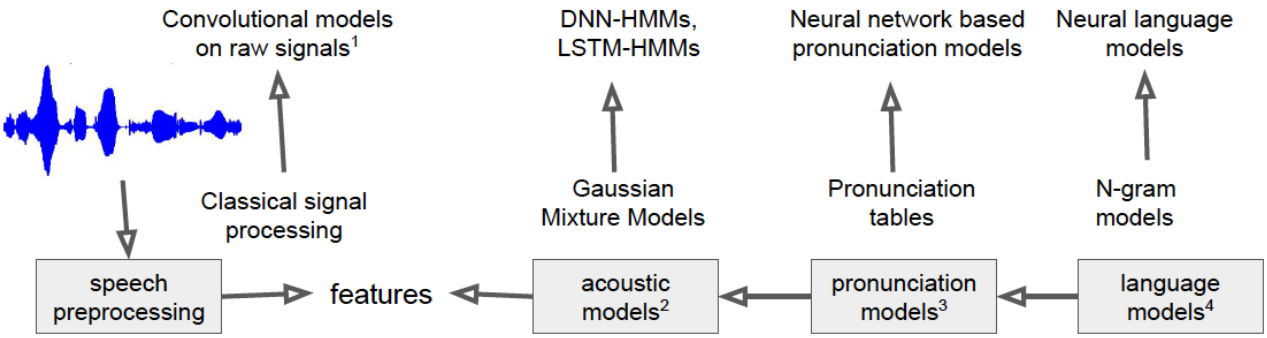
- In the past, language models were typically N-gram models, which worked very well for simple problems with limited speech input data. They are essentially tables describing the probabilities of token sequences.
- The pronunciation models were simple lookup tables with probabilities associated with pronunciations. These tables would be very large tables of different pronunciations.
- Acoustic models are built using Gaussian Mixture Models with very specific architectures associated with them.
- The speech processing was pre-defined.

Once we have this kind of model built, we can perform the recognition by doing the inference on the data received. So you get a waveform, you compute the features for it (X) and do a search for Y that gives the highest probabilities of X.

The Neural Network Invasion

Over time, researchers started noticing that each of these components could work more effectively if we used neural networks.

- Instead of the N-gram language models, we can build neural language models and feed them into a speech recognition system to restore things that were produced by a first path speech recognition system.
- Looking into the pronunciation models, we can figure out how to do pronunciation for a new sequence of characters that we've never seen before using a neural network.
- For acoustic models, we can build deep neural networks (such as LSTM-based models) to get much better classification accuracy scores of the features for the current frame.
- Interestingly enough, even the speech pre-processing steps were found to be replaceable with convolutional neural networks on raw speech signals.



However, there's still a problem. There are neural networks in each component, but they're trained independently with different objectives. Because of that, the errors in one component may not behave well with the errors in another component. So that's the basic motivation for devising a process where you can train the entire model as one big component itself.

These so-called **end-to-end models** encompass more and more components in the pipeline discussed above. The 2 most popular ones are

(1) **Connectionist Temporal Classification** (CTC), which is in wide usage these days at Baidu and Google, but it requires a lot of training; and

(2) **Sequence-To-Sequence** (Seq-2-Seq), which doesn't require manual customization.

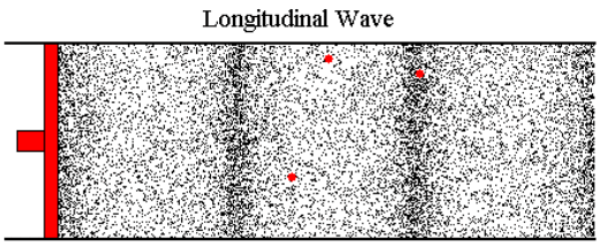
The basic motivation is that we want to do end-to-end speech recognition. We are given the audio X — which is a sequence of frames from x1 to xT, and the corresponding output text Y — which is a sequence of y1 to yL. Y is just a text sequence (transcript) and X is the audio processed spectrogram. We want to perform speech recognition by learning a probabilistic model p(Y|X): starting with the data and predicting the target sequences themselves.

Pre-processing

What is an Audio Signal?

This is pretty intuitive – any object that vibrates produces sound waves. Have you ever thought of how we are able to hear someone's voice? It is due to the audio waves. Let's quickly understand the process behind it.

When an object vibrates, the air molecules oscillate to and fro from their rest position and transmits its energy to neighbouring molecules. This results in the transmission of energy from one molecule to another which in turn produces a sound wave.



What is sampling the signal and why is it required?

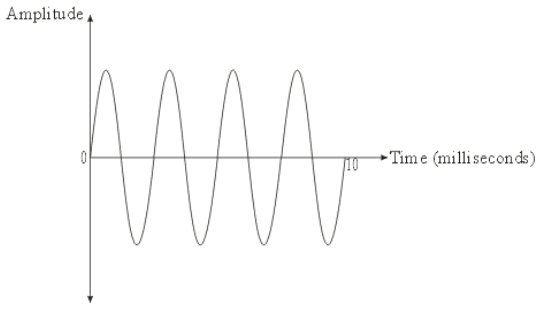
An audio signal is a continuous representation of amplitude as it varies with time. Here, time can even be in picoseconds. That is why an audio signal is an analog signal.

Analog signals are memory hogging since they have an infinite number of samples and processing them is highly computationally demanding. Therefore, **we need a technique to convert analog signals to digital signals so that we can work with them easily.**

Sampling the signal is a process of converting an analog signal to a digital signal by selecting a certain number of samples per second from the analog signal. Can you see what we are doing here? We are converting an audio signal to a discrete signal through sampling so that it can be stored and processed efficiently in memory.

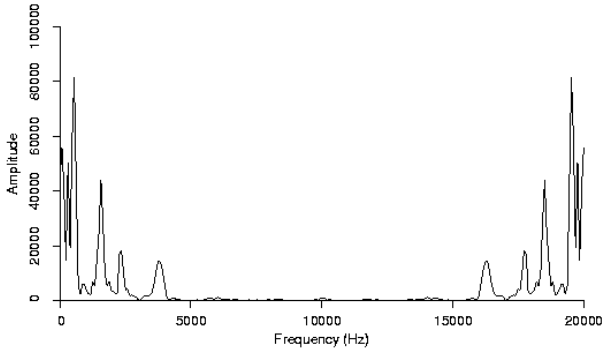
Time-domain

Here, the audio signal is represented by the amplitude as a function of time. In simple words, it is **a plot between amplitude and time**. The features are the amplitudes which are recorded at different time intervals.



Frequency domain

In the frequency domain, the audio signal is represented by amplitude as a function of frequency. Simply put – it is **a plot between frequency and amplitude**. The features are the amplitudes recorded at different frequencies.



10 seconds -> 20ms -> 500 segments

each segment -> fourier transform -> freq domain output -> o1

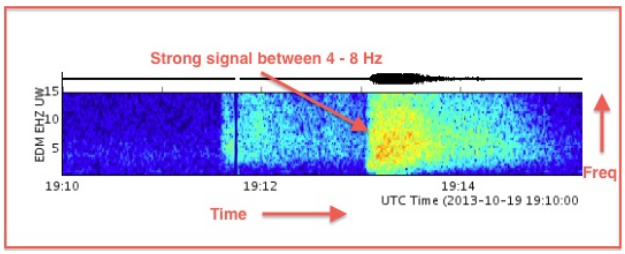
o2

o3

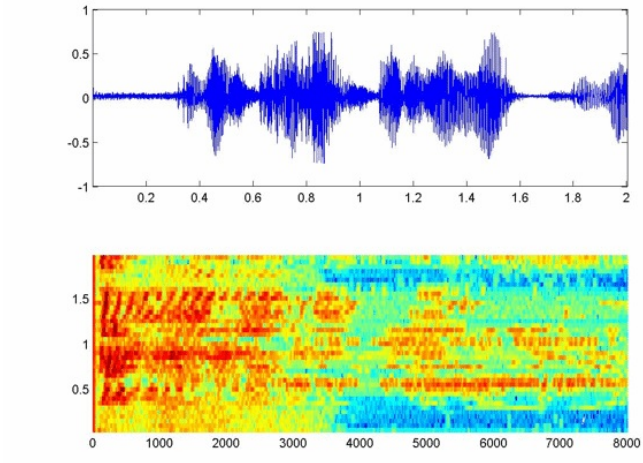
o1 o2 o3 ....

What is a spectrogram? Spectrograms represent the frequency content in the audio as colors in an image. Frequency content of milliseconds chunks is stringed together as colored vertical bars. Spectrograms are basically two-dimensional graphs, with a third dimension represented by colors.

1. **Time** runs from left (oldest) to right (youngest) along the horizontal axis.
2. The vertical axis represents **frequency**, with the lowest frequencies at the bottom and the highest frequencies at the top.
3. The amplitude (or energy or "loudness") of a particular frequency at a particular time is represented by the third dimension, **color**, with dark blues corresponding to low amplitudes and brighter colors up through red corresponding to progressively stronger (or louder) amplitudes.



For a running sound signal the spectrogram would be like this. Notice the blueness to the right of the 2nd image below. That is resulting from the low amplitude signals appearing later in time.



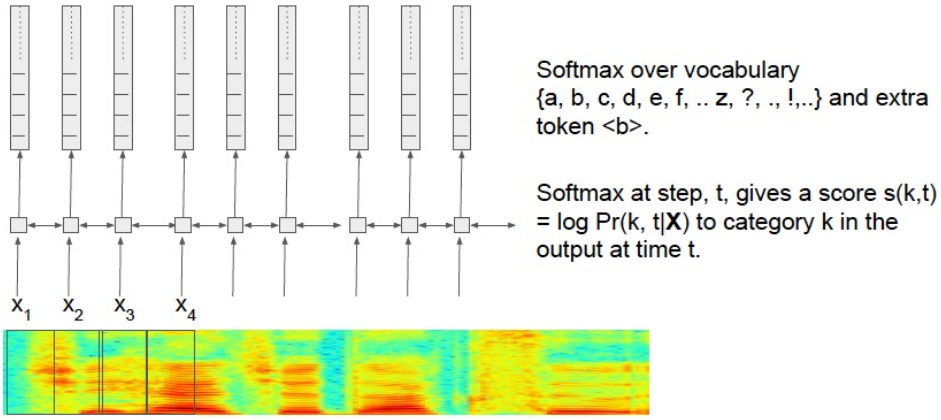
Connectionist Temporal Classification

The first of these models is called Connectionist Temporal Classification (CTC) . X is a sequence of data frames with length T: x1, x2, …, xT, and Y is the output tokens of length L: y1, y2, …, yL. Because of the way the model is constructed, we require T to be greater than L.

Hello World -> H e l l o .....

S -> Hello World

Hello World

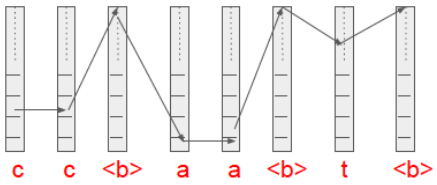


This model has a very specific structure that makes it suitable for speech:

- You get the spectrogram at the bottom (X). You feed it into a bi-directional recurrent neural network, and as a result, the arrow pointing at any time step depends on the entirety of the input data. As such, it can compute a fairly complicated function of the entire data X.
- This model, at the top, has softmax functions at every timeframe corresponding to the input. The softmax function is applied to a vocabulary with a particular length that you're interested in. In this case, you have the lowercase letters a to z and some punctuation symbols. So the vocabulary for CTC would be all that and an extra token called a blank token.
- Each frame of the prediction is basically producing a log probability for a different token class at that time step. In the case above, a score s(k, t) is the log probability of category k at time step t given the data X.

In a CTC model, if you look at just the softmax functions that are produced by the recurring neural network over the entire time step, you'll be able to find the probability of the transcript through these individual softmax functions over time.

Let's take a look at an example (below). The CTC model can represent all these paths through the entire space of softmax functions and look at only the symbols that correspond to each of the time steps.



As seen on the left, the CTC model will go through 2 C symbols, then through a blank symbol, then produce 2 A symbols, then produce another blank symbol, then transition to a T symbol, and then finally produce a blank symbol again.

Connectionist Temporal Classification(CTC) is an algorithm used to deal with tasks like speech recognition, handwriting recognition etc. where just the input data and the output transcription is available but there are no alignment details provided i.e how a particular region in audio for speech or particular region in images for handwriting is aligned to a specific character. Simple heuristics such as giving each character same time won't work since the amount of space each character takes varies in speech from person to person and time to time.

For our speech recognition use-case consider the input speech regions for a particular sentence as input *X=[x1,x2,...,xT*] while expected output as *Y*=[*y*1,*y*2,...,*yU*]* . Given X we are supposed to find accurate Y. CTC algorithm works by taking input X and giving distribution over all possible Y's using which we can make a prediction for final output.

There are challenges which get in the way of us using simpler supervised learning algorithms. In particular:

- Both X and Y can vary in length.
- The ratio of the lengths of X and Y can vary.
- We don't have an accurate alignment (correspondence of the elements) of X and Y.

https://distill.pub/2017/ctc/

The CTC algorithm is *alignment-free* — it doesn't require an alignment between the input and the output. However, to get the probability of an output given an input, CTC works by summing over the probability of all possible alignments between the two. We need to understand what these alignments are in order to understand how the loss function is ultimately calculated.

Hello -> Hulo -> HHHuuulllloooo -> Hulo

26 x 26 x26 ...

26^10

3 -> H, o, p

3 -> H, o, p

HH, Ho, Hp, ... -> HH, Ho, op -> Hello, Hullo, Hyllo, Hollo, Hollow

To motivate the specific form of the CTC alignments, first consider a naive approach. Let's use an example. Assume the input has length six and Y = [c, a, t]. One way to align X and Y is to assign an output character to each input step and collapse repeats.

X_1	X_2	X_3	X_4	X_5	X_6	input (X)
c	c	a	a	a	t	alignment
c		a			t	output (Y)

This approach has two problems.

- Often, it doesn't make sense to force every input step to align to some output. In speech recognition, for example, the input can have stretches of silence with no corresponding output.
- We have no way to produce outputs with multiple characters in a row. Consider the alignment [h, h, e, l, l, l, o]. Collapsing repeats will produce "helo" instead of "hello".

To get around these problems, CTC introduces a new token to the set of allowed outputs. This new token is sometimes called the *blank* token. We'll refer to it here as ϵ. The ϵ token doesn't correspond to anything and is simply removed from the output.

The alignments allowed by CTC are the same length as the input. We allow any alignment which maps to Y *Y* after merging repeats and removing ϵ tokens:

If Y has two of the same character in a row, then a valid alignment must have an ϵ between them. With this rule in place, we can differentiate between alignments which collapse to “hello” and those which collapse to “helo”.

While decoding the output of CTC based on the simple heuristic of highest probability for each position, we might get results which might not make any sense in the real world. To solve this we might employ a different decoder to improve the results. Let's discuss different types of decodings

1. **Best-path decoding** :- This is the generic decoding we have discussed so far. At each position we take the output of the model and find the result with the highest probability.
2. **Beam search decoding** :- Instead of taking a single output from the network every time beam search suggests keeping multiple output paths with highest every probabilities and expand the chain with new outputs and dropping paths having lesser probabilities to keep the beam size constant. The results obtained through this approach are more accurate than using the greedy approach
3. **Beam search with Language Model** :- Beam search provides more accurate results than grid search but still it won't solve the problem of having meaningful results. To solve this we can use a language model along with beam search using both probabilities from the model and the language model to generate final results.

Language models

In order to keep it as simple as possible, just think of a language model as a function taking a sentence as input, which is often only partly constructed, and returning the probability of the last word given all the previous words. What is this good for? Imagine we have the following partly constructed sentence: “tell us a fairy”. Now we hear the next word, but are not able discern whether it should be “tale” or “tail”. These words are homophones, which means they sound alike, although they are spelled differently. A well trained language model should be able to tell us that “tale” is much more probable than “tail”, making the choice straight forward. For further background reading, you can check out the Wikipedia page on language models.