Adam Geitgey Follow
Interested in computers and machine learning. Likes to write about it.
Dec 24, 2016 · 11 min read

Machine Learning is Fun Part 6: How to do Speech Recognition with Deep Learning

Update: *This article is part of a series. Check out the full series: Part 1, Part 2, Part 3, Part 4, Part 5, Part 6 and Part 7!*

You can also read this article in 普通话, 한국어 or Русский.

Speech recognition is invading our lives. It's built into our phones, our game consoles and our smart watches. It's even automating our homes. For just $50, you can get an Amazon Echo Dot—a magic box that allows you to order pizza, get a weather report or even buy trash bags—just by speaking out loud:

Alexa, order a large pizza!

The Echo Dot has been so popular this holiday season that Amazon can't seem to keep them in stock!

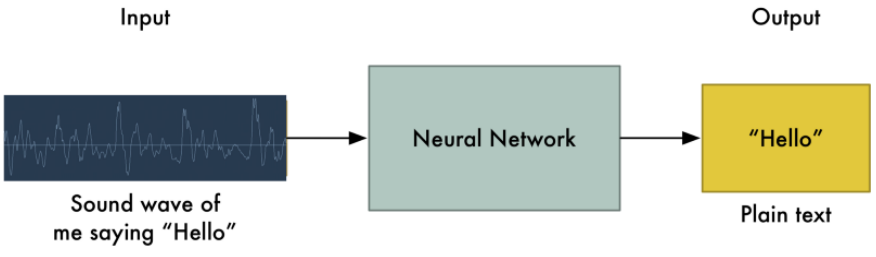
But speech recognition has been around for decades, so why is it just now hitting the mainstream? The reason is that deep learning finally made speech recognition accurate enough to be useful outside of carefully controlled environments.

Andrew Ng has long predicted that as speech recognition goes from 95% accurate to 99% accurate, it will become a primary way that we interact with computers. The idea is that this 4% accuracy gap is the difference between *annoyingly unreliable* and *incredibly useful*. Thanks to Deep Learning, we're finally cresting that peak.

Let's learn how to do speech recognition with deep learning!

Machine Learning isn't always a Black Box

If you know how neural machine translation works, you might guess that we could simply feed sound recordings into a neural network and train it to produce text:



That's the holy grail of speech recognition with deep learning, but we aren't quite there yet (at least at the time that I wrote this—I bet that we will be in a couple of years).

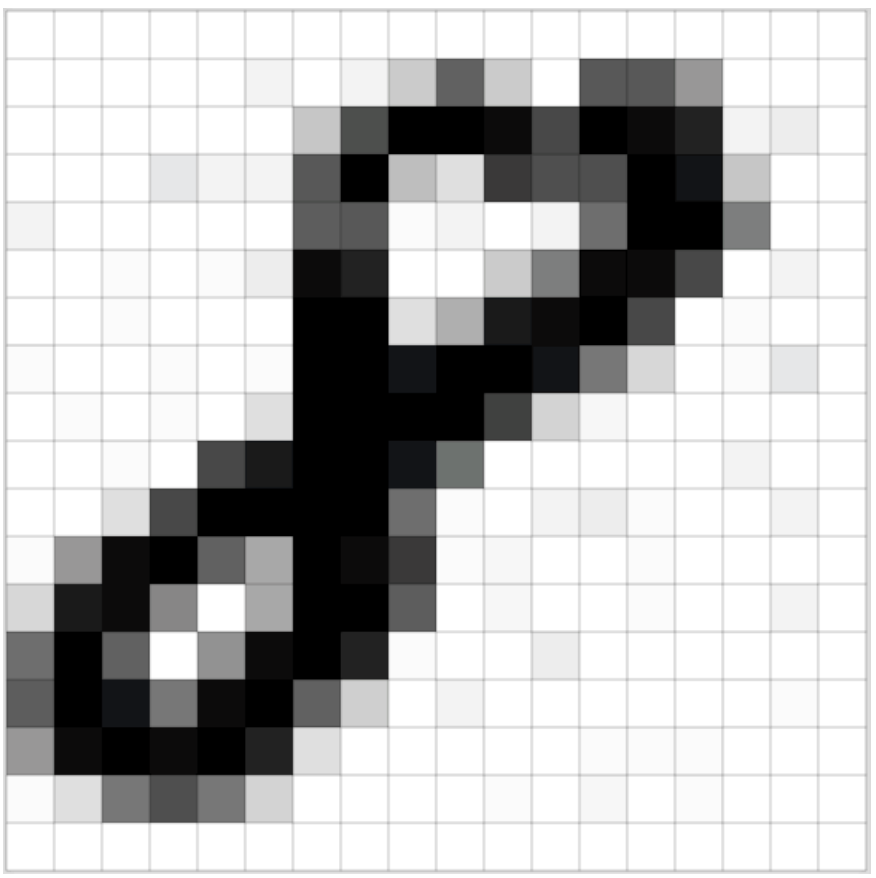
The big problem is that speech varies in speed. One person might say "hello!" very quickly and another person might say "heeeellllllllllllooooo!" very slowly, producing a much longer sound file with much more data. Both both sound files should be recognized as exactly the same text—"hello!" Automatically aligning audio files of various lengths to a fixed-length piece of text turns out to be pretty hard.

To work around this, we have to use some special tricks and extra precessing in addition to a deep neural network. Let's see how it works!

Turning Sounds into Bits

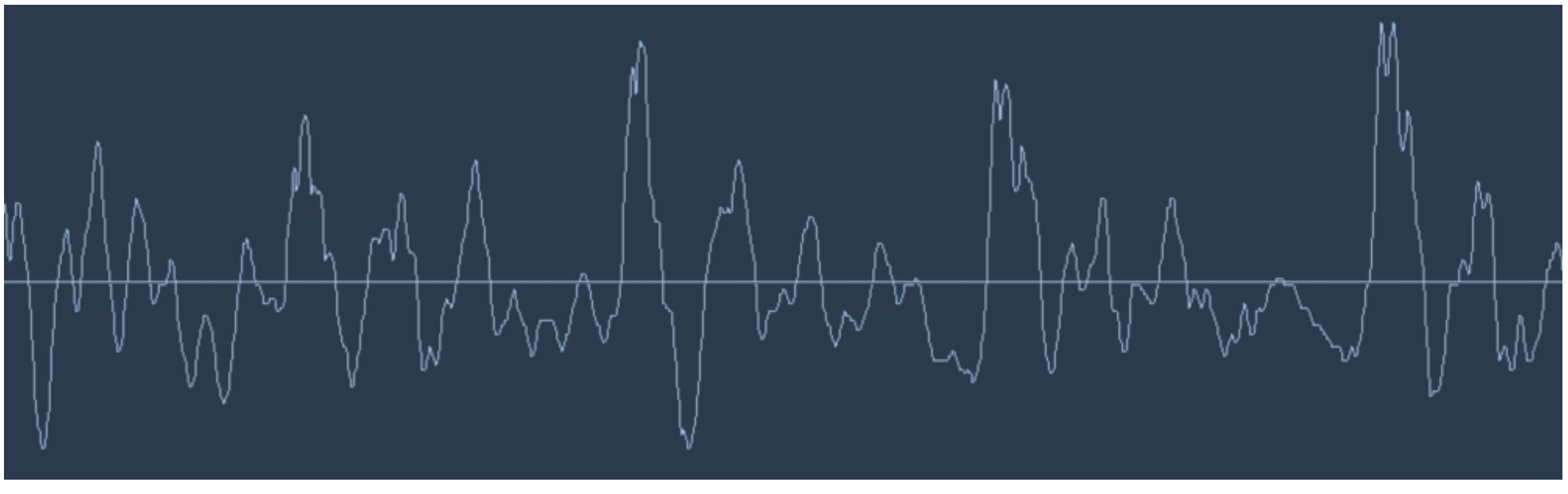
The first step in speech recognition is obvious—we need to feed sound waves into a computer.

In Part 3, we learned how to take an image and treat it as an array of numbers so that we can feed directly into a neural network for image recognition:



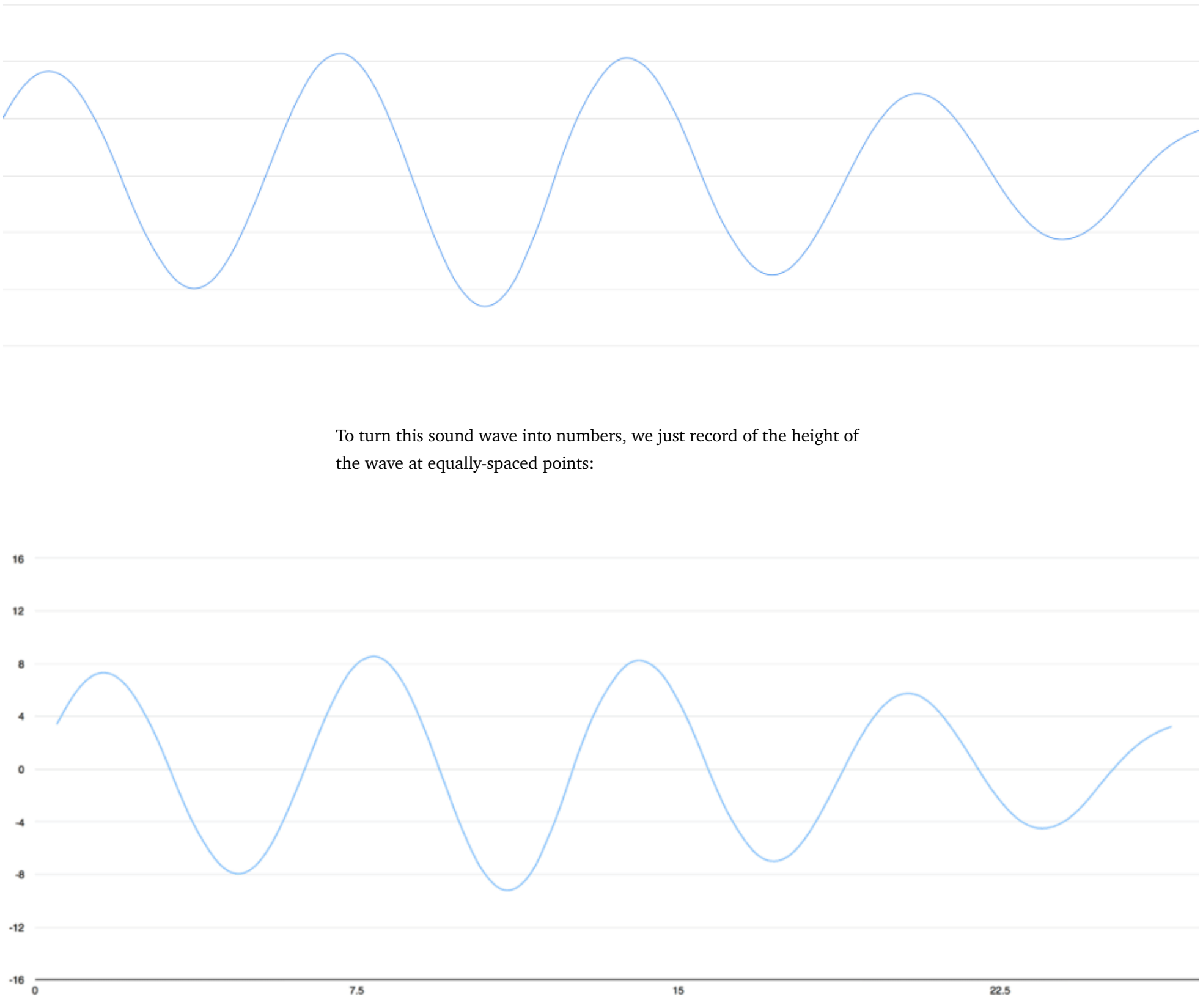
Images are just arrays of numbers that encode the intensity of each pixel

But sound is transmitted as *waves*. How do we turn sound waves into numbers? Let's use this sound clip of me saying "Hello":



A waveform of me saying “Hello”

Sound waves are one-dimensional. At every moment in time, they have a single value based on the height of the wave. Let’s zoom in on one tiny part of the sound wave and take a look:



Sampling a sound wave

To turn this sound wave into numbers, we just record of the height of the wave at equally-spaced points:

This is called *sampling*. We are taking a reading thousands of times a

second and recording a number representing the height of the sound wave at that point in time. That's basically all an uncompressed .wav audio file is.

"CD Quality" audio is sampled at 44.1khz (44,100 readings per second). But for speech recognition, a sampling rate of 16khz (16,000 samples per second) is enough to cover the frequency range of human speech.

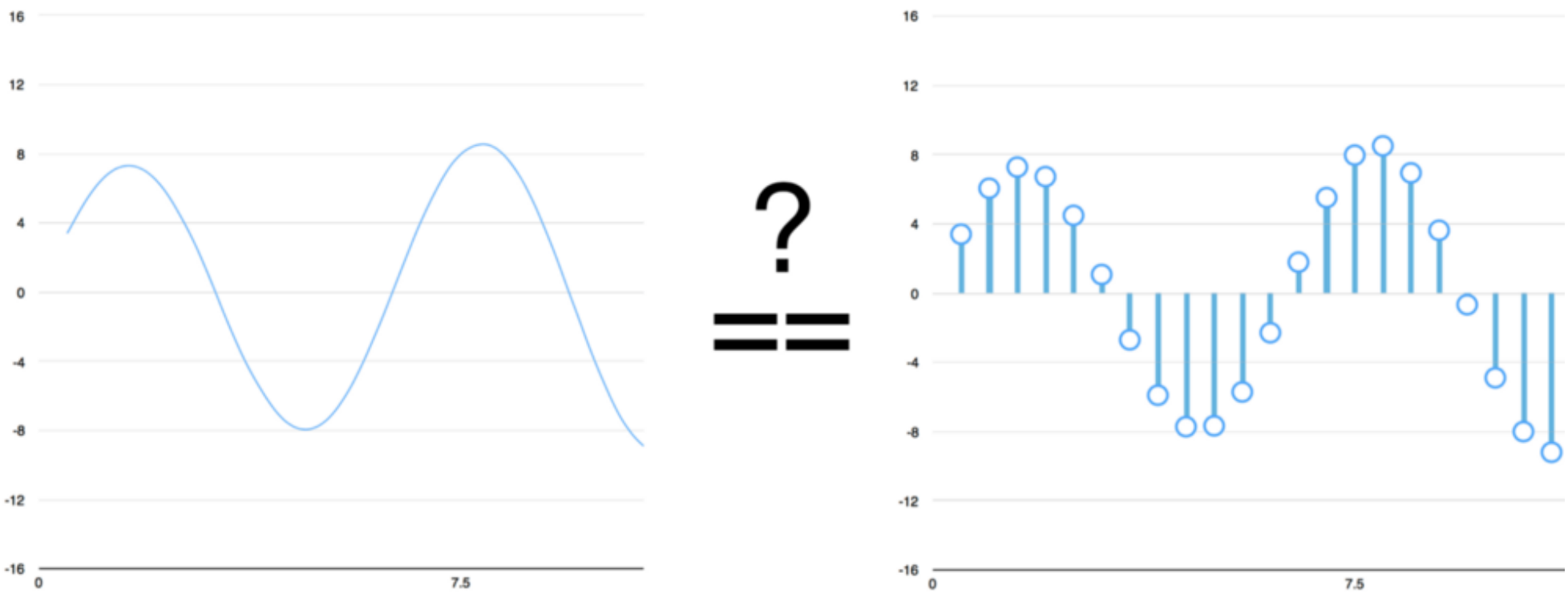
Lets sample our "Hello" sound wave 16,000 times per second. Here's the first 100 samples:

```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41, -169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448
, -397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451, 1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
 4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499, -488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148, -
1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325, 350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544]
```

Each number represents the amplitude of the sound wave at 1/16000th of a second intervals

A Quick Sidebar on Digital Sampling

You might be thinking that sampling is only creating a rough approximation of the original sound wave because it's only taking occasional readings. There's gaps in between our readings so we must be losing data, right?



Can digital samples perfectly recreate the original analog sound wave? What about those gaps?

But thanks to the Nyquist theorem, we know that we can use math to perfectly reconstruct the original sound wave from the spaced-out samples—as long as we sample at least twice as fast as the highest frequency we want to record.

I mention this only because nearly everyone gets this wrong and assumes that using higher sampling rates always leads to better audio quality. It doesn't.

</end rant>

Pre-processing our Sampled Sound Data

We now have an array of numbers with each number representing the

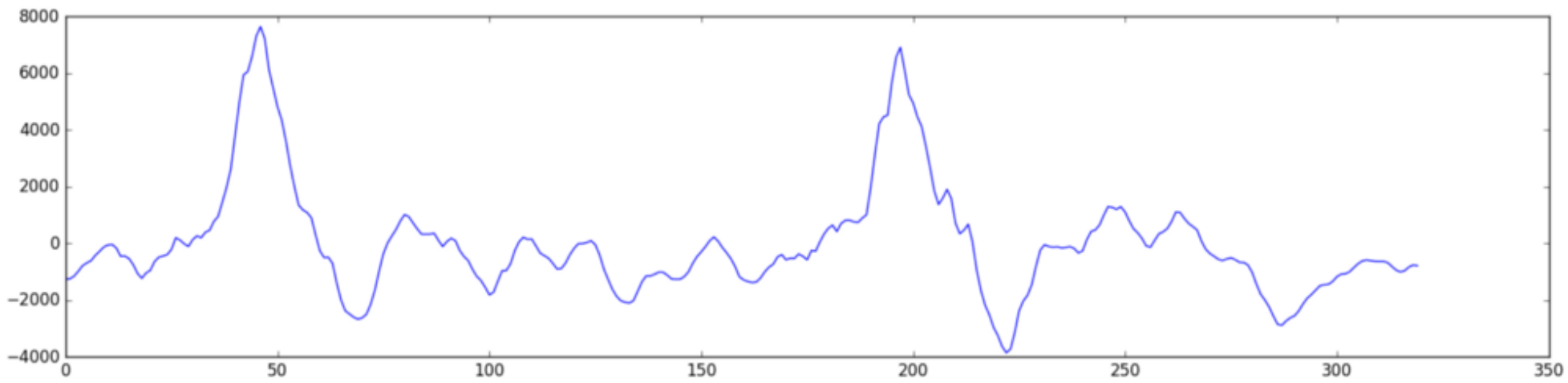
sound wave's amplitude at 1/16,000th of a second intervals.

We *could* feed these numbers right into a neural network. But trying to recognize speech patterns by processing these samples directly is difficult. Instead, we can make the problem easier by doing some pre-processing on the audio data.

Let's start by grouping our sampled audio into 20-millisecond-long chunks. Here's our first 20 milliseconds of audio (i.e., our first 320 samples):

```
[-1274, -1252, -1160, -986, -792, -692, -614, -429, -286, -134, -57, -41, -169, -456, -450, -541, -761, -1067, -1231, -1047, -952, -645, -489, -448
, -397, -212, 193, 114, -17, -110, 128, 261, 198, 390, 461, 772, 948, 1451, 1974, 2624, 3793, 4968, 5939, 6057, 6581, 7302, 7640, 7223, 6119, 5461,
 4820, 4353, 3611, 2740, 2004, 1349, 1178, 1085, 901, 301, -262, -499, -488, -707, -1406, -1997, -2377, -2494, -2605, -2675, -2627, -2500, -2148, -
1648, -970, -364, 13, 260, 494, 788, 1011, 938, 717, 507, 323, 324, 325, 350, 103, -113, 64, 176, 93, -249, -461, -606, -909, -1159, -1307, -1544,
-1815, -1725, -1341, -971, -959, -723, -261, 51, 210, 142, 152, -92, -345, -439, -529, -710, -907, -887, -693, -403, -180, -14, -12, 29, 89, -47, -
398, -896, -1262, -1610, -1862, -2021, -2077, -2105, -2023, -1697, -1360, -1150, -1148, -1091, -1013, -1018, -1126, -1255, -1270, -1266, -1174, -10
03, -707, -468, -300, -116, 92, 224, 72, -150, -336, -541, -820, -1178, -1289, -1345, -1385, -1365, -1223, -1004, -839, -734, -481, -396, -580, -52
7, -531, -376, -458, -581, -254, -277, 50, 331, 531, 641, 416, 697, 810, 812, 759, 739, 888, 1008, 1977, 3145, 4219, 4454, 4521, 5691, 6563, 6909,
6117, 5244, 4951, 4462, 4124, 3435, 2671, 1847, 1370, 1591, 1900, 1586, 713, 341, 462, 673, 60, -938, -1664, -2185, -2527, -2967, -3253, -3636, -38
59, -3723, -3134, -2380, -2032, -1831, -1457, -804, -241, -51, -113, -136, -122, -158, -147, -114, -181, -338, -266, 131, 418, 471, 651, 994, 1295,
 1267, 1197, 1291, 1110, 793, 514, 370, 174, -90, -139, 104, 334, 407, 524, 771, 1106, 1087, 878, 703, 591, 471, 91, -199, -357, -454, -561, -605,
-552, -512, -575, -669, -672, -763, -1022, -1435, -1791, -1999, -2242, -2563, -2853, -2893, -2740, -2625, -2556, -2385, -2138, -1936, -1803, -1649,
 -1495, -1460, -1446, -1345, -1177, -1088, -1072, -1003, -856, -719, -621, -585, -613, -634, -638, -636, -683, -819, -946, -1012, -964, -836, -762,
 -788]
```

Plotting those numbers as a simple line graph gives us a rough approximation of the original sound wave for that 20 millisecond period of time:



This recording is only *1/50th of a second long*. But even this short recording is a complex mish-mash of different frequencies of sound. There's some low sounds, some mid-range sounds, and even some high-pitched sounds sprinkled in. But taken all together, these different frequencies mix together to make up the complex sound of human speech.

To make this data easier for a neural network to process, we are going to break apart this complex sound wave into it's component parts. We'll break out the low-pitched parts, the next-lowest-pitched-parts, and so on. Then by adding up how much energy is in each of those frequency bands (from low to high), we create a *fingerprint* of sorts for this audio snippet.

Imagine you had a recording of someone playing a C Major chord on a piano. That sound is the combination of three musical notes— C, E and G—all mixed together into one complex sound. We want to break apart that complex sound into the individual notes to discover that they were C, E and G. This is the exact same idea.

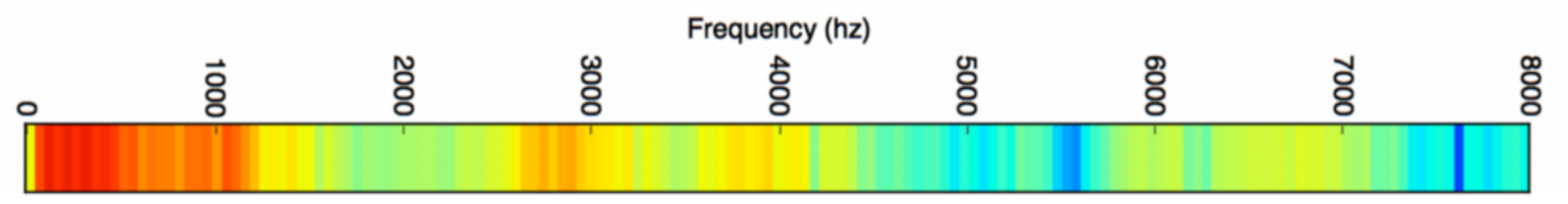
We do this using a mathematic operation called a *Fourier transform*. It breaks apart the complex sound wave into the simple sound waves that make it up. Once we have those individual sound waves, we add up how much energy is contained in each one.

The end result is a score of how important each frequency range is, from low pitch (i.e. bass notes) to high pitch. Each number below represents how much energy was in each 50hz band of our 20 millisecond audio clip:

[110.97481594791122, 166.61537247955155, 180.43561044211469, 175.09309469913353, 180.0168691095916, 176.00619977472167, 179.79737781786582, 173.53025213548219, 176.87177119846058, 170.42684732853121, 159.26023828556598, 163.24469810981628, 149.15527353931867, 154.34196586290136, 151.46179061113972, 152.99674239973979, 143.98878156117371, 156.6033737693738, 155.78237530428544, 157.17930941017838, 146.28632297509679, 164.37233032929228, 158.1282656446088, 147.23266451005145, 133.26597973863801, 116.5170100028831, 116.85501120577126, 115.40519005123537, 120.85619013711488, 112.48406123161091, 111.80244759457571, 92.590676871856431, 105.75863927434719, 95.673146446282971, 90.391748128064208, 79.355818055314899, 86.080143147713926, 84.748200268709567, 83.050569583779065, 86.207180262242758, 90.252031938154076, 89.361567351948437, 90.917307309643206, 90.746777849123049, 86.726552726337033, 85.709412745066928, 95.938840816664865, 99.09254575917069, 96.632437741434885, 103.23961231666669, 105.80328302591124, 109.53029281234707, 116.46408227060996, 129.20890691592615, 130.43460361780441, 138.15581799444712, 128.25056761852832, 138.14492240466387, 140.0352714810314, 128.15138139429752, 123.93018478493934, 121.19289035588113, 119.03159255422509, 114.23027889344033, 119.1717342154997, 101.02560719093093, 110.91192243698025, 106.04872005953503, 100.86977927980999, 92.123301579000341, 94.376766266598295, 97.850709698634489, 113.37126364077845, 110.24526597732718, 113.72249347908021, 120.63960942628063, 122.06482553759932, 117.96716716036715, 120.87682744817975, 125.06097381947157, 111.57319012901624, 115.54483708595507, 116.99850750130265, 114.40659619324526, 79.869543980883975, 104.83111191845597, 104.66218602004588, 104.91691734582642, 97.143620527536072, 78.434597811117835, 82.214144782667248, 67.246072805959614, 66.578937262360313, 74.100307226086798, 64.861423011415653, 59.167561212002269, 62.479712687304911, 63.568362396107467, 55.906096471453267, 42.790802909362839, 55.693923524361097, 50.776364877715011, 41.196111220671298, 51.062413666348945, 58.493563858289065, 53.081835042922769, 73.060663128159547, 68.21625202122361, 66.7701034934517, 59.766255124915202, 35.413635503802389, 22.705615809958832, 16.458048045346381, 44.910670465379937, 59.282513769840705, 69.241393677323856, 81.778634874076346, 88.409923803546008, 94.688033733251245, 96.640867526244051, 91.806226496828543, 94.570526932206619, 99.250924315589074, 97.899164767741183, 75.176507616277235, 80.947474423758905, 71.859103451990862, 93.863684037461738, 96.757146539348298, 96.528614354976241, 99.366456533638413, 102.18717608176904, 102.06596663023235, 101.78493139911082, 103.7883358299547, 99.915220403870748, 107.43478470929935, 104.46449552620618, 105.70789868195298, 101.10596541338749, 100.75737831526195, 91.742897073196886, 88.307278943069093, 90.936627732905492, 71.134275744339803, 72.504304977841457, 76.233185506299705, 63.281284410272761, 45.380164336858961, 43.018963766250437, 49.133789791276826, 53.507751009532953, 48.586423555688746, -4.4730776113028883, 50.833000650183408, 51.003802143009629, 39.577356593427531, 47.096919248906332, 55.442197175664383, 56.967128095484341, 49.383247263177985]

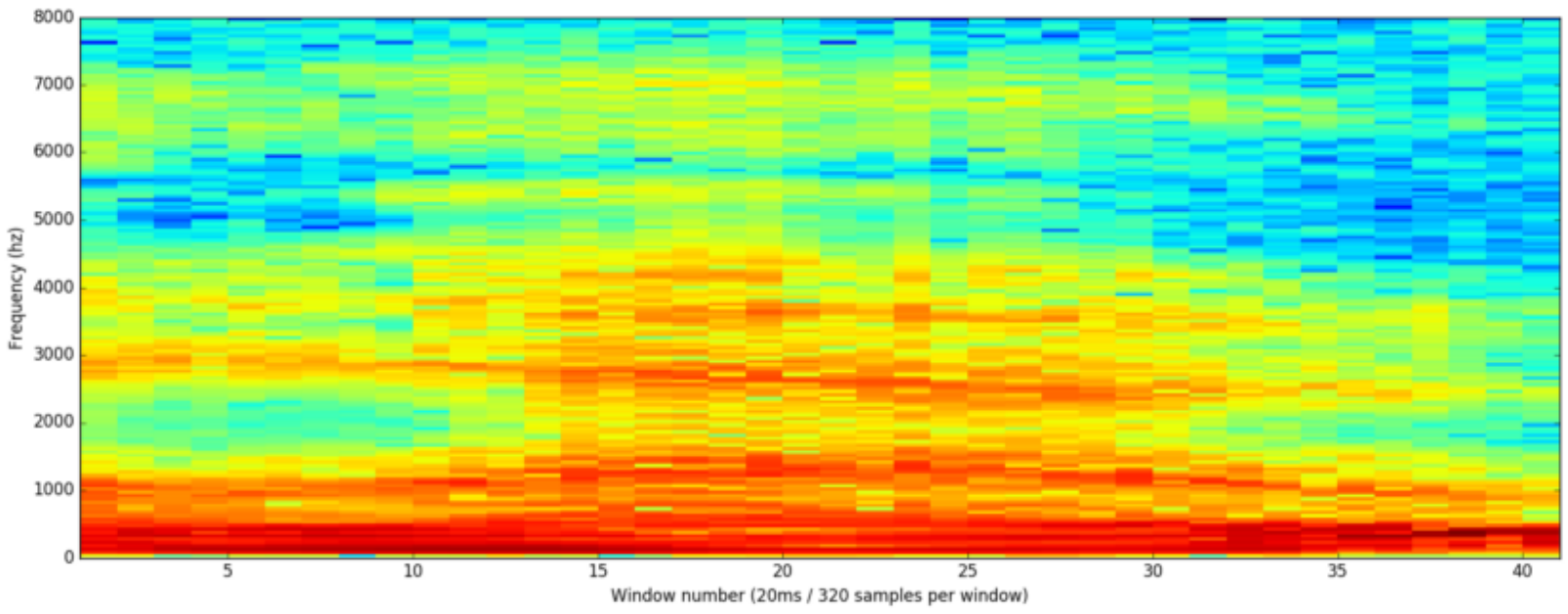
Each number in the list represents how much energy was in that 50hz frequency band

But this is a lot easier to see when you draw this as a chart:



You can see that our 20 millisecond sound snippet has a lot of low-frequency energy and not much energy in the higher frequencies. That's typical of "male" voices.

If we repeat this process on every 20 millisecond chunk of audio, we end up with a spectrogram (each column from left-to-right is one 20ms chunk):



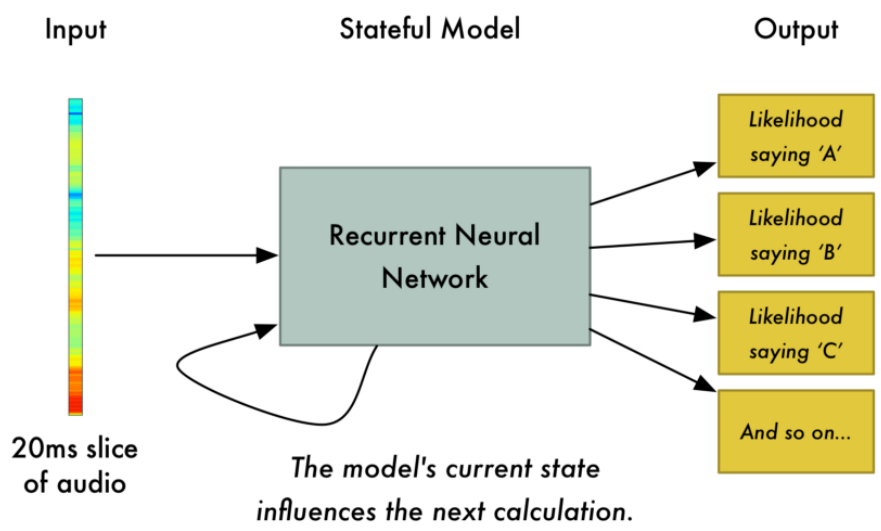
The full spectrogram of the "hello" sound clip

A spectrogram is cool because you can actually *see* musical notes and other pitch patterns in audio data. A neural network can find patterns

in this kind of data more easily than raw sound waves. So this is the data representation we'll actually feed into our neural network.

Recognizing Characters from Short Sounds

Now that we have our audio in a format that's easy to process, we will feed it into a deep neural network. The input to the neural network will be 20 millisecond audio chunks. For each little audio slice, it will try to figure out the *letter* that corresponds the sound currently being spoken.



We'll use a recurrent neural network—that is, a neural network that has a memory that influences future predictions. That's because each letter it predicts should affect the likelihood of the next letter it will predict too. For example, if we have said "HEL" so far, it's very likely we will say "LO" next to finish out the word "Hello". It's much less likely that we will say something unpronounceable next like "XYZ". So having that memory of previous predictions helps the neural network make more accurate predictions going forward.

After we run our entire audio clip through the neural network (one chunk at a time), we'll end up with a mapping of each audio chunk to the letters most likely spoken during that chunk. Here's what that mapping looks like for me saying "Hello":

- Our neural net is predicting that one likely thing I said was “HHHEE_LL_LLLOOO”. But it also thinks that it was possible that I said “HHHUU_LL_LLLOOO” or even “AAAUU_LL_LLLOOO”.

  We have some steps we follow to clean up this output. First, we’ll replace any repeated characters a single character:

  - HHHEE_LL_LLLOOO becomes HE_L_LO
  - HHHUU_LL_LLLOOO becomes HU_L_LO
  - AAAUU_LL_LLLOOO becomes AU_L_LO

  Then we’ll remove any blanks:

  - HE_L_LO becomes HELLO
  - HU_L_LO becomes HULLO
  - AU_L_LO becomes AULLO

  That leaves us with three possible transcriptions—“Hello”, “Hullo” and “Aullo”. If you say them out loud, all of these sound similar to “Hello”. Because it’s predicting one character at a time, the neural network will come up with these very *sounded-out* transcriptions. For example if you say “He would not go”, it might give one possible transcription as “He wud net go”.

  The trick is to combine these pronunciation-based predictions with likelihood scores based on large database of written text (books, news articles, etc). You throw out transcriptions that seem the least likely to be real and keep the transcription that seems the most realistic.

  Of our possible transcriptions “Hello”, “Hullo” and “Aullo”, obviously “Hello” will appear more frequently in a database of text (not to mention in our original audio-based training data) and thus is

probably correct. So we'll pick "Hello" as our final transcription instead of the others. Done!

Wait a second!

You might be thinking *"But what if someone says 'Hullo'? It's a valid word. Maybe 'Hello' is the wrong transcription!"*

"Hullo! Who dis?"

Of course it is possible that someone actually said "Hullo" instead of "Hello". But a speech recognition system like this (trained on American English) will basically never produce "Hullo" as the transcription. It's just such an unlikely thing for a user to say compared to "Hello" that it will always think you are saying "Hello" no matter how much you emphasize the 'U' sound.

Try it out! If your phone is set to American English, try to get your phone's digital assistant to recognize the world "Hullo." You can't! It refuses! It will always understand it as "Hello."

Not recognizing "Hullo" is a reasonable behavior, but sometimes you'll find annoying cases where your phone just refuses to understand something valid you are saying. That's why these speech recognition models are always being retrained with more data to fix these edge cases.

Can I Build My Own Speech Recognition System?

One of the coolest things about machine learning is how simple it sometimes seems. You get a bunch of data, feed it into a machine learning algorithm, and then magically you have a world-class AI system running on your gaming laptop's video card… *Right*?

That sort of true in some cases, but not for speech. Recognizing speech is a hard problem. You have to overcome almost limitless challenges: bad quality microphones, background noise, reverb and echo, accent variations, and on and on. All of these issues need to be present in your training data to make sure the neural network can deal with them.

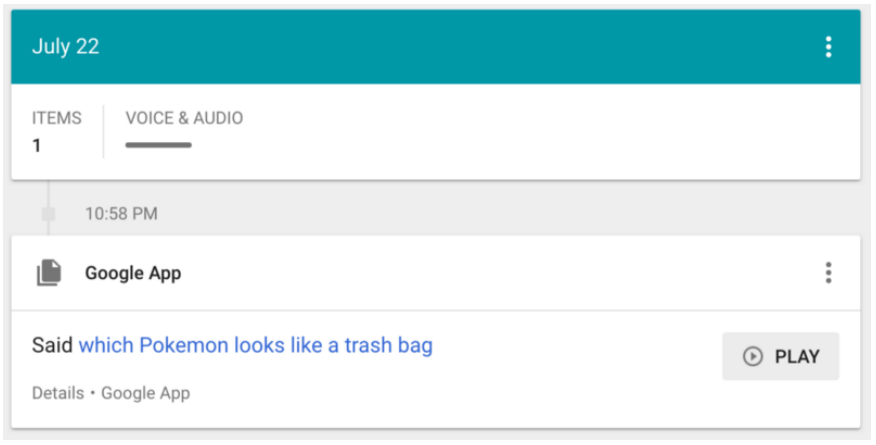
Here's another example: Did you know that when you speak in a loud room you unconsciously raise the pitch of your voice to be able to talk over the noise? Humans have no problem understanding you either way, but neural networks need to be trained to handle this special case. So you need training data with people yelling over noise!

To build a voice recognition system that performs on the level of Siri,

Google Now!, or Alexa, you will need a *lot* of training data—far more data than you can likely get without hiring hundreds of people to record it for you. And since users have low tolerance for poor quality voice recognition systems, you can't skimp on this. No one wants a voice recognition system that works 80% of the time.

For a company like Google or Amazon, hundreds of thousands of hours of spoken audio recorded in real-life situations is *gold*. That's the single biggest thing that separates their world-class speech recognition system from your hobby system. The whole point of putting *Google Now!* and *Siri* on every cell phone for free or selling $50 *Alexa* units that have no subscription fee is to get you to **use them as much as possible**. Every single thing you say into one of these systems is ***recorded forever*** and used as training data for future versions of speech recognition algorithms. That's the whole game!

Don't believe me? If you have an Android phone with *Google Now!*, click here to listen to actual recordings of yourself saying every dumb thing you've ever said into it:



You can access the same thing for Amazon via your Alexa app. Apple unfortunately doesn't let you access your Siri voice data.

So if you are looking for a start-up idea, I wouldn't recommend trying to build your own speech recognition system to compete with Google. Instead, figure out a way to get people to give you recordings of themselves talking for hours. The data can be your product instead.

Where to Learn More

The algorithm (roughly) described here to deal with variable-length audio is called Connectionist Temporal Classification or CTC. You can read the original paper from 2006.

Adam Coates of Baidu gave a great presentation on Deep Learning for Speech Recognition at the Bay Area Deep Learning School. You can watch the video on YouTube (his talk starts at 3:51:00). Highly recommended.

. . .

If you liked this article, please consider **signing up for my Machine Learning is Fun! email list**. I'll only email you when I have something new and awesome to share. It's the best way to find out when I write more articles like this.

You can also follow me on Twitter at @ageitgey, email me directly or find me on linkedin. I'd love to hear from you if I can help you or your team with machine learning.

Now continue on to Machine Learning is Fun! Part 7!