Intro

How was everyone's new year?

(audience)

Sounds like everyone had fun. I wasted mine remaking 50-year-old tech from scratch.

Look, I was bored on winter break and I was playing around with an old text-to-speech software called Software Automatic Mouth (SAM). SAM was one of the earliest consumer TTS software's available. Released in 1982, it didn't have the luxury of neural networks running locally like we do today. There's no AI in this bad boy, it had to run on the god damn Atari.

So it made me really curious. How does an ancient piece of software like SAM manage to turn a sentence into spoken words.

And uhh it was winter break so I figured it would be a fun afternoon project. That was 10 afternoons ago.

Okay so my text-to-speech system has four parts:

- You've got normalization, where you convert unspeakable text to a more speakable form. For example, "123" becomes "one hundred twenty-three."
- Tokenization, where you break up the sentence into word sized chunks called "tokens."
- Phoneme-ization (I tried to make it work man), this is where you turn each one of those tokens into phonemes (which are the smallest unit of speech in the English language).
- and finally Synthesization, where you turn that list of phonemes into actual sounds to be spoken.

Normalization

The first step, normalization, is the most annoying step and easiest to fuck up.

The approach I took towards it is a fuck ton of regular expressions. I defined a ruleset that looked something like this.

That first value there is the pattern. It's what the computer looks for in the text. And if it finds that pattern, it replaces it with that second value.

For example, the rule for the Doctor abbreviation looks something like this. The pattern here searches for the text "DR period" and the second value specifies that that should be replaced with the word "doctor."

Rinse and repeat about a hundred times and you've got normalized text.

I wrote tons of tests for this step to make sure everything works as intended! And first time around, it did not. It took me many, many hours of debugging for all the tests to pass and I was finally able to move onto tokenization.

Tokenization

This step was comparatively pretty easy. Afterall, python already has a function to do this exact thing.

But unfortunately, I can't just throw everything into split and call it a day. What about silences? What about pitch? It's sometimes useful to keep data about a word that can't be inferred from just the word alone.

For this reason, I wrote 2 classes, one for tokens and one for a list of tokens. This just makes it super easy to work with this data in the future because I've put all the information I need into one object.

Turn Into Phonemes

The next step is to take this list of tokens and turn each one of those tokens into phonemes.

But let's take a step back. Why the hell are we turning the words into phonemes anyways? The letters are right there! Just pronounce those.

That might work in a language that actually makes sense, like Spanish. But unfortunately, we speak English and our spelling rules are nonexistent.

The letter "E" alone can be pronounced 9 different ways depending on the word.

So instead of just pronouncing each letter, we convert each word into a list of phonemes.

The specific set of phonemes that we'll be using comes from the Advanced Research Projects Agency (ARPA). Back in the 70s they were studying computerized speech and made a list of standardized phonemes that's easy for computers to deal with called the ARPAbet. And it looks something like this.

The reason I'm using this list in particular is because the Speech Group over at Carnegie Mellon released a big list of about 130k words and their phonemes in this format and it's going to save us a ton of time. So everyone say "Thank you, Carnegie Mellon."

(audience)

So I downloaded it and I converted it into an SQL database just so it's easier and faster to work with.

And just in case a word isn't in the dictionary, I added a fallback pronunciation function which is basically just to say each letter. There are more complicated algorithms for this but my goal here was just to get something working. So this function can easily be upgraded later when I feel like looking at this stupid codebase again.

Synthesis

Okay we've got basically everything we need to start synthesizing. I'm gonna be glossing over some of this because human speech is very, very complicated and there are a million nuances and caveats to what I'm saying. So this is gonna be pretty broad.

The goal with this synthesizer is to mimic a human vocal tract. When you speak, your vocal cords vibrate, and your mouth and tongue shape creates resonances called formants that turn that vibration into a recognizable speech sound.

You can think of these formants as filters, and if you're a linguist in the audience pretend you didn't hear that.

So what we can do is have a base sound, and through different filters we can create different vocal sounds.

For my system, we use 3 filters but more filters will create a more "human" sound (albeit with diminishing returns).

Let's take a look at an example. Here at the top, is a simulated raw vibration that would come from your vocal cords. That sounds like this.

(play raw buzz)

Here in the middle is that same buzz after we've applied 3 filters to it to make it sound like an "E" sound.

(play final vowel sound)

And here at the bottom is a real human “E” sound for comparison. See how it has a fat line at the bottom, a thinner line above it, and then a faint line above that? Well, we can see the same pattern in our synthetic “E” sound. Fat line at the bottom, thinner line above it, faint line on top.

And with better filters we can get closer to a human “E” sound.

This approach works great for vowels, but consonants are different beasts entirely.

Instead of clean formants, something like an "S" sound uses filtered noise.

(play S sound)

While a "T" is just silence followed by a burst.

(play T sound)

So our synthesizer needs different strategies for different sound types. But we’ll be here all night if I went over each one.

Now, all that’s left to do is to find these filter values for the vowels and figure out the noise parameters for everything else. Luckily, the work is already done for us thanks to a paper from Bell Labs published in 1952 by Gordon E. Peterson and Harold L. Barney. Everyone say “Thank you Peterson and Barney”

(audience)

After slapping in a chart from them and playing around with the numbers, I got something sort of human sounding.

(play demo audio)

And that’s as good as I care to make it.

Now, we just loop over each token, synthesize the phonemes in that token into sound, then play that sound.

Live Demo

Here’s the part I’m nervous for.

(Show off the TTS in real time with audience provided phrases after doing a simple “I love the Dallas Hacker’s Association” demo)

Outro

Thank you all for listening. Everything is of course on my GitHub or you can just scan this QR code here. In the meantime, I believe I have some time for questions.