CS 229, Project Progress Report

SUNet ID: 06044535
Name: Ajay Shanker Tripathi

Title: "Voice Transmogrifier: Spoofing My Girlfriend's Voice"

Project Category: Audio and Music

The project idea is an easy-to-state supervised learning problem: the input is me saying something, and the output is that same thing being said in my girlfriend's voice. So, for example, I can say "Ajay is always right" in my deep american-accent voice, and my system should output my girlfriend's cute vietnamese-accent voice saying "Ajay is always right". Aside from this more personal application, there is obviously the application to sabotage. E.g. emulating the voice of a politician might be an easy way to spread discord (if you're into that kind of thing).

For brevity, I'll be refering to my girlfriend as Anh, her first name.

At a high level, the main plan of attack is to do the input to output tranformation by using a neural network. After trying many candidates, I ended up with a home-made architecture which I affectionately named ANH-NET, which is short for **A**uxiliary wave**N**et **H**armonizing neural **NET**work.

The data to learn from is relatively simple: I do a reading of some long and varied passage, and I have Anh do the same exact reading. The only tricky part is, to learn the correct input-to-output mapping, I need to make sure the input-output pairs I'm learning from actually match. In particular, the two readings need to be perfectly aligned, with her speaking right on top of me.

Now that I've given the high level overview of the problem, what follows is the concrete details for getting the voice transmogrifier to work.

I made a simple reading consisting of ~1300 words, containing varied passages from articles, Wikipedia, and a few pages of Harry Potter. I read through the passage twice, taking approximately 16 minutes. My girlfriend read through the exact same passage, taking 20 minutes. In order to cheaply increase the amount of data, I always independently generate additive white gaussian noise at 1% the volume each time I train on the data. This also has the added benefit of making the learned voice transmogrifier more robust to simple noise.

The alignment of my and my girlfriend's readings is done via a Dynamic-Time-Warping (DTW) algorithm and a variable phase vocoder. The DTW tells a function $t_{out} = f(t_{in})$, which maps times in the original signal to times in the warped signal such that the warped signal will match up with the reference. This can be seen in the following figure.

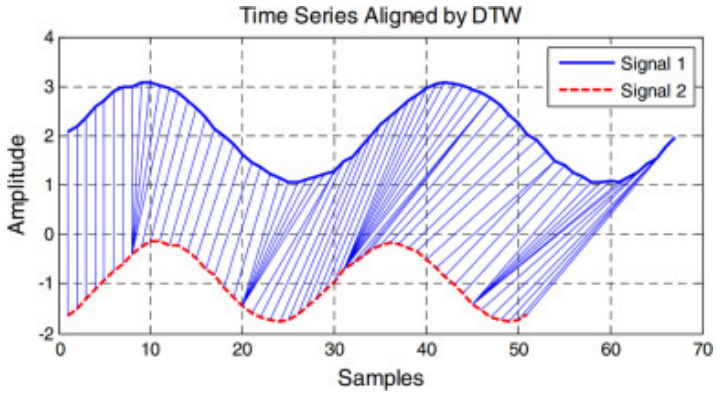

Figure 1: Example time warping, showing how times in an original signal that's to be warped map to times in a reference signal.

Crudely stated, this is done by breaking each audio signal into 20ms chunks, featurizing those chunks, and then running the edit-distance dynamic programming algorithm. The edit-distance algorithm is an efficient way of aligning two sequences with the fewest number of insertions, deletions, and replacements. By overwhelming consensus, the best featurization for voice data is the Mel-frequency Cepstral coefficients (MFCCs), which, kind of like the STFT, maps each windowed chunk of the audio signal to a vector of d "frequency coefficients". This featurization is known to grasp all of the important properties of voice data, while having very nice properties like the same phonemes being said by different people are still similar in this feature space. So two chunks are said to be aligned when the cosine-similarity between their MFCCs is high. In this manner, the DTW is performed to align chunks from my audio signal and Anh's audio signal. Here, I used the Librosa python audio processing library.

Because whole insertions and deletions of chunks of the signal will lead to the audio skipping, I take the discrete edits made and do a smooth interpolation through them (using a monotonic smooth spline interpolator, since time can't be allowed to go backwards). This intelligent smoothing step is of my own invention. Doing so gives us the continuous warping function f. An example summarizing all the steps to get to this warping function can be seen in the following time-warping plot.

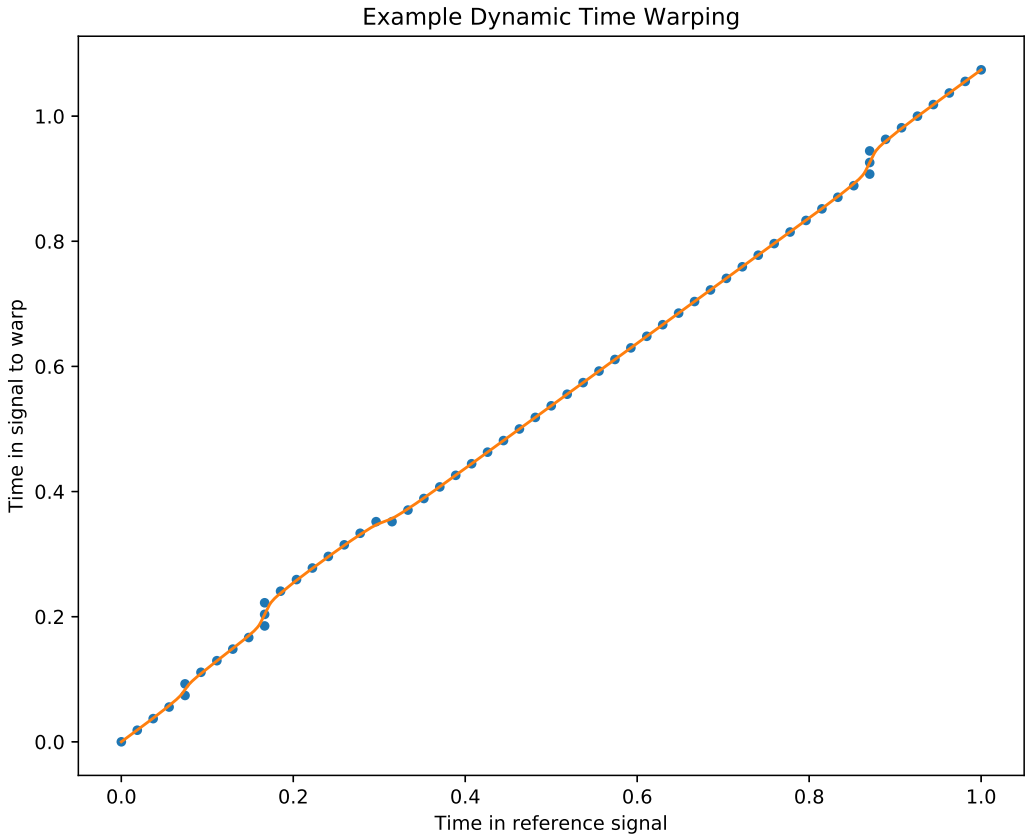

Figure 2: Plot showing the results of the DTW algorithm. The x axis represents a time in the reference signal, and the y axis represents a time in the original signal to be warped. The dots represent discrete chunks which the DTW program says are times that should line up in the two signals. The orange line is the smooth monotonic interpolation function f through those discrete chunks. Notice how there are periods where the discrete DTW is going purely vertically (because the signal-to-be-warped is too slow, corresponding to "deletions"). The smooth interpolator makes it so that, instead of having huge jumps in the audio, there will be a smooth speedup/slowdown.

Given the warping function, we use a variable phase vocoder, which essentially speeds-up/slows-down an audio signal without changing the pitch (i.e. no "chipmunk effect"). Roughly, it does this by doing interpolation in the STFT domain of the signal.

Though stately quickly, a lot of hairy audio-processing went into getting the input/output readings to align, but the point of this paper is the machine learning. Since the hairy details of this pre-processing step are a lengthy tangent, suffice it to say that I mostly successfully got the input and output readings aligned so that it sounds like two people reading at the same time perfectly in sync. However, there were still problems.

Anh has a very heavy Vietnamese accent, so she has non-standard pronounciations, emphases,

and accents littered throughout her reading. For example, she has trouble pronouncing "girl" and "world" because of the juxtaposition of "r" and "l". As another example, she often says the word "food" with an upward inflection. While these sound similar for a human, the actual underlying waveforms are vastly different, and unfortunately, even under the MFCC featurization, aren't similar. So around 10% of the time, there are heavy desyncs in the audio. These desyncs render that 10% of the data worse than useless, because my neural net will still try to fit to them even though they aren't at all reflective of the desired mapping. Luckily, however, I still end up succeeding despite this 10% of the data hurting me.

After completing the allignment, I had my data, which consisted of 16 minutes of audio from me and Anh. I downsample the audio from 44 kHz to 16 kHz since, for voice data, that's more than good enough to not detect any distortion. So I finally end up with the training input and output sequences consisting of ~15 million samples each.

Next comes with the actual system that learns to do the input-output mapping from my voice to Anh's voice. Traditionally, in signal processing, this is done using LTI filters. So as a baseline, I considered an LTI filter with 8 causal IIR taps, and 17 non-causal FIR taps. Finding the best weights for this baseline was just a simple linear regression problem, which I solved in a few lines of python.

Next, I considered various neural network architectures. One major option was to use Recurrent Neural Networks (RNNs) which could operate well for analyzing sequences. In many audio processing contexts, such as classifying the sex or age of a speaker, running RNNs on a featurization of the audio can be a very successful approach. However, for audio generation in particular, they are not known to perform very well. For audio generation, the strongest model known currently is Wavenet, which was developed recently by Google's Deepmind. I ended up using this model for inspiration to create ANH-NET.

Essentially, Wavenet is a bunch of successive dilated causal convolution layers. The following figure provides this high level overview of Wavenet.

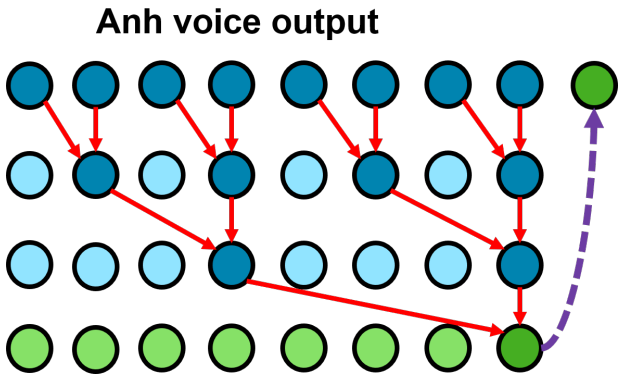

Figure 3: High level overview of Wavenet's successive dilated convolutions. Each circle represents a sample from the sequence.

The power of the dilated convolutions is that it gets you access to an exponentially large receptive field quite cheaply. As can be seen in the figure, each output of the first conv layer gets to

see two samples of the input. The second layer's outputs get to see 4 samples of the input. The third layer's output gets to see 8 samples of the input. As was suggested in the original wavenet paper, I use $D = 9$ layers, which gives the last layer a dilation factor of 512. Furthermore, I repeat this growing dilation from 1 to 512 a total of $K = 3$ times. This ends up giving the Wavenet a receptive field of ~3000 samples, or about 190ms. That is to say, the next sample is predicted using the previous 190ms of samples.

One final large point of the Wavenet architecture is that, instead of using ReLU's as the activation function for the convolutions (as is customary), a gated activation is used, very similar to LSTMs. So, in particular, instead of using

$$a = ReLU(Wx)$$

for the activation, Wavenet uses

$$a = tanh(W_1 x) \circ sigm(W_2 x)$$

which is called a "gated" activation. This is used because, like in LSTM's, it helps gradients propogate through long sequences when training.

True to its name, what ANH-NET does is takes a wavenet and harmonizes it with an auxiliary wavenet. So I have one causal wavenet for Anh's signal, but I also have another non-causal wavenet for my signal, which provides past and future context that can help guide the wavenet for Anh's signal. So the high level overview of ANH-NET is the following figure.

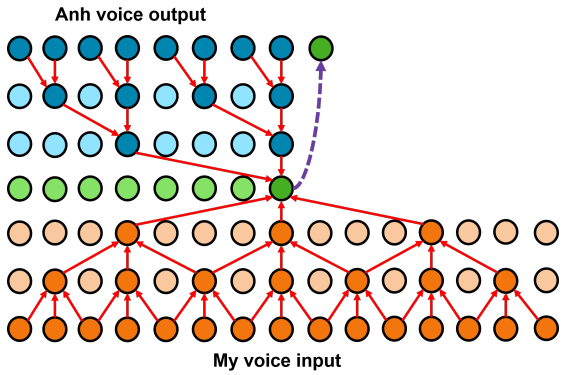

Figure 4: High level overview of ANH-NET. The top half is just vanilla Wavenet's causal dilated convolutions. The bottom half shows the non-causal dilated convolutions on the input. The top Wavenet takes guidance from the bottom Wavenet, so that Anh's voice output is "harmonized" with my voice input.

Before presenting the detailed architecture for ANH-NET, one final thing to note is that, as is done in the original Wavenet paper, I discretize the audio waveforms. Normally, the audio takes on continuous values between -1 and 1. I discretize this range into 256 bins via the so-called μ-law.

$$b \leftarrow \mu \cdot round\left(1 + sgn(x)\frac{\log(1+\mu|x|)}{\log(1+\mu)}\right)$$

Here, $x \in [-1, 1]$ is the input waveform value, $\mu = 255$ is the discretization factor (number of bins), and $b \in \{0, 1, \ldots, 255\}$ is the index of the bin that x is mapped to. Essentailly, this is just a binning that has smaller, finer discretization closer to 0, and coarser, wider discretization at the extremes of -1 and 1. This μ-law discretization is mostly imperceptible by the human ear. So now the problem becomes guessing the correct category (i.e. bin) that the waveform will lie in at each time. As such, a softmax can be used for the final output activation and the categorical cross entropy loss can be used as the training metric.

With all that said, the following is a detailed block diagram for ANH-NET.

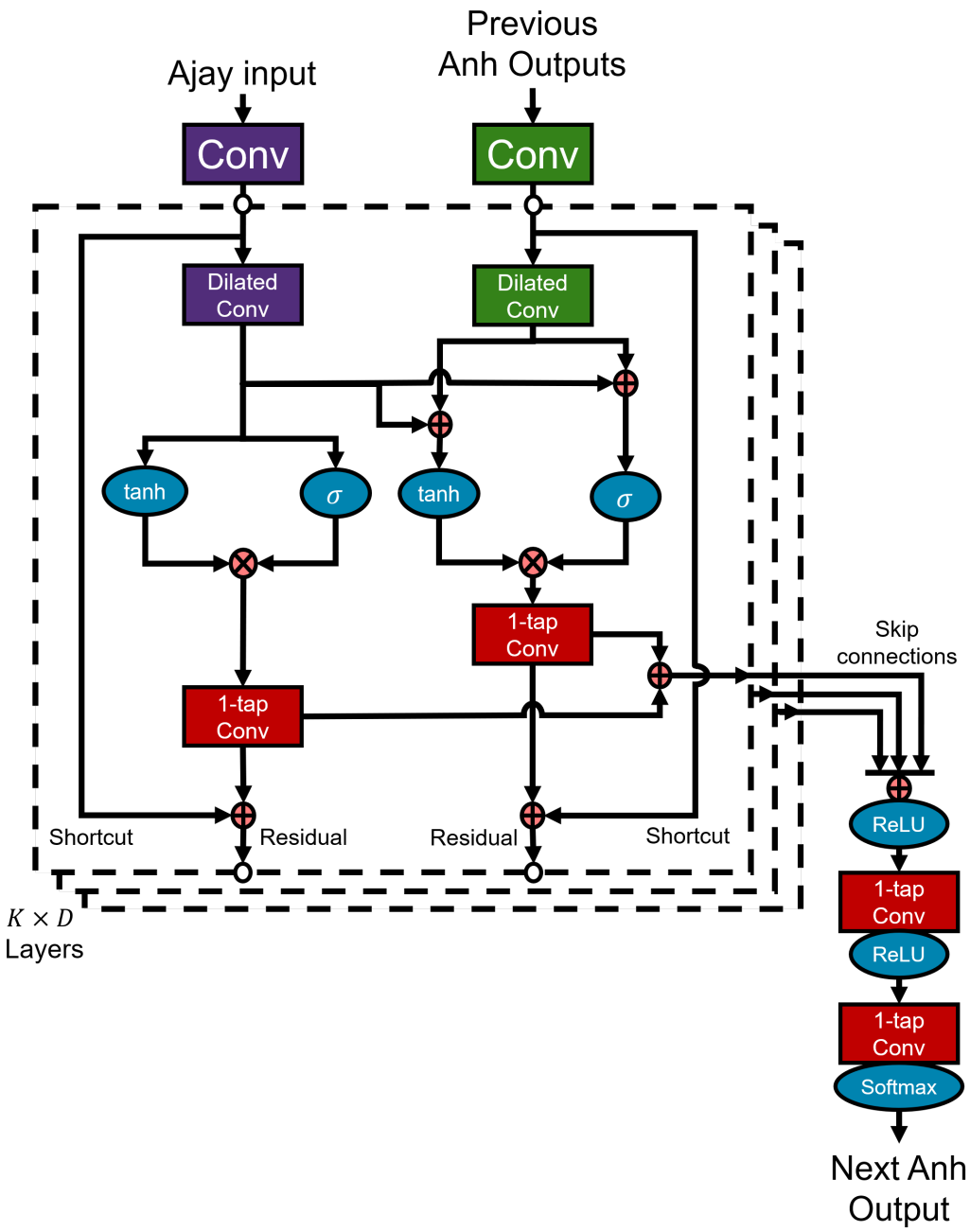

Figure 5: The architecture for ANH-NET. Green represents causal, and purple represents non-causal. 1-tap convolutions are essentially dense fully-connected layers applied across time.

After a pre-processing initial convolution, the waveforms are fed into $K \times D$ successive dilated convolution layers. Note that the Wavenet for the Ajay input is constantly feeding into the Wavenet for Anh (right before the gated activation). This is how Anh's Wavenet is harmonized with the Ajay Wavenet. This takes inspiration from the original Wavenet paper, where instead meta-data was fed

in at this point, like a one-hot encoding of whether the speaker is a man or a woman. This was so, when generating audio, one could possible condition on having a male or female voice.

There are a few extra connections in the architecture to help make it train much better. Firstly, "shortcut" connections are made, which jump over all convolutions and activations. So, as a result, even the deepest layer has a direct line to the original input, which is a technique to help gradients propogate through the network and make training easier. The biggest example of the use of these shortcut connections is the so-called Highway networks.

The other technique used is "skip" connections. Instead of only feeding the very deepest layer into the output layer, each layer feeds its premature output into the output layer. In other words, each layer has a connection that makes its output skip going through the rest of the network and go straight to the output layer. This is also to help with training the deep network.

One final thing to note is that the original Wavenet paper only gave a very high level architecture like this, but gave no details about regularization, dropout, batch normalization, etc.. So I had to experiment quite a bit to find to set those extra things to make the network train well. Essentially, I added a very slight amount of $L1$ and $L2$ regularization to each of the convolutions. Furthermore, I added dropout with 15% probability to each of the skip connections as well as the the output of the final ReLU before the softmax. Also, I used 100 filters for each of the convolutions except the very last output convolutions, for which I used 256 filters.

The resulting ANH-NET had approximately 600k parameters. Since the number of samples I have is over an order of magnitude larger the number of parameters in my model, it's safe to assume that whatever training error I get will fully generalize.

I implemented ANH-NET in the Keras library, which is a wrapper for Tensorflow. I ran the training for ANH-NET on a top-of-the line NVIDIA GeForce GTX 1080 GPU. Even so, as Wavenets are notoriously huge, training over 30 epochs took around 28 hours.

The final accuracy the network achieved was between 23% and 30%. That is to say, my network guesses the correct bin for the next sample 23-30% of the time. I give this range because, if you recall, the data has many sections which are bad and errors don't mean anything. While this seems somewhat low, it is actually quite a good level of accuracy. To illustrate this, if you are guessing uniformly between the correct bin, the bin above it, and the bin below it, then you will get an accuracy of 33% when you are only off by one bin. The bin width is around 4.31e-5, so being off by that much is a level of noise which is hardly perceptible by our ears, less than even 1% of the total volume of the signal.

Comparing to the baseline, ANH-NET performs around 60 times better, which is a significant improvement over an order of magnitude.

I would further provide audio samples for you to listen to, but the problem is that actually generating audio with Wavenets takes an enormous amount of time. When training, I can run on huge batches in parallel and do many other speedups. But for audio generation, each time I crank the input through the network, it only outputs 1 sample of audio. And each sample must be generated successively from the previous samples, so everything must run in series. For even 1 second of audio, the enormous network has to be run 16000 times in series. Google themselves reported taking extremely large amounts of time to generate audio. And so I didn't have enough time to generate audio for you to listen to. However, I note here that, very recently, Google made a paper

on Fast Wavenet, which allegedly offers a 1000x speedup by reducing redundant calculations.

To conclude, I successfully developed ANH-NET (**A**uxiliary wave**N**et **H**armonizing neural **NET**work) to transform my voice to my girlfriend's voice. To end with, here's an obligatory picture of me and my girlfriend ♡.