Appendices

A. WaveNet Architecture and Details

The WaveNet consists of a conditioning network $c = C(v)$, which converts low-frequency linguistic features v to the native audio frequency, and an auto-regressive process $P(y_i|c, y_{i-1}, \ldots, y_{i-R})$ which predicts the next audio sample given the conditioning for the current timestep c and a context of R audio samples. R is the receptive field size, and is a property determined by the structure of the network. A sketch of the WaveNet architecture is shown in Figure 3. The network details are described in the following subsections.

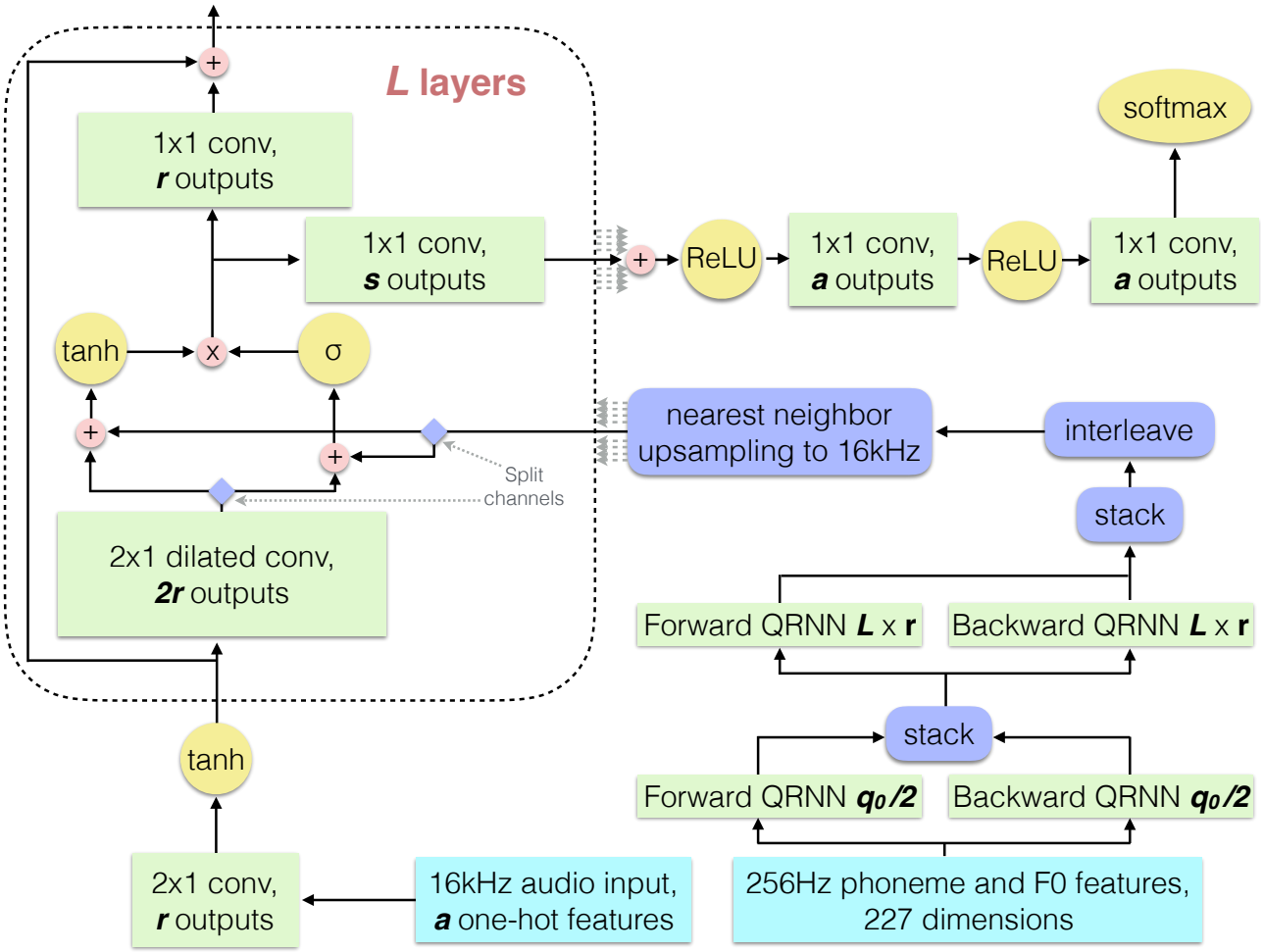

Figure 3. The modified WaveNet architecture. Components are colored according to function: teal inputs, green convolutions and QRNNs, yellow unary operations and softmax, pink binary operations, and indigo reshapes, transposes, and slices.

A.1. Auto-regressive WaveNet

The structure of the auto-regressive network is parameterized by the number of layers ℓ, the number of skip channels s, and the number of residual channels r.

Audio is quantized to $a = 256$ values using μ-law companding, as described in Section 2.2 of WaveNet. The one-hot encoded values go through an initial 2x1 convolution which generates the input $x^{(0)} \in \mathbb{R}^r$ for the first layer in the residual stack:

$$x^{(0)} = W_{\text{embed}} * y + B_{\text{embed}}, \tag{11}$$

where $*$ is the one-dimensional convolution operator. Since the input audio y is a one-hot vector, this convolution can be done via embeddings instead of matrix multiplies. Each subsequent layer computes a hidden state vector $h^{(i)}$ and then (due to the residual connections between layers) adds to its input $x^{(i-1)}$ to generate its output $x^{(i)}$:

$$h^{(i)} = \tanh\left(W_h^{(i)} * x^{(i-1)} + B_h^{(i)} + L_h^{(i)}\right) \cdot \sigma\left(W_g^{(i)} * x^{(i-1)} + B_g^{(i)} + L_g^{(i)}\right) \tag{12}$$

$$x^{(i)} = x^{(i-1)} + W_r^{(i)} \cdot h^{(i)} + B_r^{(i)}, \tag{13}$$

where $L^{(i)}$ is the output for that layer of the conditioning network. Since each layer adds its output to its input, the dimensionality of the layers must remain fixed to the number of residual channels, r. Although here this is written as two convolutions, one for W_h and one for W_g, it is actually done more efficiently with a single convolution with r input and $2r$ output channels. During inference, this convolution is replaced with two matrix-vector multiplies with matrices W_{prev} (the left half of the convolution) and W_{cur} (the right half). Thus we can reformulate the computation of $h^{(i)}$ for a specific timestep t as follows:

$$h'^{(i)} = W_{\text{prev}}^{(i)} \cdot x_{t-d}^{(i-1)} + W_{\text{cur}}^{(i)} \cdot x_t^{(i-1)} + B^{(i)} + L^{(i)} \tag{14}$$

$$h^{(i)} = \tanh\left(h'^{(i)}_{0:r}\right) \cdot \sigma\left(h'^{(i)}_{r:2r}\right), \tag{15}$$

where $L^{(i)}$ and is a concatenation of $L_h^{(i)}$ and $L_g^{(i)}$ and $B^{(i)}$ and is a concatenation of $B_h^{(i)}$ and $B_g^{(i)}$.

The hidden state $h^{(i)}$ from each of the layers 1 through ℓ is concatenated and projected with a learned W_{skip} down to the number of skip channels s:

$$h = \begin{bmatrix} h^{(1)} \\ h^{(2)} \\ \vdots \\ h^{(\ell)} \end{bmatrix} \qquad h \in \mathbb{R}^{\ell r} \tag{16}$$

$$z_s = \text{relu}\left(W_{\text{skip}} \cdot h + B_{\text{skip}}\right), \qquad z_s \in \mathbb{R}^s \tag{17}$$

where $\text{relu}(x) = \max(0, x)$.

z_s is then fed through two fully connected relu layers to generate the output distribution $p \in \mathbb{R}^a$:

$$z_a = \text{relu}\left(W_{\text{relu}} \cdot z_s + B_{\text{relu}}\right), \qquad z_a \in \mathbb{R}^a \tag{18}$$

$$p = \text{softmax}\left(W_{\text{out}} \cdot z_a + B_{\text{out}}\right) \tag{19}$$

A.2. Conditioning Network

When trained without conditioning information, WaveNet models produce human-like "babbling sounds", as they lack sufficient long-range information to reproduce words. In order to generate recognizable speech, every timestep is conditioned by an associated set of linguistic features. This is done by biasing every layer with a per-timestep conditioning vector generated from a lower-frequency input signal containing phoneme, stress, and fundamental frequency features.

The frequency of the audio is significantly higher than the frequency of the linguistic conditioning information, so an upsampling procedure is used to convert from lower-frequency linguistic features to higher-frequency conditioning vectors for each WaveNet layer.

The original WaveNet does upsampling by repetition or through a transposed convolution. Instead, we first pass our input features through two bidirectional quasi-RNN layers (Bradbury et al., 2016) with fo-pooling and 2x1 convolutions. A unidirectional QRNN layer with fo-pooling is defined by the following equations:

$$\tilde{h} = \tanh\left(W_h * x + B_h\right) \tag{20}$$

$$o = \sigma\left(W_o * x + B_o\right) \tag{21}$$

$$f = \sigma\left(W_f * x + B_f\right) \tag{22}$$

$$h_t = f_t \cdot h_{t-1} + (1 - f_t) \cdot \tilde{h}_t \tag{23}$$

$$z_t = o_t \cdot h_t \tag{24}$$

A bidirectional QRNN layer is computed by running two unidirectional QRNNs, one on the input sequence and one on a reversed copy of the input sequence, and then stacking their output channels. After both QRNN layers, we interleave the channels, so that the $\tanh$ and the sigmoid in the WaveNet both get channels generated by the forward QRNN and

backward QRNN.

Following the bidirectional QRNN layers, we upsample to the native audio frequency by repetition. (Upsampling using bilinear interpolation slowed convergence and reduced generation quality by adding noise or causing mispronunciations, while bicubic upsampling led to muffled sounds. Upsampling by repetition is done by computing the ratio of the output frequency to the input frequency and repeating every element in the input signal an appropriate number of times.)

We find that the model is very sensitive to the upsampling procedure: although many variations of the conditioning network converge, they regularly produce phoneme mispronunciations.

A.3. Input Featurization

Our WaveNet is trained with 8-bit μ-law companded audio which is downsampled to 16384 Hz from 16-bit dual-channel PCM audio at 48000 Hz. It is conditioned on a 256 Hz phoneme signal. The conditioning feature vector has 227 dimensions. Of these two are for fundamental frequency. One of these indicates whether the current phoneme is voiced (and thus has an F0) and the other is normalized log-frequency, computed by normalizing the log of F0 to minimum observed F0 to be approximately between -1 and 1. The rest of the features describe the current phoneme, the two previous phonemes, and the two next phonemes, with each phoneme being encoded via a 40-dimensional one-hot vector for phoneme identity (with 39 phonemes for ARPABET phonemes and 1 for silence) and a 5-dimensional one-hot vector for phoneme stress (no stress, primary stress, secondary stress, tertiary stress, and quaternary stress). Not all of the datasets we work with have tertiary or quaternary stress, and those features are always zero for the datasets that do not have those stress levels.

In our experiments, we found that including the phoneme context (two previous and two next phonemes) is crucial for upsampling via transposed convolution and less critical but still important for our QRNN-based upsampling. Although sound quality without the phoneme context remains high, mispronunciation of a subset of the utterances becomes an issue. We also found that including extra prosody features such as word and syllable breaks, pauses, phoneme and syllable counts, frame position relative to phoneme, etc, were unhelpful and did not result in higher quality synthesized samples.

In order to convert from phonemes annotated with durations to a fixed-frequency phoneme signal, we sample the phonemes at regular intervals, effectively repeating each phoneme (with context and F0) a number proportional to its duration. As a result, phoneme duration is effectively quantized to $1/256$ sec ≈ 4ms.

We use Praat (Boersma et al., 2002) in batch mode to compute F0 at the appropriate frequency, with a minimum F0 of 75 and a maximum F0 of 500. The Praat batch script used to generate F0 is available at https://github.com/baidu-research/deep-voice/blob/master/scripts/f0-script.praat and can be run with `praat --run f0-script.praat`.

A.4. Sampling from Output Distribution

At every timestep, the synthesis model produces a distribution over samples, $P(s)$, conditioned on the previous samples and the linguistic features. In order to produce the samples, there are a variety of ways you could choose to use this distribution:

- **Direct Sampling:** Sample randomly from $P(y)$.
- **Temperature Sampling:** Sample randomly from a distribution adjusted by a temperature t
$$\tilde{P}_t(y) = \frac{1}{Z} P(y)^{1/t}, \tag{25}$$
where Z is a normalizing constant.
- **Mean:** Take the mean of the distribution $\mathrm{E}_P[y]$.
- **Mode:** Take the most likely sample, argmax $P(y)$.
- **Top k:** Sample from an adjusted distribution that only permits the top k samples
$$\tilde{P}_k(y) = \begin{cases} 0 & \text{if } y < k\text{th}(P(y)) \\ P(y)/Z & \text{otherwise,} \end{cases} \tag{26}$$
where Z is a normalizing constant.

We find that out of these different sampling methods, only direct sampling produces high quality outputs. Temperature sampling produces acceptable quality results, and indeed outperforms direct sampling early on in training, but for converged models is significantly worse. This observation indicates that the generative audio model accurately learns a conditional sample distribution and that modifying this distribution through the above heuristics is worse than just using the learned distribution.

A.5. Training

We observed several tendencies of the models during training. As expected, the randomly initialized model produces white noise. Throughout training, the model gradually increases the signal to noise ratio, and the volume of the white noise dies down while the volume of the speech signal increases. The speech signal can be inaudible for tens of thousands of iterations before it dominates the white noise.

In addition, because the model is autoregressive, rare mistakes can produce very audible disturbances. For example, a common failure mode is to produce a small number of incorrect samples during sampling, which then results in a large number incorrect samples due to compounding errors. This is audible as a brief period of loud noise before the model stabilizes. The likelihood of this happening is higher early on in training, and does not happen in converged models.

B. Phoneme Model Loss

The loss for the n^{th} phoneme is

$$L_n = |\hat{t}_n - t_n| + \lambda_1 \text{CE}(\hat{p}_n, p_n) + \lambda_2 \sum_{t=0}^{T-1} |\widehat{F0}_{n,t} - F0_{n,t}| + \lambda_3 \sum_{t=0}^{T-2} |\widehat{F0}_{n,t+1} - \widehat{F0}_{n,t}|, \tag{27}$$

where λ_i's are tradeoff constants, $\widehat{t}_n$ and t_n are the estimated and ground-truth durations of the n^{th} phoneme, $\hat{p}_n$ and p_n are the estimated and ground-truth probabilities that the n^{th} phoneme is voiced, CE is the cross-entropy function, $\widehat{F}_{n,t}$ and $F_{n,t}$ are the estimated and ground-truth values of the fundamental frequency of the n^{th} phoneme at time t. T time samples are equally spaced along the phoneme duration.

C. Nonlinearity Approximation Details

During inference, we replace exact implementations of the neural network nonlinearities with high-accuracy rational approximations. In this appendix, we detail the derivation of these approximations.

C.1. tanh and sigmoid approximation

Denoting $\tilde{e}(x)$ as an approximation to $e^{|x|}$, we use the following approximations for $\tanh$ and σ:

$$\tanh(x) \approx \text{sign}(x) \frac{\tilde{e}(x) - 1/\tilde{e}(x)}{\tilde{e}(x) + 1/\tilde{e}(x)} \tag{28}$$

$$\sigma(x) \approx \begin{cases} \frac{\tilde{e}(x)}{1+\tilde{e}(x)} & x \geq 0 \\ \frac{1}{1+\tilde{e}(x)} & x \leq 0 \end{cases} \tag{29}$$

We choose a forth-order polynomial to represent $\tilde{e}(x)$. The following fit produces accurate values for both $\tanh(x)$ and $\sigma(x)$:

$$\tilde{e}(x) = 1 + |x| + 0.5658x^2 + 0.143x^4 \tag{30}$$

By itself, $\tilde{e}(x)$ is not a very good approximate function for $e^{|x|}$, but it yields good approximations when used to approximate $\tanh$ and σ as described in Equations 28 and 29.

C.2. e^x approximation

We follow the approach of (Stephenson, 2005) to calculate an approximate e^x function. Instead of approximating e^x directly, we approximate 2^x and use the identity $e^x = 2^{x/\ln 2}$.

Let $\lfloor x \rfloor$ to be the floor of $x \in \mathbb{R}$. Then,

$$\begin{aligned} 2^x &= 2^{\lfloor x \rfloor} \cdot 2^{x - \lfloor x \rfloor} \\ &= 2^{\lfloor x \rfloor} \cdot \Big(1 + (2^{x - \lfloor x \rfloor} - 1)\Big) \end{aligned}$$

where $0 \leq 2^{x-\lfloor x \rfloor} - 1 < 1$ since $0 \leq x - \lfloor x \rfloor < 1$. If we use a 32-bit float to represent 2^x, then $\lfloor x \rfloor + 127$ and $2^{x-\lfloor x \rfloor} - 1$ are represented by the exponent and fraction bits of 2^x. Therefore, if we interpret the bytes pattern of 2^x as a 32-bits integer (represented by I_{2^x}), we have

$$I_{2^x} = (\lfloor x \rfloor + 127) \cdot 2^{23} + (2^{x - \lfloor x \rfloor} - 1) \cdot 2^{23}. \tag{31}$$

Rearranging the Equation 31 and using $z = x - \lfloor x \rfloor$ results to

$$I_{2^x} = (x + 126 + \{2^z - z\}) \cdot 2^{23} \tag{32}$$

If we can accurately approximate $g(z) = 2^z - z$ over $z \in [0, 1)$, then interpreting back the byte representation of I_{2^x} in Equation 32 as a 32-bits float, we can accurately approximate 2^x. We use a rational approximation as

$$g(z) \approx -4.7259162 + \frac{27.7280233}{4.84252568 - z} - 1.49012907z, \tag{33}$$

which gives are maximum error 2.4×10^{-5} for $x \in (-\infty, 0]$.

D. Persistent GPU Kernels

A NVIDIA GPU has multiple Streaming Multiprocessors (SMs), each of which has a register file and a L1 cache. There is also a coherent L2 cache that is shared by all SMs. The inference process needs to generate one sample every 61 μs. Due to the high latency of a CUDA kernel launch and of reading small matrices from GPU memory, the entire audio generation process must be done by a single kernel with the weights loaded into the register file across all SMs. This raises two challenges—how to split the model across registers in a way to minimize communication between SMs and how to communicate between SMs given the restrictions imposed by the CUDA programming model.

We split the model across the register file of 24 SMs, numbered SM1 $\cdot$ SM24, of a TitanX GPU. We do not use SM24. SM1 to SM20 store two adjacent layers of the residual stack. This means SM1 stores layers 1 and 2, SM2 stores layers 3 and 4 and so on and so forth. Each layer has three matrices and three bias vectors—W_{prev}, B_{prev}, W_{cur}, B_{cur}, that are for the dilated convolutions and W_r, B_r. Thus SMi generates two hidden states $h^{(2i)}$ and $h^{(2i+1)}$ and an output $x^{(2i)}$. Each SM also stores the rows of the W_{skip} matrix that will interact with the generated hidden state vectors. Thus W_{skip} is partitioned across 20 SMs. Only SM20 needs to store B_{skip}. SM21 stores W_{relu} and B_{relu}. Finally, W_{out} is split across two SMs—SM22 and SM23 because of register file limitations and SM23 stores B_{out}.

The next challenge is to coordinate the data transfer between SMs, since the CUDA programming model executes one kernel across all SMs in parallel. However we want execution to go sequentially in a round robin fashion from SM1 to SM23 and back again from SM1 as we generate one audio sample at a time. We launch our CUDA kernel with 23 thread blocks and simulate such sequential execution by spinning on locks, one for each SM, that are stored in global memory and cached in L2. First SM1 executes two layers of the WaveNet model to generate $h^{(1)}$, $h^{(2)}$ and $x^{(2)}$. It then unlocks the lock that SM2 is spinning on and sets its own lock. It does this by bypassing the L1 cache to write to global memory so that all SMs have a coherent view of the locks. Then SM2 does the same for SM3 and this sequential locking and unlocking chain continues for each SM. Finally SM23 generates the output distribution p for timestep t and unlocks SM1 so that entire process can repeat to generate p for timestep $t + 1$.

Just like locks, we pass data between SMs, by reading and writing to global memory by bypassing the L1 cache. Since NVIDIA GPUs have a coherent L2 cache, a global memory write bypassing the L1, followed by a memory fence results in a coherent view of memory across SMs.

This partitioning scheme however is quite inflexible and only works for specific values of l, r and s shown in Table 2. This is because each SM has a fixed sized register file and combined with the relatively inflexible and expensive communication mechanism between SMs implies that splitting weight matrices between SMs is challenging. Any change in those parameters means a new kernel has to be written, which is a very time consuming process.

There are two main reasons why the GPU kernels are slower than CPU kernels. Firstly, synchronization between SMs in a GPU is expensive since it is done by busy waiting on locks in L2 cache. Secondly even though we divide the model in a way that will fit in the register file of each SM, the CUDA compiler still spills to L1 cache. We hope that with handcrafted

assembly code, we will be able to match the performance of CPU kernels. However, the lack of parallelism in WaveNet inference makes it difficult to hide the latencies inherent in reading and writing small matrices from GPU memory which are exposed in the absence of a rich cache hierarchy in GPUs.

E. Performance model

We present a performance model for the autoregressive WaveNet architecture described in Appendix A.1. In our model a dot product between two vectors of dimension r takes $2r$ FLOPS—r multiplications and r additions. This means that a matrix-vector multiply between W, an $r \times r$ matrix and x, a $r \times 1$ vector takes $2r \cdot r = 2r^2$ FLOPs. Thus calculating $h'^{(i)}$ uses

$$\text{Cost}\left(h'^{(i)}\right) = (2r \cdot 2r) + (2r \cdot 2r) + 2r + 2r + 2r \ \ \text{FLOPs} \tag{34}$$

Let division and exponentiaton take f_d and f_e FLOPs respectively. This means $\tanh$ and σ takes $(f_d + 2f_e + 1)$ FLOPs. Thus calculating $h^{(i)}$ takes $2r \cdot (f_d + 2f_e + 1) + r$ FLOPs. Finally calculating $x^{(i)}$ for each layer takes $r + (2r \cdot r) + r$ FLOPs. This brings the total FLOPs for calculating one layer to

$$\text{Cost (layer)} = 10r^2 + 11r + 2r(f_d + f_e) \ \ \text{FLOPs} \tag{35}$$

Under the same model, calculating z_s takes $(\ell \cdot 2r) \cdot s + s + s$ FLOPs, where we assume that relu takes 1 FLOP. Similarly, calculating z_a takes $2s \cdot a + a + a$ FLOPs and $W_{\text{out}} \cdot z_a + B_{\text{out}}$ takes $2a \cdot a + a$ FLOPs.

Calculating the numerically stable softmax takes one $\max$, one subtract, one exponentiation, one sum and one division per element of a vector. Hence calculating p takes $3a + a(f_d + f_e)$ FLOPs.

Adding it all up, our final performance model to generate each audio sample is as follows:

$$\text{Cost (sample)} = \ell\left(10r^2 + 11r + 2r(f_d + f_e)\right) + s(2r \cdot \ell + 2) + a(2s + 2a + 3) + a(3 + f_d + f_e) \ \ \text{FLOPS} \tag{36}$$

If we let $\ell = 40$, $r = 64$, and $s = a = 256$, and assume that $f_d = 10$ and $f_e = 10$, with a sampling frequency of 16384Hz, we have approximately 55×10^9 FLOPs for every second of synthesis.