Deep Voice: Real-time Neural Text-to-Speech

Sercan Ö. Arık [*1] **Mike Chrzanowski** [*1] **Adam Coates** [*1] **Gregory Diamos** [*1] **Andrew Gibiansky** [*1]
Yongguo Kang [*2] **Xian Li** [*2] **John Miller** [*1] **Andrew Ng** [*1] **Jonathan Raiman** [*1] **Shubho Sengupta** [*1]
Mohammad Shoeybi [*1]

Abstract

We present Deep Voice, a production-quality text-to-speech system constructed entirely from deep neural networks. Deep Voice lays the groundwork for truly end-to-end neural speech synthesis. The system comprises five major building blocks: a segmentation model for locating phoneme boundaries, a grapheme-to-phoneme conversion model, a phoneme duration prediction model, a fundamental frequency prediction model, and an audio synthesis model. For the segmentation model, we propose a novel way of performing phoneme boundary detection with deep neural networks using connectionist temporal classification (CTC) loss. For the audio synthesis model, we implement a variant of WaveNet that requires fewer parameters and trains faster than the original. By using a neural network for each component, our system is simpler and more flexible than traditional text-to-speech systems, where each component requires laborious feature engineering and extensive domain expertise. Finally, we show that inference with our system can be performed faster than real time and describe optimized WaveNet inference kernels on both CPU and GPU that achieve up to 400x speedups over existing implementations.

1. Introduction

Synthesizing artificial human speech from text, commonly known as text-to-speech (TTS), is an essential component in many applications such as speech-enabled devices, navigation systems, and accessibility for the visually-impaired. Fundamentally, it allows human-technology interaction without requiring visual interfaces. Modern TTS systems are based on complex, multi-stage processing pipelines, each of which may rely on hand-engineered features and heuristics. Due to this complexity, developing new TTS systems can be very labor intensive and difficult.

Deep Voice is inspired by traditional text-to-speech pipelines and adopts the same structure, while replacing all components with neural networks and using simpler features: first we convert text to phoneme and then use an audio synthesis model to convert linguistic features into speech (Taylor, 2009). Unlike prior work (which uses hand-engineered features such as spectral envelope, spectral parameters, aperiodic parameters, etc.), our only features are phonemes with stress annotations, phoneme durations, and fundamental frequency (F0). This choice of features makes our system more readily applicable to new datasets, voices, and domains without any manual data annotation or additional feature engineering. We demonstrate this claim by retraining our entire pipeline without any hyperparameter changes on an entirely new dataset that contains solely audio and unaligned textual transcriptions and generating relatively high quality speech. In a conventional TTS system this adaptation requires days to weeks of tuning, whereas Deep Voice allows you to do it in only a few hours of manual effort and the time it takes models to train.

Real-time inference is a requirement for a production-quality TTS system; without it, the system is unusable for most applications of TTS. Prior work has demonstrated that a WaveNet (van den Oord et al., 2016) can generate close to human-level speech. However, WaveNet inference poses a daunting computational problem due to the high-frequency, autoregressive nature of the model, and it has been hitherto unknown whether such models can be used in a production system. We answer this question in the affirmative and demonstrate efficient, faster-than-real-time WaveNet inference kernels that produce high-quality 16 kHz audio and realize a 400X speedup over previous WaveNet inference implementations (Paine et al., 2016).

[*]Listed alphabetically [1]Baidu Silicon Valley Artificial Intelligence Lab, 1195 Bordeaux Dr. Sunnyvale, CA 94089 [2]Baidu Corporation, No. 10 Xibeiwang East Road, Beijing 100193, China. Correspondence to: Andrew Gibiansky <gibianskyandrew@baidu.com>.

Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017. Copyright 2017 by the author(s).

2. Related Work

Previous work uses neural networks as substitutes for several TTS system components, including grapheme-to-phoneme conversion models (Rao et al., 2015; Yao & Zweig, 2015), phoneme duration prediction models (Zen & Sak, 2015), fundamental frequency prediction models (Pascual & Bonafonte, 2016; Ronanki et al., 2016), and audio synthesis models (van den Oord et al., 2016; Mehri et al., 2016). Unlike Deep Voice, however, none of these systems solve the entire problem of TTS and many of them use specialized hand-engineered features developed specifically for their domain.

Most recently, there has been a lot of work in parametric audio synthesis, notably WaveNet, SampleRNN, and Char2Wav (van den Oord et al., 2016; Mehri et al., 2016; Sotelo et al., 2017). While WaveNet can be used for both conditional and unconditional audio generation, SampleRNN is only used for unconditional audio generation. Char2Wav extends SampleRNN with an attention-based phoneme duration model and the equivalent of an F0 prediction model, effectively providing local conditioning information to a SampleRNN-based vocoder.

Deep Voice differs from these systems in several key aspects that notably increase the scope of the problem. First, Deep Voice is completely standalone; training a new Deep Voice system does not require a pre-existing TTS system, and can be done from scratch using a dataset of short audio clips and corresponding textual transcripts. In contrast, reproducing either of the aforementioned systems requires access and understanding of a pre-existing TTS system, because they use features from another TTS system either at training or inference time.

Second, Deep Voice minimizes the use of hand-engineered features; it uses one-hot encoded characters for grapheme to phoneme conversion, one-hot encoded phonemes and stresses, phoneme durations in milliseconds, and normalized log fundamental frequency that can be computed from waveforms using any F0 estimation algorithm. All of these can easily be obtained from audio and transcripts with minimal effort. In contrast, prior works use a much more complex feature representation, that effectively makes reproducing the system impossible without a pre-existing TTS system. WaveNet uses several features from a TTS system (Zen et al., 2013), that include values such as the number of syllables in a word, position of syllables in the phrase, position of the current frame in the phoneme, and dynamic features of the speech spectrum like spectral and excitation parameters, as well as their time derivatives. Char2Wav relies on vocoder features from the WORLD TTS system (Morise et al., 2016) for pre-training their alignment module which include F0, spectral envelope, and aperiodic parameters.

Finally, we focus on creating a production-ready system, which *requires* that our models run in real-time for inference. Deep Voice can synthesize audio in fractions of a second, and offers a tunable trade-off between synthesis speed and audio quality. In contrast, previous results with WaveNet require several minutes of runtime to synthesize one second of audio. We are unaware of similar benchmarks for SampleRNN, but the 3-tier architecture as described in the original publication requires approximately 4-5X as much compute during inference as our largest WaveNet models, so running the model in real-time may prove challenging.

3. TTS System Components

As shown in Fig. 1, the TTS system consists of five major building blocks:

- The **grapheme-to-phoneme model** converts from written text (English characters) to phonemes (encoded using a phonemic alphabet such as ARPABET).
- The **segmentation model** locates phoneme boundaries in the voice dataset. Given an audio file and a phoneme-by-phoneme transcription of the audio, the segmentation model identifies where in the audio each phoneme begins and ends.
- The **phoneme duration model** predicts the temporal duration of every phoneme in a phoneme sequence (an utterance).
- The **fundamental frequency model** predicts whether a phoneme is voiced. If it is, the model predicts the fundamental frequency (F0) throughout the phoneme's duration.
- The **audio synthesis model** combines the outputs of the grapheme-to-phoneme, phoneme duration, and fundamental frequency prediction models and synthesizes audio at a high sampling rate, corresponding to the desired text.

During inference, text is fed through the grapheme-to-phoneme model or a phoneme dictionary to generate phonemes. Next, the phonemes are provided as inputs to the phoneme duration model and F0 prediction model to assign durations to each phoneme and generate an F0 contour. Finally, the phonemes, phoneme durations, and F0 are used as local conditioning input features to the audio synthesis model, which generates the final utterance.

Unlike the other models, the segmentation model is not used during inference. Instead, it is used to annotate the training voice data with phoneme boundaries. The phoneme boundaries imply durations, which can be used to train the phoneme duration model. The audio, annotated with phonemes and phoneme durations as well as fundamental frequency, is used to train the audio synthesis model.

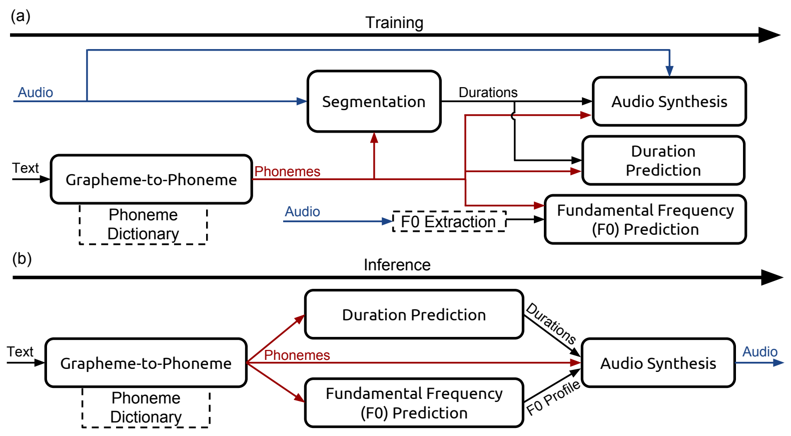

Figure 1. System diagram depicting (a) training procedure and (b) inference procedure, with inputs on the left and outputs on the right. In our system, the duration prediction model and the F0 prediction model are performed by a single neural network trained with a joint loss. The grapheme-to-phoneme model is used as a fallback for words that are not present in a phoneme dictionary, such as CMUDict. Dotted lines denote non-learned components.

In the following sections, we describe all the building blocks in detail.

3.1. Grapheme-to-Phoneme Model

Our grapheme-to-phoneme model is based on the encoder-decoder architecture developed by (Yao & Zweig, 2015). However, we use a multi-layer bidirectional encoder with a gated recurrent unit (GRU) nonlinearity and an equally deep unidirectional GRU decoder (Chung et al., 2014). The initial state of every decoder layer is initialized to the final hidden state of the corresponding encoder forward layer. The architecture is trained with teacher forcing and decoding is performed using beam search. We use 3 bidirectional layers with 1024 units each in the encoder and 3 unidirectional layers of the same size in the decoder and a beam search with a width of 5 candidates. During training, we use dropout with probability 0.95 after each recurrent layer.

For training, we use the Adam optimization algorithm with $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$, a batch size of 64, a learning rate of 10^{-3}, and an annealing rate of 0.85 applied every 1000 iterations (Kingma & Ba, 2014).

3.2. Segmentation Model

Our segmentation model is trained to output the alignment between a given utterance and a sequence of target phonemes. This task is similar to the problem of aligning speech to written output in speech recognition. In that domain, the connectionist temporal classification (CTC) loss function has been shown to focus on character alignments to learn a mapping between sound and text (Graves et al., 2006). We adapt the convolutional recurrent neural network architecture from a state-of-the-art speech recognition system (Amodei et al., 2015) for phoneme boundary detection.

A network trained with CTC to generate sequences of phonemes will produce brief peaks for every output phoneme. Although this is sufficient to roughly align the phonemes to the audio, it is insufficient to detect precise phoneme boundaries. To overcome this, we train to predict sequences of phoneme *pairs* rather than single phonemes. The network will then tend to output phoneme pairs at timesteps close to the boundary between two phonemes in a pair.

To illustrate our label encoding, consider the string “Hello!”. To convert this to a sequence of phoneme pair labels, convert the utterance to phonemes (using a pronunciation dictionary such as CMUDict or a grapheme-to-phoneme model) and pad the phoneme sequence on either end with the silence phoneme to get “sil HH EH L OW sil”. Finally, construct consecutive phoneme pairs and get “(sil, HH), (HH, EH), (EH, L), (L, OW), (OW, sil)”.

Input audio is featurized by computing 20 Mel-frequency cepstral coefficients (MFCCs) with a ten millisecond stride. On top of the input layer, there are two convolution layers (2D convolutions in time and frequency), three bidirectional recurrent GRU layers, and finally a softmax output layer. The convolution layers use kernels with unit stride, height nine (in frequency bins), and width five (in time) and the recurrent layers use 512 GRU cells (for each direction). Dropout with a probability of 0.95 is applied after the last convolution and recurrent layers. To compute the phoneme-pair error rate (PPER), we decode using beam search. To decode phoneme boundaries, we perform a beam search with width 50 with the constraint that neighboring phoneme pairs overlap by at least one phoneme and keep track of the positions in the utterance of each phoneme pair.

For training, we use the Adam optimization algorithm with

$\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$, a batch size of 128, a learning rate of 10^{-4}, and an annealing rate of 0.95 applied every 500 iterations (Kingma & Ba, 2014).

3.3. Phoneme Duration and Fundamental Frequency Model

We use a single architecture to jointly predict phoneme duration and time-dependent fundamental frequency. The input to the model is a sequence of phonemes with stresses, with each phoneme and stress being encoded as a one-hot vector. The architecture comprises two fully connected layers with 256 units each followed by two unidirectional recurrent layers with 128 GRU cells each and finally a fully-connected output layer. Dropout with a probability of 0.8 is applied after the initial fully-connected layers and the last recurrent layer.

The final layer produces three estimations for every input phoneme: the phoneme duration, the probability that the phoneme is voiced (i.e. has a fundamental frequency), and 20 time-dependent F0 values, which are sampled uniformly over the predicted duration.

The model is optimized by minimizing a joint loss that combines phoneme duration error, fundamental frequency error, the negative log likelihood of the probability that the phoneme is voiced, and a penalty term proportional to the absolute change of F0 with respect to time to impose smoothness. The specific functional form of the loss function is described in Appendix B.

For training, we use the Adam optimization algorithm with $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$, a batch size of 128, a learning rate of 3×10^{-4}, and an annealing rate of 0.9886 applied every 400 iterations (Kingma & Ba, 2014).

3.4. Audio Synthesis Model

Our audio synthesis model is a variant of WaveNet. WaveNet consists of a conditioning network, which upsamples linguistic features to the desired frequency, and an autoregressive network, which generates a probability distribution $\mathbb{P}(y)$ over discretized audio samples $y \in \{0, 1, \ldots, 255\}$. We vary the number of layers ℓ, the number of residual channels r (dimension of the hidden state of every layer), and the number of skip channels s (the dimension to which layer outputs are projected prior to the output layer).

WaveNet consists of an upsampling and conditioning network, followed by ℓ 2×1 convolution layers with r residual output channels and gated $\tanh$ nonlinearities. We break the convolution into two matrix multiplies per timestep with W_{prev} and W_{cur}. These layers are connected with residual connections. The hidden state of every layer is concatenated to an ℓr vector and projected to s skip channels with W_{skip}, followed by two layers of 1×1 convolutions (with weights W_{relu} and W_{out}) with relu nonlinearities.

WaveNet uses transposed convolutions for upsampling and conditioning. We find that our models perform better, train faster, and require fewer parameters if we instead first encode the inputs with a stack of bidirectional quasi-RNN (QRNN) layers (Bradbury et al., 2016) and then perform upsampling by repetition to the desired frequency.

Our highest-quality final model uses $\ell = 40$ layers, $r = 64$ residual channels, and $s = 256$ skip channels. For training, we use the Adam optimization algorithm with $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$, a batch size of 8, a learning rate of 10^{-3}, and an annealing rate of 0.9886 applied every 1,000 iterations (Kingma & Ba, 2014).

Please refer to Appendix A for full details of our WaveNet architecture and the QRNN layers we use.

4. Results

We train our models on an internal English speech database containing approximately 20 hours of speech data segmented into 13,079 utterances. In addition, we present audio synthesis results for our models trained on a subset of the Blizzard 2013 data (Prahallad et al., 2013). Both datasets are spoken by a professional female speaker.

All of our models are implemented using the TensorFlow framework (Abadi et al., 2015).

4.1. Segmentation Results

We train on 8 TitanX Maxwell GPUs, splitting each batch equally among the GPUs and using a ring all-reduce to average gradients computed on different GPUs, with each iteration taking approximately 1300 milliseconds. After approximately 14,000 iterations, the model converges to a phoneme pair error rate of 7%. We also find that phoneme boundaries do not have to be precise, and randomly shifting phoneme boundaries by 10-30 milliseconds makes no difference in the audio quality, and so suspect that audio quality is insensitive to the phoneme pair error rate past a certain point.

4.2. Grapheme-to-Phoneme Results

We train a grapheme-to-phoneme model on data obtained from CMUDict (Weide, 2008). We strip out all words that do not start with a letter, contain numbers, or have multiple pronunciations, which leaves 124,978 out of the original 133,854 grapheme-phoneme sequence pairs.

We train on a single TitanX Maxwell GPU with each iteration taking approximately 150 milliseconds. After approximately 20,000 iterations, the model converges to a phoneme error rate of 5.8% and a word error rate of 28.7%, which are on par with previous reported results (Yao & Zweig, 2015). Unlike prior work, we do not use a language model during decoding and do not include words with multiple pronunciations in our data set.

4.3. Phoneme Duration and Fundamental Frequency Results

We train on a single TitanX Maxwell GPU with each iteration taking approximately 120 milliseconds. After approximately 20,000 iterations, the model converges to a mean absolute error of 38 milliseconds (for phoneme duration) and 29.4 Hz (for fundamental frequency).

4.4. Audio Synthesis Results

We divide the utterances in our audio dataset into one second chunks with a quarter second of context for each chunk, padding each utterance with a quarter second of silence at the beginning. We filter out chunks that are predominantly silence and end up with 74,348 total chunks.

We trained models with varying depth, including 10, 20, 30, and 40 layers in the residual layer stack. We find that models below 20 layers result in poor quality audio. The 20, 30, and 40 layer models all produce high quality recognizable speech, but the 40 layer models have less noise than the 20 layer models, which can be detected with high-quality over-ear headphones.

Previous work has emphasized the importance of receptive field size in determining model quality. Indeed, the 20 layer models have half the receptive field as the 40 layer models. However, when run at 48 kHz, models with 40 layers have only 83 milliseconds of receptive field, but still generate high quality audio. This suggests the receptive field of the 20 layer models is sufficient, and we conjecture the difference in audio quality is due to some other factor than receptive field size.

We train on 8 TitanX Maxwell GPUs with one chunk per GPU, using a ring allreduce to average gradients computed on different GPUs. Each iteration takes approximately 450 milliseconds. Our model converges after approximately 300,000 iterations. We find that a single 1.25s chunk is sufficient to saturate the compute on the GPU and that batching does not increase training efficiency.

As is common with high-dimensional generative models (Theis et al., 2015), model loss is somewhat uncorrelated with perceptual quality of individual samples. While models with unusually high loss sound distinctly noisy, models that optimize below a certain threshold do not have a loss indicative of their quality. In addition, changes in model architecture (such as depth and output frequency) can have a significant impact on model loss while having a small effect on audio quality.

To estimate perceptual quality of the individual stages of our TTS pipeline, we crowdsourced mean opinion score (MOS) ratings (ratings between one and five, higher values being better) from Mechanical Turk using the CrowdMOS toolkit and methodology (Ribeiro et al., 2011). In order to separate the effect of the audio preprocessing, the WaveNet model quality, and the phoneme duration and fundamental frequency model quality, we present MOS scores for a variety of utterance types, including synthesis results where the WaveNet inputs (duration and F0) are extracted from ground truth audio rather than synthesized by other models. The results are presented in Table 1. We purposefully include ground truth samples in every batch of samples that raters evaluate to highlight the delta from human speech and allow raters to distinguish finer grained differences between models; the downside of this approach is that the resulting MOS scores will be significantly lower than if raters are presented *only* with synthesized audio samples.

First of all, we find a significant drop in MOS when simply downsampling the audio stream from 48 kHz to 16 kHz, especially in combination with μ-law companding and quantization, likely because a 48 kHz sample is presented to the raters as a baseline for a 5 score, and a low quality noisy synthesis result is presented as a 1. When used with ground truth durations and F0, our models score highly, with the 95% confidence intervals of our models intersecting those of the ground truth samples. However, using synthesized frequency reduces the MOS, and further including synthesized durations reduces it significantly. We conclude that the main barrier to progress towards natural TTS lies with duration and fundamental frequency prediction, and our systems have not meaningfully progressed past the state of the art in that regard. Finally, our best models run slightly slower than real-time (see Table 2), so we demonstrate that synthesis quality can be traded for inference speed by adjusting model size by obtaining scores for models that run 1X and 2X faster than real-time.

We also tested WaveNet models trained on the full set of features from the original WaveNet publication, but found no perceptual difference between those models and models trained on our reduced feature set.

4.5. Blizzard Results

To demonstrate the flexibility of our system, we retrained all of our models with identical hyperparameters on the Blizzard 2013 dataset (Prahallad et al., 2013). For our experiments, we used a 20.5 hour subset of the dataset segmented into 9,741 utterances. We evaluated the model using the procedure described in Section 4.4, which encourages raters to compare synthesized audio directly with the

Type	**Model Size**	MOS±CI
Ground Truth (48 kHz)	None	4.75 ± 0.12
Ground Truth	None	4.45 ± 0.16
Ground Truth (companded and expanded)	None	4.34 ± 0.18
Synthesized	$\ell = 40, r = 64, s = 256$	3.94 ± 0.26
Synthesized (48 kHz)	$\ell = 40, r = 64, s = 256$	3.84 ± 0.24
Synthesized (Synthesized F0)	$\ell = 40, r = 64, s = 256$	2.76 ± 0.31
Synthesized (Synthesized Duration and F0)	$\ell = 40, r = 64, s = 256$	2.00 ± 0.23
Synthesized (2X real-time inference)	$\ell = 20, r = 32, s = 128$	2.74 ± 0.32
Synthesized (1X real-time inference)	$\ell = 20, r = 64, s = 128$	3.35 ± 0.31

Table 1. Mean Opinion Scores (MOS) and 95% confidence intervals (CIs) for utterances. This MOS score is a relative MOS score obtained by showing raters the same utterance across all the model types (which encourages comparative rating and allows the raters to distinguish finer grained differences). Every batch of samples also includes the ground truth 48 kHz recording, which makes all our ratings comparative to natural human voices. 474 ratings were collected for every sample. Unless otherwise mentioned, models used phoneme durations and F0 extracted from the ground truth, rather than synthesized by the duration prediction and frequency prediction models, as well as a 16384 Hz audio sampling rate.

Model	**Platform**	**Data Type**	**Number of Threads**	**Speed-up Over Real-time**
$\ell = 20, r = 32, s = 128$	CPU	`float32`	6	**2.7**
$\ell = 20, r = 32, s = 128$	CPU	`float32`	2	**2.05**
$\ell = 20, r = 64, s = 128$	CPU	`int16`	2	**1.2**
$\ell = 20, r = 64, s = 128$	CPU	`float32`	6	**1.11**
$\ell = 20, r = 64, s = 128$	CPU	`float32`	2	0.79
$\ell = 40, r = 64, s = 256$	CPU	`int16`	2	0.67
$\ell = 40, r = 64, s = 256$	CPU	`float32`	6	0.61
$\ell = 40, r = 64, s = 256$	CPU	`float32`	2	0.35
$\ell = 20, r = 32, s = 128$	GPU	`float32`	N/A	0.39
$\ell = 20, r = 64, s = 128$	GPU	`float32`	N/A	0.29
$\ell = 40, r = 32, s = 128$	GPU	`float32`	N/A	0.23
$\ell = 40, r = 64, s = 128$	GPU	`float32`	N/A	0.17

Table 2. CPU and GPU inference kernel benchmarks for different models in float32 and int16. At least one main and one auxiliary thread were used for all CPU kernels. These kernels operate on a single utterance with no batching. CPU results are from a Intel Xeon E5-2660 v3 Haswell processor clocked at 2.6 GHz and GPU results are from a GeForce GTX Titan X Maxwell GPU.

ground truth. On the held out set, 16 kHz companded and expanded audio receives a MOS score of 4.65 ± 0.13, while our synthesized audio received a MOS score of 2.67 ± 0.37.

5. Optimizing Inference

Although WaveNet has shown promise in generating high-quality synthesized speech, initial experiments reported generation times of many minutes or hours for short utterances. WaveNet inference poses an incredibly challenging computational problem due to the high-frequency, autoregressive nature of the model, which requires orders of magnitude more timesteps than traditional recurrent neural networks. When generating audio, a single sample must be generated in approximately 60 μs (for 16 kHz audio) or 20 μs (for 48 kHz audio). For our 40 layer models, this means that a single layer (consisting of several matrix multiplies and nonlinearities) must complete in approximately 1.5 μs. For comparison, accessing a value that resides in main memory on a CPU can take 0.1 μs. In order to perform inference at real-time, we must take great care to never recompute any results, store the entire model in the processor cache (as opposed to main memory), and optimally utilize the available computational units. These same techniques could be used to accelerate image synthesis with PixelCNN (Oord et al., 2016) to fractions of a second per image.

Synthesizing one second of audio with our 40 layer WaveNet model takes approximately 55×10^9 floating point operations (FLOPs). The activations in any given layer depend on the activations in the previous layer and the previous timestep, so inference must be done one timestep and one layer at a time. A single layer requires only 42×10^3 FLOPs, which makes achieving meaningful parallelism difficult. In addition to the compute requirements, the model has approximately 1.6×10^6 parameters, which equate to about 6.4 MB if represented in single precision. (See Appendix E for a complete performance model.)

On CPU, a single Haswell or Broadwell core has a peak single-precision throughput of approximately 77×10^9 FLOPs and an L2-to-L1 cache bandwidth of approximately 140 GB/s (assuming two 8-wide AVX FMA instructions every cycle and an L2-to-L1 bandwidth of 64 bytes per cycle.). The model must be loaded from cache once per timestep, which requires a bandwidth of 100 GB/s. Even if the model were to fit in L2 cache, the implementation would need to utilize 70% of the maximum bandwidth and 70% of the peak FLOPS in order to do inference in real-time on a single core. Splitting the calculations across multiple cores reduces the difficulty of the problem, but nonetheless it remains challenging as inference must operate at a significant fraction of maximum memory bandwidth and peak FLOPs and while keeping threads synchronized.

A GPU has higher memory bandwidth and peak FLOPs than a CPU but provides a more specialized and hence restrictive computational model. A naive implementation that launches a single kernel for every layer or timestep is untenable, but an implementation based on the persistent RNN technique (Diamos et al., 2016) may be able to take advantage of the throughput offered by GPUs.

We implement high-speed optimized inference kernels for both CPU and GPU and demonstrate that WaveNet inference at faster-than-real-time speeds is achievable. Table 2 lists the CPU and GPU inference speeds for different models. In both cases, the benchmarks include only the autoregressive, high-frequency audio generation and do *not* include the generation of linguistic conditioning features (which can be done in parallel for the entire utterance). Our CPU kernels run at real-time or faster-than-real-time for a subset of models, while the GPU models do not yet match this performance.

5.1. CPU Implementation

We achieve real-time CPU inference by avoiding any recomputation, doing cache-friendly memory accesses, parallelizing work via multithreading with efficient synchronization, minimizing nonlinearity FLOPs, avoiding cache thrashing and thread contention via thread pinning, and using custom hardware-optimized routines for matrix multiplication and convolution.

For the CPU implementation, we split the computation into the following steps:

1. **Sample Embedding:** Compute the WaveNet input causal convolution by doing two sample embeddings, one for the current timestep and one for the previous timestep, and summing them with a bias. That is,
$$x^{(0)} = W_{\text{emb,prev}} \cdot y_{i-1} + W_{\text{emb,cur}} \cdot y_i + B_{\text{embed}} \quad (1)$$

2. **Layer Inference:** For every layer j from $j = 1$ to ℓ with dilation width d:
   (a) Compute the left half of the width-two dilated convolution via a matrix-vector multiply:
   $$a^{(j)}_{\text{prev}} = W^{(j)}_{\text{prev}} \cdot x^{(j-1)}_{i-d} \quad (2)$$
   (b) Compute the right half of the dilated convolution:
   $$a^{(j)}_{\text{cur}} = W^{(j)}_{\text{cur}} \cdot x^{(j-1)}_{i} \quad (3)$$
   (c) Compute the hidden state $h^{(j)}$ given the conditioning vector $L^{(j)}_h$:
   $$a^{(j)} = a^{(j)}_{\text{prev}} + a^{(j)}_{\text{cur}} + B^{(j)}_h + L^{(j)}_h \quad (4)$$
   $$h^{(j)} = \tanh\left(a^{(j)}_{0:r}\right) \cdot \sigma\left(a^{(j)}_{r:2r}\right), \quad (5)$$
   where $v_{0:r}$ denotes the first r elements of the vector v and $v_{r:2r}$ denotes the next r elements. Then, compute the input to the next layer via a matrix-vector multiply:
   $$x^{(j)} = W^{(j)}_{\text{res}} \cdot h^{(j)} + B^{(j)}_{\text{res}} \quad (6)$$
   (d) Compute the contribution to the skip-channel matrix multiply from this layer, accumulating over all layers, with $q^{(0)} = B_{\text{skip}}$:
   $$q^{(j)} = q^{(j-1)} + W^{(j)}_{\text{skip}} \cdot h^{(j)} \quad (7)$$

3. **Output:** Compute the two output 1×1 convolutions:
$$z_s = \text{relu}\left(q^{(\ell)}\right) \quad (8)$$
$$z_a = \text{relu}\left(W_{\text{relu}} \cdot z_s + B_{\text{relu}}\right) \quad (9)$$
$$p = \text{softmax}\left(W_{\text{out}} \cdot z_a + B_{\text{out}}\right) \quad (10)$$
Finally, sample y_{i+1} randomly from the distribution p.

We parallelize these across two groups of threads as depicted in Figure 2. A group of main threads computes $x^{(0)}$, $a^{(j)}_{\text{cur}}$, $h^{(j)}$, and $x^{(j)}$, z_a, and p. A group of auxiliary threads computes $a^{(j)}_{\text{prev}}$, $q^{(j)}$, and z_s, with the $a^{(j)}_{\text{prev}}$ being computed for the next upcoming timestep while the main threads compute z_a and p. Each of these groups can consist of a single thread or of multiple threads; if there are multiple threads, each thread computes one block of each matrix-vector multiply, binary operation, or unary operation, and thread barriers are inserted as needed. Splitting the model across multiple threads both splits up the compute and can also be used to ensure that the model weights fit into the processor L2 cache.

Pinning threads to physical cores (or disabling hyperthreading) is important for avoiding thread contention and cache thrashing and increases performance by approximately 30%.

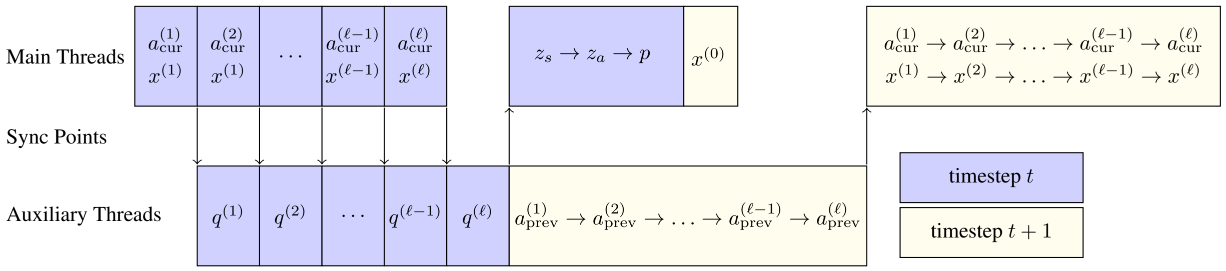
Main Threads
$a_{\text{cur}}^{(1)}$ $x^{(1)}$
$a_{\text{cur}}^{(2)}$ $x^{(1)}$
$\ldots$
$a_{\text{cur}}^{(\ell-1)}$ $x^{(\ell-1)}$
$a_{\text{cur}}^{(\ell)}$ $x^{(\ell)}$
$z_s \to z_a \to p$
$x^{(0)}$
$a_{\text{cur}}^{(1)} \to a_{\text{cur}}^{(2)} \to \ldots \to a_{\text{cur}}^{(\ell-1)} \to a_{\text{cur}}^{(\ell)}$
$x^{(1)} \to x^{(2)} \to \ldots \to x^{(\ell-1)} \to x^{(\ell)}$
Sync Points
Auxiliary Threads
$q^{(1)}$ $q^{(2)}$ $\ldots$ $q^{(\ell-1)}$ $q^{(\ell)}$
$a_{\text{prev}}^{(1)} \to a_{\text{prev}}^{(2)} \to \ldots \to a_{\text{prev}}^{(\ell-1)} \to a_{\text{prev}}^{(\ell)}$
timestep t
timestep $t+1$

Figure 2. Two groups of threads run in parallel. Computation of the W_{skip} is offloaded to the auxiliary threads while the main threads progress through the stack of WaveNet layers. While the main threads are computing the output layer, the auxiliary threads prepare the left W_{prev} half of the WaveNet layer convolutions for the upcoming timestep. Arrows indicate where one thread group waits on results from the other thread group, and are implemented as spinlocks.

Depending on model size, the nonlinearities ($\tanh$, sigmoid, and softmax) can also take a significant fraction of inference time, so we replace all nonlinearities with high-accuracy approximations, which are detailed in Appendix C. The maximum absolute error arising from these approximations is 1.5×10^{-3} for $\tanh$, 2.5×10^{-3} for sigmoid, and 2.4×10^{-5} for e^x. With approximate instead of exact nonlinearities, performance increases by roughly 30%.

We also implement inference with weight matrices quantized to `int16` and find no change in perceptual quality when using quantization. For larger models, quantization offers a significant speedup when using fewer threads, but overhead of thread synchronization prevents it from being useful with a larger number of threads.

Finally, we write custom AVX assembly kernels for matrix-vector multiplication using PeachPy (Dukhan, 2015) specialized to our matrix sizes. Inference using our custom assembly kernels is up to 1.5X faster than Intel MKL and 3.5X faster than OpenBLAS when using `float32`. Neither library provides the equivalent `int16` operations.

5.2. GPU Implementation

Due to their computational intensity, many neural models are ultimately deployed on GPUs, which can have a much higher computational throughput than CPUs. Since our model is memory bandwidth and FLOP bound, it may seem like a natural choice to run inference on a GPU, but it turns out that comes with a different set of challenges.

Usually, code is run on the GPU in a sequence of kernel invocations, with every matrix multiply or vector operation being its own kernel. However, the latency for a CUDA kernel launch (which may be up to 50 μs) combined with the time needed to load the entire model from GPU memory are prohibitively large for an approach like this. An inference kernel in this style ends up being approximately 1000X slower than real-time.

To get close to real-time on a GPU, we instead build a kernel using the techniques of persistent RNNs (Diamos et al., 2016) which generates all samples in the output audio in a single kernel launch. The weights for the model are loaded to registers once and then used without unloading them for the entire duration of inference. Due to the mismatch between the CUDA programming model and such persistent kernels, the resulting kernels are specialized to particular model sizes and are incredibly labor-intensive to write. Although our GPU inference speeds are not quite real-time (Table 2), we believe that with these techniques and a better implementation we can achieve real-time WaveNet inference on GPUs as well as CPUs. Implementation details for the persistent GPU kernels are available in Appendix D.

6. Conclusion

In this work, we demonstrate that current Deep Learning approaches are viable for all the components of a high-quality text-to-speech engine by building a fully neural system. We optimize inference to faster-than-real-time speeds, showing that these techniques can be applied to generate audio in real-time in a streaming fashion. Our system is trainable without any human involvement, dramatically simplifying the process of creating TTS systems.

Our work opens many new possible directions for exploration. Inference performance can be further improved through careful optimization, model quantization on GPU, and `int8` quantization on CPU, as well as experimenting with other architectures such as the Xeon Phi. Another natural direction is removing the separation between stages and merging the segmentation, duration prediction, and fundamental frequency prediction models directly into the audio synthesis model, thereby turning the problem into a full sequence-to-sequence model, creating a single end-to-end trainable TTS system, and allowing us to train the entire system with no intermediate supervision. In lieu of fusing the models, improving the duration and frequency models via larger training datasets or generative modeling techniques may have an impact on voice naturalness.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, Corrado, Greg S., Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Goodfellow, Ian, Harp, Andrew, Irving, Geoffrey, Isard, Michael, Jia, Yangqing, Jozefowicz, Rafal, Kaiser, Lukasz, Kudlur, Manjunath, Levenberg, Josh, Mané, Dan, Monga, Rajat, Moore, Sherry, Murray, Derek, Olah, Chris, Schuster, Mike, Shlens, Jonathon, Steiner, Benoit, Sutskever, Ilya, Talwar, Kunal, Tucker, Paul, Vanhoucke, Vincent, Vasudevan, Vijay, Viégas, Fernanda, Vinyals, Oriol, Warden, Pete, Wattenberg, Martin, Wicke, Martin, Yu, Yuan, and Zheng, Xiaoqiang. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Amodei, Dario, Anubhai, Rishita, Battenberg, Eric, Case, Carl, Casper, Jared, Catanzaro, Bryan, Chen, Jingdong, Chrzanowski, Mike, Coates, Adam, Diamos, Greg, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv preprint arXiv:1512.02595*, 2015.

Boersma, Paulus Petrus Gerardus et al. Praat, a system for doing phonetics by computer. *Glot international*, 5, 2002.

Bradbury, James, Merity, Stephen, Xiong, Caiming, and Socher, Richard. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

Chung, Junyoung, Gulcehre, Caglar, Cho, KyungHyun, and Bengio, Yoshua. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Diamos, Greg, Sengupta, Shubho, Catanzaro, Bryan, Chrzanowski, Mike, Coates, Adam, Elsen, Erich, Engel, Jesse, Hannun, Awni, and Satheesh, Sanjeev. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of The 33rd International Conference on Machine Learning*, pp. 2024–2033, 2016.

Dukhan, Marat. Peachpy meets opcodes: direct machine code generation from python. In *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, pp. 3. ACM, 2015.

Graves, Alex, Fernández, Santiago, Gomez, Faustino, and Schmidhuber, Jürgen. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pp. 369–376, New York, NY, USA, 2006. ACM.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Mehri, Soroush, Kumar, Kundan, Gulrajani, Ishaan, Kumar, Rithesh, Jain, Shubham, Sotelo, Jose, Courville, Aaron, and Bengio, Yoshua. Samplernn: An unconditional end-to-end neural audio generation model. *arXiv preprint arXiv:1612.07837*, 2016.

Morise, Masanori, Yokomori, Fumiya, and Ozawa, Kenji. World: a vocoder-based high-quality speech synthesis system for real-time applications. *IEICE TRANSACTIONS on Information and Systems*, 99(7):1877–1884, 2016.

Oord, Aaron van den, Kalchbrenner, Nal, and Kavukcuoglu, Koray. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

Paine, Tom Le, Khorrami, Pooya, Chang, Shiyu, Zhang, Yang, Ramachandran, Prajit, Hasegawa-Johnson, Mark A, and Huang, Thomas S. Fast wavenet generation algorithm. *arXiv preprint arXiv:1611.09482*, 2016.

Pascual, Santiago and Bonafonte, Antonio. Multi-output rnn-lstm for multiple speaker speech synthesis with α-interpolation model. *way*, 1000:2, 2016.

Prahallad, Kishore, Vadapalli, Anandaswarup, Elluru, Naresh, et al. The blizzard challenge 2013indian language task. In *In Blizzard Challenge Workshop 2013*, 2013.

Rao, Kanishka, Peng, Fuchun, Sak, Haşim, and Beaufays, Françoise. Grapheme-to-phoneme conversion using long short-term memory recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 4225–4229. IEEE, 2015.

Ribeiro, Flávio, Florêncio, Dinei, Zhang, Cha, and Seltzer, Michael. Crowdmos: An approach for crowdsourcing mean opinion score studies. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 2416–2419. IEEE, 2011.

Ronanki, Srikanth, Henter, Gustav Eje, Wu, Zhizheng, and King, Simon. A template-based approach for speech synthesis intonation generation using lstms. *Interspeech 2016*, pp. 2463–2467, 2016.

Sotelo, Jose, Mehri, Soroush, Kumar, Kundan, Santos, Joao Felipe, Kastner, Kyle, Courville, Aaron, and Bengio, Yoshua. Char2wav: End-to-end speech synthesis. In *ICLR 2017 workshop submission*, 2017. URL `https://openreview.net/forum?id=B1VWyySKx`.

Stephenson, Ian. *Production Rendering, Design and Implementation*. Springer, 2005.

Taylor, Paul. *Text-to-Speech Synthesis*. Cambridge University Press, New York, NY, USA, 1st edition, 2009. ISBN 0521899273, 9780521899277.

Theis, Lucas, Oord, Aäron van den, and Bethge, Matthias. A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*, 2015.

van den Oord, Aäron, Dieleman, Sander, Zen, Heiga, Simonyan, Karen, Vinyals, Oriol, Graves, Alex, Kalchbrenner, Nal, Senior, Andrew, and Kavukcuoglu, Koray. Wavenet: A generative model for raw audio. *CoRR abs/1609.03499*, 2016.

Weide, R. *The CMU pronunciation dictionary 0.7*. Carnegie Mellon University, 2008.

Yao, Kaisheng and Zweig, Geoffrey. Sequence-to-sequence neural net models for grapheme-to-phoneme conversion. *arXiv preprint arXiv:1506.00196*, 2015.

Zen, Heiga and Sak, Haşim. Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pp. 4470–4474. IEEE, 2015.

Zen, Heiga, Senior, Andrew, and Schuster, Mike. Statistical parametric speech synthesis using deep neural networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pp. 7962–7966, 2013.