Optimizing WaveNet Inference using Caching and Quantization

Yamini Vibha Ananth
Applied Math and Computer Science
Columbia University
New York, New York
yva2002@columbia.edu

Stan Liao
Applied Math and Computer Science
Columbia University
New York, New York
sl4239@columbia.edu

***Abstract*—WaveNet is a groundbreaking generative model that allows for the generation of human-like speech and other sounds, including unique and sonically pleasant music. However, because WaveNet is autoregressive, only one value can be inferenced at a time, making it prohibitively slow for usage in situations where swift inference is critical. We implement a PyTorch based method that caches recurrent nodes for for dramatically reducing inference time for WaveNet, as introduced by le Paine et al. We further demonstrate the effects of static and dynamic quantization on the training time, inference time, and dynamic ranges of generated output. We demonstrate that applying dynamic mu-law quantization after training can effectively limit the dynamic range of samples generated by WaveNet.**

***Index Terms*—WaveNet, Generative AI, Caching, Quantization**

I. Background

A. WaveNet

In this series of experiments, we explore the effects of quantization on a PyTorch implementation of WaveNet. WaveNet is a deep generative model of raw audio waveforms, capable of mimicking both human voice (for text-to-speech systems) as well as other audio signals such as music. WaveNet is a fully probabilistic and autoregressive model with predictive distribution for each proceeding sample conditioned on all previous samples (Van de Oord, 2016).

The model is autoregressive and constructed largely of causal convolutions with increasing dilation. Using dilation allows the receptive field to increase exponentially with depth, which is important for modeling the long-range temporal dependencies in audio signals. WaveNet uses a single preprocessing causal convolution layer followed by several blocks of dilated causal convolutional layers that skip and residual connections, then post-processing (including a softmax layer), as visible in Figure 1.

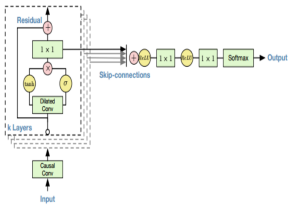

Fig. 1. Architecture of Wavenet

B. Inference Runtime

The original WaveNet paper's implementation for inference is on the order of $\mathcal{O}(2^L)$, where L is the number of layers in the network, due to the increasing dilation size throughout the convolutional layers (see Figure 2. In practice, arguably the most popular application of WaveNet, Text-to-Speech (TTS), is incredibly restricted by the runtime of inference. Therefore, such a time-intensive inference operation is non-feasible particularly for consumer facing applications of WaveNet. However, using caching, it is possible to reduce the inference runtime to a mere $\mathcal{O}(L)$, as shown by le Paine et al later in 2016. The basic idea is that, given a specific set of recurrent nodes in the graph of WaveNet, the current output can be computed without examining all the nodes in the graph (see Figure 3. Further, caching these recurrent states can allow for efficient inference compared to recomputing them from scratch for each new sample.

C. Quantization for Audio

WaveNet trains on raw audio waveforms. This is done by taking continuous analog audio waveforms

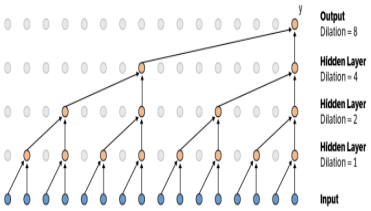

Fig. 2. Computation graph for naive implementation of inference in WaveNet, requires $\mathcal{O}(2^L)$ computations per output

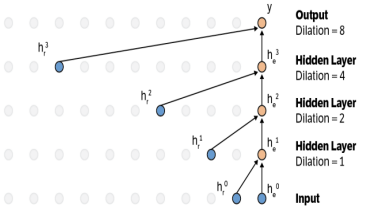

Fig. 3. Computation graph for fast generation WaveNet, using caching, requires $\mathcal{O}(L)$ computations per output

and converting them to time-series data by capturing their descriptive values at successive points in time and chaining them together. The rate at which samples are captured is referred to as "bit rate", while the number of bits used to represent each captured sample is referred to as the "bit depth".

Quantization is well known as a process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. It reduces both the memory requirement and computational cost of using neural networks. The major tradeoff with quantization is in accuracy, since networks are not representing information precisely. However, in audio generation, this is less of an overt concern.

By the Weber-Fechner law, the relation between the actual change in a physical stimulus and the perceived change in the stimulus is relatively low for audio. For example, in speech, big frequency jumps (eg between high and low pitched sounds, pauses, etc) are relatively more important than smaller variances in intonation, dynamics, etc. to get the bigger picture of what is being conveyed. Thus, quantization offers many potential benefits in reducing model size and training/inference time, and its major drawback in reduced accuracy is a worthy tradeoff.

II. High Level Approach

In order to avoid the process of encoding and decoding associated with text-to-speech, and given the project

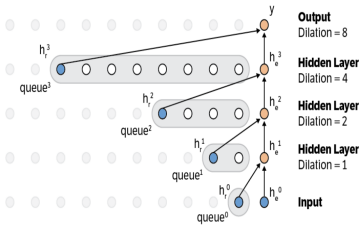

Fig. 4. Adding caching scheme for efficient generation. Due to dilated convolutions, the size of the queue at the l-th hidden layer is 2^l

scope around the quantization and benchmarking of WaveNet directly, a music generation task was selected for training WaveNet. In particular, Bach's Chaconne violin concerto series was selected as the training dataset.

First, we implemented a method for fast generation of samples.

Second, we pretrained a model on CPU and applied dynamic quantization to reduce the dynamic range of generated samples.

Third, we constructed and trained a version of WaveNet with the necessary quantization stub layers to perform static quantization. Using this version of WaveNet, we applied static quantization after training to observe the difference on inference time and constraint of dynamic range.

III. Implementation Details

A. Fast Inference

The core idea in implementing fast inference is that the same scheme can be applied anytime the weights need to be updated. There are two main components to the caching algorithm for fast inference:

1) *Generation Model* - Behaves like a single step of a multi layer RNN
2) *Convolution Queues* - Queues containing recurrent states computed by the layer below

The algorithm works in the following manner, as laid out in Figure 4:

1) Initialize the generation model using a pre-trained Wavenet model. Initialize each convolution queue with all recurrent states as zeros.
2) To generate samples, execute the following two steps every time.
    a) For each convolution queue, pop the recurrent state and feed it to the corresponding state of the generation model, then calculate the new hidden states and output.

b) For each new hidden state push it into the convolution queue of the layer above.

We adapt a version of fast training written for WaveNet for Tensorflow by the authors le Paine et al.

B. Post-Training Dynamic Quantization

In dynamic quantization, weights are quantized with activations read/stored in floating point and quantized for compute.

To implement dynamic quantization, we use the PyTorch dynamic quantization API. In PyTorch, this method converts all weights to 'int8' and converting activations to 'int8' on the fly dynamically. Computations can therefore be sped up thanks to the usage of 'int8' matrix multiplication and convolution implementations.

We use a vanilla pretrained WaveNet model trained for one epoch and apply dynamic quantization with builtin PyTorch functionality. For the specific implementation function of dynamic quantization, we use the mu-law companding transformation which allows for the compression of a waveform (representation with significantly less bits) while maintaining the important characteristics of the original data.

C. Post-Training Static Quantization

In static quantization, both weights and activations are quantized, and calibration is required post training.

To improve further on latency, static quantization converts networks to use both 'int' and 'int8' memory accesses. Essentially, data is first fed through the network, the distribution of different activations is computed, and the specific quantization of each activation at inference time is computed. In this case, the range of activation is divided into 256 levels. Quantized values can therefore be passed between operations rather than conversions between 'int' and 'float' values, resulting in significant speedup of inference.

In this case, we use the PyTorch static quantization API. The model architecture is modified to use Quant and DeQuant stubs to bookend the network. Then, the model is fused and static quantization is implemented with the default strategy of dividing activations into 256 levels. Afterwards, calibration is applied.

Quantization is not a CPU-specific technique (e.g. NVIDIA's TensorRT can be used to implement quantization on GPU). However, inference time on GPU is already usually "fast enough", and CPUs are more attractive for large-scale model server deployment. Consequently, as of PyTorch 1.6, only CPU backends are available in the native API for quantization. As a result, we were not able to train WaveNet using GPU for the purposes of quantization testing. Due to the prohibitively lengthy training time of WaveNet, the limitations of the compute resources available for use in this project, and that we were primarily interested in the inference performance gains provided by quantization, the model was only trained for one epoch. The fact that audio generation is not concerned with "accuracy" as an outbound metric of model performance also contributed to this decision.

IV. Benchmarking Methodology and Results

A. Fast Generation

We use a pretrained CPU model and apply dynamic quantization. This way, we have the exact same model both before and after quantization. For the purposes of benchmarking, we generate 50,000 samples which equates to roughly 2 seconds of audio. In practice, ideally many more seconds of audio would be generated for real-world examples; however, the goal here was simply to benchmark inference. For both the non quantized and quantized models, we generate 100 samples, timing the inference on each step and ensuring that all cores are synchronized.

In our model, we have a relatively small number of layers (10). According to le Paine et al, the fast generation method outperform the regular generate method most intensely for values of L=12 and above. Thus, we can observe a small quantity of speedup, however effects of speedup would be more prominent with more layers.

We see not much speedup after scaling to 4 CPUs (and, in fact, very little even from 2→4). As mentioned below, our machine only fully utilizes 2 CPUs at a time. Thus, while we have some speedup moving from 2 → 4, adding additional CPUs does not lead to speedup.

However, it is possible that if we did hyperparameter optimization on our model to include more dataloader workers and a higher batch size, we could see some more speedup, which could be made possible with more memory.

B. Comparing static vs dynamic quantization

In order to implement static quantization, it was necessary to add quantization stubs to the WaveNet constructor and retrain on the Bach-Chaconne dataset. Because quantization on PyTorch is not supported over GPU, we unfortunately could not accelerate training using parallelization, distributed training, or the accelerated compute power of the GPU. Thus, trained for only one epoch; however, due to the dense convolutional layers

TABLE I
RUNTIME AND SPEEDUP OF GENERATION AFTER APPLYING CACHING

CPUs	**Memory (GB)**	**Mean generate runtime (secs)**	**Mean fast-generate runtime (secs)**	**Speedup**
2	13	150.98	118.88	1.2700
4	26	144.16	113.35	1.2718
8	52	145.88	115.83	1.2594

TABLE II
COMPARING EFFECTS OF POST TRAINING STATIC AND DYNAMIC QUANTIZATION VS. NO QUANTIZATION MODEL

Model Type	Mean per-epoch training time (hr)	Mean post-training quantization - time to convert (s) over 100 trials	Model Size (MB)	Mean fast_generate runtime (s) over 100 trials	CPU utilization of fast_generate (%)
No Quantization	3.22	—	6.998	118.293	200
Post Training Dynamic Quantization	—	11.09	6.998	100.234	200
Post Training Static Quantization	3.48	16.30	3.526	98.938	200

and sample density in the training data, this process still took approximately 6.23 hours.

To measure model size, we specifically considered the size of the model parameters, inputs, buffers, and intermediate values (eg values, gradients). To compute the CPU Utilization of fast_generate for each model, we use the top Linux command after running our generation in the background. Our values are above 100% because they indicate usage compared to one core, while our machine had 4 cores; ergo, a CPU utilization of 200% would indicate a true utilization of 50%.

For inference, both static and dynamic quantization outperform the non-quantized model. In this particular situation, on average, the dynamic quantization model is 15.26% faster, and the static quantization method is 17.00% faster than the non-quantized model for inference. While this does not seem very high, quantization is focused on saving memory and power, which we did accomplish successfully.

Further considering the results, static quantization does not provide significant performance or memory gains over dynamic quantization in the implementation. When further accounting for the training time, as retraining is necessary using quantization stubs, the overhead and compute cost associated with static quantization is significantly greater than that of dynamic quantization. As a result, we conclude that in the context of WaveNet, dynamic quantization provides sufficient gains.

V. CONCLUSION

In this paper, we explored ways to optimize inference for WaveNet, a generative audio neural network. Using our fast generation algorithm, we saw significant speedup in inference by caching our weight updates. After implementing quantization, both dynamic and static, on WaveNet, we see small but measurable speedup as well as a smaller model size for static quantization. In general, we conclude that dynamic quantization may offer some performance gains that are worth it for improving performance slightly. Quantization - and especially post-training quantization (PTQ) - is a promising frontier in optimizing inference for WaveNet.

VI. FUTURE WORK

Because of the resource constraints for compute, we were not able to experiment with training Wavenets with different numbers of layers; however, this analysis would be useful for further benchmarking the fast generation method, as it is known to be more performant for high numbers of layers (eg 14 and greater).

Furthermore, there is also a NVIDIA CUDA compatible version of WaveNet that has been written without PyTorch libraries in between. Using this version of WaveNet and applying optimizations such as caching and quantization could lead to further performance gains which outdo PyTorch implementations due to the usage of more performant languages.

VII. Acknowledgments

Thank you to Professor Kauotar and Professor Parijat Dube for holding the course COMS6998 High Performance Machine Learning at Columbia University in Spring 2023. We learned a great deal from the course and will carry the experience through future professional work and research.

References

[1] Paine, T. L., Khorrami, P., Chang, S., Zhang, Y., Ramachandran, P., Hasegawa-Johnson, M. A., & Huang, T. S. (2016). Fast Wavenet Generation Algorithm.*ArXiv [Cs.SD]*. Retrieved from http://arxiv.org/abs/1611.09482

[2] van den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., … Kavukcuoglu, K. (2016). WaveNet: A Generative Model for Raw Audio.*ArXiv [[Cs.SD](Http://Cs.Sd/)]*. Retrieved from http://arxiv.org/abs/1609.03499

[3] Herrmann, V. (2017). Open Source PyTorch Wavenet implementation. Retrieved from https://github.com/vincentherrmann/pytorch-wavenet/

[4] Krishnamoorthi, R. (2020, March 26). *Quantization in PyTorch*. PyTorch. Retrieved May 11, 2023, from https://pytorch.org/blog/introduction-to-quantization-on-pytorch/

[5] Kuchaiev, O., Ginsburg, B., Gitman, I., Lavrukhin, V., Li, J., Nguyen, H., … Micikevicius, P. (2018). Mixed-Precision Training for NLP and Speech Recognition with OpenSeq2Seq.*ArXiv [Cs.CL]*. Retrieved from http://arxiv.org/abs/1805.10387

[6] Pednekar, S., Krishnadas, A., Cho, B., & Makris, N. C. (2023). Weber's Law of perception is a consequence of resolving the intensity of natural scintillating light and sound with the least possible error.*Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*,*479*(2271), 20220626. doi:10.1098/rspa.2022.0626