Optimizing WaveNet with Caching and Quantization

Yamini Ananth and Stan Liao
Columbia University

code

Executive Summary

Goal: WaveNet is an autoregressive generative network for generating raw audio waveforms. Due to its autoregressive nature, inference is prohibitively slow. This project aims to optimize WaveNet's inference performance through implementations using caching and quantization.

Approach: We demonstrate the performance of caching-based optimization for inference, post-training dynamic quantization and post-training static quantization,

Benefit: Caching-based method is universally applicable across CNN architectures. Quantization method has multiple benefits: speedup & reducing model size

Problem Motivation

1. WaveNet is powerful, achieves more realistic TTS than previous world-class models at Google, etc. However, it is autoregressive and probabilistic so inference is on the order of 2^L by number of layers *L*. Inference time is crucial for most applications of TTS and other audio generation models. We aim to speedup the inference with caching.
2. WaveNet is a memory-intensive model both in size and bandwidth. We aim to reduce model size using quantization in the 2 main methods: post-training dynamic and post-training static. This is useful for efficiently deploying the model.

Background Work

- We build upon the original architecture of WaveNet as introduced by Van de Oort et al. in 2016
- We provide a PyTorch based algorithm for caching-based implementation for fast generation introduced by le Paine et al, building off of the open source PyTorch implementation of WaveNet led by Vincent Herrmann.
- In order to implement post training static and dynamic quantization, we use the PyTorch Quantization API, as described by Krishnamoorthi et al.

Technical Challenges

Quantization in PyTorch

- Fully understand nuances of PyTorch implementation of quantization:
- Need to retrain model with a modified architecture
- Training is slow on CPU, but can’t use GPU as PyTorch does not support GPU quantization

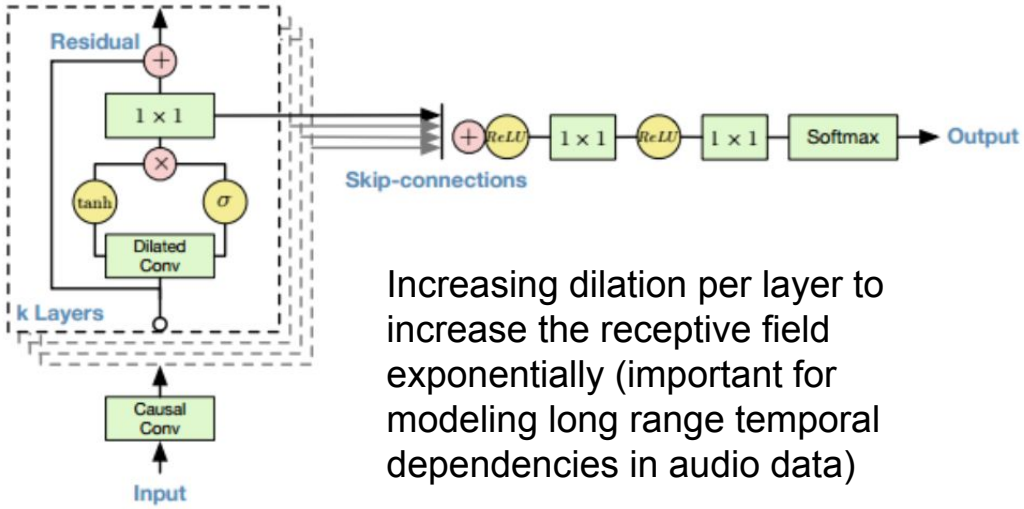
Fast Generation (caching)

- Understand & implement DP algorithm on a neural network

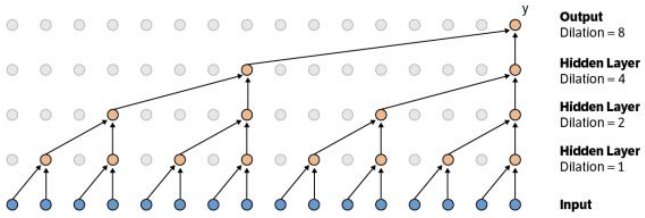
Approach

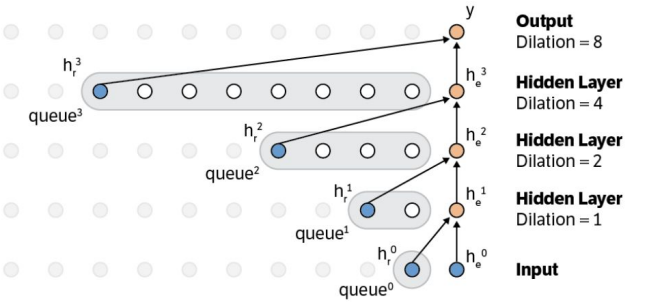
- To avoid the process of encoding and decoding text-to-speech, and given project scope around quantization and benchmarking WaveNet directly, chose music generation task with 10layers
  - Training data: Bach’s Chaconne violin concerto series
- 1st: implement algo for fast generation of samples
- 2nd: pretrained model using dynamic quantization
- 3rd: restructure & retrain model w/ stub layers, perform static quantization

Solution Diagram/Architecture



Increasing dilation per layer to increase the receptive field exponentially (important for modeling long range temporal dependencies in audio data)





Implementation Details

Fast inference:

- **Generation Model:** Behaves like a single step of a multi layer RNN
- **Convolution Queues:** Queues containing recurrent states computed by the layer below
- Initialize the generation model using a pre-trained Wavenet model. Initialize each convolution queue with all recurrent states as zeros.
    - 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.
    - For each new hidden state push it into the convolution queue of the layer above.

Quantization

- Post training dynamic
    - Using pretrained model on CPU
    - apply dynamic quantization - weights quantized, activations quantized on the fly
    - measure the time it takes to convert the model to quantized + measure inference runtime for 100 trials of inference of 30000 samples each.
- Post training static quantization
    - Static quantization requires backend to be prepped for quantization (Quantized CPU) - Activations quantized
    - Add quant & dequant stub layers
    - Retrain on CPU
    - Ran fast_generate for 30000 samples, 100 trials for benchmarking

Experiment Design Flow

1. Construct fast_generation method
2. Benchmark fast_generation vs normal generation on 3 types of machines (different vCPU and memory count)
3. Pretrain model on CPU; dynamically quantize post-training
4. Train model on CPU with static quantization stubs, statically quantize post-training
5. Benchmark model size, mean inference runtime, and CPU utilization on inference for all 3 cases.

Demo

- Original Bach-Chaconne Violin Concerto (sample, 5 secs)
- Vanilla (non-quantized) inference sample
- Post-Training Dynamically Quantized sample
- Post-Training Statically Quantized sample

Normal vs. Fast Generation Speedup

CPU cores	CPU memory	Mean generate runtime (secs) over 100 trials	Mean fast_generate runtime (secs) over 100 trials	Speedup
2 vCPU	13GB	150.98 s	118.88 s	1.27002018843
4 vCPU	26GB	144.16 s	113.35 s	1.27181296868
8 vCPU	52GB	145.88 s	115.83 s	1.2594319261

Comparing Quantization Methods

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	9.22	—	6.998	118.293	200
Post Training Dynamic Quantization	—	11.09	6.998	100.234	200
Post Training Static Quantization	—	16.30	3.526	98.938	200

Conclusion

- Caching based algorithm is incredibly powerful for **increasing speedup in inference** for WaveNet
- Quantization offers **minimal increase in inference** performance
- Post-training Dynamic Quantization is the most flexible option for increasing inference performance slightly with very low time taken to quantize and no retraining required.
- Post-training Static Quantization reduces model size twice as much as Dynamic Quantization, but requires investment in model retraining.

References

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

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

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

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

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

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/1609.0349