FAST GENERATION FOR CONVOLUTIONAL AUTOREGRESSIVE MODELS

Prajit Ramachandran*, Tom Le Paine*, Pooya Khorrami, Mohammad Babaeizadeh, Shiyu Chang, Yang Zhang, Mark Hasegawa-Johnson, Roy Campbell, & Thomas Huang

University of Illinois at Urbana-Champaign
{prmchnd2, paine1, pkhorra2, mb2,
chang87, yzhan143, jhasegaw, rhc, t-huang1}@illinois.edu

ABSTRACT

Convolutional autoregressive models have recently demonstrated state-of-the-art performance on a number of generation tasks. While fast, parallel training methods have been crucial for their success, generation is typically implemented in a naïve fashion where redundant computations are unnecessarily repeated. This results in slow generation, making such models infeasible for production environments. In this work, we describe a method to speed up generation in convolutional autoregressive models. The key idea is to cache hidden states to avoid redundant computation. We apply our fast generation method to the Wavenet and PixelCNN++ models and achieve up to $21\times$ and $183\times$ speedups respectively.

1 INTRODUCTION

Autoregressive models are a powerful class of generative models that factorize the joint probability of a data sample x into a product of conditional probabilities. Autoregressive models such as Wavenet (van den Oord et al., 2016a), ByteNet (Kalchbrenner et al., 2016a), PixelCNN (van den Oord et al., 2016b;c), and Video Pixel Networks (Kalchbrenner et al., 2016b) have shown strong performance in audio, textual, image, and video generation. Unfortunately, generating in a naïve fashion is typically too slow for practical use. For example, generating a batch of 16 32×32 images using PixelCNN++ (Salimans et al., 2017) takes more then 11 minutes on commodity hardware with a Tesla K40 GPU.

The ability to do fast generation is useful for many applications. Production environments have tight latency constraints, so real-time speech generation, machine translation, and image super-resolution (Dahl et al., 2017) all require fast generation. Furthermore, quick simulation of environment dynamics is important for fast training in model-based reinforcement learning (Oh et al., 2015). However, slow generation hampers the use of convolutional autoregressive models in these situations.

In this work, we present a method to significantly speed up generation in convolutional autoregressive models. The contributions of this work are as follows:

1. We present a general method to enable fast generation for autoregressive models through caching. We describe specific implementations of this method for Wavenet (van den Oord et al., 2016a) and PixelCNN++ (Salimans et al., 2017). We demonstrate our fast generation achieves up to $21\times$ for Wavenet and $183\times$ for PixelCNN++ over their naïve counterparts.
2. We open-source our implementation of fast generation for Wavenet[1] and PixelCNN++[2]. Our generation code is compatible with other open-source implementations of these models that also implement training.

2 METHODS

Naïve generation for convolutional autoregressive models recalculates the entire receptive field at every iteration (we refer readers to van den Oord et al. (2016a); Salimans et al. (2017) for details). This results in exponential time and space complexity with respect to the receptive field. In this section, we propose a method that avoids this cost by caching previously computed hidden states and using them in the subsequent iterations.

*Denotes equal contribution.
[1]https://github.com/tomlepaine/fast-wavenet
[2]https://github.com/PrajitR/fast-pixel-cnn

2.1 Caching for dilated convolutions

To generate a single output y, computations must be performed over the entire receptive field which is exponential with respect to the number of layers. A naïve generation method repeats this computation over the entire receptive field at every step, which is illustrated in Figure 1A. However, this is wasteful because many hidden states in the receptive field can be re-used from previous iterations. This naïve approach has been used in open-source implementations of Wavenet[3].

Instead of recomputing all of the hidden states at every iteration, we propose caching hidden states from previous iterations. Figure 1B illustrates this idea, where each layer maintains a cache of previously computed hidden states. During each generation step, hidden states are popped off the cache to perform the convolutions. The newly generated hidden states are then pushed back into the cache for future computation. Therefore, the computation and space complexity are linear in the number of layers instead of exponential.

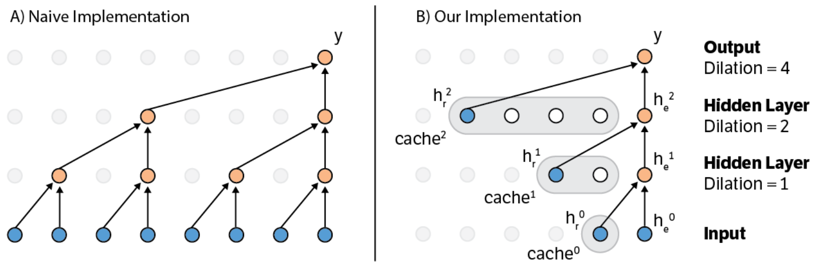

Figure 1: **Comparison of naïve implementation of the generation process and our proposed method.** Orange nodes are computed in the current timestep, blue nodes are previously cached states, and gray nodes are not involved in the current timestep. Notice that generating a single sample requires $O(2^L)$ operations for the naïve implementation where L is number of layers in the network. Meanwhile, our implementation only requires $O(L)$ operations to generate a single sample.

2.2 Caching for strided convolutions

The caching algorithm for dilated convolutions is straightforward because the number of hidden states in each layer is equal to the number of inputs. Thus, each layer can simply maintain a cache that is updated on every step. However, strided convolutions pose an additional challenge since the number of hidden states in each layer is different than the number of inputs.

A downsampling (strided convolutional) layer will not necessarily generate an output at each timestep (see the first hidden layer in Figure 2) and may even skip over some inputs (see the second hidden layer in Figure 2). On the other hand, an upsampling (strided transposed convolutional) layer will produce hidden states and outputs for multiple timesteps (see the last hidden layer in Figure 2). As a result, the cache cannot be updated in every timestep. Thus, each cache has an additional property *cache_every*, where the cache is only updated every *cache_every* steps. Every downsampling layer increases the *cache_every* property of the layer by the downsampling factor (2 in the case of Figure 2). Conversely, every upsampling layer decreases the *cache_every* property of the layer by the upsampling factor (also 2 in the case of Figure 2).

2.3 Model-specific details

Wavenet uses 1D dilated convolutions. Our fast implementation of Wavenet follows directly from the components outlined in Section 2.1.

PixelCNN++ improves upon PixelCNN (van den Oord et al., 2016c) through a variety of modifications, including using strided convolutions and transposed convolutions instead of dilation for speed. Our method scales from 1D to 2D with very few changes. The caches for each layer are now 2D, with a height equal to the filter height and a width equal to the image width. After an entire row is generated, the oldest row of the cache is popped and the new row is pushed. Because strided convolutions are used, we use the *cache_every* idea detailed in Section 2.2. For full details please refer to our code.

[3] `https://github.com/ibab/tensorflow-wavenet`

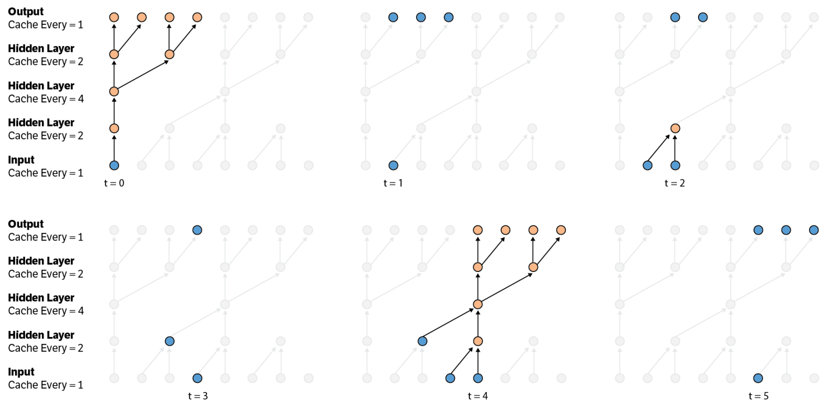

Figure 2: **Fast generation for a network with strided convolutions**. We show an example model with 2 convolutional and 2 transposed convolutional layers each with a stride of 2 (Dumoulin & Visin, 2016). Due to the stride, each layer has fewer states than network inputs. Orange nodes are computed in the current timestep, blue nodes are previously cached states, and gray nodes are not involved in the current timestep. In the first timestep ($t = 0$), the first input is used to compute and cache all nodes for which there is sufficient information to generate, including the first four outputs. At $t = 1$, there are no nodes that have sufficient information to be computed, but the output for $t = 1$ has already been computed at $t = 0$. At $t = 2$, there is one new node that now has sufficient information to be computed, although the output for $t = 2$ has also been computed at $t = 0$. The $t = 3$ scenario is similar to $t = 1$. At $t = 4$, there is enough information to compute multiple hidden states and generate the next four outputs. This is analogous to the $t = 0$ scenario. $t = 5$ is analogous to $t = 1$, and this cycle is followed for all future time steps.

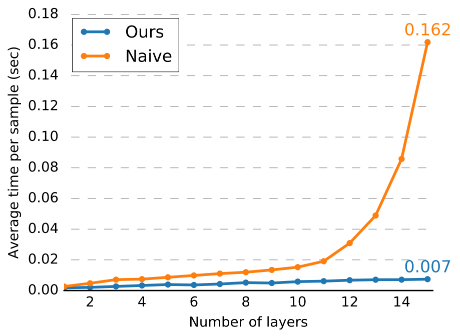

Figure 3: **Wavenet timing experiments.** We generated from a model with 2 sets of L dilation layers each, using a naïve implementation and ours. Results are averaged over 100 repeats. When L is small, the naïve implementation performs better than expected due to GPU parallelization of the convolution operations. When L is large, the difference in performance is more pronounced.

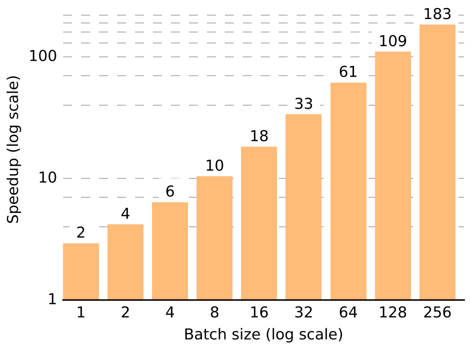

Figure 4: **PixelCNN++ timing experiments.** We generated images using the model architecture described in (Salimans et al., 2017). Due to the huge number of convolution operations in the naïve implementation, GPU utilization is always high and there is no room for parallelization across batch. Since our method avoids redundant computations, larger batch sizes result in larger speedups.

3 Experiments

We implemented our methods for Wavenet (van den Oord et al., 2016a) and PixelCNN++ (Salimans et al., 2017) in TensorFlow (Abadi et al., 2016). We compare our proposed method with a naïve implementation of Wavenet[4] and a naïve implementation of PixelCNN++[5] in Figures 3 and 4 respectively. The results indicate significant speedups, up to 21× for Wavenet and 183× for PixelCNN++.

[4]https://github.com/ibab/tensorflow-wavenet
[5]https://github.com/openai/pixel-cnn

REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

Ryan Dahl, Mohammad Norouzi, and Jonathon Shlens. Pixel recursive super resolution. *arXiv preprint arXiv:1702.00783*, 2017.

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016a.

Nal Kalchbrenner, Aaron van den Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. Video pixel networks. *arXiv preprint arXiv:1610.00527*, 2016b.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pp. 2863–2871, 2015.

Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

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

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

Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328*, 2016c.