Generative Gated Convolutional Networks for Raw Audio

Spencer Kresge **Neale Ratzlaff**
kresges@oregonstate.edu nealeratzlaff@gmail.com

Abstract

In this work we expand on the growing body of work pertaining to dense signal generation using deep neural networks. We present relevant heuristics pertaining to generating raw audio from a fully convolutional neural network, trained on millions of audio samples. We used the completely probabilistic and autoregressive architecture described in [1] to learn a joint conditional probability for each audio sample, dependent on all previous samples. The generated audio samples, without post-processing, outperform previous works that attempt also to generate dense audio vectors. Furthermore, we discuss implementation and strategies for effective model creation: including a discussion of training data selection and hyperparameter search. Most importantly, because modeling music is hard to quantify, we provide a library of generated audio samples using different training data and network configurations. This library provides insight into an inherently subjective modeling problem.

1 Introduction

Since the advent of trainable neural networks in 2012, there has been an overwhelming surge in data produced about deep learning in general. Most of the development in deep learning has been on the application side, as the systems that can solve real-world problems are developed. This growth of the solution space has produced much research into improving techniques where old initiatives failed. Fully conditional, parametric models have been shown in recent years to be able to learn and generate many forms of dense vector data such as images, text, and audio. Most famously, convolutional neural networks were shown to be both competent feature extractors and learners when applied to data with large spatial correlations [2]. By learning a conditional distribution for each sample, we are able to invert a discriminitive convolutional neural network into a generative one. Traditionally, learning conditional dependencies on large datasets has been restricted to recurrent neural networks, which connect layers and accumulate gradients through time steps. This allows them to learn parameter representations that know about past and future data. Recurrent nets have been used with much success in areas such as gene interpolation and language modeling [6], where data is one dimensional and has clear ordered dependencies. More recently, the PixelRNN [3] has shown that recurrent nets can be applied in multiple dimensions to model images. PixelRNN was able to generate images on par with adversarial networks without the architectural complexity [7]. It is well known that recurrent nets are relatively slow and hard to train due to their long gradient pipelines. Information doesn't flow as well through a recurrent net as it does a convolutional one. Convolutions are faster and more easily parallelizable, and recurrent connections have been modeled by convolutional layers using masked convolutions [4]. Modeling recurrent connections as convolutional layers allowed PixelRNN to be reformulated as PixelCNN, a faster network that matches the performance of PixelRNN on Imagenet. Therefore we can use a fully convolutional network with masked convolutions to learn a conditional distribution for each sample, for data in any dimension.

The rest of the paper is organized as follows. Section 2 details the model architecture and design decisions. Training and implementation details of each model are contained in Section 3. Results and evaluation of each of the models performance are detailed in Section 4, and concluding remarks are made in Section 5.

2 Architecture

Here we present the details of the model that we used, along with discussion of the various design decisions. The model was implemented using Google's Tensorflow framework [9] and analysis was preformed using the attached Tensorboard tools. Tensorboard was essential to our aim of providing design heuristics, due to the provided ability to visualize flow graphs, loss metrics, and model outputs. As specified in section 1, we modeled our network after the architecture presented in [1], but here are the important points. The model is a fully convolutional neural network that uses masked convolutions to learn a true conditional distribution for each sample with the purpose of generating similar data.

$$p(x) = \prod_{i=1}^{N} (x_t | x_1, x_2, ..x_{t-1})$$

We do this using a stack of dilated causal convolutions without pooling. We do not use pooling layers because we want our output to have the same dimension as the input. Pooling is meant to provide a different level of granularity to extract higher level features; we accomplish this instead by using dilated convolutions, of which we go into more detail later. Our network uses standard back propagation to pass gradients through the network, and is optimized to maximize the log-likelihood on the data. The network uses a softmax distribution at the output to categorize the next sample with respect to the samples $(x_{t-1} \quad ... \quad x_{t-n})$.

2.1 Causal Convolutions

Causal, or masked convolutions have been described in [4] and implemented in PixelCNN with success. When learning conditional distributions, parameters for any given sample must depend solely on the samples before. Standard convolutional layers are applied globally, with the shape of the window being a square, which forces the parameters to be learned independently of the ordering of the data. In short, standard convolutional layers infer some local connectivity within the window, but do not impose any strict ordering to the data. In order to do this, we need to change the shape of the window to better reflect the ordering of the data. The result is a causal convolution that masks out data that occurs further on in the ordering.

2.2 Dilated Convolutions

With a dense data representation such as raw audio, each second contains many thousands of samples. It is then intractable to apply convolutional layers to the entire dataset at each training step. Therefore we use dilated convolutions [8] to capture dependencies with decreasing granularity. This is a very natural way to handle long range dependencies, as the correlations between samples will decrease with distance. When applied to audio, we can simplify the problem to 1-D convolutions. This enables each sampled data point to develop its own conditional distribution dependent only on prior data. We the

n stack many layers of these dilated-causal blocks to form a deep network of variable size.

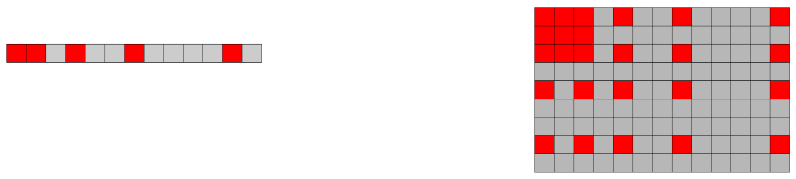

(a) A dilated convolution in one dimension (b) A dilated convolution in two dimensions

Figure 1: Figures showing dilated convolutions in one and two dimensions. Red squares represent the portion of that data that is being convolved with the parameters. Grey squares are portions of the data that are not operated on. Setting the window striding to 1 ensures that each sample will be considered, without the computational cost of doing a global convolution. This has the added benefit of mimicking the tendency of ordered data to depend more on samples in its immediate neighborhood than data very far away from itself.

Here its important to note that the receptive field size of each layer gives the window size that each sample has access to. Larger receptive fields will be able to capture more past information than a smaller one. The WaveNet architecture in [1] uses a uniform stack of dilations that double up from 1 to 512:

$$1, 2, 4, 8, 16, \quad ... \quad 512$$

Using a 16kHz sample rate and n layers, this gives a receptive field size in seconds of:

$$\frac{n * 1024}{16000} = n(0.065)$$

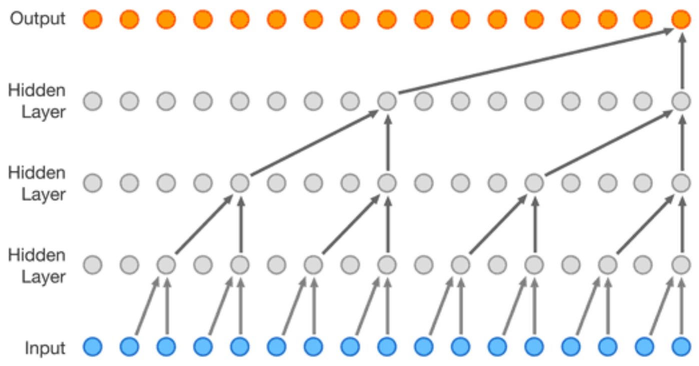


Figure 2: A Stack of Dilated Causal Convolutions [1]

Heuristically we can see that we would need to stack many of these layers before each sample could have a large enough data pool to properly learn to model the input. In order to do this we would need a larger computing infrastructure than we had access to. Instead we generalized and made up for using less layers by creating much larger dilation stacks. In our models, we used fewer layers but increased the dilation limit from 512 to 16384:

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, \quad ... \quad 16384$$

Less layers means that the network spends less time doing local computation, because there aren't as many operations involving dilation of $1, 2, 4...$ etc. But longer dilation stacks allow us to see further back with each filter, allowing the network to learn potentially longer range dependencies than the original proposed architecture.

2.3 Gated Residual Blocks

Very deep network architectures often result in unresponsive neurons as early layers suffer the vanishing gradient problem. Ideally, if any layer fails to transform the input beyond the representation it received, then it should output an identity mapping to preserve the input through to the next layer. In practice though, very deep networks have a hard time with linear mappings and more often the gradient just dies. This results in both poor convergence time and inference capability. In order to allow our network to train with a large amount of parameters, we use both gating and residual blocks in our network. Gating has been shown to vastly improve gradient propagation, and keep it from either vanishing or exploding by squashing input signals with Sigmoid or $\tanh$ units [6] [4]. It has also been shown that a residual function $x \to F(x) - x$ is much easier to learn than a direct mapping from $x \to F(x)$ in deep networks. We use these gated residual blocks at each layer in our network.

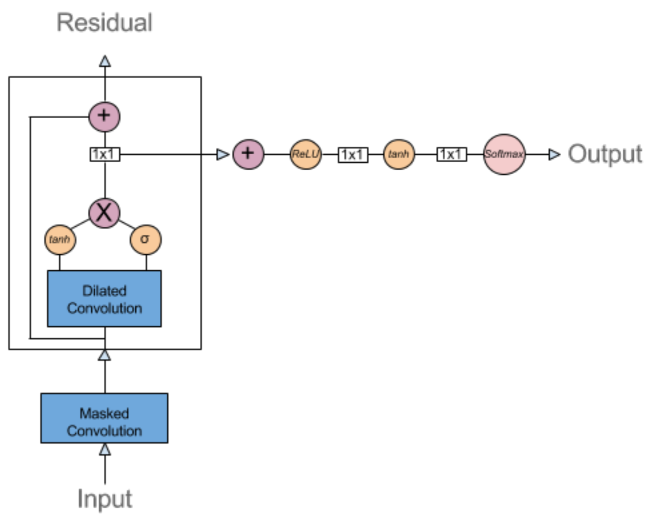


Figure 3: A single layer, containing a gated residual block, with the output stream for the network.

We use the same gating function as PixelCNN [4]:

$$Q(x) = \tanh(W * X) \odot \sigma(W * X)$$

Where $\odot$ represents the Hadamard product, W is the parameter tensor, and X is the output of the dilated convolution. We preserve X as the notation for the input because we retain the same dimensionality through the convolutional steps. Again, both gating and residual connections are efforts to aid the propagation of the gradient through a deep network. Because of these efforts, we found that the additional ReLU units in the WaveNet model were unnecessary. We omitted these ReLu units or replaced them with $\tanh$ units. In particular, we feel the scaled $\tanh$ instead of a ReLU at the output improved performance. A scaled $\tanh$ allowed us to limit the possible output values. Limiting excessive output values was vital, since generated noise (high frequency data) was fed back into the network and tended to cause the model to diverge into generating pure noise.

3 Classification

Here all the details relating to the training of the models is discussed. This includes implementation details as well as metrics used to determine design choices about the models' architectures.

3.1 Training

We trained our models using mainly RMSprop, which we found to converge much better than standard SGD. Since our models would train for many days before converging, it was necessary to use an adaptive optimizer rather than a simple annealing scheme for decaying the learning rate. Adam was also considered as a possible solver, but RMSprop tended to converge faster and more reliably in our experiments. Loss was calculated by taking the SSE with respect to the next incoming sample. The output of the network attempts to model the following input sample as close as it can. The loss was calculated from the distance between the generated sample and the proceeding input sample. A model with 0 loss would perfectly reproduce the input. We trained the network on a dataset of solo piano music that we constructed from YouTube videos; we scraped 20 hours of video and extracted the audio files to form the dataset. Each file was segmented into 10 second chunks of audio for input into the network. Most importantly, the data was not shuffled in any way since it was imperative that order be preserved. Even if the receptive frame of the dilated convolutions could not capture data beyond the local segment, ordering affected how the network converged. Oscillations in the loss occurred when uptempo audio was shuffled into lower tempo audio. The resulting audio was fed as a raw waveform into the network. Given our loss scheme, we were able to easily test our model on a validation set by seeding the generator with a validation file and measuring the SSE between the two.

3.2 Implementation

We implemented our models using Google's Tensorflow framework [9], which allowed us to easily visualize the computational graph and its outputs. Tensorflow was also preferred because it allowed us to easily offload computations onto a GPU for a 10x - 100x training speed increase. We used a NVIDIA GTX 970 for our smaller models, and we upgraded to a newer GTX 1070 for larger networks with more dilation layers. With large convolutional models, training on a CPU is unfeasible due to the low number of cores that can run in parallel at any time. Convolution kernels have a low computational cost, but need to be done hundreds of thousands of times. GPUs have many more cores than any CPU, and therefore can speed up training of large models from months to days.

3.3 Heuristics

Because of the extensive training time, hyperparameter search presented a problem. We were unable to do exhaustive cross-validation for the optimal hyperparameters, so we instead developed heuristics to streamline the process of testing.

- Use an adaptive solver

  Learning rate is always a problem when manually tuned, and usually requires many runs and warm starts to get right. Instead we opted for an adaptive learning rate like RMSprop or Adam as discussed above.

- Craft a homogeneous dataset

  We found that training was near impossible if the dataset is not somewhat homogeneous. Experiments with classical music failed because there was too much data encoded in each sample. With a small receptive window, classical music sounds like noise to model and person alike. Instead restricting the data to a single instrument with a similar style of expression proved to be the most responsive to learning.

- Use less, longer dilation layers

  The depth of the dilation stack is what enables the models ability to characterize the input, but also adds enormous complexity to achieve a usable receptive field. If compute power is limited, instead opt for larger dilations and fewer layers. 5 layers of dilations 1..4096 creates a receptive field of nearly 3 seconds. The same receptive field using 1...512 stacks would require 45 layers, which is impossible to fit on a consumer GPU.

4 Experiments

We trained the model on a few different datasets, a collection of music from classical composers, and a collection of solo piano from YouTube.

- **Classical Music** : The classical data set was trained for 2 days and showed little response with regard to the SSE. The loss remained high at 5.5 and dropped less than 1.0 for the duration of the training period. We hypothesize our receptive field size was too small to capture any meaningful features of classical music. The multiple instruments created too much information to be learned with a small model. Attempting again with a larger model and compute power to match should yield positive results
- **Solo Piano Music** : This model was trained for over a week and showed much better response than the classical music dataset. The loss converged to less than 1.5 after just over 5000 iterations, or approximately 14 hours of audio samples.

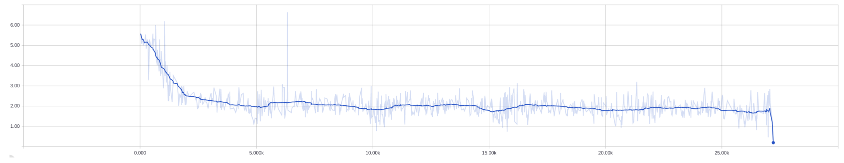
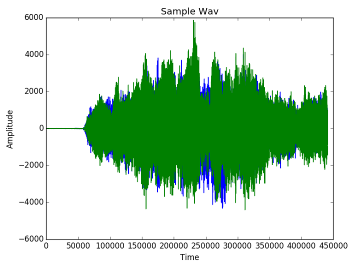
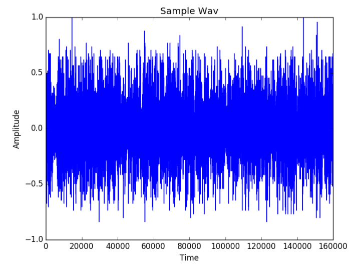


Figure 4: Graph showing the SSE over time of the training data, on the solo piano dataset

The generated samples from the two models differed drastically, reflecting the relative success of each model. Because we lack a sufficient metric to judge the quality of music. We can instead take a look at the corresponding waveforms and see that the solo piano generations more closely match their input.



(a) Waveform of sample from classical dataset

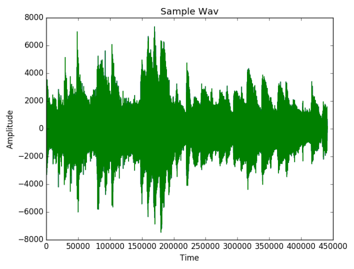


(b) Generated waveform trained on classical dataset

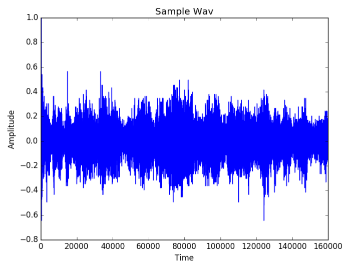
From looking at the ground truth next to the generated counterpart, we can see that the distributions of points do not match at all. There is much more variation in the actual sample and we can almost infer some intention of what the audio would sound like just from the graph. We can also see that the generated waveform would likely sound like uniform noise. There is little variation in the generated data to speak of; we can tell that the model failed to characterize the input.

However, there is a clear difference when comparing the ground truth to the generated samples on the solo piano dataset. The two waveforms more closely resemble each other and we can tell that the generated sample is more than wide band noise. In practice, the samples drawn from our solo piano model sounded much better than any other model we tried. The model learned clear division of notes and noticeable tonal quality.

While we can show that our model learns from raw audio input, we fail to produce samples



(a) Waveform of sample from solo piano dataset



(b) Generated waveform trained on solo piano

as beautiful as provided by DeepMind in [1]. We believe this to be an artifact of our lack of computing resources. We found that the longer the model trained for, the better the resulting sample would sound. This would occur without much change in the loss, leading us to believe that there is little difference numerically with respect to our loss function between a noisy sample and one that sounds pleasing to human ears. Further work on developing a robust loss metric for good ”sounding” audio would allow us to train beyond optimizing for a minimum SSE. An alternative explanation would be that the difference between a model that has completely overfit, and a model that performs well may be extremely small. RMSprop continues to decay the learning rate, and the extremely small steps taken after days of training may make the largest difference in the subjective quality of the generated audio. As an aside, we have posted elsewhere a small library of samples that we have generated. Along with the output, we describe the model used to create/generate the sample.

5 Conclusion

In this work we have provided an implementation of a generative convolutional neural network for raw audio. We use a stack of dilated convolutions along with feedback control in the form of gated residual blocks, to learn a representation of multiple seconds of 16KHz audio. Because this method outputs true conditional joint probabilities, it can be scaled as opposed to GANs which require heavy monitoring by a human to keep track of stability. We also provided heuristics to allow for effective training on low-compute systems, due to the high cost of many small, parameter dense layers.

6 References

[1] Aron Van Den Oor, Karen Simonyan, Nal Kalchbrenner, Sander Dieleman, Oriol Vinyals, Andrew Senior, Heiga Zen, Alex Graves, Koray Kavucuoglu. ”WaveNet: A Generative Model for Raw Audio.” DeepMind. 2016.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. ”Imagenet classification with deep convolutional neural networks.” Advances in neural information processing systems. 2012.

[3] Aron Van Den Oor, Nal Kalchbrenner, Koray Kavukcuoglu. ”Pixel Recurrent Neural Networks.” DeepMind. 2016.

[4] Aron Van Den Oor, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. ”Conditional Image Generation with PixelCNN Decoders.” DeepMind. 2016.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. ”Deep Residual Learning for Image Recognition.” Microsoft Research. 2015.

[6] Alex Graves. ”Generating Sequences with Recurrent Neural Networks.” University of Toronto. 2014.

[7] Alec Radford, Luke Metz, Soumith Chintala. ”Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks.” 2016

[8] Vladlen Koltun, Fisher Yu. ”Multi-Scale Context Aggregation by Dilated Convolutions.” 2015

[9] Martn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. ”TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems.” Google Research. 2015.