Raw Audio Music Generation Using Deep Neural Networks

Fabian Stahl

September 16, 2019

Contents

Abbreviations

CNN	Convolutional Neural Network
DCGAN	Deep Convolutional Generative Adverserial Networks
GAN	Generative Adversarial Network
GRU	Gated Recurrent Unit
KLD	Kullback–Leibler Divergence
LSTM	Long Short-Term Memory
MIDI	Musical Instrument Digital Interface
MLP	Multilayer Perceptrons
NLL	Negative Log-Likelihood
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
VAE	Variational Auto-encoder
VRNN	Variational Recurrent Neural Network

1 Motivation

Music can be found in almost every movie, video game or public location. Its targeted use can change people's mood, encourage buying decisions or add context to accompanying content. However, composing, recording and mixing music is a creative process that takes a lot of time and skill to master. This poses the question if pleasing music can be generated autonomously.

Deep neural networks have been known to solve a wide range problems. This paper will introduce three different network architectures to generate sample based music, show how they can be trained and compare the results.

2 Related Work

In the field of audio generation there is a wide variety of approaches. Most of them work with symbolic music representations, such as ASCII-text, musical scores or spectrogram data. Especially the MIDI format is commonly used, a binary format specifying musical notes. This makes the format very slim and easy to work with. MIDI-based approaches include e.g. Generate Adversarial Neural Networks with convolutions layers [20, 15, 15], Variational Auto-encoders paired with Long Short-term memory cells [18, 19, 9] and Recurrent neural networks with restricted Boltzmann machines [1]. However, musical nuances like tonal colors, scratch and breath noises or intonations are necessary to make most music genres more realistic. While sometimes these small subtle variations are not even perceived consciously, their absence can affect the overall impression. Having MIDI output only, the musical interpretation, (the raw audio sent to the sound port), is up to an external MIDI synthesizer that cannot produce these nuances.

There are commercial music streaming services that claim to provide generated music that fits the customer's mood. The most prominent of them being brain.fm [10] and [11]. However, the brain.fm algorithm only learns certain tasks, such as arranging motifs over long timescales, while melodies, instrument choices and chord progressions are pre-composed by humans. Mubert also uses handcrafted single instrument MIDI loops, but generates some

MIDI layers using decision trees, random forests and musical analysis. The services music is limited to electronic genres only, which narrows musical variety. While the application developers like to emphasize the machine learning aspect, their algorithms are not fully autonomous. The two biggest reasons for that could be the high complexity of generating music sample by sample, especially with real time constraints, as well as the ineptitude of the synthesized music for commercial use.

3 SampleRNN

Recurrent Neural Networks (RNNs) are used to process sequences of data. Output data is not only based upon input data, but also on a hidden state vector, that encodes previous input. RNNs consist of arrays of small memory units. While the original *vanilla cells* were prone to the vanishing gradient problem, more advanced structures like Long Short-Term Memory cells (LSTMs) and Gated Recurrent Units (GRUs) are able to learn to keep important information and forget redundant or irrelevant previous input.

In audio data correlation can exist between neighbouring samples as well as between samples that are more than thousand frames apart. Mehri et al. states that common RNNs don't scale very good with a very high temporal resolution and proposes to use multiple hierarchical stacked RNNs with different temporal resolutions to capture all audio features [14]. This model is called *SampleRNN*. Originally tested only for piano music, [21] and [2] showed, that SampleRNN is especially good to generate loud music genres, like Metal and Dark Ambient. This makes the SampleRNN a promising candidate for music generation.

The implementation used in this work was built on top of a project called *Deepsound-project* [16]. However, in a few cases this version differs from the original proposal. The main differences to the original implementation include

- works with an arbitrary number of frame-level tiers
- upsampling layers prior to RNN cells

- 1D convolutions with stride 1 instead of fully-connected layers (equivalent operations, but better performance)

3.1 Architecture

The (Deepsound-project) SampleRNN consists of multiple hierarchically stacked *frame-level modules* and a single *sample-level module*. Each frame-level module is an RNN that generates a conditioning vector for the next lower tier based on a state vector that summarizes previous input. The same audio data is passed to all Modules, but while the number of chunks increases with the lower tiers, the chunks itself become smaller, i. e. the clock rate gets faster.

At the bottom of the model is a single *sample-level module*. It puts the combined conditioning vector from above into consideration as well as the raw audio samples to generate a probability distribution for each chunk.

3.2 Training

To allow an efficient training of the model, truncated backpropagation through time (TBTT) is used. Since it is computational very expensive to propagate gradients back after a full song, training on shorter sub-sequences of 8 seconds is much more feasible. The initial state vectors of all GRUs was learned.

Figure 1 shows a training cycle of a SampleRNN trained with the following parameters:

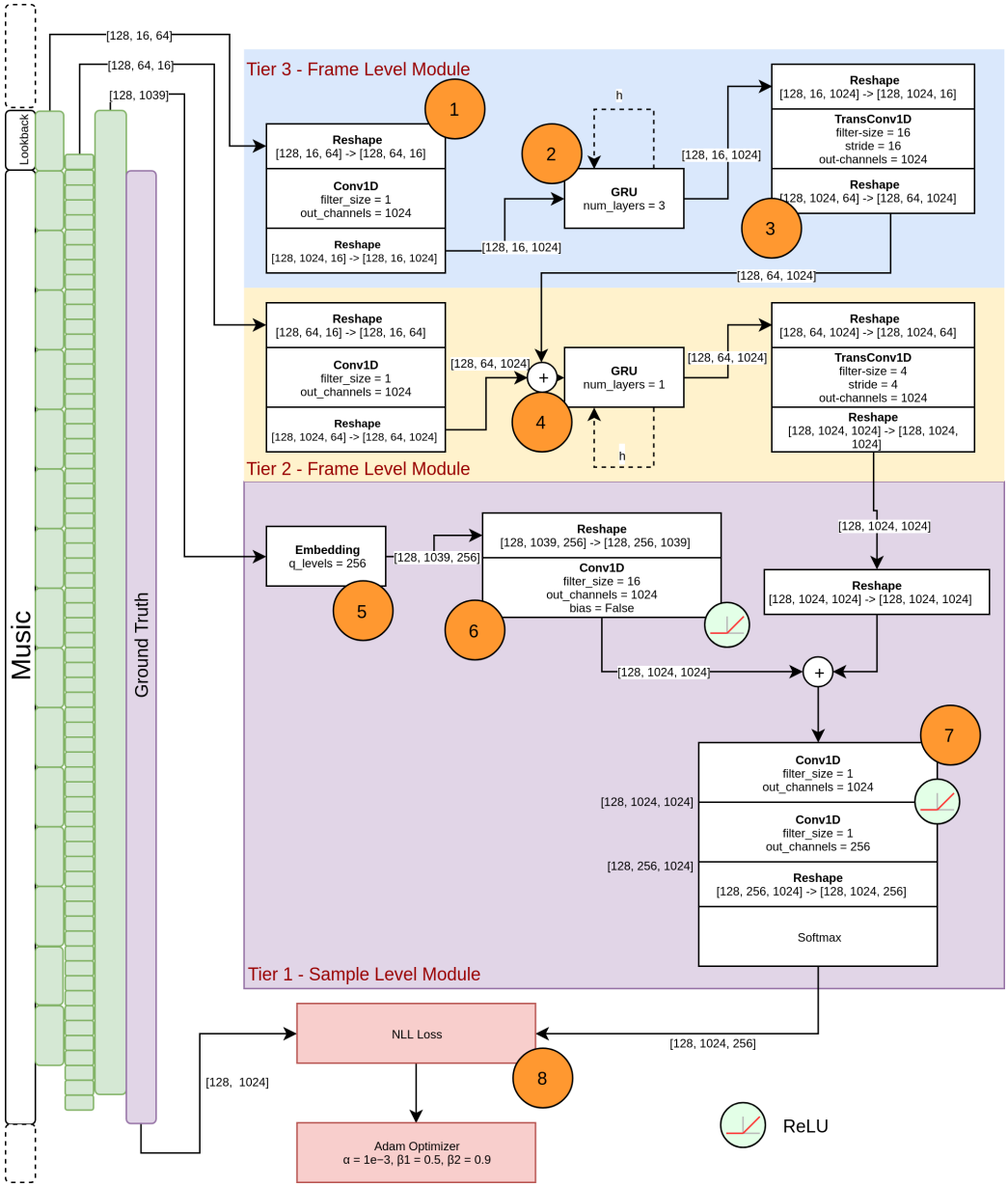

Figure 1: The SampleRNN architecture with the best output results consists of two stacked frame-level modules and a sample-level module. All tiers receive the same input but depending on the tier it is chunked differently. The output is a q-way Softmax which means for each of the 128 batch items and for each of the 1024 chunks a probabilities for 256 output values are assigned.

variable name	value	description
frame sizes	16, 4	numbers of frame-level chunks per upper tier chunk
sequence length	1024	how many samples to include in each training step
quantization levels	256	number of bins in quantization of audio samples
num rnns	3	number of stacked GRUs per frame-level
dim	1024	dimension of the state vectors h, neurons in sample-level
learning rate	0.001	
batch size	128	
sample rate	16000	

First the upper frame-level module chunks the input data into 16 chunks, each containing 64 samples. Using a 1D convolution with stride 1 each 64 dimensional sample chunk is expanded to a 1024 dimensional sample chunk (1). These chunks are fed into three stacked GRUs, that predict the follow up sequences based upon a state vector h_t^1 (2). To be able to pass the condition vectors to the next lower tier with a faster clock rate, a transposed 1D convolution helps to upsample 16 to 64 chunks (3). The middle tier repeats step 1, but uses a smaller clock rate (64 chunks, each with 16 samples) and adds the expanded result to the conditioning vector from above (4). The signal than passes the middle tier GRU and upsampling layers. The sample-level module first quantizes the whole input sequence to 256 bins and maps them to a real-values vector embedding (5). This signal is passed through a 1D convolution to match the conditioning signal from the middle tier (6). The combined signal passes a Multilayer Perceptron (expressed with 1D convolutions) as well as a Softmax layer to calculte 256 probability vaues for each batch item for each chunk (7). According to [14] a discrete output distribution using a q-way Softmax yields better results than a continual distribution using for example Gaussian Mixture Models. Finally the output signal is compared to the ground truth data and the negative log likelihood (NLL) is calculated. Based on this score the Adam optimizer adjusts the models parameters (8).

SampleRNN experiments were stopped after no more validation progress was achieved. On a NVIDIA RTX2080 GPU, this took about 35 hours and almost the full 8Gb RAM.

3.3 Generation

At generation time the sample-level module is run repeatedly, where each sample is fed back into the model. Conditioning vectors from the frame-evel modules are refreshed using a lower clock rate if enough samples are generated to feed a new non-overlapping input chunk to the RNNs.

4 VRNN

4.1 Idea

Another approach to generate sound data is the *Variational Recurrent Neural Network (VRNN)*, proposed 2016 by Chung et al. [3]. The architecture was trained on the tasks of sound generation as well as handwriting. However, experiments for the first only included training with speech datasets and non-linguistic human-made sounds like coughing and screaming. This poses the question if the VRNN can be used to generate music as well.

Chung et al. states that there is an issue in the way that RNNs model output variety, since it is often the only source of true randomness to draw values from an output probability density. Data that is both highly variable and highly structured can be hard to model this way. On the one hand small input variations should lead to potentially very large variations in the hidden state vector h_t, on the other hand, different high level structures need to be mapped to distinct state vectors as well. This forces the network to compromise between the encoding of low-level and high-level features. The main idea of the VRNN is to extend a recurrent neural network with latent random variables, therefore improving the representational power of the model. Such high-level variables are commonly used in Variable Auto-encoders (VAEs) to model multimodal conditional distributions, approximating real world data variety [12]. Chung et al. proposes to extend the VEA model by inducing temporal dependencies using the hidden state vector of a RNN across neighboring timesteps.

4.2 Training

Figure 2 shows all necessary steps for a training cycle. First the raw audio data is passed through φ^x to extract features from x_t (1). According to [3] φ^z as well as φ^x are crucial to learn complex sequences. Since the Kullback–Leibler divergence (KLD) is used to make the outputs of φ^{prior} and φ^{enc} as similar as possible, the prior needs to be included in the model. It handles the state vector h_t and generates both μ^{prior} and σ^{prior}(2). The encoder φ^{enc} is used during training only. It takes φ^x and the state vector h_t and calculates μ^{enc} and σ^{enc} (3). Now z is drawn from $\mathcal{N}(\mu^{enc}, \sigma^{enc})$ (4). Another network φ^z is used to extract features from z (5). The decoder is used to replicate the input data x. It therefore takes the concatenated φ^z and h_t into consideration (6). Also, z and h are passed to the RNN to determine a new state vector h_{t+1} (7). Lastly the overall loss is calculated (8). It is composed of the KLD from φ^{prior} and φ^{dec} to make them similar, as well as a reconstruction loss that rates the networks ability to reconstruct input data. Using probabilistic terms $q(\mathbf{z}_t|\mathbf{x}_{\leq t}, \mathbf{z}_{<t})$ for the encoder distribution, $p(\mathbf{x}_t|\mathbf{z}_{\leq t}, \mathbf{x}_{<t})$ for the decoder distribution and $p(\mathbf{z}_t|\mathbf{x}_{<t}, \mathbf{z}_{<t})$ for the prior distribution the loss is calculated as follows:

$$E_{\mathbf{z}\leq T|\mathbf{x}\leq T} = \left[\sum_{t=1}^{T} -(KLD(q(\mathbf{z}_t|\mathbf{x}_{\leq t}, \mathbf{z}_{<t})||p(\mathbf{z}_t|\mathbf{x}_{<t}, \mathbf{z}_{<t})) + log(p(\mathbf{x}_t|\mathbf{z}_{\leq t}, \mathbf{x}_{<t}))\right]$$

The KLD between the two distributions Q and P with respect to Q is a measure of similarity between two distributions. During training the prior is conditioned by the encoder, respectively the approximate posterior, to be yield similar output. The property is forced by minimizing the KLD, which is equivalent to maximizing the lower variational bound [12]. The reconstruction error is the Gaussian negative log-likelihood and forces the network to learn to reconstruct the input data during training. In this work Gaussian distribution models are used to model real world variability. However, in theory the model can be trained using different probability models like the Gaussian Mixture Model (GMM). All input samples are normalized using the global dataset mean and standard deviation.

For music experiments the following parameters proved to be most effective

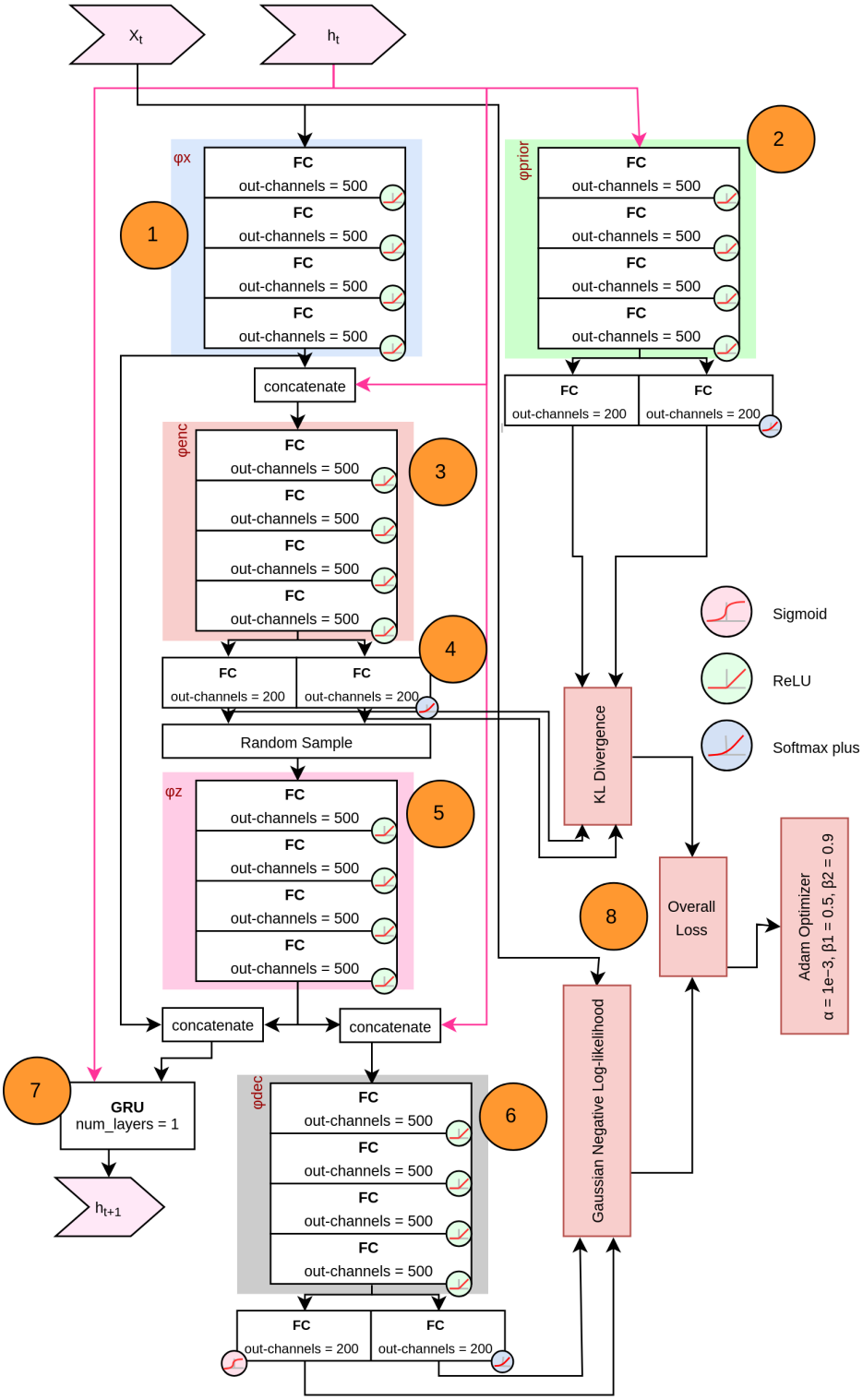

Figure 2: The VRNN consists of all components from conventional Variational Auto-encoder but is paired with recurrent layers to allow learning from sequential data using a single state vector h.

variable name	value	description
frame length	200	numbers of samples per input chunk
num rnns	1	number of stacked GRUs per Frame-Level
clip	0.1	clips gradient to this length to avoid gradient explosion
dim ϕ^x, ϕ^{dec}	600	number of hidden units
dim ϕ^z, ϕ^{enc}, ϕ^{prior}	500	number of hidden units
dim h	2000	number of hidden units in the state vector h
learning rate	0.0003	
batch size	128	
sample rate	16000	

Since the original source code uses the Theano ML library, I started from a public PyTorch implementation [6] for generating handrwritten digits and expanded the models architecture according to the original description used for the Blizzard experiment.

4.3 Generation

Conditioned on the input vector h_t, φ^{prior} is used to calculate μ^{prior} and σ^{prior} values so that the latent variable z can be drawn from $\mathcal{N}(\mu^{prior}, \sigma^{prior})$. To extract features from z the signal passes φ_z. Finally φ^{dec} is used to decode z. Since the decoder is trained with Gaussian KLD loss, the decoder yields μ^{dec} and σ^{dec}. Output values can be drawn from $\mathcal{N}(\mu^{dec}, \sigma^{dec})$. However, in the original implementation μ^{dec} is used as output.

To produce sound of more than 200 samples, h_t needs to be adjusted and fed back into the model. In order to do that x is drawn from $\mathcal{N}(\mu^{dec}, \sigma^{dec})$ and passed through the feature extractor φ^x. Finally the encoded x and z are passed to the RNN and h_{t+1} can be calculated.

5 WaveGAN

In 2014 Goodfellow et al. introduced a new approach to train networks called Generative Adversarial Networks (GANs) [7]. Such networks consist

of a generator and a discriminator sub-network. While the discriminator tries to distinguish between real and generated data, it is the generators task to fool the discriminator into thinking its output is real. Both networks are trained simultaneously and try to outperform each other.

While GANs have been used widely for tasks related to images, there are only few approaches using audio data. Donahue et al. presented two GAN approaches to generate sound data called *WaveGAN* and *SpecGAN* [4]. Both models are adaptions of a model class called Deep Convolutional Generative Adverserial Networks or DCGANs, first proposed by Radford et al. [17]. The SpecGAN works with spectrogram images that are processed by the network in a similar way to normal images. However, the final conversion from image to sound data is known to be problematic, since a lot of noise is induced. The second architecture, the WaveGAN, is a flattened adaption of the original DCGAN model using 1D convolutions on raw audio samples. Both models have the same number of parameters and numerical operations. However, neither strategy has been tested with music data. Since SpecGANs lossy conversion poses difficulties beyond the actual AI model, only the second approach will be presented here.

Unfortunately the original model description is only capable to generate about a second of audio. By adding one more transposed convolution layer to both the encoder and the decoder, and increasing the number of intermediate output channels, the models generating capacity can be increased to about 4 seconds, possibly enabling whole musical motifs. This expanded model can be found on the authors GIT repository [5].

5.1 Architecture

See Figure 3 for a visualization of the graph with parameters. The generator network is trained to map a low-dimensional latent vector z to a high-dimensional data output. First the network applies a fully connected layer with $512d$ output nodes to z, where d is used to adjust the models size. The output vector is then reshaped, since an additional dimension is needed for the sound channel. The signal then passes 5 transposed 1D convolution layers with stride 4 and ReLU activation functions in between as well as a

Tanh function at the end.

The decoder is basically a mirrored generator with a few exceptions. Instead of ReLUs, Leaky ReLUs with $\alpha = 0.2$ are used in between convolution layers. Also a new kind of layer, called Phase-Shuffle, is introduced here. Since the generators transposed convolution is known to produce artifacts, the discriminator may learn to find generated samples based on these artifacts instead of conducting a broad content analysis. Shifting the vector randomly up to n times in a circle can prevent this learning behavior.

5.2 Training

Training GANs can be seen as a two player minimax game. While the generator G is trained to minimize the following value function, the discriminator D_w is trained to maximize it.

$$V_{WGAN}(D_w, G) = E_{x \sim P_x}[D_w(x)] - E_{z \sim P_z}[D_w(G(z))]$$

The network is trained using the Wasserstein-GAN Gradient Penalty, or short *WGAN-GP*, strategy. That means, that instead of predicting probabilities, the discriminator is trained to assist computing the wasserstein or earth movers distance between the distribution of generated data and distribution of real examples. However, minimizing this measure is only possible if the discriminator function is 1-Lipschitz, which can be done by enforcing a gradient penalty. For more details on the WGAN-GP strategy see [8].

In most WaveGAN experiments training was stopped in between 40.000 and 50.000 steps since no more progress was achieved. On a NVIDIA RTX2080 GPU, this took about a day and 7 of 8Gb RAM. All other parameter choices are similar to [4]. Adam Optimizer with $learningrate = 1e-4$, $\beta^1 = 0.5$ and $\beta^2 = 0.9$ was used for both discriminator and generator. The model was trained using both a batch size b and a scale factor d of 64.

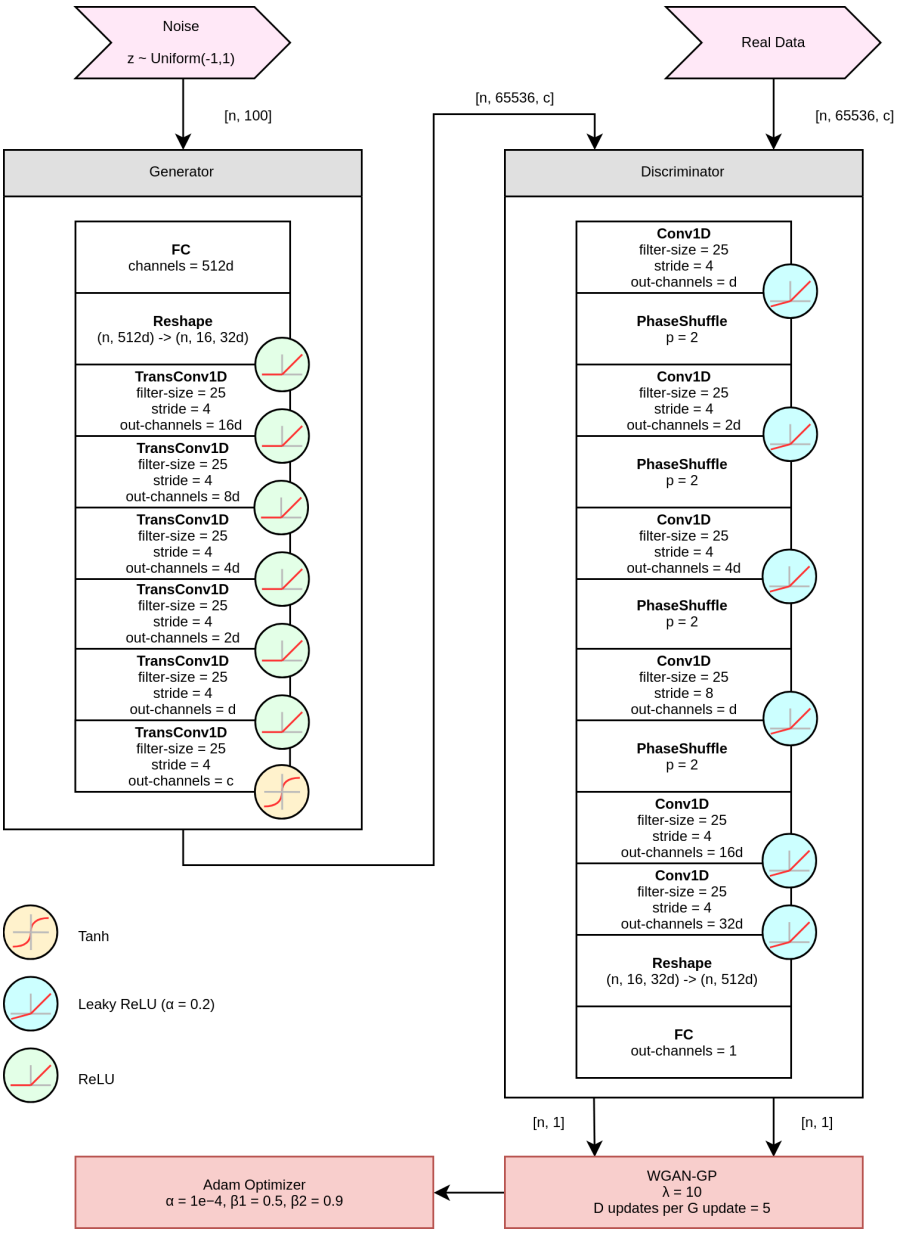

Figure 3: While the generator tries to produce realistic data using transposed convolution layers based upon a latent noise vector z, the discriminator tries to distinguish between generated and real data. The parameter d can be used to adjust the models size. n is the batch size and c is the number of output channels.

6 Experiments

Different music genres come with different sets of features. This challenges the networks to solve very diverse problems. For example a task for a network learning to produce Hip-Hop might be to arrange uniform beat elements with consistent timing to produce a rhythm section, since drums often are the only instrument in Hip-Hop. Piano music on the other hand is often very expressive and has no constant timing. Instead of a fixed timing, it is much more important for the network to learn arranging combinations of harmonizing frequencies to generate plausible motifs.

To cover a lot of musical variety SampleRNN, VRNN and WaveGAN are trained on

- Piano music (Felix Mendelssohn: Lieder ohne Worte, about 2 hours)
- Instrumental Metal (Intervals: The Way Forward + A Voice Within (Instrumental Version) + In Time, 1 hour and 53 minutes)
- Hip-Hop (Cypress Hill: Cypress Hill, 46 minutes)

For SampleRNN the music was split into 8 second, for VRNN into 0.5 second and for WaveGAN into 4 second chunks. It should be mentioned that the music format used in all experiments is mp3. Among other tricks to reduce its file size, this format relies on psychoacoustic phenomenas. For example, frequencies that cannot be heard by the human ear or are overshadowed by previous louder sound are cut out. However, these features might be important for AI models to learn music generation. Future work might investigate how training with raw uncompressed music data influences the models accuracy.

Please listen to the provided audio examples of all experiments. They can be found on the repository page **sound-results.html**.

7 Evaluation

SampleRNN
Despite its relative complicated architecture, the SampleRNN learned to produce first musical features like drum hits, scratches and screams after only about 5 minutes of training. It takes much more training to reduce noise and generate more complex features like guitar riffs. In my experiments SampleRNN was able to produce the most clean sound compared to the other models. However, sample quality varies strongly even after days of training. With the Hip-Hop dataset the model even completely regressed to produce random sounds, even though it learned to produce recognizable voices at training start (as the only contestant). These quality variations were also described by [2] who advised to restart training if output samples during training remain bad. Occasional loss deteriorations can also be seen in both trainings and validation losses (see Figure 4). Future work should investigate how to avoid this unpredictable behavior.

Another drawback of the SampleRNN is the long music generation time since every generation cycle outputs a single sample that needs to be fed back into the model. With the architecture described above it takes about half an hour to generate 5 minutes of audio. However, assuming that there is enough memory, increasing generation batch size does not affect the overall generation time since it can be parallelized.

VRNN
The VRNN model proved to be the worse candidate in terms of output quality and training difficulty. Compared to the SampleRNN, that generates assignable instrument sounds after a few epochs, the VRNN training progress is very slow, yielding sawtooh-like waves most of the time. Also exploding gradients and huge loss values cause the training to crash despite of gradient clipping with a threshold of 0.1. Despite input data normalization this seems to depend on the dataset used. While every attempt to train the model with the full Metal dataset failed because of huge gradients, using a single Metal album the loss converged (The provided example output was generated using this model). The Hip-Hop dataset caused the training to fail about 70% of the time and the Piano dataset score converged almost always. This behavior suggests that the models learning success depends to big parts on the input

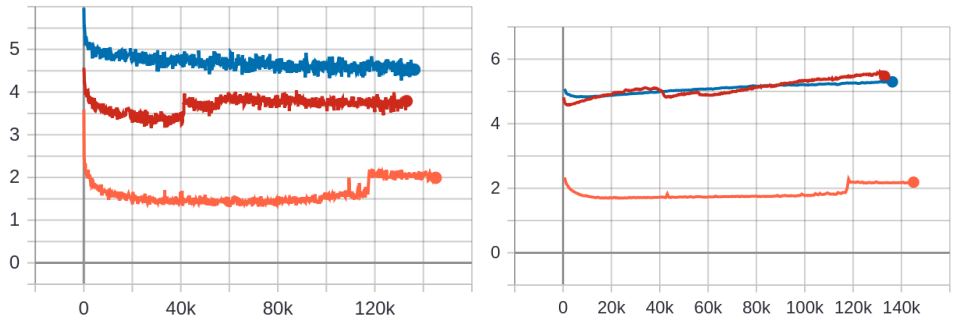

Figure 4: Occasional fluctuations can be seen in the SampleRNN trainig loss (left) as well as in the validation loss (right). There seems to be a point where the Piano dataset (orange, 120k) and the Hip-Hop dataset (red, 40k) "unlearn" certain features. Only with the Metal dataset, the model is able to lower trainings loss continuously. Experimenting with Dropout layers in the sample-level module resulted in a non-increasing validation loss, but seemed to compromise sound quality.

data. However, exploding gradients are also caused by small hyper-parameter changes, for example the dimension of h or the number of fully-connected neurons in all layers.

Despite experimenting with different parameter combinations and error functions used by [3] for speech generation, no trained model seemed capable of learning to produce whole motifs. At most very distorted features like drum hits, single notes or scratches are generated. A known problem with Variational Auto-encoders that process images is that they tend to produce blurry due to the nature of the error function [13]. Especially when listening to the Metal and the Piano experiment, one could assume that this happens with music produced by the VRNN as well.

In terms of generation speed, the VRNN is considerably faster than the SampleRNN with about 50 seconds for 5 minutes of audio. This makes the VRNN a potential candidate for real-time applications. The generation can also be parallelized. The trade-off to this is a long training time of more than 2 days on a RTX2080 using 7GB RAM (if the model is converging).

WaveGAN
Despite GANS being often hard to train, WaveGAN has the most straight-

forward architecture out of the three models. Unlike both other auto-regressive networks, where samples have to be fed back into the model, the WaveGAN audio generation is feed-forward only, which makes generation extremely fast with about 20 milliseconds for 4 seconds of audio (25 seconds for 5 minutes of non-contiguous audio). It also is the only architecture that is (out-of-the-box) capable of producing multi-channel sound output. However, the main drawback of the WaveGAN is the models limitation to generate music of more than a couple seconds. Regarding memory and complexity requirements compared to the other models, WaveGAN is very hard to scale.

In all experiments the model did learn to produce music that somewhat showed features from the input data. One can often observe instruments, small riffs and drum hits. However, in all experiments the networks never fully learned to reduce noise. This can clearly be heard in the examples and often even dominates them. Increasing the model and batch size on a GPU with more RAM might help.

8 Conclusion

In order to generate new music, the models first need to learn to extract features from real data. Since sound data can be very large it is a complex task to get compressed information out of the raw samples. I have presented three strategies to tackle this difficulty.

- The SampleRNN uses hierarchical stacked RNNs with different clock rates to detect features on different scales.
- The VRNN uses latent random variables to model real world data variance.
- The WaveGAN learns to replicate real data using the WGAN-GP strategy.

I have showed that all architectures are capable of producing music with features from the input dataset. While the SampleRNN is better for louder music generes like Metal, VRNN and WaveGAN performed better on music

with a single instrument like the piano. However, all approaches have crucial drawbacks in terms of architecture complexity, memory restrictions, consistency during training or generation efficiency. In my experiments no network was able to reliably produce realistic music with all datasets. Some experiments even completely regressed. More research is necessary to improve sound quality as well as training stability.

Sources

[1] Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*, 2012.

[2] CJ Carr and Zack Zukowski. Generating albums with samplernn to imitate metal, rock, and punk bands. *arXiv preprint arXiv:1811.06633*, 2018.

[3] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, and Yoshua Bengio. A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pages 2980–2988, 2015.

[4] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. *arXiv preprint arXiv:1802.04208*, 2018.

[5] Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In *ICLR*, 2019.

[6] emmanuel. Variationalrecurrentneuralnetwork. `https://github.com/emited/VariationalRecurrentNeuralNetwork`, 2017.

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[8] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777, 2017.

[9] Jay A Hennig, Akash Umakantha, and Ryan C Williamson. A classifying variational autoencoder with application to polyphonic music generation. *arXiv preprint arXiv:1711.07050*, 2017.

[10] BrainFM Inc. brain.fm. `https://brain.fm/faq`. Accessed: 2019-06-30.

[11] Mubert Inc. mubert. `https://mubert.com/`. Accessed: 2019-06-30.

[12] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[13] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *arXiv preprint arXiv:1512.09300*, 2015.

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

[15] Olof Mogren. C-rnn-gan: Continuous recurrent neural networks with adversarial training. *arXiv preprint arXiv:1611.09904*, 2016.

[16] Bartosz Michalak Piotr Kozakowski. samplernn-pytorch. `https://github.com/deepsound-project/samplernn-pytorch`, 2017.

[17] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[18] Adam Roberts, Jesse Engel, Colin Raffel, Curtis Hawthorne, and Douglas Eck. A hierarchical latent vector model for learning long-term structure in music. *CoRR*, abs/1803.05428, 2018.

[19] Alexey Tikhonov and Ivan P Yamshchikov. Music generation with variational recurrent autoencoder supported by history. *arXiv preprint arXiv:1705.05458*, 2017.

[20] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. Midinet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847*, 2017.

[21] Zack Zukowski and CJ Carr. Generating black metal and math rock: Beyond bach, beethoven, and beatles. *arXiv preprint arXiv:1811.06639*, 2018.