Digital Audio Generation with Neural Networks

Henrik Brustad

Thesis submitted for the degree of
Master in Robotics and Intelligent Systems
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Digital Audio Generation with Neural Networks

Henrik Brustad

© 2018 Henrik Brustad

Digital Audio Generation with Neural Networks

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Acknowledgements

I would like to thank my adviser for all the knowledge and experience he has given me, as well as the numerous motivational speeches throughout the process of writing this thesis. This would not have been possible if it wasn't for your inspiring work. I would also like to thank the University of Oslo for using their computer hardware and for letting me be a part of a great work environment.

Abstract

In this thesis I explore three different techniques for generating digital audio using neural networks. All three techniques use different network structures and architectures suitable for generating sequential data. Operating at the sample level requires each technique to model dependencies across large time lags in order to generate realistic audio. This is a hard task for even the most sophisticated techniques.

To gain an understanding of how each technique works I have implemented two neural networks of different structures based on the same architecture, as well as familiarized myself with an implementation of a network using an architecture not commonly used to model sequential data.

To compare each technique I have trained each model on a dataset containing a large number of classical piano pieces. Each model is evaluated in terms of the audio quality and musicality of their generated audio.

Results suggest that each model could be used in applications using short amounts of digital audio. It is unclear, however, if these techniques are able to generate arbitrary music with high level structures, while containing the small details necessary to generate realistic sounds.

Contents

1 Introduction **1**
1.1 Sound . . . 2
1.1.1 Characteristics . . . 2
1.2 Digital Audio . . . 4
1.2.1 Sample Rate . . . 5
1.2.2 Sample Resolution . . . 5
1.3 Generating Music using Computers . . . 6
1.3.1 Markov Processes . . . 6
1.3.2 Evolutionary Computing . . . 7
1.3.3 Neural Networks . . . 8
1.4 Contributions . . . 8

2 Neural Networks **11**
2.1 Structure . . . 11
2.2 Training algorithms . . . 13
2.2.1 The Forward Pass . . . 14
2.2.2 Loss function . . . 15
2.2.3 Backpropagation . . . 16
2.3 CNN . . . 17
2.3.1 Convolutions . . . 17
2.3.2 Applications . . . 18
2.4 RNN . . . 18
2.4.1 Structures . . . 19
2.4.2 RNN Architectures . . . 20
2.4.3 LSTM . . . 21
2.5 Musical Applications . . . 23

3 Models **25**
3.1 NaiveRNN . . . 26
3.1.1 Implementation . . . 27
3.2 SampleRNN . . . 29
3.2.1 Modules with Different Clock Rates . . . 30
3.2.2 Training vs Generation Algorithm . . . 35
3.3 WaveNet . . . 37
3.4 Conclusion . . . 39

4 Experiments **41**
4.1 Data 41
4.1.1 Preparing the data 41
4.2 Overfitting 42
4.2.1 Naive RNN 42
4.2.2 SampleRNN 47
4.2.3 WaveNet 48
4.2.4 Overfitting Results and Conclusion 49
4.3 Training on longer sequences 51
4.3.1 Results 55

5 Conclusion **59**
5.1 Model Discussion 60
5.2 Future Work 61
5.2.1 Limitations 61
5.2.2 Applications 63
5.3 Final Remarks 63

A List of Audio Examples **69**

Chapter 1

Introduction

Communication is an important part of every humans life and we rely on it to share our thoughts and emotions with other humans. We communicate with each other not only through advanced languages, but also through music. Considering how important these forms of communication is to us, it is not hard to imagine why speech and music have become popular fields within computer science.

The latest advances in machine learning and artificial neural networks have given rise to techniques that show great potential in speech and music generation. These techniques tackles the problem at its lowest level, they generate digital audio one sample at a time. This allows them to be used in both speech and music generation and have made it possible for computers to generate speech that sounds more human than with previous techniques, as well as music that sounds highly realistic. But what are these techniques?

These techniques are inspired by techniques used in applications such as text and image generation. Because there are many similarities between text and audio generation, as well as image generation, it allows to use these techniques without many modifications. Different architectures and structures have shown great potential in many domains and it is natural to wonder how these techniques would handle the difficult task of generating audio sample by sample.

Natural sound consists of highly complex and irregular information and even the most powerful neural network in the world, the human brain, rely on a sophisticated auditory system that extracts this information and converts it into a simpler format that is easier to process. Of course, converting the sound information into a simpler format makes it easier to understand specific sounds, but it makes it difficult to reproduce them. This is probably why humans have a separate system for generating certain sounds. It it possible to imaging a biological speaker, driven by well coordinated muscles, being able to reproduce sound in the same way as an electrical speaker. If so, the brain might have had to process sound on similar low-level information as audio samples. But the fact that this has not been evolved might suggest that this task is too difficult for even the most powerful neural network.

Considering the difficulty of this task, how practical and useful are

these techniques for generating musical sound? Is a future where neural networks generate expressive and creative music realistic, or do we rely on a system more similar to how humans process and generate sound?

1.1 Sound

Sound is periodic variations in atmospheric pressure [23, p. 33] and we refer to these variations as sound waves. These waves will propagate through the atmosphere, from an initial sound source, and they will eventually arrive at our ears. The human ear has evolved to convert these sound waves into electrical signals which our brain can understand. Inside the ear is the eardrum, a membrane which turns these sound waves into mechanical vibrations. The vibrations are transferred to the inner ear, by a set of tiny bones, where there are fluid-filled chambers with tiny hair receptors and each hair responds to certain frequencies depending on their position within the chamber [23, p. 56]. These hairs are converting the mechanical vibrations into electrical impulses and they are the reason we are able to perceive sound.

1.1.1 Characteristics

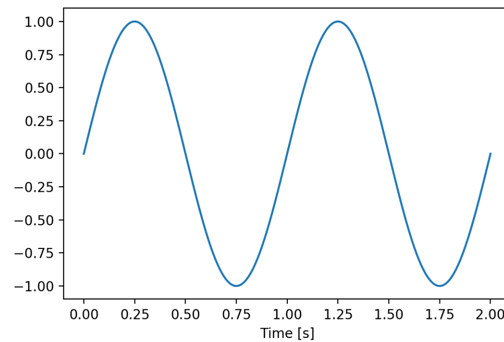
Waveforms are graphical representations of sound waves over time and they allow us to see and understand sound waves in a more intuitive way. All waveforms have fundamental characteristics which allows us to distinguish one waveform from another and I will explain three common characteristics.

Amplitude

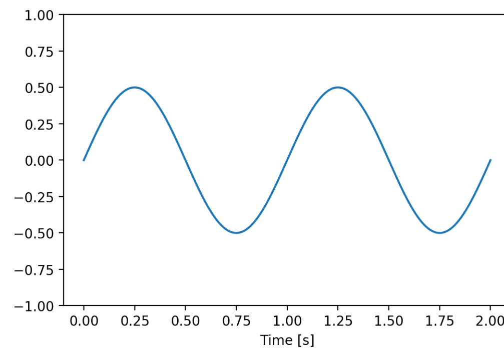
The *amplitude* of a waveform describes the intensity of the sound wave. Large variations in atmospheric pressure results in a high amplitude, which we perceive as the loudness of the sound. There are multiple ways of measuring the amplitude of a waveform, but the most simple way is to measure the distance from the center line to the highest or lowest point on the waveform. This is the measure I have used in figure 1.1a and 1.1b. Another common measure is the *root-mean-square (rms)*, which is a measure that better represents the level perceived by our ears [23, p. 36].

Frequency

All sound waves have a frequency and its frequency tells us how many cycles we have in one second, where a cycle is defined as going from one positive peak to the next positive peak, (see figure 1.2a and 1.2b). The way we perceive frequency is referred to as *pitch* and we often describe sound as being *high-pitch* or *low-pitch*.



(a) A sine waveform with an amplitude of 1



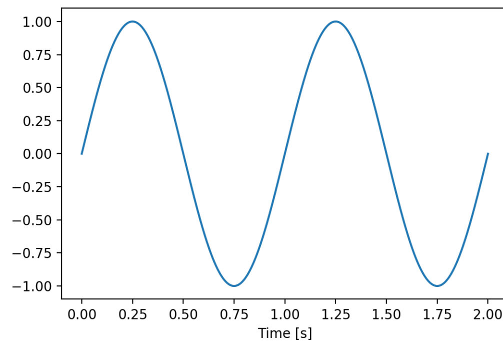
(b) A sine waveform with an amplitude of 0.5

Figure 1.1: Visualizing two waveform with different amplitudes. The zero line represents normal atmospheric pressure. A waveform allows us to see how the atmospheric pressure at a certain position in space changes over time.

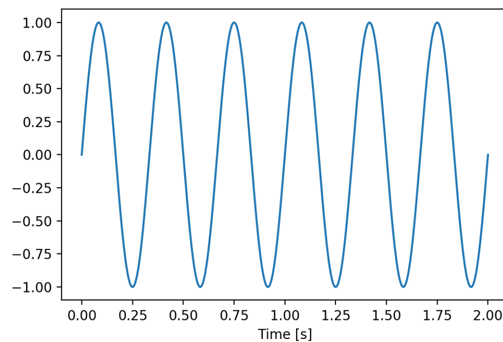
Harmonic Content

Although sine waves are useful for describing the characteristics of sound, they usually never appear in everyday life. Instead, we are surrounded by complex sound waves generated by for example acoustic musical instruments, like the waveform in figure 1.3. These complex sound waves are not only made of one specific frequency, but rather a whole spectrum of frequencies we call *overtones*. These overtones are different from instrument to instrument and are essentially what makes us able to differentiate between musical instruments. The overtones and their relative intensities are called the *timbre* of an musical instrument [23, p. 48].

The term timbre is used in various ways in music. Gounaropoulos [15] defines two concepts of timbre, *gross timbre* and *adjectival timbre*. Gross timbre describes the gross categories of sounds, e.g instrument types, the sound of certain combinations of instruments and so on. Within each gross category there are big differences in the distinctive sound qualities and the changes in those qualities that can be produced. Grey [17] explains this as an indication of some tonal quality of performance on a given instrumental source. These differences is described by the adjectival timbre term.



(a) A sine waveform with a frequency of 1 Hz.



(b) A sine waveform with a frequency of 3 Hz.

Figure 1.2: Visualizing two waveform with different frequencies. The frequency of a waveform tells us how fast the atmospheric pressure at a certain position in space changes.

Timbre is not well understood compared to other aspects of music such as rhythm and pitch. There are several reasons for that, one of them is the lack of theory and notation support [15]. Most of the research done in this field consists of musicians listening to sound and verbally describing what they hear. McAdams [32] used professional musicians, amateur musicians and nonmusicians to map gross categories into a three dimensional timbre space. He used the log-rise time, spectral centroid and and the degree of spectral variation as his three dimensions. His results shows us that different types of instruments occupy different parts of the timbre space.

Even though we have these gross categories which have their unique timbre, there are big differences within each category. These differences is the distinctive sound qualities and the changes in those qualities that can be produced.

1.2 Digital Audio

In the same matter as the human brain, computers rely on converting the sound waves into a different form. A *microphone*, often called *mic*, is transducer that converts sound waves and into electrical signals [23, p. 115]. There exists many different types of mics, but one of the

more common types are the *dynamic mic*. The dynamic mic is using electromagnetic induction to generate an electrical signal. Inside the dynamic mic is a diaphragm, much like the ear drum inside an ear, which will vibrate with the incoming sound waves. Attached to this diaphragm is a coil of electrical wire that are suspended in a magnetic field. When the diaphragm is vibrating, the coil will move up and down within the magnetic field, which will generate an electric current in the electrical wire. The generated electrical signal will look the same as the audio waveform, where the voltage level corresponds to the amplitude of the waveform. An electrical audio signal might look like the waveform in figure 1.3.

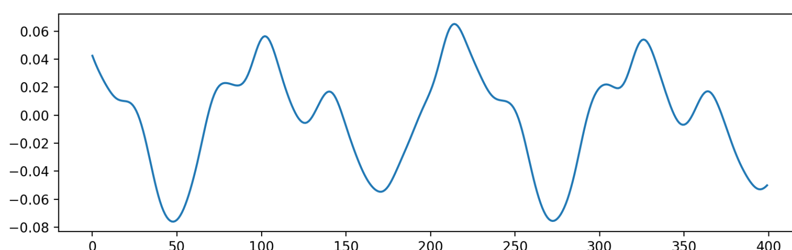


Figure 1.3: Analogue sound wave

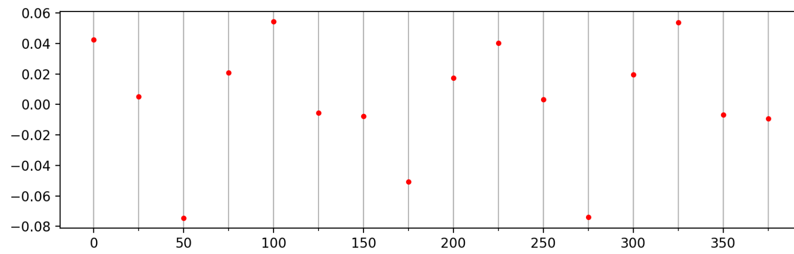
1.2.1 Sample Rate

Taking periodic *samples* of the analogue electrical signal allows us to transform the analogue signal into a sequence of bytes which can be stored in the computer. The *sample rate* is the number of measurements taken of the analogue signal in one second. According to the *Nyquist Theorem* [23, p. 219] the sample rate must be at least twice as high as the highest frequency to be recorded. Because humans can only hear frequencies between 20 Hz and 20 kHz, the standard sample rates for distributing music and speech as digital audio are 44.1 kHz and 48 kHz.

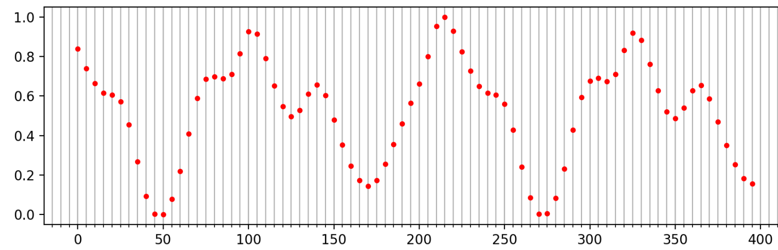
Recording frequencies higher than the one-half of the sample rate can cause unwanted frequencies to appear, which can make the audio sound different when played back. Figure 1.4 shows discrete sampled versions of the waveform shown in figure 1.3 and the effect of using a sample rate which are too low. This makes it is hard to recreate the details of the original sound wave.

1.2.2 Sample Resolution

The accuracy of each sample is determined by the *sample resolution*. Because a binary number only has a finite number of steps, the accuracy will only depend on how many bits used to measure the voltage. *Quantization* is the process of taking a high resolution signal and transforming it into a lower resolution signal, where an analogue signal in theory has an infinite resolution. The most common sample resolutions for digital audio is 16 and 24 bits.



(a) Digital sound wave with a sample rate of 1764 Hz



(b) Digital sound wave with a sample rate of 8820 Hz

Figure 1.4: Recording audio using a small sample rate can make it impossible to recreate the original audio. Increasing the sample rate makes the digital audio signal look more like the original analogue signal. The vertical lines represent each sample.

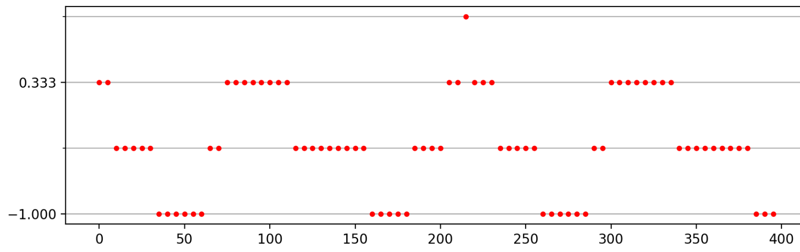
In the same way as using a low sample rate can lead to unwanted frequencies, using a low sample resolution could also lead to artifacts. This effect is visualizes in figure 1.5.

1.3 Generating Music using Computers

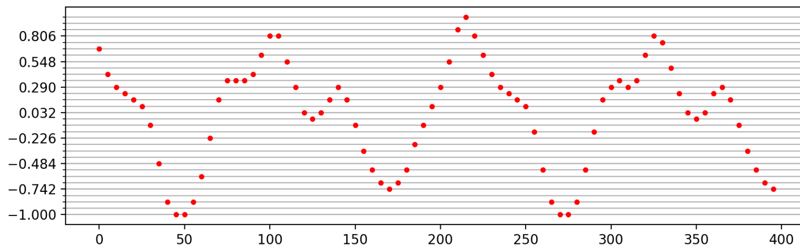
There are many techniques for generating music using computers, everything from hand written algorithms to complicated neural networks. Iannis Xenakis was a pioneer in computer music and the ideas of an automated score compositions was mentioned as early as 1957 [21]. In the following sections I will give an overview over popular techniques used to generate music.

1.3.1 Markov Processes

The *Illiac Suite* was the first music piece composed by a computer using Markov chains and was created by Lejaren Hiller and Leonard Isaacs in 1957 [1]. Markov chains use transition tables to calculate the probability of going from one state to the next. The transition tables could be composed by humans or obtained through a musical source material. Hiller and Baker used the composition *Putnam's Camp* by Charles Ives to create their transition probabilities used to compose the *Computer Cotana* [19] in 1963.



(a) Digital sound wave with 2 bits of resolution



(b) Digital sound wave with 5 bits of resolution

Figure 1.5: Visualizing the importance of using enough bits to measure the audio signal. Using only 2 bits will turn the original signal into a digital signal that is unrecognizable. The horizontal lines represent quantization steps.

While Hiller and Baker used pitch, note duration and velocity to discribe each state, Xenakis used each sample when creating *GENDYN* [21]. His goal was to use the computer throughout the entire compositional process, from creating the melody and the dynamics to synthesizing the sound of each instrument.

1.3.2 Evolutionary Computing

Evolutionary computing uses a variety of techniques and methods which are inspired by natural evolution with the idea of efficiently searching through a vast data space. Each point in the data space represents a solution to a defined problem, which might be finding an optimal design of a walking robot or composing music. The search uses the effectiveness of natural selection, mutation and reproduction, where the idea is to use the best candidates in the population to produce new offspring in hope of getting a better candidate.

There are many examples of music composition using evolutionary algorithms. Horner and Goldberg [10] used a genetic algorithm (GA) with thematic bridging, which is a method of composing music by finding a transformation, a bridge, of an initial musical pattern to a final pattern. The initial and final pattern is defined and it is up to the GA to find the bridge using a set of defined operations. Examples of these operations are adding or deleting elements, or it could be to mutate or exchange elements.

Magnus [29] uses genetic algorithms to evolve waveforms on sample level where each sample represents a single gene within a chromosome. The goal is not only to evolve waveforms that look similar to a real waveform, but also to capture the entire evolution process.

Chan et al. [5] presented an automated genetic algorithm method that determines discrete summation synthesis and hybrid sampling-wavetable synthesis parameters to match any acoustic instrument tone.

1.3.3 Neural Networks

Mozer [34] was one of the first to use a neural network to generate musical melodies. His network, called *CONCERT*, is a recurrent neural network that predicts the next note in a sequence, not only its pitch, but also its duration and harmonic chord accompaniment. He found that the architecture and training did not scale well as the length of the melody grows and as the higher-order structures increases, but this is due to the limitations of regular RNNs which we will discuss in Chapter 2.

Bown and Lexer [4] proposed the use of CTRNNs as an audio synthesis algorithm which made it possible to generate audio at a lower level. CTRNNs are a type of artificial neural networks that uses differential equations to generate an output. Depending on the configurations of the network, they are able to produce oscillations which resemble audio waveforms.

Martin and Torresen [31] used a mixture density RNN to model musical touchscreen performances. The model is connected to an interactive touchscreen music app that allows users to create short musical improvisations. Training the model on a large collection of these performances have enabled them to use the model as an agent for call-and-response style interactions with the users. Given a users call performance, the model was able to generate responses that are related in both movements and rhythm. Human evaluation has shown that a call-and-response interaction with the model has enhanced the user experience.

1.4 Contributions

In this thesis I have explored three techniques for digital audio generation using artificial neural networks. Each technique is using a different architecture commonly used to model sequential data. To gain a better understanding of what these techniques are and the differences between them, I have implemented two neural networks that are using recurrent units to store information about previous events and familiarized myself with the implementation of a convolutional neural network that is using causal convolutions. Each neural network is implemented using TensorFlow, which provides high-level functionality that makes some implementations easier than others.

I have implemented two recurrent neural networks (RNNs) that are using two different structures. The NaiveRNN are made using basic RNN

methods where only a single sample is fed to the network at each time step. The limited input size means this network will have to use its recurrent units to model the smallest details in the audio samples while simultaneously collecting information about longer temporal structures. SampleRNN [33] is using a more complex structure where the network is made of modules operating at different clock rates. This allows the network to separate different levels of temporal structures into each module, making it easier for the network to model dependencies across longer time lags.

WaveNet [38] is a convolutional neural network (CNN) that is using dilated causal convolutions to model each audio sample. This allows the network to capture dependencies thousands of samples apart while only using a few layers. Using convolutional layers allow WaveNet to utilize the parallel computational powers that the GPU is capable of. This means that WaveNet is very efficient to train compared to the two RNNs, which have a more sequential order of computations. However, while WaveNet is limited to model dependencies within its receptive field, the two RNNs are theoretically capable of modeling dependencies across arbitrary lengths [16].

In order to understand how practical these techniques are for generating musical sound I have conducted a series of experiments where I trained each model on a collection of classical piano music. Limiting the dataset to a single instrument makes the audio easier to model and will help decrease the training time. Each model is unconditional, meaning they will generate arbitrary piano music. When comparing the models I have focused on their efficiency, how fast they are able to train, and the quality of their generated audio. The audio quality is based on the musicality, the high-level musical structures such as melody and chord progressions, and timbre, the low-level audio details.

Results show that each model have the potential to generate short amounts of digital audio and could be applied to applications such as instrument sampling or generation of impulse responses. Generating arbitrary piano music was a much more difficult task and only SampleRNN and WaveNet was able to generate audio that started to sound like piano music. SampleRNN generated audio which were short, but contained musical structures such as notes and chords, while WaveNet generated audio which are longer and had a more accurate timbre. Both models could have performed better if trained over a longer period of time, as well as conditioning the input, which are proposed as possible future work.

Chapter 2

Neural Networks

Artificial neural networks are data structures inspired by the human brain. They are made of simple processing units called neurons or nodes, which are linked together in a complex way. Every neuron will produce an activation, which is a function of its inputs, and this activation is sent to other neurons. Each link between two neurons are weighted, which is how the network controls the flow of information through the network. By adjusting these weights we can make the network solve difficult problems [9].

2.1 Structure

There are many different groups of neural networks and each group has different structures. These structures come together by constructing networks to solve specific problems. Each group of networks share similar traits and structure because of the learning algorithm they use. Even though these networks might be different in structure, they are based on the same principles which are inspired by nature. I am going to focus on a big group of networks called feed-forward networks. The name of this group comes from their structure which only allow data flow going in one direction, from the input to the output of the network.

Input Layer

The first part of any neural network is the input to the network. We can think of this being the eyes of our network, capturing the continues flow of information from the outside. The input is represented by nodes and there are usually as many input nodes as there are data points in the input data. For example, a gray scale image of size 28 x 28 pixels would have 784 data points, one for every pixel, and the network would have 784 input nodes. We refer to these input nodes as the input layer of our network, which is one of the traits in these feed-forward networks, we separate nodes into sequential layers.

Output Layer

The last part of a feed-forward network is the output layer and this is where we would get the result from our network. This could be the result of a classification problem where the network would try to give the input data a label or class, or it could be the result of a regression problem where the network would try to find the a continuous function which fits the input data the best. No matter which problem we are trying to solve, the network will generate an output the exact same way, the only difference is how we interpret the output. The output layer can contain any number of nodes and each node is connected to every node in the input layer, see figure 2.1. These connections are weighted which makes it possible to control which input nodes can affect the different output nodes and how much they are affected. This kind of network are referred to as single layer network.

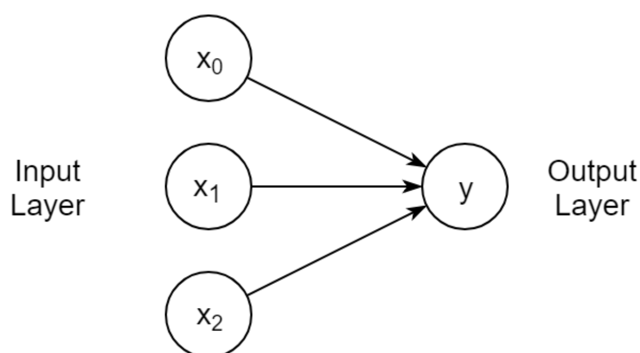


Figure 2.1: A simple neural network with three inputs and one output. To layers connected in this way, every node in one layer is connected to every node in the other layer, is referred to as a fully connected layer. The arrows represent weighted connections between nodes.

Hidden Layers

Single layer networks have been used in a variety of applications involving mapping similar input patterns to similar output patterns [41], which means that there have to be a linear relation between the input pattern and the output pattern. More complicated input patters, however, usually have non-linear relations with the output and this is when single layer networks start to struggle. A good example of such a problem is the XOR problem, table 2.1, which is a simple problem, but it is unsolvable for single layer networks [41]. There are no single linear function that allow us to classify the two inputs correctly.

There are a couple of solutions to this problem. One way is to add a bias node, which is very common in all types of neural networks. A bias node is a node which acts as an input with a constant value of 1.0. It has its own weight associated with it and it does not have any input connections. Nodes b_0 and b_1 in figure 2.3 are bias nodes. The bias node adds a third dimension to the XOR-input which makes it possible to create a linear relation to the output.

Input 1	Input 2	Output
0	0	0
1	0	1
0	1	1
1	1	0

Table 2.1: The XOR problem. The non-linear relation between the inputs and output makes this problem unsolvable for single layer networks.

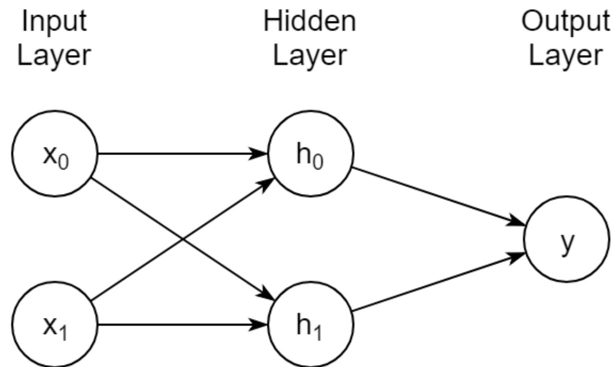


Figure 2.2: A simple multilayer network with two input node, two hidden nodes and a output node. The hidden nodes creates internal representations that allow the network to classify nonlinear patterns.

The second method is to add hidden nodes, which are nodes in between the input and output nodes as shown in figure 2.2. Hidden nodes make these networks very powerful because of the ability to make internal representations that allow the network to make the necessary mappings between its input and its output. This makes multilayer networks able to solve more complicated problems than single layer networks, although they are harder to train [9].

2.2 Training algorithms

Many methods have been developed for training neural networks over the years, each having pros and cons. The learning algorithms are split into two general areas, supervised and unsupervised learning. Although unsupervised learning have performed beyond everyone's expectations in recent years [42], I am going to focus on supervised learning.

The idea behind supervised learning is that we use labeled training data to teach our network the relationship between inputs and desired outputs in hope of it being able to predict unseen data. This is easy with single layer networks, but scientists would struggle for many years to find a general algorithm to train multilayer networks. It wasn't until year 1985 that Hinton [41] developed a more general algorithm based on the already well known delta rule, which I will describe in a later section. We use the same algorithm today, but with additional features that make the algorithm

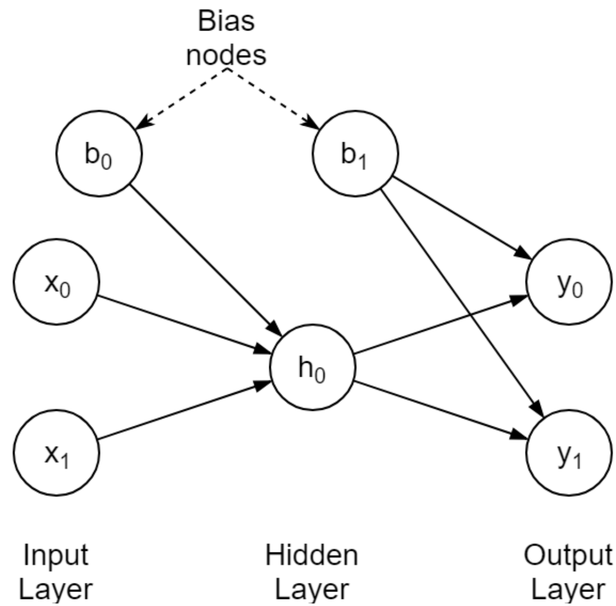


Figure 2.3: Multilayer network with two input nodes, one hidden node and two output nodes. The two bias nodes acts as input to the hidden and output layer. These nodes output a constant value of 1, but using their weights the network can learn to make use of these nodes.

perform better than the original.

2.2.1 The Forward Pass

The first step in any training algorithm is to generate an output. In feed forward networks we calculate the output of each layer sequentially. The input layer will generally output the actual input data, but it is common to process the data before it is fed into the network. Examples of preprocessing are normalizing and centering.

The output of the hidden units are calculated using equation 2.1. It is common to use nonlinear activation functions to achieve the advantages of multilayer networks [9] and it allows the networks to classify nonlinear input patterns. One of the more common activation functions are the sigmoid function.

$$y_j = \sum_{i=0}^{n} x_i W_{ij} + b_j \tag{2.1}$$

$$f_j(y_j) = \frac{1}{1 + e^{-y_j}} \tag{2.2}$$

This activation function has a range of (0, 1). Another common activation function is the *tanh* function. It is similar to the sigmoid function, but it has a range of (-1, 1). The last activation function I am going to mention is the

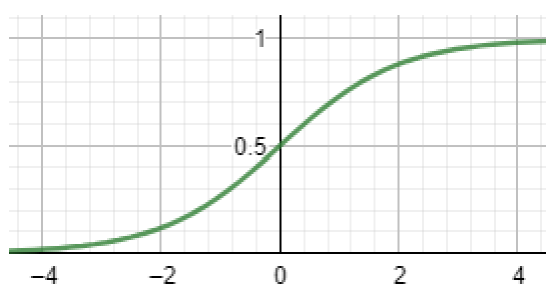


Figure 2.4: Plot of the sigmoid function

most used today. It is called the *Rectified Linear Unit* (ReLU).

$$f_j(y_j) = \begin{cases} 0 & ,y_j \leq 0 \\ y_j & ,y_j > 0 \end{cases} \tag{2.3}$$

The most important thing about activation functions is that they have to be differentiable. This allows the derivative of the neural network to be calculated so that it can be trained by gradient descent.

The output of the output neurons are calculated in the same way as the hidden neurons, except that it is not necessary to use an activation function at this point. The functions we use are determined by the problem itself and will be discussed in the next section.

2.2.2 Loss function

We use output functions to make the output of our network useful in different situations. In the case of classification, there are a couple of useful functions. If we are only interested in two classes, let's say cats and dogs, we can use the sigmoid function, equation 2.2. In this case we would only need one output neuron. We know the sigmoid function has a range from 0 to 1, where an output value below 0.5 could represent cats and a value above 0.5 could represent dogs. If we are interested in more than two classes we can use a function called *softmax*.

$$q_j(y) = \frac{e^{y_j}}{\sum_k e^{y_k}} \tag{2.4}$$

Equation 2.4 is called the softmax function and it calculates the probability distribution of the output, which tells us how likely it is that the input belongs to the different classes.

In order to make our network learn anything we need to measure its performance. Again, there are many ways to measure the performance, but in supervised learning we usually compare the predicted output with the desired output. Two of the most popular loss functions are the squared error and cross entropy. In theory an ANN can be trained equally as well by minimizing both error functions, but in practice cross entropy leads to faster convergence and better results in terms of classification error rates

[12]. This has lead to cross entropy being the favorable loss function in recent years.

$$E = \frac{1}{2} \sum_j (p_j - y_j)^2 \tag{2.5}$$

Equation 2.5 is the squared error loss function where p_j is the target output, y_j is the predicted output and n is the amount of output nodes, i.e. the number of classes. Dividing by 2 makes the derivative easier to calculate.

$$E = -\sum_j p_j log(q_j) \tag{2.6}$$

Equation 2.6 is called the cross entropy loss function where p_j is the desired output and q_j is the predicted probability calculated by the softmax function.

2.2.3 Backpropagation

We can use some simple tricks to make a neural network learn. The only thing we would have to do is to produce some output given a certain input, observe how well the network performs, change one single weight in the network, again produce on output with the same input and see if the network performs better of worse. If it performs better we keep the change, but if it performs worse we would change the weight in the opposite direction. If we do this over and over again, the network would eventually reach a state were it could make decent predictions, but it would take an enormous amount of time to reach this point if the network contains a large number of parameters. This is basically what backpropagation does, only we use mathematics to calculate how we need to change the weights in the network for it to perform better and we change all the weights at once.

Backpropagation was developed in 1985 [41] and is a more general version of the delta rule. The learning algorithm used in the delta rule is based on supervised learning where we compare some produced output with a set of target outputs. If there is no difference in the outputs there will be no learning, but if there is a difference, we can use that difference to update the weights in the network. In a single layer network, the delta rule is defined as:

$$\Delta w_{ij} = \eta(p_j - y_j)x_i = \eta\delta_j x_i \tag{2.7}$$

where p_j is the jth element in the target output, y_j is the jth element in the produced output, x_i is the ith element in the input and $\Delta w_i j$ is the amount of change between input i and output j.

It is worth noticing that this equation is negative proportional with the derivative of the squared error with respect to the weights if we use a linear activation function. This means that when we are using the delta rule, we are actually doing a gradient descent on the squared error because we are

changing the weights in the direction where the gradient is decreasing the most. Calculating the derivative of equation 2.5.

$$-\frac{\delta E_j}{\delta w_{ij}} = \delta_j x_i \tag{2.8}$$

where E_j is the squared error of the jth output node and $\delta_j = p_j - y_j$.

Hinton describes how we can use the chain rule of derivatives to propagate the error back in the network, which means that we can calculate how each weight in the network would affect the loss function. We can use the chain rule to derive the delta rule from the squared error loss function.

$$\frac{\delta E_j}{\delta w_{ij}} = \frac{\delta E_j}{\delta y_j}\frac{\delta y_j}{\delta w_{ij}} \tag{2.9}$$

We can use equation 2.5 to find the derivative of the squared error function with respect to the output of the network.

$$\frac{\delta E_j}{\delta y_j} = -(p_j - y_j) = -\delta_j \tag{2.10}$$

And then we can use equation 2.1 to find the derivative of the output with respect to the weights of the network.

$$\frac{\delta y_j}{\delta w_{ij}} = \frac{\delta}{\delta w_{ij}}(x_i w_{ij} + b_j) = x_i \tag{2.11}$$

Using the derivative of the loss function and propagating it back through the network using the chain rule, we can optimize any neural network to minimize any loss function as long as the loss function and any activation function is differentiable.

2.3 CNN

CNNs are a type of artificial neural networks which are more inspired by the visual cortex in our brain. Hubel and Wiesel [22] measured neural activity in the cat brain and discovered that the visual cortex where made of simple and complex cells. Simple cells were activated by simple features like lines or edges, and the complex cells were connected to multiple simple cells making them activate by more complex shapes. Although CNN architectures come in many variations, these findings lead to the use of convolutional layers in neural networks.

2.3.1 Convolutions

In the same way that simple neurons in the visual cortex react to features in an image, the convolutional layers serve as feature extractors. The input to the layer is convolved with trainable weights, which acts as a filter, and it will generate a feature map where each neuron represents the presence of a feature at a certain position in the image. All the neurons in a feature

map will share the same weights, but we can use multiple filters in a single layer to create many of these feature maps. We can express this as

$$Y_k = f(W_k * x)$$

where Y_k is the output feature map, x is the input, W_k is the filter weights, $*$ represents the convolution operator and f is the activation functions.

By stacking multiple convolutional layers after each other, we see that the neurons respond to more and more abstract features the deeper we go. As these convolutional layers can extract abstract features themselves, CNNs can be directly applied to complex low-level data, such as images or audio data, without developing specific feature extraction techniques.

2.3.2 Applications

Because of the resemblance with the visual cortex one of main applications for CNNs are images, either classifying images or detecting objects within an image. Since 2010, an annual competition called ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held. The competition uses a subset of the ImageNet dataset, which consists of roughly 1.2 million images of 1000 classes. Alex Krizhevsky [26] scored a top-5 error rate of 17.0% in 2010 and since then networks like ResNet-152 [18] have reached aa error rate of 3.57%, which is close to human accuracy.

Another popular field of research is image modelling, where instead of classifying images you generate new images. Aäron van den Oord et al. made a network called PixelCNN [36] which uses autoregressive connections to model images pixel by pixel. Generative adversarial networks [7, 13] have also been used to model images. These generative techniques can also be applied to audio data as we will discuss in Section 3.3.

2.4 RNN

Recurrent neural networks (RNNs) are a class of neural networks that is used to predict or generate sequences in many domains. Music is already mentioned, but text, speech and motion are other areas where RNNs are widely used. These networks are able to store and update context information from previous inputs to generate a desired output.

The following equations describes how we can use the simple RNN structure in figure 2.5 to generate an output sequence y:

$$h_t = \mathcal{H}(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \quad (2.12)$$

$$y_t = \mathcal{Y}(W_{hy}h_t + b_y) \quad (2.13)$$

where W represents the weight metrices, b is the bias term, $\mathcal{H}$ is the state activation function and $\mathcal{Y}$ is the output function. We can see that the inputs are conditioned on the previous state of the RNN.

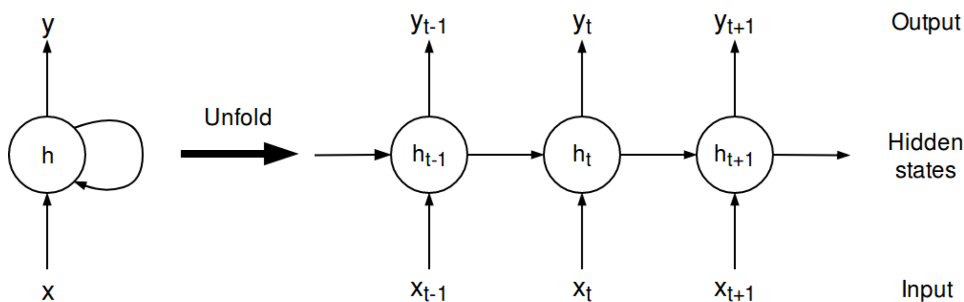


Figure 2.5: The folded representation of a RNN (left) and how the network is unfolded to create the computational graph (right).

Training an RNN involves feeding an entire sequence $X = \{x_1, x_2, ..., x_T\}$ to the network and the goal at each time step t is to predict the next value at time step $t+1$ of the input sequence. When we use the RNN to generate a sequence, however, we use predictions from previous time steps as input to the network, see figure 2.6. Using backpropagation to train networks where the output at time step t is depending on variables at earlier times is referred to as backpropagation through time (BPTT) [44].

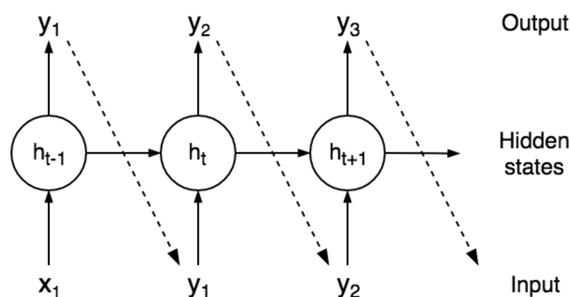
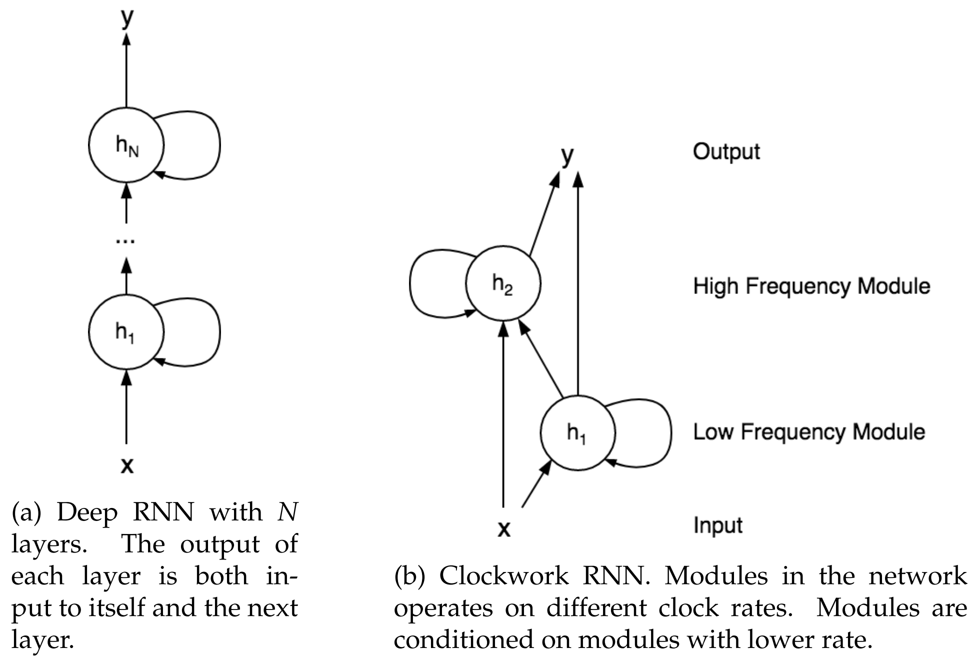


Figure 2.6: Visualizing how the output at time step $t-1$ is used to predict the next time step t during generation of a sequence. x_1 is usually a start token to begin the generation of the sequence.

2.4.1 Structures

There are many ways of constructing a RNN. Graves at el. [16] was one of the first to use a deep RNN. By stacking recurrent units on top of each other, as visualized in figure 2.7a, one can achieve greater performance in the same way as in a CNN or other multilayer networks. After this point, there have been developed many different structures. Koutnik at el. [25] constructed a RNN made of multiple modules where each module operates at a different clock rate, see figure 2.7b, which means that each module has a different input length or modules will skip steps of the input sequence. Modules with longer input sequences, so called low frequency modules, will therefore be more suited for long term dependencies and the high frequency modules, modules with shorter input, will deal with more

local, short term information. The implementation of the ClockworkRNN is complex and will be discussed in a later section.



(a) Deep RNN with N layers. The output of each layer is both input to itself and the next layer.

(b) Clockwork RNN. Modules in the network operates on different clock rates. Modules are conditioned on modules with lower rate.

Figure 2.7: Two different RNN structures.

2.4.2 RNN Architectures

RNNs have been used in many different tasks and each task requires its own form of output. The different tasks and applications have given rise to subgroups within the RNN genre.

Many to many

This type of RNN is common in many applications involving sequence generation, whether it is text generation [16] or even image generation [35]. The networks in figure 2.5 and 2.6 are of this kind. The common traits of these networks is that they will predict the next step in sequence. Given a start token, these networks will generate a sequence one step at a time until a stop token is reached, or a certain number of steps is generated. During training, the networks will generate as many outputs as there are steps in the input sequence.

Many to one

A number of tasks involve classifying sequences. Lei Ba et al. [2] constructed a RNN which classifies objects in images by evaluating small patches of the image step by step. The network is used in two ways, it will at each step predict a new location in the image to evaluate and then it will

use the last state of the RNN to classify the object. As the name suggests, these networks will create a representation of the input sequence which will be used to classify the entire sequence, see figure 2.8.

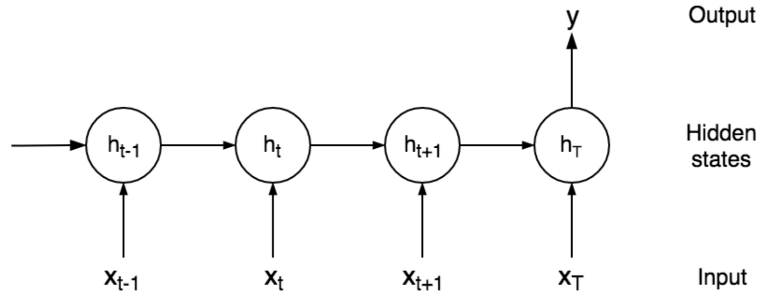


Figure 2.8: A many-to-one RNN network. This type of RNN will create a final representation of the input sequence and use it to generate a single output.

Sequence to sequence

The idea behind a sequence to sequence RNN is to first make a representation of the input sequence and then use this representation to generate a new sequence, see figure 2.9. This kind of RNN is well-suited for machine translation where the task is to generate the same sentence as the input only in a different language [45]. These networks are usually split into two parts, an encoder and a decoder. The encoder is used to make the representation of the input and the decoder will then use this representation to generate a new sequence one step at a time.

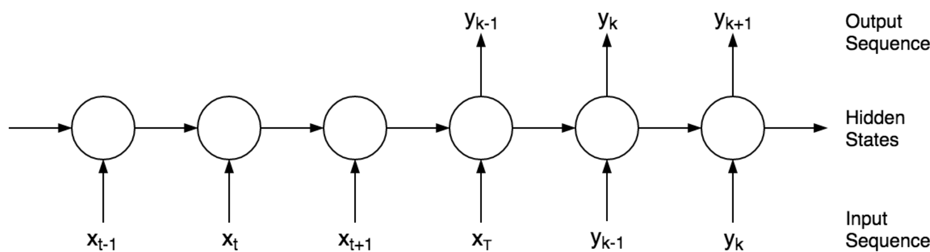


Figure 2.9: A sequence to sequence RNN. This type of RNN creates a final representation of the input sequence and then use it to generate a new sequence. This structure is often used in machine translation.

2.4.3 LSTM

Recurrent networks are very powerful and they are in theory, with a large enough network, able to generate sequences of arbitrary complexity, but experiments show that they are hard to train in tasks involving long term dependencies [3]. When training on long sequences, the gradients tends to blow up or vanish when propagating the error back through the

network. Alternative approaches like simulated annealing and discrete error propagation have been tested and they do show that there might be ways to make these networks perform better than with standard gradient descent.

Hochreiter and Schmidhuber [20] created the *Long Short-Term Memory* cell, which contains a self-connected linear unit called the *Constant Error Carousel* (CEC) which solves the vanishing error problem by allowing constant error flow. In addition to this unit, they added a multiplicative input gate unit to protect the CEC from irrelevant inputs, and also a multiplicative output gate unit to protect other CECs from irrelevant output from itself. These gates will learn to open and close, allowing error signals to pass through the CEC.

Figure 2.10 visualizes how the LSTM cell is constructed and the following equations shows how the output is calculated.

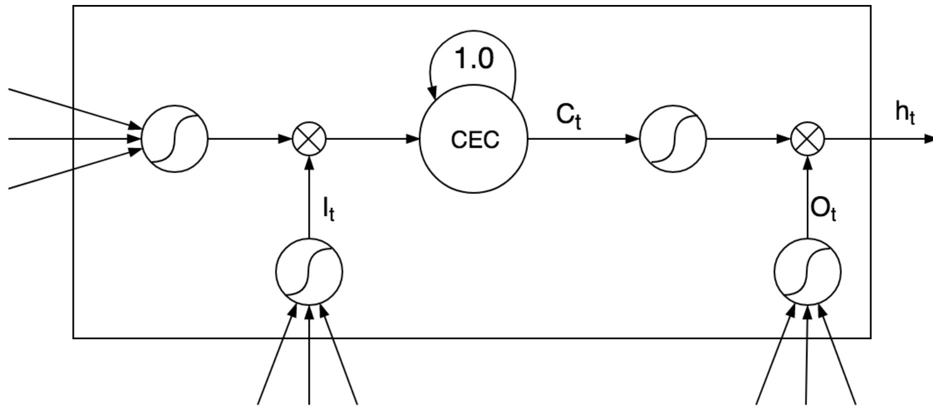


Figure 2.10: Basic LSTM cell structure. The recurrent connection to the CEC makes sure there is a constant error signal

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{2.14}$$

$$c_t = c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.15}$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{2.16}$$

$$h_t = o_t tanh(c_t) \tag{2.17}$$

where σ is the sigmoid function, i_t is the input gate, o_t is the output gate, c_t is the cell activation and h_t is the cell output.

Although these memory cells outperform regular RNNs across long time lags, they do have some limitations. One weakness is that the cell state, the activation of the CEC, often tend to grow linearly across sequences. This will lead to saturation of the h-function and the output gate will lose its function. The forget gate [11] was the solution to this problem. The forget gate replaces the self-connections' constant weight of

1.0 with a multiplicative unit which gives the cell the ability to learn to reset its memory when it is no longer useful. This is particularly important when we are doing truncated backpropagation through time, which means that we split a long sequence into small sections and then train on these small sections. The final state from each section is carried into the next section.

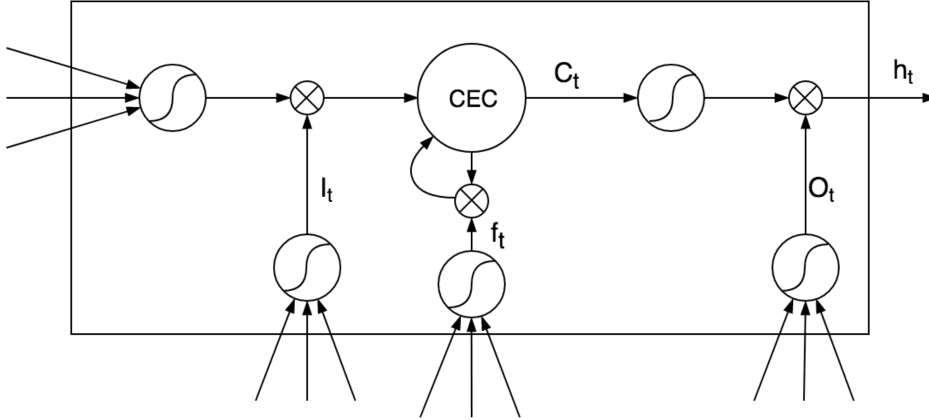


Figure 2.11: LSTM cell with forget gate

The forget gate introduces a new equation and the cell state activation is affected by this new variable f_t.

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2.18}$$

$$c_t = f_t c_{t-1} + i_t tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \tag{2.19}$$

2.5 Musical Applications

Artificial neural networks have long been used to compose or generate music. One of the main challenges when generating music using computers is evaluating the performance of the techniques. Many of these techniques explore the creative aspect of music generation, but creativity is hard to measure and results are often evaluated by human listeners.

Florian Colombo el al. [6] created the Deep Artificial Composer (DAC), a recurrent neural network trained to generate monophonic melodies close to tunes of a given musical style. One RNN is trained to model the transition distribution of the note duration, which is used as an additional input to a RNN that models the transition distribution of the note pitch. This allows the network to learn the relation between a note duration and its pitch, which is essential in learning different styles of music because similar rhythmical patters are more frequent than similar melody patterns in many musical styles. Their results show that the DAC is able to generate melodies that are consistent in style, as well as scale and rhythm.

Iman Malik et al. [30] used recurrent neural networks to translate sheet music into musical performances of different styles. The idea behind

this research is the fact that every musician has a unique interpretation of the sheet description that will lead a variety of different performances. The network consists of an interpretation layer that will convert the musical input into its own representation, and a set of subnetworks called GenreNets that will model the dynamics of the sheet music based on the interpretation. Each GenreNet allows the model to learn a specific style of music. The natural sounding performances this model produces is indistinguishable from a human performance based on human evaluations.

Wavenet [38] is a deep convolutional neural network that generates sound sample by sample. Using causal convolutions, Wavenet is able to predict samples conditioned on all previous samples. When trained on a music dataset, WaveNet is able to generate highly realistic sounding music that closely resembles the dataset, even when produced by an unconditional model. The problem with WaveNet is that it is essential to have a big receptive field in order to generate music that sound pleasing and the way to accomplish this is to use many layers. This makes WaveNet use alot of memory during training which can affect the performance when resources are limited. The architecture allows the use of parallel computations and that makes WaveNet more efficient to train in comparison to other models using the RNN architecture.

Engel et al. [8] argues that WaveNet rely on external conditioning to capture long-term dependencies. To address this problem, Engel et al. created a WaveNet Autoencoder that removes the need of this external conditioning. Using a WaveNet-like encoder to produce embeddings ditributed in time and a WaveNet decoder to effectively recreate the original audio, they are able to control the generation and produce new sounds that can be a mixture of instruments.

MidiNet [46] is generative adversarial network (GAN) [14] that generates symbolic music one bar at a time. It consists of two CNNs, a generator and a discriminator. The generator uses random noise to generate a new melody which is used as input to the discriminator, together with other real melodies. The discriminator will predict whether the input melodies are real or generated, which will inform the generator how to generate more realistic melodies. To be able to generate melodies across multiple bars, MidiNet uses a conditioning network that conditions the generation of melodies on previous bars. This allows the network to keep track of previous events without using recurrent units. The results from MidiNet was compared to Google's MelodyRNN models [43] and the result shows that MidiNet generates melodies which are as realistic and pleasant, yet more interesting, than the MelodyRNN models.

Chapter 3

Models

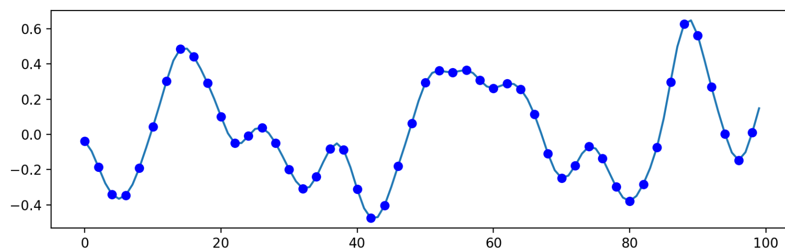
Work in data technology and music have usually been about using more abstract methods to represent music or sound. Whether it is about using MIDI data to compose music or adjusting parameters on a synthesizer to achieve a certain timbre. As I have mentioned previously, people have tried to develop new techniques for generation of sound at the sample level by for example using evolutionary algorithms, but trying to use backpropagation to train a neural network to generate sound, which is suppose to sound realistic and authentic, is an extremely difficult task which we have only started to experiment with in the last two years.

To discribe music or sound based on musical notation or specific parameters is not a very difficult task. Of course, it would take a human a few hundred or maybe thousand hours of training to be able to do this at a certain level, but just imagine how hard it would be if we were to compose music only by describing the position of a speaker element over time. This gives you an idea of the complexity of the task we are trying to achieve. A simple melody might contain eight separate notes in a sequence which would make the last note only to be depended on seven steps of events. Let us consider this melody lasting for five seconds, which would mean that the last note would depend on maybe 176 400 previous samples if we were using a standard sample rate of 44.1 kHz. Even if we are only using a third of the amount of samples we are still talking about dependencies stretching over thousands of samples. We can see from figure 3.1a that a simple melody with only a few musical events can represent many seconds of sound, compared to the sound wave in figure 3.1b which contains 100 samples which is adding up to a total of 6.8 ms of sound when using a sample rate of 14.7 kHz.

The models which are able to do these calculations are enormous and very complicated and it would take an extreme amount of calculations to find the optimal connections in these networks. The only reason we are able to train these networks is the rapid development of parallel computing which the GPU is able to offer us.

In this section I will describe the architecture of three different model, NaiveRNN, SampleRNN and WaveNet. I will in detail walk you through the implementation of the baseline model NaiveRNN and the more

(a) Simple melody



(b) 6.8ms of samples

Figure 3.1: Demonstrating the differences in high-level versus low-level musical representation

complex model SampleRNN. The WaveNet implementation I have used is inspired by the Fast WaveNet [40] and the source code is found on GitHub [39].

3.1 NaiveRNN

The NaiveRNN is a deep RNN made of multiple LSTM layers with residual connections between every layer, see figure 3.2. It is inspired by the basic RNN structures which evaluates one sample at a time. It models the probability of a sequence $x = x_0, ..., x_T$ as the product of the probabilities of each sample given all the previous samples.

$$p(x) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1}) \tag{3.1}$$

The goal with this model was to make a simple RNN so that I had a baseline comparison to the other more complex models. I also wanted to get more familiar with Tensorflow, an open source machine learning framework, and especially its RNN functionality.

The model is simple, or "naive", due to the fact that in every step of the RNN there are only one sample fed to the network and there are only sample coming out, see figure 3.3 for a visualization of the data flow. This is

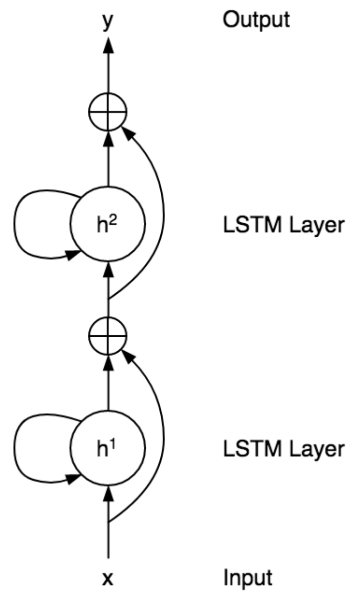


Figure 3.2: Naive RNN architecture with two layers of LSTM cells. There are residual connections between each layer.

equivalent to a RNN trying to learn how to read and write by only feeding in one character at a time instead of entire words. The network itself has to learn how to define a word and find the connections and correlation between all the words, which makes it a more challenging task.

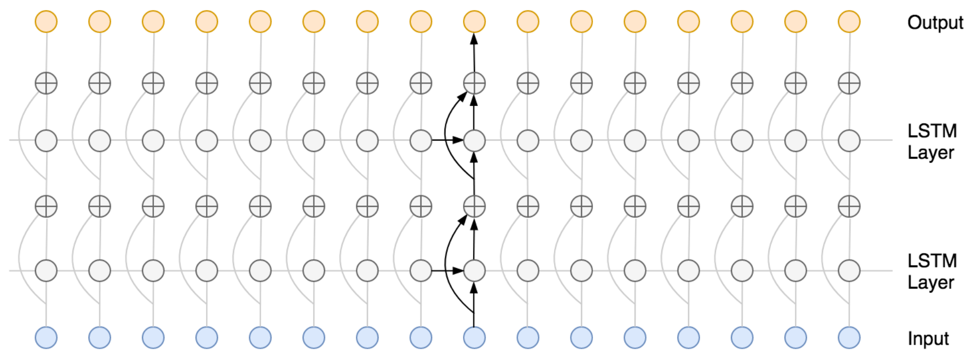


Figure 3.3: Visualizing the data flow in the NaiveRNN. Only one sample of the input is used to predict the next sample.

3.1.1 Implementation

Creating a model in Tensorflow means that we are constructing a computation graph which specifies the mathematical operations

The whole model is made up of a number of layers of LSTM cells, where there are residual connections between every layer, see figure 3.2. Tensorflow provides us with high-level functions which makes the implementation of this model relatively simple.

```python
# Create one LSTM cell with residual connections
def residual_cell(units):
    cell = tf.contrib.rnn.LSTMCell(units)
    return tf.contrib.rnn.ResidualWrapper(cell)

# Make an array of residual LSTM cells
cells = [residual_cell(units) for i in range(layers)]
# Put all cells into a convenient cell
cell = tf.contrib.rnn.MultiRNNCell(cells)
```

where *tf* is the reference to the Tensorflow framework, *units* specifies the number of units in each LSTM cell and *layers* specifies the number of LSTM layers in the model.

The residual connections, often reffered to as skip connections, makes sure the input to each LSTM cell is added to the output. These residual connections might make it easier to train the network, especially if there are many layers in the network, because they allow the error gradients to flow easier through the network [18]. They don't increase the amount of parameters within the network or make the network more complex.

Because we want to add together the input and output of each LSTM cell, we have to make sure the dimensions match. In this model, it is only a problem in the first LSTM layer because of the shape of the input to the network. To make the dimensions match we need to upsample the input.

```python
# Input to the graph
self.inputs = tf.placeholder(tf.float32, [None, None],
 ↪  name='input')
# Shape = [batch_size, n_steps]

# Upsampling weights and bias
w = tf.get_variable('input-weights', [1, units],
 ↪  initializer=tf.contrib.layers.xavier_initializer())
b = tf.get_variable('input-bias', [units],
 ↪  initializer=tf.zeros_initializer())

# Reshape input for matmul function
inputs = tf.reshape(self.inputs, [-1, 1])
# Shape = [batch_size * n_steps, 1]
# Upsample
inputs = tf.matmul(inputs, w) + b
# Shape = [batch_size * n_steps, units]
```

We reshape the input before the *matmul* function because we want to apply the same weights to every step in the sequence.

We use the *dynamic_rnn* function in Tensorflow which allows us to train on sequences of arbitrary lengths. This function expects an input with a specific shape and that is why we reshape the *inputs* matrix to the correct dimensions before sending it to the *dynamic_rnn* function.

```
# Reshape to correct shape
inputs = tf.reshape(inputs, [batch_size, -1, units])
# Provide an initial state for RNN
self.initial_state = cell.zero_state(batch_size, tf.float32)
# Create the RNN
rnn_output, self.final_state = tf.nn.dynamic_rnn(cell, inputs,
↪   initial_state=self.initial_state)
```

I make a reference to the *initial state* so that we can provide a previous state as input to the graph at a later point during training. This is particularly useful when performing truncated BPTT, where the final state of one training section will be the initial state of the next section.

The last part of the network is the softmax layer. This is where we reduce the dimension space of the RNN output to match the number of classes in our dataset, before applying the softmax function and calculating the loss. To train the network I use the Adam optimizer [24] and it is minimizing the cross entropy loss function.

```
# Prediction weights and bias
w = tf.get_variable('pred-weights', [units, CLASSES],
↪   initializer=tf.contrib.layers.xavier_initializer())
b = tf.get_variable('pred-bias', [CLASSES],
↪   initializer=tf.zeros_initializer())
# Reshape RNN output for matmul function
rnn_output = tf.reshape(rnn_output, [-1, units])
# Shape = [batch_size * n_steps, units]
# Final layer
logits = tf.matmul(rnn_output, w) + b
# Shape = [batch_size * n_steps, CLASSES]

self.targets = tf.placeholder(tf.int32, [None, None],
↪   name='targets')
# Make sure targets match the logits
labels = tf.reshape(self.targets, [-1])
# Calculate loss
loss =
↪   tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels,
↪   logits=logits))
self.optimize =
↪   tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

The Tensorflow framework allows us to implement standard RNNs using only a few lines of code.

3.2 SampleRNN

SampleRNN [33] is a model inspired by the ClockworkRNN architecture [25] where the network consists of modules operating at different clock

rates, which are connected hierarchically from lower frequency modules to higher frequency modules. SampleRNN consists of three modules, two modules containing a RNN and one module with a multilayer network, linked together as shown in figure 3.4. In this section I will describe the architecture of SampleRNN, how I have implemented it using Tensorflow and I will discuss how it is different from the implementation of the NaiveRNN.

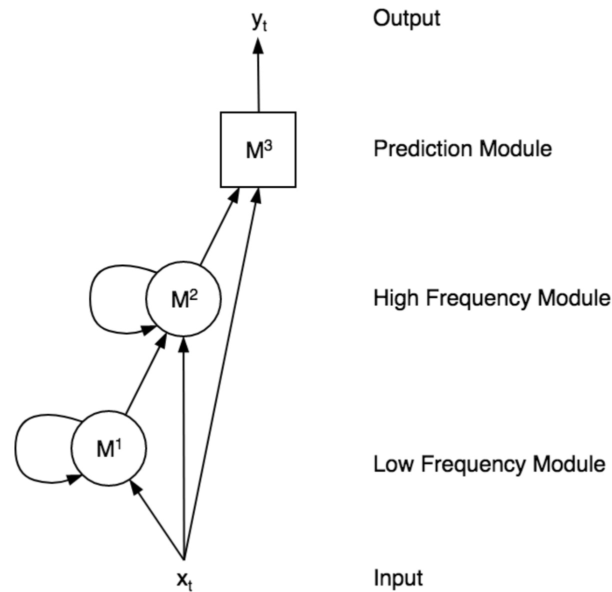


Figure 3.4: Visualizing the folded structure of SampleRNN. The modules are connected hierarchically, meaning higher frequency modules are conditioned on lower frequency modules. Modules M^1 and M^2 are deep RNNs which can consist of multiple recurrent layers.

3.2.1 Modules with Different Clock Rates

The modules in figure 3.4 get their name from the frequency which their input is fed into them. All modules will receive input from the same input sequence, it is only the frame size of the input which are different, see figure 3.5 for a visulization of the data flow. The low frequency module (LFM) will receive m values from the input sequence at each time step and the high frequency module (HFM) will receive n, where $n \leq m$. This will lead to each RNN in the different modules having a different amount of steps during each training step, the LFM will have less steps than the HFM. The goal of using this architecture is to get the different modules to learn dependencies across different time lags. It is easier to understand this by looking at figure 3.6

Figure 3.6 shows how we can represent a complex wave form, figure 3.6c, with two simple sine waves, one with a low frequency and one with i higher frequency, figure 3.6a and 3.6b respectively. This is what we want to achieve with SampleRNN, each module breaking down the input into

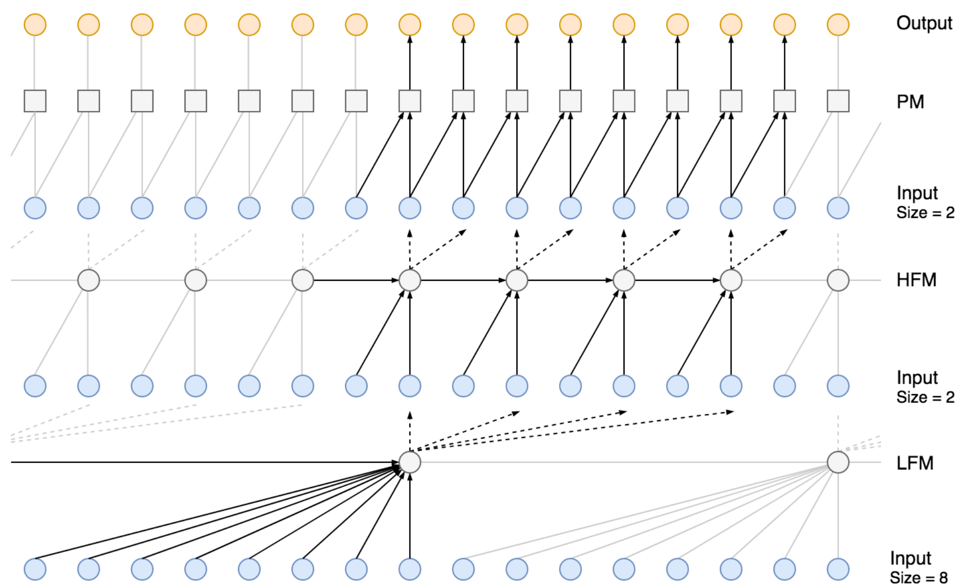


Figure 3.5: SampleRNN data flow. Dashed lines represent conditioning vectors to the next module.

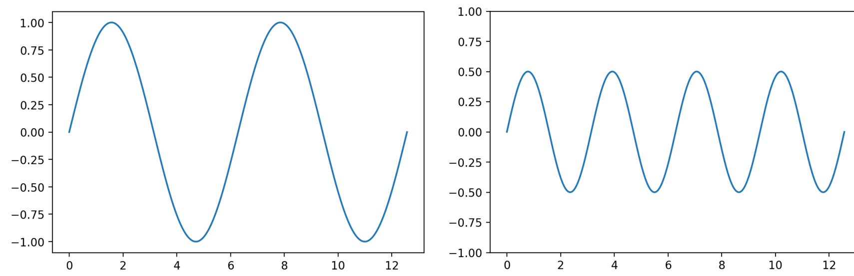
more simple structures so that the model can make better predictions.

Clockwork Implementation

I use the *dynamic_rnn* function in Tensorflow to make the computation graph and the function expects an input of size *[batch size, sequence length, input size]*, where *input size* is the dimensions of the input at each step. In order to make each module have a different clock rate, I change the input size in each of the two RNNs by reshaping the input vector into non-overlapping frames.

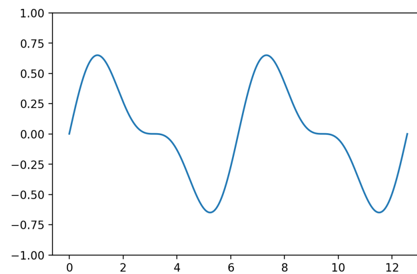
```
# Where we input the sequences
self.input = tf.placeholder(tf.float32, [None, None],
 ↪  name='input')
# Slice input, we don't need last frame
self.big_frames_input = self.input[:, :-BIG_FRAME_SIZE]
big_frames = tf.reshape(self.big_frames_input, [batch_size, -1,
 ↪  BIG_FRAME_SIZE]) # Missing dim is number of steps
# Slice input, we don't need part of the beginning and end
self.frames_input = self.input[: BIG_FRAME_SIZE -
 ↪  FRAME_SIZE:-FRAME_SIZE]
frames = tf.reshape(self.frames_input, [batch_size, -1,
 ↪  FRAME_SIZE]) # Missing dim is number of steps
```

where *big_frames* is the input to the LFM and *frames* is part of the input to the HFM. The constants *BIG_FRAME_SIZE* and *FRAME_SIZE* corresponds to the number of values from the input sequence we feed to the RNNs at each step, where *FRAME_SIZE* is less than or equal to *BIG_FRAME_SIZE*.



(a) 1 Hz sine wave. A low frequency representation of the original wave function.

(b) 2 Hz sine wave. A high frequency representation of the original wave function.



(c) Sum of the two waveforms

Figure 3.6: Visualizing a complex wave function and how we would expect a low frequency and a high frequency module to represent the original wave function.

tf is a reference to the Tensorflow API. Everywhere I use the keyword *self* are useful access points to the graph during generation of samples and will be discussed later in this section. I slice the input because of how the data flows through the network, see figure 3.5. This makes training implementation easier.

The second part of the input to the HFM is the output of the LFM and we get the output by running the *dynamic_rnn* function.

```
big_cell = tf.contrib.rnn.LSTMCell(units)
self.big_init_state = big_cell.zero_state(batch_size,
↪  tf.float32)
big_frame_out, self.big_final_state =
↪  tf.nn.dynamic_rnn(big_cell, big_frames,
↪  initial_state=self.big_init_state)
```

where *big_final_state* will be the next *big_init_state* and *units* is the number of units in the LSTM cell. All the recurrent connections are handled within the *dynamic_rnn* function.

The output space of the LFM, *big_frame_out*, doesn't match up with the input of the HFM, because of the different amounts of steps in each RNN. This is why we need to upsample the output.

```
# Upsampling weights and biases
```

```
w = tf.get_variable('big-frame-weight', [units, R * units],
↪  initializer=xavier())
b = tf.get_variable('big-frame-bias', [R * units],
↪  initializer=tf.zeros_initializer())

# Reshaping rnn's output for use with matmul function
big_frame_out = tf.reshape(big_frame_out, [-1, units])
# Upsample the rnn output
big_frame_out = tf.matmul(big_frame_out, w) + b
# Shape = [batch_size * n_steps, R * units]
# Getting the output ready for input to the next module
self.big_frame_out = tf.reshape(big_frame_out, [-1, units])
# Shape = [batch_size * n_steps * R, units]
```

where R is the ratio between the low and high frequency. What this code does is to make one step of the output from the LFM into R steps of the input to the HFM.

There is one more step we have to make in order to add the output of the LFM with the input sequence to make the input of the HFM, we have to upsample the input sequence as well. The first two dimensions, batch size and number of steps, are matching, but the last dimension is different and we need all dimensions to match in order to add them together.

```
# Upsampling weights for HFM input
w = tf.get_variable('frame-input-weights', [FRAME_SIZE, units],
↪  initializer=xavier())
b = tf.get_variable('frame-input-bias', [units],
↪  initializer=tf.zeros_initializer())

# Reshaping the frames for use with matmul function
frames = tf.reshape(frames, [-1, FRAME_SIZE])
# Shape = [batch_size * n_steps, FRAME_SIZE]
# Upsample input frames
frames = tf.matmul(frames, w) + b
# Shape = [batch_size * n_steps, units]
# Add LFM outputs
frames += self.big_frame_out
# Reshape into correct shape for dynamic_rnn function
frames = tf.reshape(frames, [batch_size, -1, units]) # Missing
↪  dim is number of steps
# Shape = [batch_size, n_steps, self.dim]
```

At this point, the *frames* are ready as input to the *dynamic_rnn* in the HFM.

```
cell = tf.contrib.rnn.LSTMCell(units)
self.init_state = cell.zero_state(batch_size, tf.float32)
frame_out, self.final_state = tf.nn.dynamic_rnn(cell, frames,
↪  initial_state=self.init_state)
```

We have to do the same procedure when going from the HFM to the prediction module (PM) because also these two modules operates at different clock rates. The procedure is to upsample the output of the HFM and the input sequence, before adding them together.

```
# Upsampling weights and biases
w = tf.get_variable('frame-output-weight', [units, R_1 *
↪  units], initializer=xavier())
b = tf.get_variable('frame-output-bias', [R_1 * units],
↪  initializer=tf.zeros_initializer())

# Reshaping rnn's output for use with matmul function
frame_out = tf.reshape(frame_out, [-1, units])
# Shape = [batch_size * n_steps, units]
# Upsampling
frame_out = tf.matmul(frame_out, w) + b
# Shape = [batch_size * n_steps, R_1 * units]
self.frame_out = tf.reshape(frame_out, [-1, units])
# Shape = [batch_size * n_steps * R_1, units]
```

where *R_1* is the ration between the frequencies of the HFM and PM.

In the PM, we want to evaluate the very near dependencies between samples, such near dependencies that we can use a standard multilayer network instead of a RNN. This will speed of training time. I use the sliding window approach to convert the input sequence into an array of overlapping frames using the *frame* function in Tensorflow, before I upsample every frame to match the output space of the HFM.

```
# Upsampling weights and bias
w = tf.get_variable('sample-weight', [FRAME_WIDTH, units],
↪  initializer=xavier())
b = tf.get_variable('sample-bias', [units],
↪  initializer=tf.zeros_initializer())

# Slice input to correct length
self.sample_input = self.input[:, BIG_FRAME_SIZE -
↪  FRAME_WIDTH:-1]
# Turning the input into overlapping frames of size FRAME_WIDTH
pred_input = tf.contrib.signal.frame(self.input, FRAME_WIDTH,
↪  1)
# Reshape for matmul function
pred_input = tf.reshape(pred_input, [-1, FRAME_WIDTH])
# Upsample input sequence
pred_input = tf.matmul(pred_input, w) + b
# Add HFM output
pred_input += self.frame_out
```

where *FRAME_WIDTH* is the width of the overlapping frames and is less than or equal to *FRAME_SIZE*.

The last part is the PM, which consists of two fully connected ReLU layers and a softmax layer at the end.

```
# Layer 1
w = tf.get_variable('pred-weight', [units, units],
↪   initializer=xavier())
b = tf.get_variable('pred-bias', [units],
↪   initializer=tf.zeros_initializer())
pred_out = tf.nn.relu(tf.matmul(pred_input, w) + b)
# Layer 2
w = tf.get_variable('pred-weight-1', [units, units],
↪   initializer=xavier())
b = tf.get_variable('pred-bias-1', [units],
↪   initializer=tf.zeros_initializer())
pred_out = tf.nn.relu(tf.matmul(pred_out, w) + b)
# Softmax layer
w = tf.get_variable('pred-weight-2', [units, CLASSES],
↪   initializer=xavier())
b = tf.get_variable('pred-bias-2', [CLASSES],
↪   initializer=tf.zeros_initializer())
logits = tf.matmul(pred_out, w) + b
# Use the linear activation in the softmax function

self.targets = tf.placeholder(tf.int32, [None, None],
↪   name='targets')
# Slice targets to correct length
targets = self.targets[:, BIG_FRAME_SIZE:]
# Reshape targets to match logits dimension space
targets = tf.reshape(targets, [-1])
# Calculate loss using cross entropy
cost =
↪   tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits,
↪   labels=self.targets)
self.loss = tf.reduce_mean(cost)

# Use AdamOptimizer to train the network
self.optimize =
↪   tf.train.AdamOptimizer(learning_rate).minimize(self.loss)
```

3.2.2 Training vs Generation Algorithm

Running a training step in the model is as easy as feeding the whole training sequence , as well as the target sequence, to the graph and the model will deal with slicing and shaping the input to each module.

```
# l = length of training sections
for i in range(inputs.shape[1] // (l - BIG_FRAME_SIZE)):
    # Make sure not to skip important samples
```

```
    start = i * (l - BIG_FRAME_SIZE)
    # Feeding l samples to the graph
    feed = { self.input: inputs[:, start:start+l],
             self.targets: targets[:, start:start+l] }
    # Add states if we have one
    if state is not None:
        feed[self.init_state_tuple] = state

    state = self.session.run([self.final_state_tuple,
     ↪  self.optimize], feed_dict=feed)[0]
```

where *init_state_tuple* is a tuple containing the initial states for both RNNs and *final_state_tuple* allows us to get both of the RNNs finale state. The *run* function will execute and return the results of the specified nodes in the graph and the *session* object contains the graph.

The generation algorithm is more complicated because we have to think about the frequency of which a module is run. For every run of the LFM, we will have multiple runs of the HFM. This is why it is useful to have all the access points to the graph, it allows us to run specific sections of the graph where the input data to those sections are already calculated.

```
# Generate a sample at time step t
# First frame of the sequence is padding
for t in range(BIG_FRAME_SIZE, steps):
    if  t % BIG_FRAME_SIZE == 0:
        big_frame_feed = { self.big_frame_input: sample[:,
         ↪  t-BIG_FRAME_SIZE:t] }
        # Add state if there is one
        if big_frame_state is not None:
            big_frame_feed[self.big_init_state] =
             ↪  big_frame_state
        big_frame_out, big_frame_state =
         ↪  self.session.run([self.big_frame_out,
         ↪  self.big_final_state], feed_dict=big_frame_feed)
        big_frame_out = np.reshape(big_frame_out, [batch_size,
         ↪  -1, units])

    if t % FRAME_SIZE == 0:
        frame_feed = { self.frame_input: sample[:,
         ↪  t-FRAME_SIZE:t], self.big_frame_out:
         ↪  big_frame_out[:, t % R] }
        # Add state if there is one
        if frame_state is not None:
            frame_feed[self.init_state] = frame_state
        frame_out, frame_state =
         ↪  self.session.run([self.frame_out,
         ↪  self.final_state], feed_dict=frame_feed)
        frame_out = np.reshape(frame_out, [batch_size, -1,
         ↪  units])
```

```
sample_feed = { self.sample_input: sample[:,
 ↪  t-FRAME_WIDTH:t], self.frame_out: frame_out[:, t % R_1]
 ↪  }
predictions = self.session.run(self.prediction,
 ↪  feed_dict=sample_feed)
sample[:, t] = bins[predictions]
```

We can use the modulo operator, %, to control when each module is run. The frequency of each module depends on its input size and we will only run a module when we have generated enough samples to fill its entire input. We reshape the output of the LFM into a batch of sequences with length R and make sure that the correct output is fed to the HFM. We do the same with the output of the HFM.

3.3 WaveNet

WaveNet [38] is a model which is inspired by the PixelCNN [36] architecture and has proven to be the state of the art when it comes to audio generation. While other models use recurrent connections to generate sequential data, WaveNet uses causal convolutions. These are one dimensional convolutions which makes the output at time step t not dependant on any future timesteps, see figure 3.7. This is possible by using filters with a width of two.

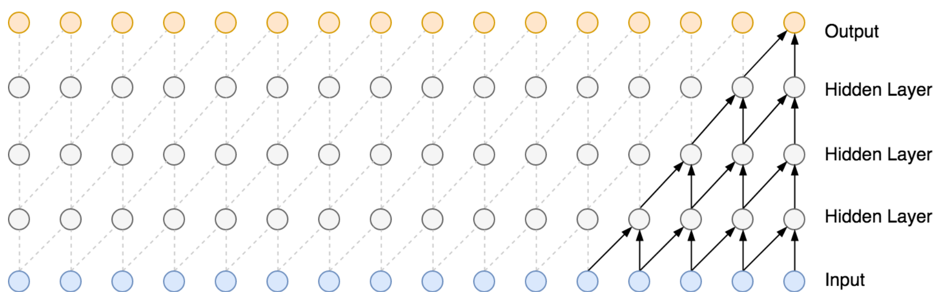


Figure 3.7: Causal convolutions used in WaveNet

The problem with using filters which are only two wide is that the receptive field of the output nodes are not big enough, which is essential when you think about how many samples one single time step might be dependant on. By looking at figure 3.7, we can see that using these filters gives us a receptive field in layer n equal to $n + 1$, which means that we would have to use thousands of layers to get a receptive field big enough to see the connection between relevant samples. Van den Oord used dialated causal convolutions as a solution to this problem, see figure 3.8.

In this kind of convolutions we simply skip parts of the input. This is equivalent to increasing the filter width and changing the value of the inputs we are not interested in to zero. By increasing the filter width every

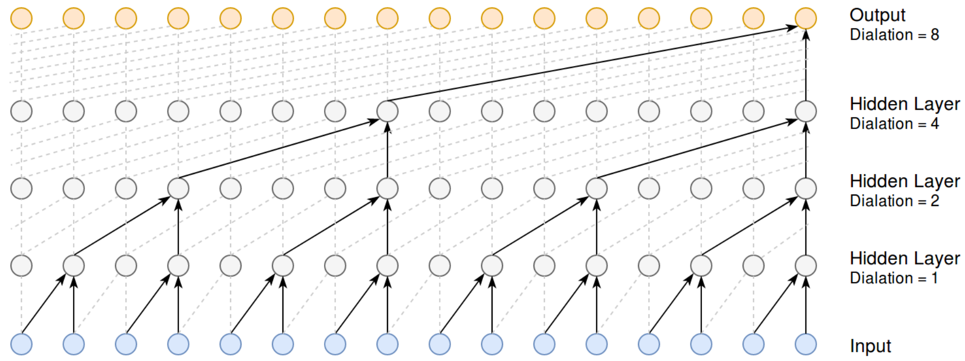


Figure 3.8: Dialated causal convolutions used in WaveNet

layer n, we can achieve a receptive field equal to 2^n, which is a significantly bigger number than $n + 1$. WaveNet is built using multiple blocks of these dilated convolutional layers stacked on top of each other. This results in an alternating pattern in the receptive field.

$$1, 2, 4, ..., 512, 1, 2, 4, ..., 512, 1, 2, 4, ..., 512$$

The total receptive field can then be calculated as $R = b2^n$, where b is number of blocks and n is the number of layers in each block. Using stacked dilated convolutions in this manner enables WaveNet to have a very large receptive field using only a small number of layers while still preserving the input resolution throughout the network.

Even though we use dilated causal convolution, we are still interested in using filter with a width of two. This is because the increased filter width would increase the amount of unnecessary computations, when we are only interested in a fraction of the input. In layer twelve, we would have a filter width of 4096, but we are only actually interested in the first and last value of the filter. We can shape the input matrix in a particular way which allows us to use filters which are two wide. Given the two input sequences presented below.

$$\begin{array}{c|c|c|c|c|c|c|c|c|c} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \end{array} \tag{3.2}$$

The first thing we have to do is to transpose the matrix, which would

make it look like this

0	10
1	11
2	12
3	13
4	14
5	15
6	16
7	17
8	18
9	19

(3.3)

If we find our selves in layer one, we can see from figure 3.8 that we are using a dialation of one, which means that are skipping every other value in every convolutional operation. This is the same as splitting each of the two sequences into two subsections. In our case, one of the subsequences would contain the even numbers and the other would contain the odds. We can achieve this by reshaping the matrix above so that each column of the matrix will have the correct length as the subsequences. In our case, the subsequences will have a length of 5. The matrix will now look like this

0	10	1	11
2	12	3	13
4	14	5	15
6	16	7	17
8	18	9	19

(3.4)

The only thing we have to do to make the matrix correct is to again transpose it so that each row of the matrix contains each subsequence. Now we can use this matrix with the convolution operator and we are able to use a filter with a width of two. In our case, we have ended up with a set of sequences which is twice as big, but each sequence within the set is half the length of the original sequences.

0	2	4	6	8
10	12	14	16	18
1	3	5	7	9
11	13	15	17	19

(3.5)

After we have done the convolutions on the new matrix we have to turn each subsequence back into one sequence so that we get the correct input shape into the next layer in the network. If we do all the steps presented above in reverse order, we will turn the output of the convolutions into sequences with the same length as the input.

3.4 Conclusion

Even though the NaiveRNN and SampleRNN use the same basic RNN principles, their implementation is very different. Because there are no

functions in the Tensorflow framework that makes it easy to implement modules with different clock rates, the implementation will be longer and more complicated than compared to the NaiveRNN. The main difference is that in the NaiveRNN we can compute the output with one call of the *dynamic_rnn* function, but in the SampleRNN we have to construct each module as its own *dynamic_rnn*. This will lead to a difference in the order of the computations. While the NaiveRNN will compute all the layers in one step, a left to right approach, the SampleRNN will compute all the steps in one module, a bottom to top approach. Despite the differences, I don't expect any of the two approaches to perform better than the other. If we would have used a bottom to top approach in the NaiveRNN, the unfolded graph would still be the same as with the left to right approach.

If we compare the data flow of all three models, it is clear that SampleRNN has similarities between both the NaiveRNN and WaveNet. We can see from figure 3.5 that the HFM and PM is closely related to the causal convolutions we find in WaveNet.

Chapter 4

Experiments

4.1 Data

I use a data set created by Bernd Krueger [27] that contains a large collection of classical piano music. Although the data set contains more than a hundred musical pieces, I have selected a smaller section of pieces that adds up to about forty minutes of music. This is because these pieces are similar in terms of audio quality, use of reverb and timbre, which might make it easier to train the networks.

4.1.1 Preparing the data

The first step in preparing the data is to split each track of music into four-seconds long sections on which we will perform truncated BPTT. In theory, it is possible to train on entire pieces, but splitting each piece into smaller sections where we reset the state of the RNN after each section, will lower the risk of saturating the cell activation function.

I normalize all sections individually to make sure all data is ranging from -1 to 1. This will lead to better use of the entire sample resolution, however, we will loose the long term velocity changes present in the music. This will help the network focus on how the piano sounds, but it might end up generating music which is flat and not dynamic.

Each model outputs a discrete distribution with 256 possible quantized values of x_t. This corresponds to a sample resolution of 8 bits. When generating samples, each predicted sample is used as input to the next step and to make the input sequences correspond better to the generated sequences, I quantize every input sequence to match the output distribution of the models. I divide the range from -1 to 1 into 256 bins and I use those bins to get the bin index of each sample. Those indices are used as targets in every model and they are used to retrieve the quantized floating point value of the bins.

The music is recorded with a sample rate of 44.1 kHz, but the small sample resolution might result in having multiple proceeding samples with the same value. This can lead to artifacts, and more importantly unnecessary calculations, and it is the reason why we down sample the

music to 14.7 kHz, which corresponds to every third sample of the original track. Even though it will decrease the audio quality, 14.7 kHz is still a sufficient sample rate to capture the fundamental frequencies of every note on a piano, where the highest possible note is a high C, which corresponds to frequency of 4.186 kHz.

Decreasing the sample resolution and rate can ease the learning procedure while still generating audio with reasonable quality.

4.2 Overfitting

Before we start training the networks on a big data set it is important to find the optimal hyper parameters for each network. These parameters varies from network to network and from data set to data set. It is hard to know what these parameters should be and the only way to find out is to train the network with different values. If we train the networks on a small section of the data set we are able to do many short runs and observe how these parameters affect the training and then try the parameters which gave the best result on a bigger data set.

Many networks share common parameters such as learning rate, number of hidden layers and number of nodes in each layer, but some networks might have less common parameters that are specific to their architecture.

In this experiment I have trained all the models using a very small section of the data set, a one-second long sequence of a single piece of piano music. I have done multiple training runs of the NaiveRNN with different parameter combinations to get a better understanding of how each parameter affect the training of the network. For the other two models, hyperparameters from other authors' experiments have been used. At the end of the experiment, I compare the best performing NaiveRNN with the other two models.

All three models are trained using the Adam optimizer [24] with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ and an initial learning rate of 0.001, which are the default values in Tensorflow.

4.2.1 Naive RNN

I started the overfitting with a base configuration where I used two LSTM layers, each LSTM layer has 256 units and I performed truncated BPTT on sequences with 256 steps. I have experimented with different amounts of layers, hidden units and sequence lengths to try to find the optimal combination of parameters. Table 4.1 displays the loss results after testing six different combinations.

It turns out that the base configuration I chose was able to reproduce the small data set, see appendix A, example 1. This suggests that the NaiveRNN, with the 2-layer, 256-units, 256-steps configuration, is a good baseline comparison against other configurations.

Layers	Units	Length	Steps	Time	Loss
2	256	256	44.83k	1:43:13	0.027
3	256	256	32.93k	1:49:18	0.086
2	512	256	29.13k	1:21:57	0.049
2	256	512	23.24k	1:45:51	0.124
2	256	128	68.06k	1:18:20	0.007
2	512	128	42.24K	1:00:55	0.015

Table 4.1: Training loss and time for the NaiveRNN when overfitted on a single example with different hyperparameters. I stopped the training if a configuration used longer time to converge than the base configuration. The network with 2 layers, 512 units and a sequence length of 128 was the most efficient.

Adding LSTM layers

Some neural networks will benefit from having deeper structures and with this in mind I wanted to see how the NaiveRNN performed with one more LSTM cell.

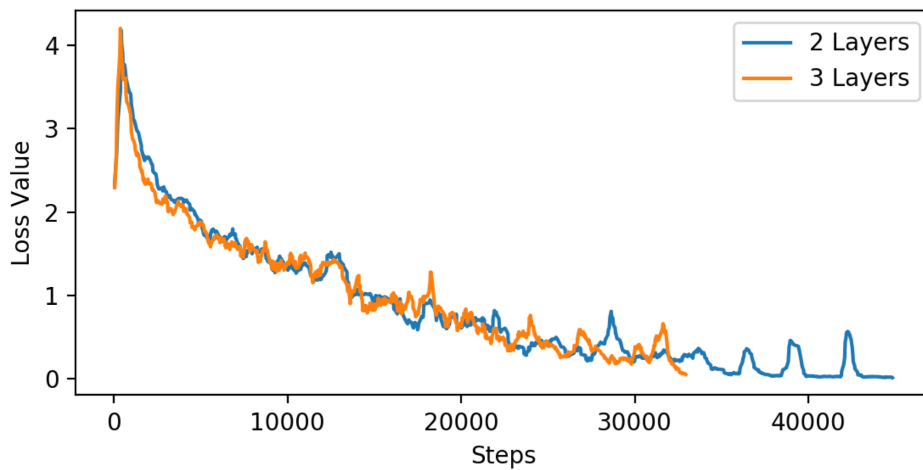


Figure 4.1: Training loss when overfitting the NaiveRNN with two and three LSTM layers. The x axis is training steps. Y axis shows the rolling average over 10 steps. Adding layers doesn't improve the learning efficiency of the network.

Figure 4.1 displays how the error value changes with the amount of training steps. This allows us to compare the learning efficiency of the two configurations. My definition of efficiency is the rate of change in loss value with respect to training steps. Even though the rate of change is not calculated directly, the graphs gives us a visual indication of how each parameter affect the rate of change.

We can see that the two graphs behaves similarly, which means that there are no substantial improvement by adding one more cell. If the extra cell had been beneficial for the model, I would have expected the error

value to decrease at a faster rate when compared to the base model. Since this is not the case, I believe that there is no need to make any deeper representation of the input. Compared to images, where objects might be pieced together by many different and complex features, sound might contain many of the same and simple features. This makes me think that a wider network, where there might be more connections between lower level features, would be beneficial at this specific task.

Adding more units

With the results I got from the previous test I wanted to see if my assumptions was correct. I doubled the amount of units to a total of 512 within each LSTM cell and ran the test again.

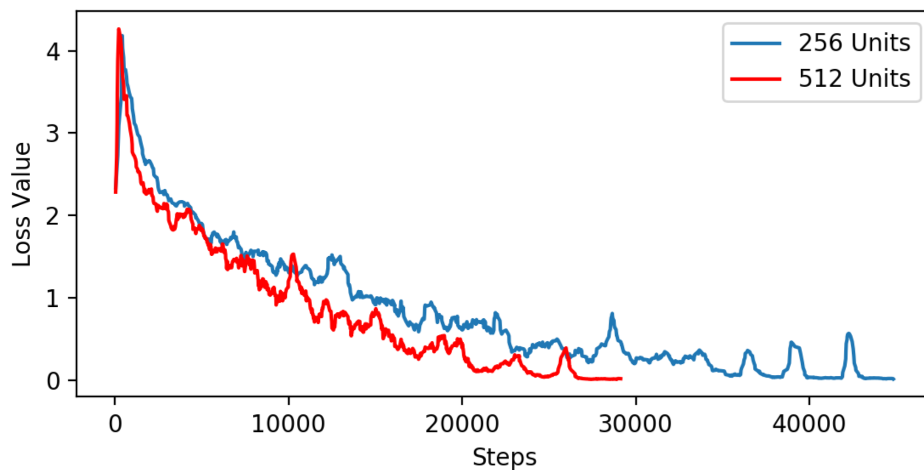


Figure 4.2: Training loss when overfitting the NaiveRNN with 256 and 512 units in each LSTM cell. Results show that the network with 512 units needs less steps to reach the same loss value as compared to 256 units.

If we take a look at the graphs in figure 4.2, then it seems like the model with twice as many units is indeed more efficient per training step than the base configuration. This makes the model converge in a shorter amount of time even though each training step takes a bit longer to calculate.

Changing the sequence length

As I have explained earlier, there are dependencies between samples which are thousands of steps apart when dealing with sample level audio and because of this one might think that it would be better to perform BPTT on longer sequences. I increased the sequence length to 512 steps to see how this would impact the performance of the network.

It is hard to say if the longer sequences does improve how well the network is able to learn the data set, figure 4.3. One thing that is clear, however, is that increasing the sequence length will increase the training time, it will actually increase linearly with the sequence length. This is due

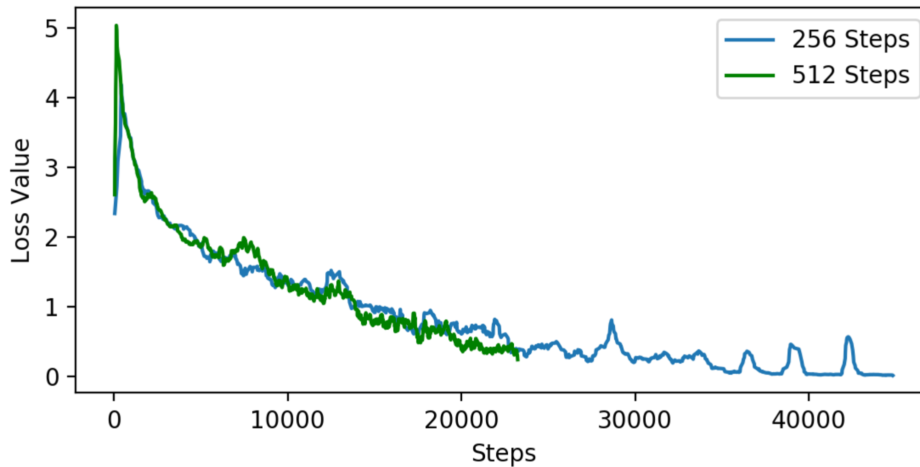


Figure 4.3: Training loss when overfitting the NaiveRNN with a sequence length of 256 and 512 steps. Although the network training on 512 steps might learn depencies across longer time lags, it doesn't seem to increase the efficiency per step.

to the architecture of RNNs and how the calculations are more sequential than compared to CNNs, which might lead to not being able to take full advantage of the computing capabilities of the GPU. In this case and in the case of doubling the amount of units, the model is twice as big as the base configuration, but the difference in training time between these two cases are huge, doubling the amount of steps doubles the training time. This significant increase may not translate into improved results.

The learning efficiency is more clear when decreasing the sequence length. The graphs in figure 4.4 shows that it takes more training steps to converge when performing BPTT on shorter sequences, but even though it takes more steps, these steps are faster to calculate and the model does converge at a shorter amount of time.

Increasing the sequence length to 512 steps might not be a big enough increase to capture essential dependencies within one step of the training algorithm, which is why I continued with shorter sequences.

Increasing the amount of units in each LSTM cell increased the efficiency of each training step. Because of this finding I wanted to keep the shorter sequence length and increase the amount of units to 512.

This configuration, with 2 layers, 512 units and a sequence length of 128, performed close to the base configuration in terms of the amount of training steps it took to converge, but it spent less time training. Even though it used a relatively short sequence length, this configuration was still able to reproduce the input data with great precision, see audio example 2.

Conclusion

This experiment shows that the NaiveRNN doesn't benefit from having a deep structure, the 2 layer configurations performs as well as the 3 layer

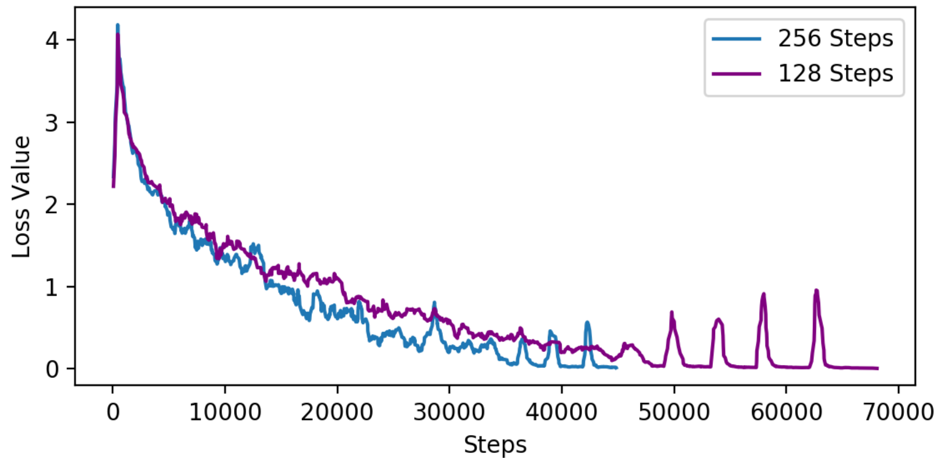


Figure 4.4: Decreasing the sequence length.

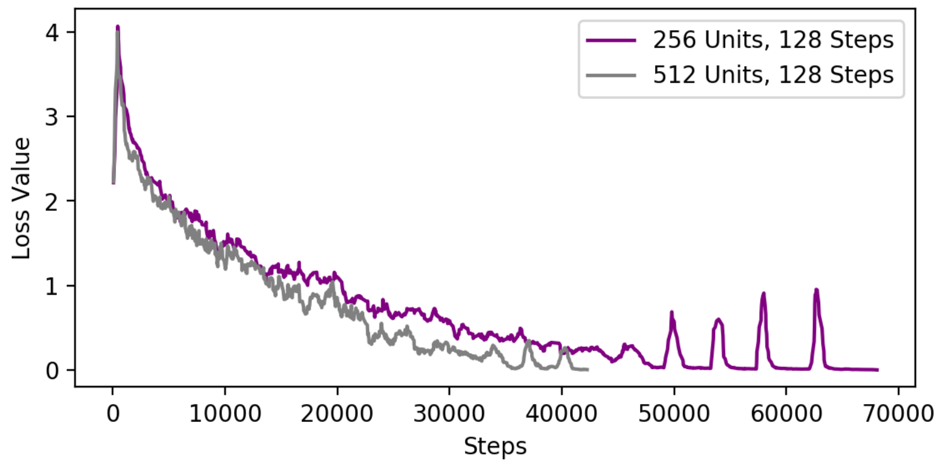


Figure 4.5: Decreasing the sequence length, increasing number of units.

configuration. Having a wider structure, 512 units instead of 256 units, makes each training step more efficient and suggests that the network benefits from having more lower-level feature extractions. The sequence length has an effect on the training and longer sequences tend to be more efficient per training step. But the network was able to learn dependencies across the entire data set when using the shortest sequence length and it did that in the shortest amount of time. As a result, I chose to use the 2-layer, 512-units, 128-steps configuration in the next experiments.

Even though some configurations of parameters perform better than other on this small dataset doesn't mean they are guaranteed to perform better on the big dataset. This experiment only suggests that some parameters might work better than others, however it has allowed us to explore potential relationships between the structure of audio data and the NaiveRNN network.

4.2.2 SampleRNN

To get an understanding of how NaiveRNN compare against SampleRNN I trained SampleRNN on the same data. This will provide an indication of how the training time and efficiency compares between the two models. Mehri et al. [33] conducted a hyper-parameter search and the SampleRNN configuration used in this experiment is based on their findings with some exceptions.

The structure of the network is the same as described in section 3.2. The LFM is made of one LSTM layer with 512 units and an input size of 8. The HFM is made of one LSTM layer with 512 units and an input size of 2. The PM contains three fully connected ReLU layers where the two first layers contain 512 nodes and the last layer contains 256 nodes, equal to the amount of classes in the dataset. The input size of the PM is also 2. To avoid gradients blowing up, all gradients are clipped to remain in the [-1, 1] range. Merhi et al. used 1024 units in both LSTM layers and in the two first layers of the PM, but because I used 512 units in each LSTM layer in the NaiveRNN model I think the comparison between these two models will be more clear when using 512 units in SampleRNN as well. I performed truncated BPTT on sequences of length 512.

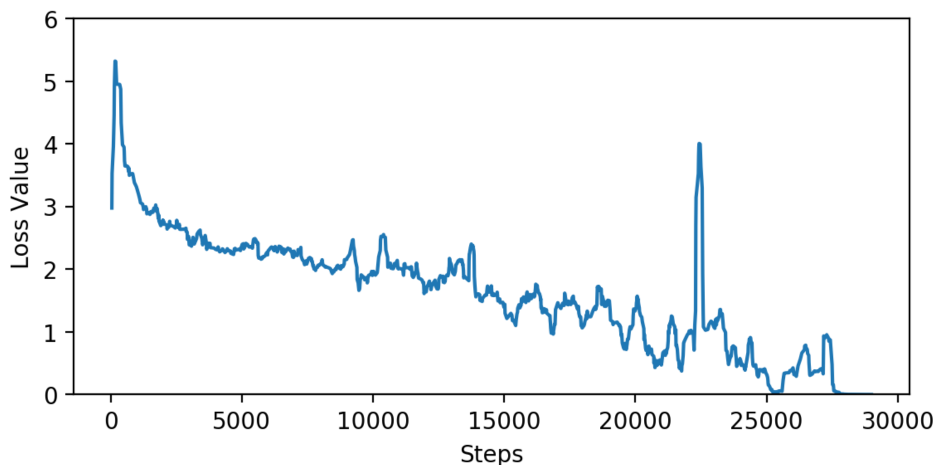


Figure 4.6: Training loss when overfitting SampleRNN. The network converged after about 28000 training steps, which corresponds to 1000 epochs. Y axis is the rolling average loss over 10 training steps.

As expected, also this network converges and is able to generate audio that matches the dataset with good precision, see audio example 3. It took about 28000 training steps for the network to converge, which corresponds to 1000 epochs of the data. Figure 4.7 shows an overview of the original audio in the dataset and the audio generated by SampleRNN. The two waveforms are almost identical. Figure 4.8 shows a detailed view of the same two waveforms, which makes the subtle difference more clear.

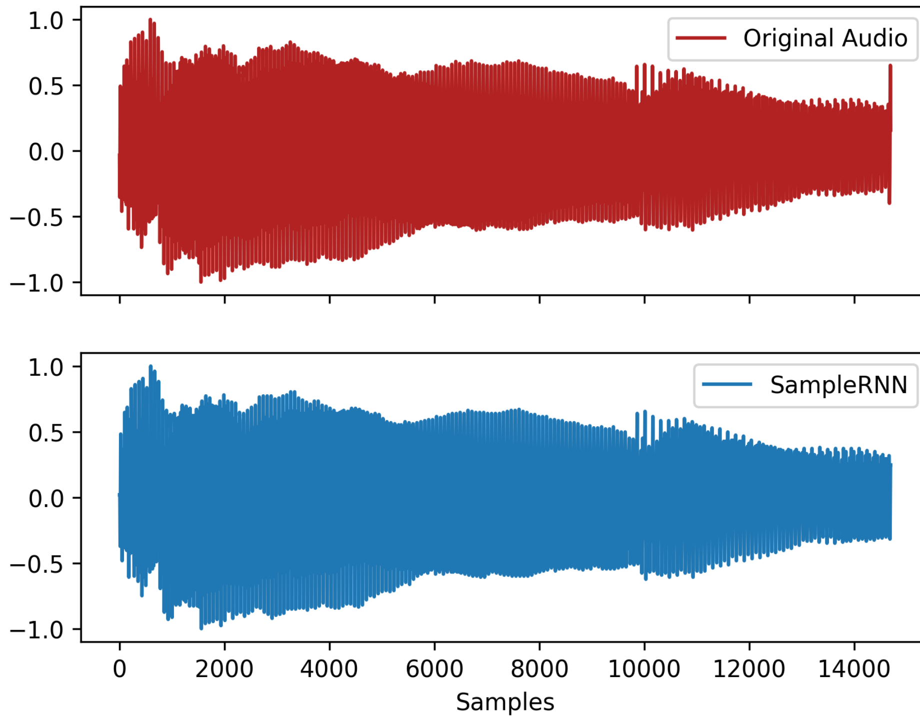


Figure 4.7: Comparing the original audio used to overfit the models and the generated audio by SampleRNN. The generated audio looks identical to the original, although subtle differences makes the generated audio look more smooth, especially at the 8000th sample mark.

4.2.3 WaveNet

In the same way as the two RNN models, I wanted to see how WaveNet handled the small dataset. I tested two configurations with different amount of blocks and layers.

The first configuration I tested was the same as the one described by Mehri et al. [33], using four dilated blocks with ten layers each, giving the model a receptive field of 4096 samples. This model, however, was not able to learn the dataset. Changing the configuration to two blocks with twelve layers would drastically change the networks ability to learn the dataset. The reason for this could be that the configuration has a receptive field of 8192 samples. Both configurations are using 128 filters in each hidden layer and 256 filters in the output layer to match number of classes.

It would appear that the four block, ten layer configuration contains too many parameters and is not able to find the necessary weight updates using this small amount of data. Increasing the dataset could allow this configuration to use all the parameters and learn more details present in the audio samples.

The two block, twelve layer configuration was able to converge after about 2100 training steps, which corresponds to 2100 epochs. This is because WaveNet is able to predict the entire output sample each training

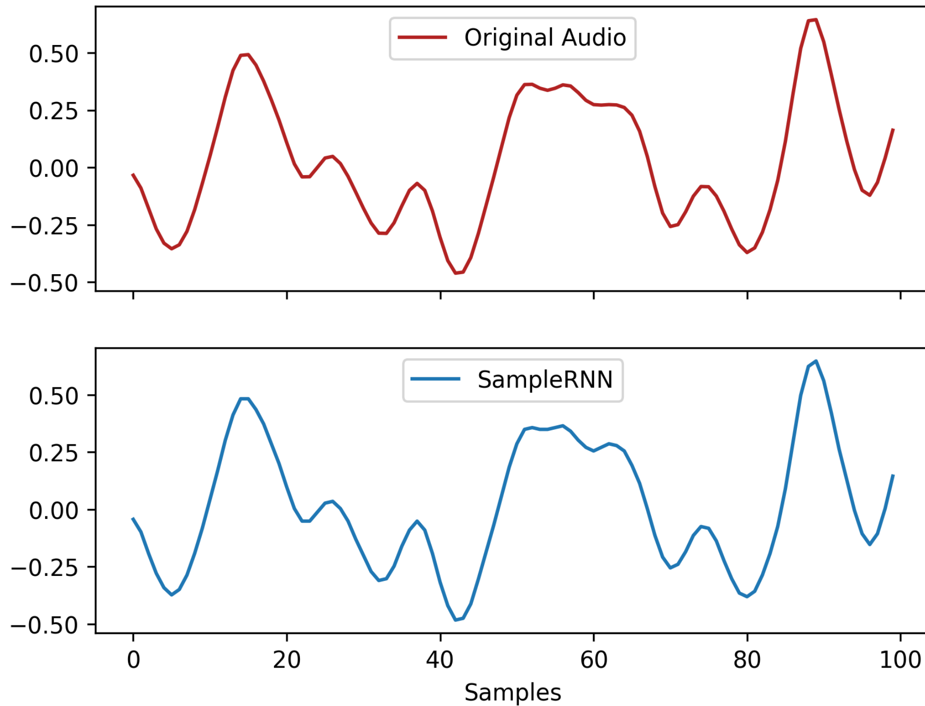


Figure 4.8: A detailed view of the original audio and the generated audio by SampleRNN. It is hard to see any difference in the two waveform, except at the 60th sample mark where there are subtle differences.

step and this makes WaveNet able to train quite quickly. The generated audio is near identical to the original audio, see audio examples 4.

4.2.4 Overfitting Results and Conclusion

To find the an optimal configuration of the NaiveRNN I conducted a search for hyper-parameters where I tested six different configurations of the number of layers, number of hidden units and the sequence length. In order to see how this network performs compared to the other two models I trained those models on the same data.

RNN Models

Both RNN models were able to learn the dataset and the two generated audio tracks are of almost identical quality. Although there are noticeable background noise in the generated audio, a subtle, but constant hiss, this noise is most likely a result of down sampling the original data from 44.1 kHz to 14.7 kHz as well as decreasing the sample resolution.

The NaiveRNN converged after about 41000 training steps compared to SampleRNN's 28000, but although more training steps, this corresponds to about 360 epochs which is less than half the amount of epochs it took SampleRNN to converge. The difference in the amount of training steps is because the two models is trained on different sequence lengths,

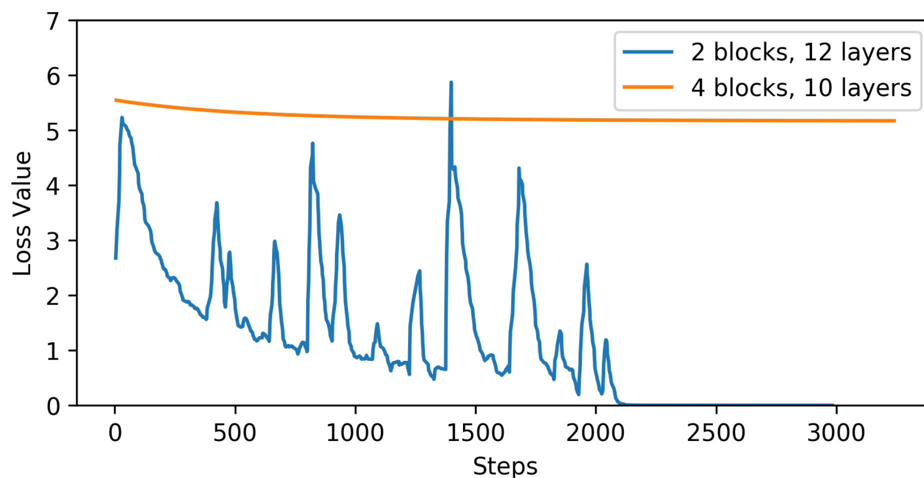


Figure 4.9: Training loss when overfitting WaveNet. Y axis is the rolling average loss over 10 training steps. The network using two blocks with twelve layers was able to converge after about 2100 training steps, although the training process was not very smooth. The network using four blocks with ten layers was not able to learn the dataset.

NaiveRNN was trained on sequences of length 128 while SampleRNN used 512, which makes NaiveRNN have more training steps per epoch. The ratio between the training steps vs epochs suggests that NaiveRNN, the more simple model of the two, responds better to the small dataset.

Increasing the sequence length has a big impact on convergence time. Even though SampleRNN used a sequence length four times as long as NaiveRNN, its convergence time was shorter. This is because RNNs are slow to train. Despite the longer sequence length , the RNN modules in SampleRNN will only have 4 and 256 steps because of the input sizes of 8 and 2 in the LFM and HFM respectively. Using a standard multilayer network in the PM helps reduce the training time. This allows SampleRNN to train on longer sequences, which results in a lower loss [33], while keeping convergence time on par with NaiveRNN.

RNN vs CNN

Also WaveNet was able to learn the dataset and generate audio close to the original track. There are no clear difference between the audio generated by the RNNs and Wavenet.

The difference in architecture between the two RNN models and WaveNet makes a big difference in convergence time. While the two RNN models trained for about an hour each, WaveNet converged in only two minutes time. The causal convolutions allow WaveNet to parallelize its computations, while the sequential architecture of RNNs require a more ordered set of computations. RNNs might not be able to utilize the entire capacity of the GPU beacuse of this, which will result in slower training. This problem is more apparent when training with a small batch size,

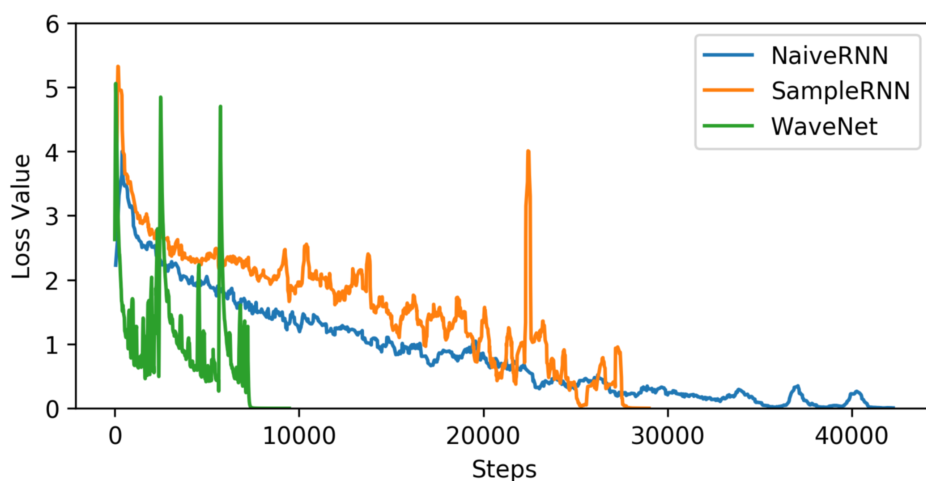


Figure 4.10: Training loss when overfitting all three models. It is clear that WaveNet is more efficient per training step and it allows it to converge faster than the two RNN models.

because increasing the batch size will increase the amount of parallel computations, which will help utilize more of the GPU capacity.

While the two RNN models performs truncated BPTT on small sections of the entire input, WaveNet is able to do backpropagation on the entire input sequence. This allows WaveNet to perform more efficient training steps compared to the RNNs when using the same batch size. Each weight update in a single convolutional layer is the average over the entire input sequence, which might help WaveNet learn more general filters even when the batch size is small. Figure 4.10 displays this effect, which enables WaveNet to converge after only about 7300 training steps. To get the same effect with the RNNs would require an increase in batch size, although it is unlikely to achieve the same efficiency.

4.3 Training on longer sequences

Each model has proven itself able to generate audio as complex as a piano note. The problem is that every network overfitted on its dataset is not able to generate anything else than the original audio in the dataset. This suggests that the networks might not have learned the building blocks beneath the piano timbre and the changes in pitch, which is essential for generating arbitrary piano music. To get a better understanding of the models potential to create a representation of the piano timbre and how it changes with the change in pitch, I train the models with a considerably larger dataset than used to overfit the models.

The size of the dataset used in this experiment is relatively small compared to the datasets used by Mehri et al. [33] and van den Oord et al. [38] in their experiments. Merhi et al. used a collection of Beethoven's piano sonatas adding up to 10 hours of piano music, while van den Oord

et al. used the MagnaTagATune dataset, a total of 200 hours of music, as well as about 60 hours of piano music obtained from YouTube videos. Considering the task at hand, comparing three different techniques of generating digital audio, I consider my dataset, containing forty minutes of piano music , as a sufficient amount of data to explore the potential of the different models.

Each model is trained for about 48 hours on a single GeForce GTX 1080Ti GPU. The dataset is split into mini batches of size 128 which are shuffled after each epoch.

NaiveRNN

Looking at the training loss in figure 4.11 suggests that this task is too difficult for the NaiveRNN. Although the training loss decreases over the first $4e^5$ steps, the progression up to this point suggests that the network is not able to learn any better representations.

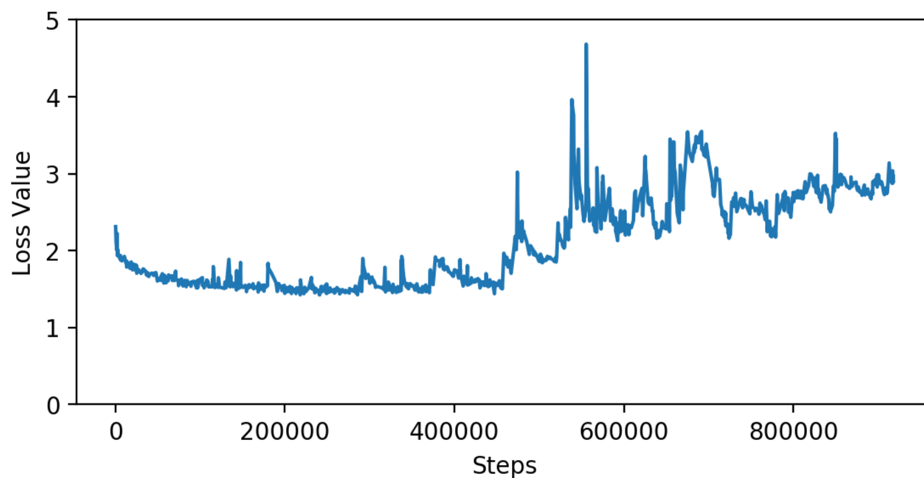


Figure 4.11: Training loss when training the NaiveRNN on the big dataset. The loss value is decreasing up to around step $4e^5$ when the optimiser made a bad decision it could not recover from.

The audio generated by the network after about one million training steps is not what I expected. After feeding the first sample from the first piano piece in the dataset, all preceding predictions was -1.0, which corresponds to class zero. This is unexpected because always predicting class 127, which would be the average sample value over the entire dataset, might have resulted in a lower average loss.

Even though the expectations of this model generating realistic piano music was low, I would have expected it to generate some kind of noise or converging towards the average sample value over the entire dataset.

SampleRNN

Figure 4.12 show the training loss over the entire training session. Although the training loss has a wave-like shape it never converges, which suggests that the network still has more to learn. The training loss is at its lowest at around step 100000.

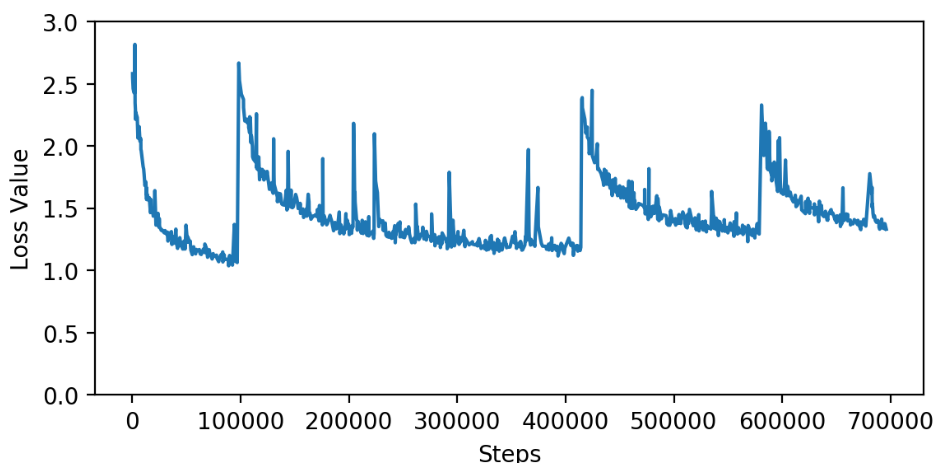


Figure 4.12: Training loss when training SampleRNN on the big dataset.

The audio generate by SampleRNN after $6.75e^5$ training steps is only about 170 ms long before it converges, but it has a timbre which makes it sound like a synthesizer, as well as sounding like it is produced by hitting a key on a piano, see audio example 5. This suggests that the network is starting learn representations of the piano timbre. The timbre is clearly based around multiple frequencies, creating a chord consisting of even frequencies that gives it a more pleasant sound.

Figure 4.13 shows that the generated audio converges towards a value near 0.6. Although the dataset contains audio in the [-1, 1] range, the predicted samples of SampleRNN are only in the [0, 1] range, the center line has shifted. Together with noise created by the network, this might be the reason why the generated audio dies out after only a couple of thousand samples.

Comparing this audio example with the audio example generated at step $5.75e^5$ (audio example 6), right before the last wave, gives us an indication that the network is in fact still learning. The audio at these two points has the same shape, converging after just a few thousand samples, but the amount of details, timbre and present frequencies in the last example, suggests that the network would benefit from more training.

When generating audio, we give the network eight samples as a start token. Usually, these eight samples are equal to zero, but there is no reason why these can not be collected from some other waveform. Using eight samples from the dataset as a start token resulted in a generated audio which were longer, had a more specific pitch and a timbre close to that of a piano, audio examples 7. Although, this was inconsistent and most start

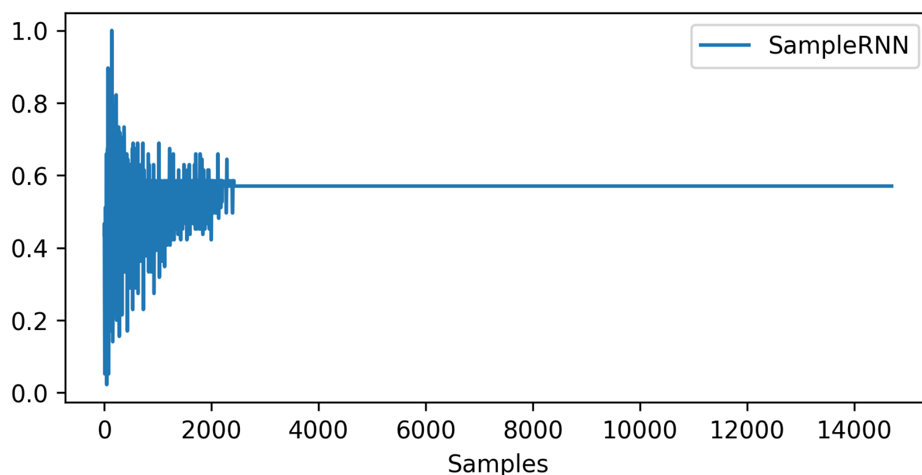


Figure 4.13: Audio generated by SampleRNN. The audio signal dies out after about 2500 generated samples. The zero line seems to have shifted and that could be the cause of the signal dying.

token ended up generating audio similar to audio example 5.

WaveNet

The way WaveNet is constructed makes us have to reconsider the size of each mini batch. The weights in each convolutional layer is updated with the average over the entire input sequence, which means that longer input sequences can result in more general weight updates in the same way as increasing the batch size. The difference in increasing the sequence length, as opposed to increasing the batch size, is that it can lead to weight updates that are better for long term dependencies, which can be essential for generating longer continuous music. Training on many longer sequences at the same time, however, can lead to memory problems due to the size of the input. It is therefor important to balance the length of each sequence and the size of each mini batch to get a good result.

Using the two blocks, twelve layer configuration allows us to train the network using a sequence length of one second and a batch size of 16. Figure 4.14 shows the training loss when training this configuration of WaveNet on the big dataset. Comparing two audio examples generated at different times during training indicates that the network is struggling to learn the necessary connections. Example 8 and 9 are 150k training steps apart, but even though there are differences, the network has not learned anything to generate music. The network has a wide receptive field, but using only two blocks will not be a sufficient amount of parameters to capture the details of the audio samples. The network is not able to generate audio similar to the dataset, it is only generating noise.

The four blocks, ten layer configuration is a considerably larger model than the two blocks, twelve layer configuration. The larger model and a sequence length of one second, gave us only room for a batch size of 8,

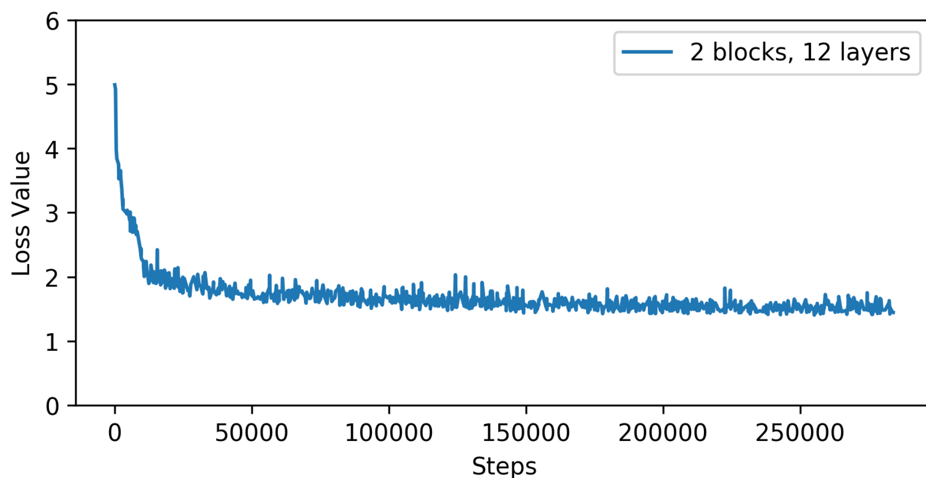


Figure 4.14: Training loss when training the two block, twelve layer WaveNet configuration on the big dataset. Loss value is decreasing throughout the training session, but the generated audio indicates that this configuration is too small to capture the necessary details in the audio samples.

which could be the reason why the learning is not as smooth as with the previous configuration, as shown in figure 4.15. Because the receptive field is only 4096 samples, the added parameters are focused more on short term dependencies and will enable the network to capture more detailed audio structures.

The audio generated by WaveNet is quite noisy and unpleasant, except from the middle part of the example, audio example 10. Figure 4.16 shows the generated audio and we can see there is a longer continuous section of audio from about sample 21k to 37k. This part sounds like there is a person hitting every key on the piano while pressing the sustain pedal. Although it is not musically interesting, this proves that the network is starting to learn what a piano sounds like. Unfortunately, the network made a mistake at the end of the training session, but the audio example is generated at step 325k.

4.3.1 Results

Even though it appears as if the NaiveRNN performed better than SampleRNN in the overfitting experiments, it is now clear that the task of generating arbitrary piano music is too difficult for the NaiveRNN. SampleRNN, on the other hand, seem to respond better in terms of the ability to learn complex internal representations.

The two architectures generates audio which both sounds and behaves quite different. The RNNs can only "see" one step of the sequence and depends on a good flow of the hidden state from one step to the next. If the state weights are not fully optimized, it appear that the generated audio tends to die out. WaveNet, on the other hand, will always be able to look at

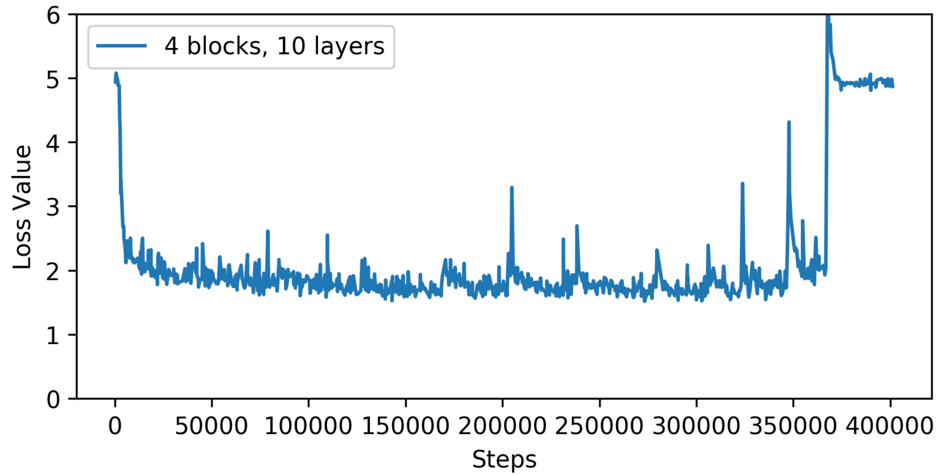


Figure 4.15: Training loss when training WaveNet on the big dataset. The optimiser appears to have made a bad decision at the end of the training session.

a certain number of previously generated samples, and even though these samples might not be similar to the dataset, the audio might not die out in the same way. The problem with this, however, is that the network is not able to evaluate longer dependencies than its receptive field. This can lead to sudden changes in musical style and tonality, or in this case, sudden changes in noise and piano-like sound.

The generation time of each model is also different. The NaiveRNN, being the smallest network, uses the least amount of time generating audio. It takes 16 seconds of GPU time to generate one second of audio on average, compared to SampleRNNs 25 seconds and WaveNets 28 seconds. This means that none of the models are useful in any task involving generating real time audio.

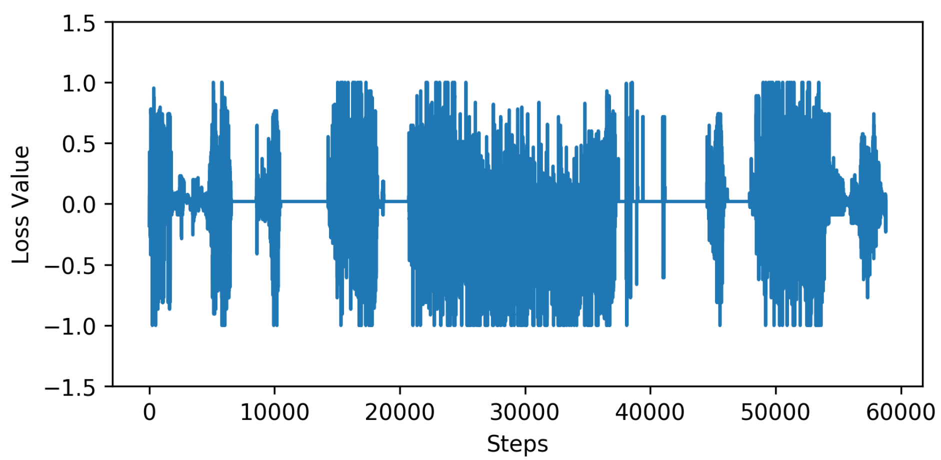


Figure 4.16: Four seconds of generated audio from WaveNet.

Chapter 5

Conclusion

In this thesis I have explored three different techniques for digital audio generation. These techniques are using neural networks of different structure and architecture to generate audio on a sample level. In order to understand how each technique works and what sets them apart from each other I have implemented two RNN networks using Tensorflow and familiarized myself with the implementation of a CNN network.

To gain an understand of how practical and useful these techniques are for music generation, I have conducted experiments where I trained all three neural networks on a dataset containing a large collection of solo piano music and compared the audio generated by each network. It is unclear how these techniques compare against each other and I have for that reason arranged the experiments in such a way that the results from each technique, especially the ones using the RNN architecture, are more comparable.

When trained on a small dataset, all techniques show great potential, although it is unclear if the networks are able to create internal representations that allow them to generate arbitrary piano music. With this idea in mind, I conducted an experiment using a larger dataset which would force each network to rely on their ability to create these internal representations to minimize the cross entropy loss function. After training each network for about 48 hours, none of the networks were able to produce audio similar to the dataset. This supports the initial thoughts about how difficult the task of generating raw digital audio is.

While these results were disappointing, taken together, the experiments show that these techniques could be practical for generating small amounts of digital audio. There are many applications in music that use short audio files as a tool and it is not unlikely that neural networks are going to be used in these applications in the future. While all three models could be successful at these tasks, WaveNet appears to be the most effective as I will discuss in the next section.

5.1 Model Discussion

While SampleRNN [33] and WaveNet [38] are both well documented in their original papers, using basic RNN techniques to generate raw digital audio is not as well documented. In order to find an optimal configuration of hyper-parameters to compare against the other models, I tested six different configurations to gain a better understanding of how each hyper-parameter affects the performance of the NaiveRNN. Because this network is not expected to perform as well as the two other models, I configured the SampleRNN network in a way that would not show its best potential, but rather to better compare against the NaiveRNN. I measured the convergence time of the best NaiveRNN configuration and then configured SampleRNN to match that convergence time.

Efficiency

The different architectures in the three models make them train and learn at different rates. This is more clear in the overfitting experiments where each model converges towards a loss value of zero. The amount of time and training steps it takes for each model to converge might give us an indication of the potential of each model, as well as their learning efficiency.

The structure of SampleRNN allows the network to process longer sequences while keeping the convergence time similar to the NaiveRNN. This will enable SampleRNN to more easily find the long term dependencies. However, the clockwork implementation is complex and time consuming compared to the NaiveRNN implementation.

In the overfitting experiment, WaveNet outperformed the two RNNs. The WaveNet's ability to utilize parallel computations, as well as the fact that it can process the entire input sequence each training step, makes it more efficient than the two RNNs. WaveNet is also easier to implement compared to SampleRNN.

Audio Quality

The audio files generated in the overfitting experiment are, as far as I can hear, identical in terms of audio quality. They all share the same background noise. The audio generated after training for 48 hours are more different. The NaiveRNN did not generate any audio, but SampleRNN and WaveNet generate audio which sounds and behaves quite different. SampleRNN generated audio which were short, but in my opinion it was musically interesting. There are clearly frequencies that work well together, meaning they do not sound harsh or unpleasant. The piano timbre, however, is not quite fully developed. The note generated in example 7 sounds more like an electric piano, which are sounds inspired by the piano, but made using a synthesizer. The WaveNet audio, example 10, was different. It is not musically interesting in the same way as example 7, but it does sound like someone is hitting every key on a piano.

It is hard to say whether SampleRNN and WaveNet are any better then the other in terms of audio quality. SampleRNN should in theory be able to produce audio with longer dependencies due to the fact that WaveNet is only able to produce dependencies as long as its receptive field. There is no defined limit to how far the dependencies can be when using LSTM cells, but the fact that they are hard to train can make them struggle to learn longer dependencies than WaveNet.

Limitations

It is hard to measure the performance of these models. Only measuring the performance by looking at how the loss value changes over time might not be sufficient, because even though the loss value does not seem to decrease in the case of the NaiveRNN and WaveNet, there might be other measures that would better display the learning process. Two potential alternative measures could be the absolute difference between the input and output or one could use cross correlation.

Conditioning the input could have made it easier for the models to generate audio that sounds like piano. The fact that each model have to learn the piano timbre, how the timbre changes with pitch and the higher musical structures makes this problem not solvable within two days of training. Removing one of these variables by for example condition the input on MIDI data would make each model not have to learn the higher musical structures and use more resources on the timbre.

Generating audio using these three models are sequential processes and is not well suited for real time use. While new research have proposed new ways of speeding up audio generation using Parallel WaveNet [37], it is hard to see any use for RNNs in tasks requiring real time audio generation.

The WaveNet's performance is limited by the memory capacity of the GPU. The architecture requires a certain amount of layers to achieve a receptive field big enough to capture long term dependencies, as well as a certain amount of blocks to capture the details of the audio samples. In order to capture long-term dependencies means the network has to train on longer input sequences. Training on many longer sequences simultaneously can lead to memory issues due to the necessary amount of parameters to model audio data.

5.2 Future Work

5.2.1 Limitations

More Training

Considering that the networks struggled to generate audio similar to the dataset suggests that they were not trained long enough to reach their full potential. Of course, the NaiveRNN would probably not benefit from more training because of the limitations of the model. It is hard to say how long

it would take each model to reach a state where they are able to generate realistic piano music, but there are ways to speed up the training process.

Tensorflow supports the use of multiple GPUs and implementing each model to take advantage of this feature would speed up training. One way of to implement this is to make two identical networks sharing the same weights, placing them on separate GPUs and calculating the average loss over the output of both networks when calculating the gradients. A second way is to place each layer, or a number of layers, on a separate GPU. This would enable us to calculate each layer in parallel.

Test More Configurations

The SampleRNN configuration is inspired by the results described in the paper [33], but it could have been interesting to conduct my own hyper-parameter search with SampleRNN in the same way as I did with the NaiveRNN. Testing different input sizes of each module and testing the effect of adding or removing modules are examples of different hyper-parameters that could affect the generative performance of the model.

While the training progresses, the networks could be more sensitive to every weight update. This might be the reason why the loss value will sometimes jump and cause the network to not progress any further. To prevent this from happening, one could lower the learning rate after a period time to make each weight update more precise. The only problem is that it is difficult to know when to lower the learning rate, lowering it too soon could make the network fall into a local minima and stop the learning process, but lowering it too late can make the network take a completely different route which it is not able to recover from.

Data Preprocessing

There are audio processing techniques which could prevent artifacts from appearing when down sampling the audio from 44.1 kHz to 14.7 kHz. These artifacts can create noise and inconsistencies which might cause the models to make bad decisions they cannot recover from. Applying a low pass filter enables us to cut away unwanted frequencies, frequencies that might be the cause of the noise and inconsistencies. Using a sample rate of 14.7 kHz will only allow us to accurately reproduce frequencies up to 7350 Hz, any frequencies above this point will only lower the audio quality.

Normalizing each four seconds long sequences separately causes the center line to shift a small amount. Although it might be an inconsiderable amount, shifting only a couple of quantization steps, having different center lines throughout the dataset is something that could lead to learning mistakes. Using a compressor or a limiter would make sure that highest positive peak amplitude is only as loud as the lowest negative peak amplitude. This would prevent the center line to shift and it would be consistant throughout the entire datset.

5.2.2 Applications

Using neural networks to generate digital audio on a sample level could be applied to many of the musical applications discussed in this thesis. I have selected a few applications I find interesting.

Audio Generation from MIDI Data

Conditioning the audio generation on MIDI data would enable us to train a network to generate a unique performance in the same way as StyleRNN [30]. Generating a performance on a sample level would not be limited to model the dynamics of different musical styles, it would enable us to capture and predict the playing style of specific musicians and as well as the timbre of their instruments. For example, every guitarist have their own unique way of playing a note on the guitar, using their fingers or a pick, and all that information is present in the audio samples. This would make it possible to generate a saxophone solo played with the style of guitarist Jimi Hendrix.

Generation of Unique Sounds

These techniques can be used as a tool for musicians and music technicians to generate sounds never heard before. This is a field of research already being explored [8].

Impulse Responses

Impulse responses are audio files ranging from just a few milliseconds to a few seconds. These audio files are often used to capture the reverb of specific rooms or locations which are then applied, using convolution, to a source audio track. This would give the impression of the source audio actually being recorded at that location. Being audio waveforms, there are no reason why these techniques can not be used to generate impulse responses of different kinds. Because impulse responses are usually short audio files, this could even be an application suitable for the NaiveRNN.

5.3 Final Remarks

Machine learning in music and audio generation is an interesting field of science. Music is always changing and it is difficult to predict what kind of music is going to be popular in the future. There might emerge new genres where all music is composed and generated be artificial intelligence, or musicians use it as a tool to express their creative ideas. We have already seen WaveNet been used in Google Duplex [28], an AI system for booking appointments using the phone. These are conversations made entirely using machine learning and the conversations sound remarkably human. It is exciting to see what machine learning and artificial intelligence will bring in the future.

I feel I have learned a lot about what it means to apply machine learning to a difficult problem, the process of finding a dataset containing the kind of data the problem is using and the long process of training each model for days. The actual amount of training necessary to generate something other than noise was surprising to me. This made me think it was something wrong with the way I had implemented the models or the way I processed the data. Writing this thesis have made me more aware of how to approach a machine learning task of this complexity, I only wish the awareness had come earlier in the process.

Bibliography

[1] Charles Ames. "The Markov Process as a Compositional Model: A Survey and Tutorial". In: *Leonardo* 22.2 (1989), pp. 175–187. ISSN: 0024094X, 15309282. URL: http://www.jstor.org/stable/1575226.

[2] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. "Multiple Object Recognition with Visual Attention". In: *CoRR* abs/1412.7755 (2014). arXiv: 1412.7755. URL: http://arxiv.org/abs/1412.7755.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[4] Oliver Bown and Sebastian Lexer. "Continuous-Time Recurrent Neural Networks for Generative and Interactive Musical Performance". In: *Applications of Evolutionary Computing*. Ed. by Franz Rothlauf et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 652–663. ISBN: 978-3-540-33238-1.

[5] San-kuen Chan, Jennifer Yuen, and Andrew B Horner. "Discrete summation synthesis and hybrid sampling-wavetable synthesis of acoustic instruments with genetic algorithms". In: *Proceedings of the 1996 International Computer Music Conference, Hong Kong, August 19-24, 1996, Hong Kong University of Science and Technology, Hong Kong*. 1996.

[6] Florian Colombo, Alexander Seeholzer, and Wulfram Gerstner. "Deep Artificial Composer: A Creative Neural Network Model for Automated Melody Generation". In: *Computational Intelligence in Music, Sound, Art and Design*. Ed. by João Correia, Vic Ciesielski, and Antonios Liapis. Cham: Springer International Publishing, 2017, pp. 81–96. ISBN: 978-3-319-55750-2.

[7] Emily L. Denton et al. "Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks". In: *CoRR* abs/1506.05751 (2015). arXiv: 1506.05751. URL: http://arxiv.org/abs/1506.05751.

[8] Jesse Engel et al. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders". In: *CoRR* abs/1704.01279 (2017). arXiv: 1704.01279. URL: http://arxiv.org/abs/1704.01279.

[9] L.V. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice-Hall international editions. Prentice-Hall, 1994. ISBN: 9780133341867. URL: https://books.google.no/books?id=ONylQgAACAAJ.

[10] "Genetic algorithms and computer-assisted music composition". In: *Urbana*. 51.61801 (1991). ISSN: 0798-0523.

[11] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM". In: (1999).

[12] Pavel Golik, Patrick Doetsch, and Hermann Ney. "Cross-entropy vs. squared error training: a theoretical and experimental comparison." In: *Interspeech*. Vol. 13. 2013, pp. 1756–1760.

[13] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[14] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[15] Alex Gounaropoulos and Colin Johnson. "Synthesising timbres and timbre-changes from adjectives/adverbs". In: *Workshops on Applications of Evolutionary Computation*. Springer. 2006, pp. 664–675.

[16] Alex Graves. "Generating sequences with recurrent neural networks". In: *arXiv preprint arXiv:1308.0850* (2013).

[17] John M. Grey. "An Exploration of Musical Timbre". MA thesis. Stanford, California: Stanford University, 1975. URL: https://ccrma.stanford.edu/files/papers/stanm2.pdf.

[18] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[19] Lejaren A. Hiller and Robert A. Baker. "Computer Cantata: A Study in Compositional Method". In: *Perspectives of New Music* 3.1 (1964), pp. 62–90. ISSN: 00316016. URL: http://www.jstor.org/stable/832238.

[20] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[21] Peter Hoffmann. "Towards an "automated art": Algorithmic processes in Xenakis' compositions". In: *Contemporary Music Review* 21.2-3 (2002), pp. 121–131. DOI: 10.1080/07494460216650. eprint: https://doi.org/10.1080/07494460216650. URL: https://doi.org/10.1080/07494460216650.

[22] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[23] David Miles Huber. *Modern recording techniques*. eng. Boston, 2005.

[24] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[25] Jan Koutnık et al. "A Clockwork RNN". In: *CoRR* abs/1402.3511 (2014). arXiv: 1402.3511. URL: http://arxiv.org/abs/1402.3511.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[27] Bernd Krueger. *Classical Piano Midi Page*. [Piano music archive available online]. 2016. URL: http://www.piano-midi.de.

[28] Yaniv Leviathan and Yossi Matias. *Google Duplex: An AI System for Accomplishing Real-World Tasks Over the Phone*. [Post on Google's AI Blog]. May 2018. URL: https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html.

[29] Cristyn Magnus. "Evolutionary Sound: a Non-Symbolic Approach to Creating Sonic Art With Genetic Algorithms". In: (2010).

[30] Iman Malik and Carl Henrik Ek. "Neural Translation of Musical Style". In: *CoRR* abs/1708.03535 (2017). arXiv: 1708.03535. URL: http://arxiv.org/abs/1708.03535.

[31] Charles P. Martin and Jim Tørresen. "RoboJam: A Musical Mixture Density Network for Collaborative Touchscreen Interaction". In: *CoRR* abs/1711.10746 (2017). arXiv: 1711.10746. URL: http://arxiv.org/abs/1711.10746.

[32] Stephen McAdams et al. "Perceptual scaling of synthesized musical timbres: Common dimensions, specificities, and latent subject classes". In: *Psychological research* 58.3 (1995), pp. 177–192.

[33] Soroush Mehri et al. "SampleRNN: An Unconditional End-to-End Neural Audio Generation Model". In: *CoRR* abs/1612.07837 (2016). arXiv: 1612.07837. URL: http://arxiv.org/abs/1612.07837.

[34] MICHAEL C. MOZER. "Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Multi-scale Processing". In: *Connection Science* 6.2-3 (1994), pp. 247–280. DOI: 10.1080/09540099408915726. eprint: https://doi.org/10.1080/09540099408915726. URL: https://doi.org/10.1080/09540099408915726.

[35] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel recurrent neural networks". In: *arXiv preprint arXiv:1601.06759* (2016).

[36] Aäron van den Oord et al. "Conditional Image Generation with PixelCNN Decoders". In: *CoRR* abs/1606.05328 (2016). arXiv: 1606.05328. URL: http://arxiv.org/abs/1606.05328.

[37] Aäron van den Oord et al. "Parallel WaveNet: Fast High-Fidelity Speech Synthesis". In: *CoRR* abs/1711.10433 (2017). arXiv: 1711.10433. URL: http://arxiv.org/abs/1711.10433.

[38] Aäron van den Oord et al. "WaveNet: A Generative Model for Raw Audio". In: *CoRR* abs/1609.03499 (2016). URL: http://arxiv.org/abs/1609.03499.

[39] Tom Le Paine. *Fast Wavenet Source Code*. Dec. 2016. URL: https://github.com/tomlepaine/fast-wavenet.

[40] Tom Le Paine et al. "Fast Wavenet Generation Algorithm". In: *arXiv preprint arXiv:1611.09482* (2016).

[41] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[42] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[43] Elliot Wait. *Generator Long-Term Structure in Songs and Stories*. Aug. 2016. URL: https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn.

[44] Paul J Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[45] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: *CoRR* abs/1609.08144 (2016). arXiv: 1609.08144. URL: http://arxiv.org/abs/1609.08144.

[46] Li-Chia Yang, Szu-Yu Chou, and Yi-Hsuan Yang. "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions". In: *CoRR* abs/1703.10847 (2017). arXiv: 1703.10847. URL: http://arxiv.org/abs/1703.10847.

Appendix A

List of Audio Examples

All audio files can be downloaded from https://doi.org/10.5281/zenodo.1247475

1. NaiveRNN-2-256-256.wav
2. NaiveRNN-2-512-128.wav
3. SampleRNN-overfit.wav
4. WaveNet-2-12-overfit.wav
5. SampleRNN-675k.wav
6. SampleRNN-575k.wav
7. SampleRNN-575k-starttoken.wav
8. WaveNet-2-12-100k.wav
9. WaveNet-2-12-250k.wav
10. WaveNet-4-10.wav