LAURA CABELLO PIQUERAS

AUTOREGRESSIVE MODEL BASED ON A DEEP CONVOLUTIONAL NEURAL NETWORK FOR AUDIO GENERATION

Master of Science thesis

Examiner: Tuomas Virtanen, Heikki Huttunen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 7th December 2016

ABSTRACT

LAURA CABELLO PIQUERAS: Autoregressive model based on a deep convolutional neural network for audio generation
Tampere University of Technology
Master of Science thesis, 54 pages
March 2017
Master's Degree in Telecommunication Engineering
Major: Signal Processing
Examiner: Tuomas Virtanen, Heikki Huttunen
Keywords: audio generation, convolutional neural network, deep learning

The main objective of this work is to investigate how a deep convolutional neural network (CNN) performs in audio generation tasks. We study a final architecture based on an autoregressive model of deep CNN that operates directly at the waveform level.

In first place, we study different options to tackle the task of audio generation. We define the best approach as a classification task with one-hot encode data; generation is based on sequential predictions: after next sample of an input sequence is predicted, it is fed back into the network to predict the next sample.

We present the basics of the preferred architecture for generation, adapted from *WaveNet* model proposed by *DeepMind*. It is based on dilated causal convolutions which allows an exponential growth of the receptive field size with depth of the network. Bigger receptive fields are desirable when dealing with temporal sequences since it increases the model capacity to model temporal correlations at longer timescales.

Due to the lack of an objective method to assess the quality of new synthesized signals, we firstly test a wide range of network settings with pure tones so the network is capable to predict the same sequences. In order to overcome the difficulties of training a deep network and to accelerate the research adjusted to our computational resources, we constrain the input database to a mixture of two sinusoids within an audible range of frequencies. In generation phase, we acknowledge the key role of training a network with a large receptive field and large input sequences. Likewise, the amount of examples we feed to the network every training epoch exert a decisive influence in any studied approach.

CONTENTS

TERMS AND DEFINITIONS

ANN	Artificial Neural Network
CE	Cross Entropy
CNN	Convolutional Neural Network
DNN	Deep Neural Network
FNN	Feedforward Neural Network
GPU	Graphics Processor Unit
HMM	Hidden Markov Model
LSTM	Long Short-Term Memory
MLP	Multilayer Perceptron
MSE	Mean Squared Error
PDF	Probability Density Function
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SPSS	Statistical Parametric Speech Synthesis
NN	Neural Network

1. INTRODUCTION

Breakthroughs in machine learning over the last decade lead us to a new era of artificial intelligence. Nowadays computers can learn. But not only that, they can potentially understand the world around us. In the conventional approach to programming, we tell the computer what to do, breaking big problems up into many small, precisely defined tasks that the computer can easily execute. By contrast, in a neural network (NN) we do not tell the computer how to solve our problem. Instead, it learns patterns from observational data and figure out its own solution to the current problem.

Automatically learning from data sounds promising. However, until 2006 we did not know how to train neural networks to outperform more traditional approaches, except for a few specialized problems. In 2006 Hinton and Salakhutdinov [15] proposed a layer-wise pre-training which favored this recent surge of popularity and an introduction to new techniques of learning known as *deep learning*. This incredible revival of neural networks within *deep learning* field in the past five to ten years is partly due to improvements of mathematical algorithms, partly because we have much more data, but a big part is thanks to the advances in computational resources and the decrease in the price of powerful GPUs.

Deep learning can be summed up as a sub field of machine learning studying statical models called deep neural networks. The latter are able to learn complex and hierarchical representations from raw data, extracting new set of features that enhance traditional, hand crafted models. They have been further developed and today deep neural networks achieve outstanding performance on diverse tasks such as computer vision, speech recognition or natural language processing.

Thus, larger and deeper architectures are trained on bigger databases to achieve better performance every year. It is worth to highlight *AlexNet*, a deep convolutional neural network (shorted as CNN or ConvNet) developed in 2012 by Krizhevsky, Sutskever and Hinton [23]. *AlexNet* became a milestone in the use of deep CNNs for image classification and ever since then they are widely used in a wide range of contexts. Despite they were firstly intended to work with images as input data, language processing [5, 6] or audio modeling [14, 42] -where audio generation is included- are some of the last applications that benefit from CNN properties.

Audio generation aims to give a machine the ability to compose new pieces of audio. New compositions must be meaningful accordingly to the purpose of generation: compelling piano melodies, realistic jazz rhythms or simply sounds that are pleasant to listen to if that is what we are looking for.

Many studies have been conducted for the analysis and generation of musical sequences [19, 28, 29]. The handling of memory and computational cost are core challenges in music modeling. Whereas the widely used bag-of-features approach, based on haphazard collections of local data descriptors, neglects any sequential relation between musical events, common N-gram based methods for the representation of musical sequences usually set a maximal fixed length of context [24]. This leads to exponentially growing storage needs to allow the model to account for more complex structures. The solution offered by *WaveNet* model [42] handles larger input sequences than traditional methods without greatly increasing computational cost. It is based on a deep convolutional neural network that combines causal filters with dilated convolutions to allow its receptive field to grow exponentially with depth, which is important to model the long-range temporal dependencies in audio signals.

Motivated by the recent success of deep CNNs, in this work we decided to analyze its performance on audio generation tasks. Taking *WaveNet* architecture as reference for generation, we study how to tackle the problem of raw audio generation and the implication of different hyperparameters of the network. Finally, the quality of the synthesis reveals whether the methods used to generate new waveforms have been adequate.

This thesis is structured as follows. Chapter 2 serves as an introduction to machine learning and neural networks, as well as presents the theoretical background necessary to understand the research and methodology accomplished in this work. Chapter 3 describes the methodology and process followed in our study with the ultimately objective of achieve a good generation. The experimental cases and results of testing a set of proposed configurations are presented in Chapter 4. Finally, Chapter 5 gathers concluding remarks and a proposal for further research.

2. BACKGROUND

This chapter reviews the general theory needed later. A deep convolutional neural network (CNN or ConvNet) is the approach that we study in this work to predict and generate audio signals. Hence, this chapter first talks about the basic concept of artificial neural networks (ANNs, or simply NNs), with a special focus on CNNs and its architecture, and then introduce the concept of deep learning; finally the basis of audio generation are presented.

The area of study of NNs was originally inspired by the goal of modeling biological neural systems, but has since diverged and become a matter of engineering and achieving good results in machine learning tasks. Machine learning is viewed as a programming paradigm which allows a computer to learn from observational data. In computer science, a NN is an instance of machine learning and it is frequently described as a computing system made up of simple, highly connected processing elements which processes information by its dynamic state response to external input [3].

Hence, an NN's topology can be described as a finite subset of simple processing units (called *nodes* or *neurons*) and a finite set of weighted connections between nodes that scale the strength of the transmitted signal, mimicking synapses in the human brain. The behavior of the network is determined by a set of real-valued, modifiable parameters or *weights* $\mathbf{W}=\{\mathbf{w_1},\ \mathbf{w_2},\ \mathbf{...}\}$ which are tuned in every event, known as *epoch*, of the training process. Neurons in the network are grouped into *layers*. There is one input layer, a variable number of hidden layers that perform intermediate computations and one output layer.

Supervised and unsupervised learning Neural networks do not follow the conventional approach to programming, where we tell the computer what to do, breaking big problems up into many smaller tasks that the computer can easily perform. By contrast, a neural network learns itself from observational data, figuring out its own solution to a current problem. [30]

Typically, the network reads an input $\mathbf{x}$ and associate one or more labels $\mathbf{y}$. If the network predicts a label for new unseen data, we say it performs a *classification* task. When a database has a sufficient amount of pairs ($\mathbf{x}$,$\mathbf{y}$), we can make a computer learn how to classify new unseen data by training it on the known instances from the database. It is the so-called *supervised learning*, that try to find patterns in data as useful as possible to predict the labels.

Hence, it is desirable the network learns to classify new unseen instances and not only the training set. We want to prevent our model from overfitting, i.e., from memorizing training pairs instead of generalizing patterns to any example. A classic methodology to ensure the model has not overfitted is to test it on unseen data whose labels are known and evaluate the accuracy.

In contrast to supervised learning, *unsupervised learning* is another type of machine learning technique that learns patterns in data without neither label information nor an specific prediction task.

2.1 Perceptron

In order to understand how neurons and NNs work, it is worth to introduce first the baseline unit for modern research: the perceptron. Perceptron was defined in 1957 by the scientist Frank Rosenblatt [34], inspired by earlier work by Warren McCulloch and Walter Pitts [27]. A perceptron takes several binary inputs and combines them linearly to produce a single binary output. Figure 2.1 depicts a perceptron with several inputs $\{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}\} \in \mathbb{R}$. Rosenblatt proposed a simple rule to compute the output: the neuron's output, 0 or 1, is determined whether the weighted sum is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. Equation 2.1 defines it in algebraic terms:

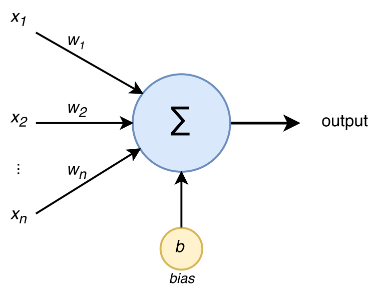


Figure 2.1 *Model of a perceptron.*

$$
output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold}, \end{cases} \tag{2.1}
$$

where it is easy to infer that by varying the weights and the threshold we can get different models of decision-making. However, Equation 2.1 can be simplified making two notational changes. First, both inputs and weights can be seen as vectors $[\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}]^T$ and $\mathbf{w}$ respectively, which allows us to rewrite the summation as a dot product. The second change is to move the threshold to the other side of the inequality, and replace it by what is known as the perceptron's bias, $b \equiv -threshold$. The bias can be seen as a measure of how easy is to get the perceptron to output a 1 [30]. Thus, the perceptron rule can be rewritten into Equation 2.2:

$$
output = \begin{cases} 0 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \end{cases} \tag{2.2}
$$

We can devise a network of perceptrons that we would like to use to learn how to solve a problem. For instance, the inputs to the network might be the raw audio from a soundtrack. And we want the network to learn weights and biases so that the output from the network correctly classifies the chord that is being played one at a time. We can now devise a learning algorithm which can automatically tune the weights and biases to get our network to behave in the manner we want after several epochs. The learning algorithm gradually adjusts the weights and biases in response to external stimuli, without direct intervention by a programmer.

The problem is that this is not possible if our network contains perceptrons, since a small change in the weights (or bias) of any single perceptron in the network could cause the output of that perceptron to completely flip, say from 0 to 1. And that flip may then cause the behavior of the rest of the network to entirely change in some very complicated way [30].

It is possible to overcome this problem by introducing new types of neurons with a nonlinear behavior, which lead us to introduce a new concept: activation functions. The main purpose of nonlinear activation functions is to enable the use of nonlinear classifiers.

2.2 Activation Function

An activation function scales the activation of a neuron into an output signal. Any function could serve as an activation function, however there are few activation functions commonly used in NNs:

- *Sigmoid Function.* This is a smooth approximation of the step function used in perceptrons. It is often used for output neurons in binary classification tasks, since the output is in the range [0,1]. It is sometimes referred to as *logistic function.* Mathematically,

$$\sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.3}$$

- *Rectified Linear Unit (ReLU).* This function avoids saturation problems and vanishing gradients, two of the major problems that arise in deep networks. It is depicted in red in Figure 2.2, where we can see how ReLU grows unbounded for positive values of x,

$$ReLU(x) = max(0, x). \tag{2.4}$$

- *Hyperbolic Tangent (tanh).* This function is used as an alternative to the sigmoid function. Hyperbolic tangent is vertically scaled to output in the range [-1,1]. Thus, big negative inputs to the *tanh* will map to negative outputs and only zero-valued inputs are mapped to zero outputs. These properties make the network less likely to get stuck during training, which could be possible with *sigmoid function* for strongly negative inputs. Mathematically,

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.5}$$

2.3 Neural Networks

Multilayer perceptrons (MLP) constitute one of the simplest type of feedforward NNs (FNNs) and the most popular network for classification and regression [13]. An MLP consists of a set of source nodes forming the input layer, one or more hidden layers of computation nodes, and an output layer. Figure 2.3 depicts the architecture of an MLP with a single hidden layer.

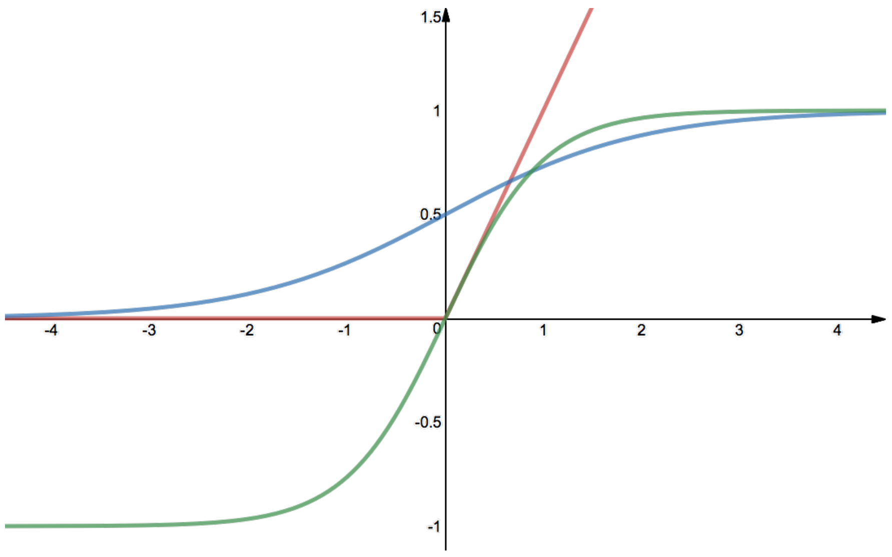


Figure *2.2 Visual representation of sigmoid (blue), rectified linear unit (ReLU, red) and hyperbolic tangent (tanh, green) activation functions. It can be seen that sigmoid and tanh are both bounded functions.*

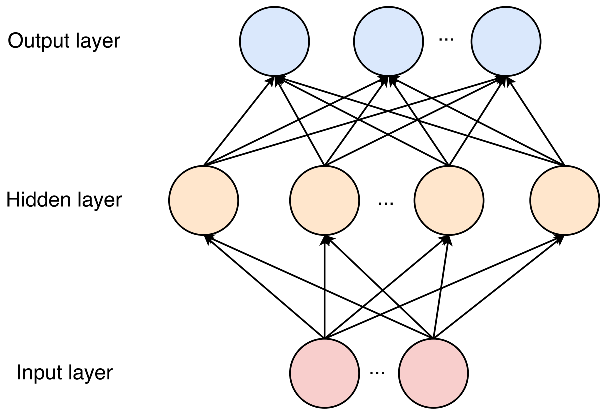


Figure *2.3 Signal-flow graph of an MLP with one hidden layer. Output layer computes a linear operation.*

For an input vector **x**, each neuron computes a single output by forming a linear combination according to its input weights and then, possibly applying a nonlinear activation function. The computation performed by an MLP with a single hidden layer with a linear output can be written mathematically as:

$$\widehat{\mathbf{y}} = \mathbf{W}^{\mathbf{hy}} \cdot \Phi(\mathbf{W}^{\mathbf{xh}}\mathbf{x} + \mathbf{b}^{h}) + \mathbf{b}^{y}, \tag{2.6}$$

where, in vector notation, $\mathbf{W}{**}$ denotes the weight matrices connecting two layers, i.e., $\mathbf{W}^{xh}$ are the weights from input to hidden layer and $\mathbf{W}^{hy}$ from hidden to output layer, $\mathbf{b*}$ are the bias vectors, and the function $\Phi(\cdot)$ is an element-wise non-linearity.

The power of an MLP network with only one hidden layer is surprisingly large. As Hornik et al. and Funahashi showed in 1989 [17, 9], such networks, like the one in Equation 2.6, are capable of approximating any continuous function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ to any given accuracy, provided that sufficiently many hidden units are available.

For an input $\mathbf{x}$ a prediction $\hat{\mathbf{y}}$ is computed at the output layer, and compared to the original target $\mathbf{y}$ using a *cost function* $E(\mathbf{W}, \mathbf{b}; \mathbf{x}, \mathbf{y})$, or just E for simplicity. The network is trained to minimize E for all input samples $\mathbf{x}$ in the training set, formally:

$$E(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{N} E(\mathbf{W}, \mathbf{b}; \mathbf{x}_n, \mathbf{y}_n) \tag{2.7}$$

where N is the number of training samples. Since the *cost function* (also known as *loss* or *objective function*) is a measure of how well our network did to achieve its goal in every epoch, it is a single value. *Mean squared error (MSE)* and *cross entropy* [1] ($H(p, q)$ with p and q two probability distributions) are among the most common cost function to train NNs for classification tasks:

$$E_{MSE} = \frac{1}{N} \sum_{n=1}^{N} \|y_n - \widehat{y_n}\|^2 \tag{2.8}$$

$$E_{CE} = \frac{1}{N} \sum_{n=1}^{N} H(p_n, q_n) = -\frac{1}{N} \sum_{n=1}^{N} y_n \log \widehat{y_n} + (1 - y_n) \log(1 - \widehat{y_n}). \tag{2.9}$$

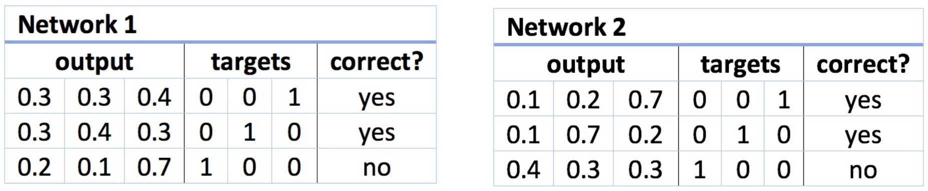
Furthermore, categorical cross entropy is a more granular way to compute error in multiclass classification tasks than simply accuracy or classification error. Let us consider the following example to endorse this statement. Suppose we have two neural networks working on the same problem whose outputs are the probability of belonging to each class, shown in Figure 2.4. We choose the class with the highest probability as the solution and then compare it with the known right answer (targets); since both networks classified two items correctly, both present a classification error of $1/3 = 0.33$ and thus, same accuracy. However, while the first network barely classify the first two training items (similar probabilities among all of them), the

[1] In information theory, the entropy of a random variable is a measure of the variability associated with it. Shannon defined the entropy of a discrete random variable X as: $H(X) = -\sum_x \mathbb{P}(X = x) \log \mathbb{P}(X = x)$. From this definition we can deduce straightforward the entropy between two variables (*cross entropy*).

second network distinctly gets them correct. Should we consider now the average cross entropy error for every network,

$$\begin{cases} E_{CE}^{1} = -(\log(0.4) + \log(0.4) + \log(0.1))/3 = 1.38, \\ E_{CE}^{2} = -(\log(0.7) + \log(0.7) + \log(0.3))/3 = 0.64, \end{cases} \tag{2.10}$$

we can notice that the second network has a lower value which indicates it actually performed better. The $log()$ in cross entropy takes into account the closeness of a prediction.



Network 1						
output			**targets**			**correct?**
0.3	0.3	0.4	0	0	1	yes
0.3	0.4	0.3	0	1	0	yes
0.2	0.1	0.7	1	0	0	no

Network 2						
output			**targets**			**correct?**
0.1	0.2	0.7	0	0	1	yes
0.1	0.7	0.2	0	1	0	yes
0.4	0.3	0.3	1	0	0	no

Figure 2.4 *Example of two networks' output for the same classification problem with three training samples and three different classes. Networks output the probability of belonging to each class; the class with the highest probability is chosen as the solution and compared to the target to decide whether it is correct or not.*

NNs are constructed as differentiable operators and they can be trained to minimize the differentiable cost function using *gradient descent* based methods. An efficient algorithm widely used to compute the gradients for all the weights in the network is the *backpropagation* algorithm, an implementation of the chain rule for partial derivatives along the network. The *backpropagation* algorithm is the most popular learning rule for performing supervised learning tasks [7] and it was proposed for the MLP model in 1986 by Rumelhart, Hinton, and Williams [35]. Later on, the *backpropagation* algorithm was discovered to have already been invented in 1974 by Werbos [44].

Due to *backpropagation*, MLP can be extended to many hidden layers. In order to understand how the algorithm works, we will use the following notation: Φ' is the first derivative of the activation function Φ; w_{ji}^{l} is the weight connecting the i^{th} neuron in the layer $l-1$ to the j^{th} neuron in the layer l; z_j^l is the weighted input to the j^{th} neuron in layer l, expressly:

$$z_j^l = \sum_i w_{ji}^l \Phi(z_i^{l-1}) + b_j^l = \sum_i w_{ji}^l h_i^{l-1} + b_j^l, \tag{2.11}$$

where h_i^{l-1} is the activation of the i^{th} neuron in the layer $l-1$. The cost function can be minimized by applying the gradient descent procedure. It requires to compute the derivative of the cost function with respect to each of the weights and bias terms in the network., i.e., $\frac{\partial E}{\partial w_{ji}^l}$ and $\frac{\partial E}{\partial b_j^l}$. Once these gradients have been computed, the corresponding parameters in the network can be updated by taking a small step towards the negative direction of the gradient. Should we use stochastic gradient descent (SGD),

$$w \equiv w - \eta \nabla E(w), \tag{2.12}$$

the weights are updated via the following:

$$\Delta w_i(\tau + 1) = -\eta \nabla E(w_i) = -\eta \frac{\partial E}{\partial w_i}, \tag{2.13}$$

where τ is the index of training iterations (epochs); η is the *learning rate* and it can be either a fixed positive number or it may gradually decrease during the epochs of the training phase. The same update rule applies to the bias terms, with b in place of w.

Backpropagation is a technique that efficiently computes the gradients for all the parameters of the network. Unfortunately, computing $\frac{\partial E}{\partial w_{ji}^l}$ and $\frac{\partial E}{\partial b_j^l}$ is not so trivial. For MLP, the relationship between the error term and any weight anywhere in the network needs to be calculated. This involves propagating the error term at the output nodes backwards through the network, one layer at a time. First, for each neuron j in the output layer L an error term δ_j^l is computed:

$$\delta_j^L \equiv \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial h_j^L} \frac{\partial h_j^L}{\partial z_j^L} \tag{2.14}$$

We can then compute the backpropagated errors δ_j^l at the l^{th} layer in terms of the backpropagated error δ_j^{l+1} in the next layer applying the chain rule:

$$\delta_j^l \equiv \frac{\partial E}{\partial z_j^l} = \sum_i \frac{\partial E}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial z_j^l}. \tag{2.15}$$

The first factor of Equation 2.15 can be rewritten directly from definition in 2.14 as

$$\frac{\partial E}{\partial z_i^{l+1}} \equiv \delta_i^{l+1}, \tag{2.16}$$

the second factor in Equation 2.15 can be derived using Equation 2.11

$$\frac{\partial z_i^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l} \sum_i w_{ij}^{l+1} h_j^l + b_i^{l+1} = \sum_i w_{ij}^{l+1} \Phi'(z_j^l), \tag{2.17}$$

hence, we can simplify Equation 2.15

$$\frac{\partial E}{\partial z_j^l} = \sum_i \delta_i^{l+1} w_{ij}^{l+1} \Phi'(z_j^l). \tag{2.18}$$

Finally, the gradients can be expressed in terms of the error δ_j^l

$$\frac{\partial E}{\partial w_{ji}^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{ji}^l} = h_i^{l-1} \delta_j^l \tag{2.19}$$

$$\frac{\partial E}{\partial b_j^l} = \frac{\partial E}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{2.20}$$

Note that all weights and bias must be initialized to give the algorithm a place to start from. The values are typically drawn randomly and independently from uniform or Gaussian distributions.

The SGD, defined in Equation 2.12, is convergent in the mean if $0 < \eta < \frac{2}{\lambda_{max}}$, where λ_{max} is the largest eigenvalue of the autocorrelation of the input vector X. When λ is too small, the possibility of getting stuck at a local minimum of the error function is increased. In contrast, the possibility of falling into oscillatory traps is high when λ is too large. This fact added to the slow convergence of the algorithm lead to several variations to improve performance and convergence speed.

Following with SGD as the *cost function*, it can also be used in a smarter way to speed up the learning. The idea is to estimate the gradient $\nabla E(w)$ by computing $\nabla E_x(w)$ for a small sample of randomly chosen training inputs, called *batch*, whose size is m so that $m < n$, with n the size of the complete input dataset. By averaging over this sample, provided that the batch size m is large enough, it quickly gets a good estimate of the true gradient:

$$\frac{\sum_{j=1}^{m} \nabla E_{x_j}(w)}{m} \approx \frac{\sum_{i=1}^{n} \nabla E_{x_i}(w)}{n} = \nabla E(w).^2 \tag{2.21}$$

Adam [22] is a recent alternative to SGD. It is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. Adam was designed to combine the advantages of two other popular techniques: AdaGrad [8], which works well with sparse gradients, and RMSProp [40], which works well in on-line and non-stationary settings.

In this section we have presented the MLP network, which is the baseline model for FNN. In Section 2.4 another type of FNN, Convolutional Neural Networks (CNNs), are described in detail since it will be used in the subsequent sections of this work. However, before offering an insight into CNNs, we briefly present Recurrent Neural Network (RNN).

Recurrent Neural Network An architecture is referred to as RNN when connections between neurons form a directed cycle (see Figure 2.5). This creates an internal state in the network, which allows it to exhibit dynamic temporal behavior, i.e., the feedback connections provide the network with past context information. Due to this property RNNs are often better for tasks that involve sequential inputs such as audio, video and text. When we consider the outputs of the hidden units at different discrete time steps as if they were the outputs of different neurons in a deep multilayer network (Figure 2.5, right), it becomes clear how we can apply backpropagation to train RNNs.

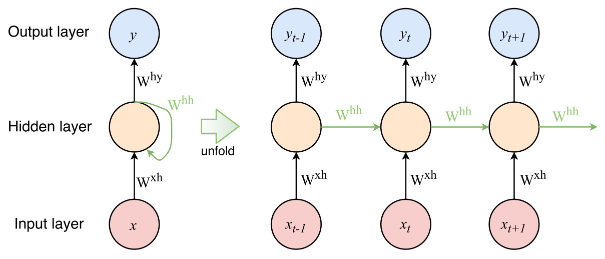


Figure 2.5 *A recurrent neural network with one hidden layer and a single neuron. On the right, the unfolding in time of the steps involved in its forward computation.*

RNNs, once unfolded in time, can be seen as very deep feedforward networks in which all the layers share the same weights. Although their main purpose is to

[2]Conventions vary about scaling of the cost function and batch updates. We can omit $\frac{1}{n}$, summing over the costs of individual training examples instead of averaging. This is particularly useful when the total number of training examples isn't known in advance

learn long-term dependencies [24], theoretical and empirical evidence shows that it is difficult to learn to store information for very long. To correct for that, an effective alternative to conventional RNN are Long Short-Term Memory (LSTM) networks [16], that use special hidden units to augment the network with an explicit memory. Other proposals include the Neural Turing Machine [12] and memory networks [45].

2.4 Convolutional Neural Networks

There have been numerous applications of convolutional neural networks going back to the early 1990s, but it was since the early 2000s when CNNs have been applied with great success to detection, segmentation and recognition of objects and regions in images. Recently, they have achieved major results in face recognition [39], speech recognition [1] and raw audio generation [42]. The model presented in [42] by DeepMind, which inspired us to undertake this work, also reaches state-of-the-art performance in text-to-speech applications.

Despite these successes, CNNs were largely forsaken by the mainstream computer-vision and machine-learning communities until the ImageNet competition in 2012. The spectacular results achieved by A.Krizhevsky, I.Sutskever and G.Hinton [23] came from the efficient use of GPUs, ReLUs, a new regularization technique to avoid overfitting called dropout, and techniques to generate even more training examples by deforming the existing ones. This success has brought about a revolution in computer vision; CNNs are now the dominant solution for almost all recognition and detection tasks and approach human performance on some others [24].

The Convolution Operation. The operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields such as engineering or pure mathematics. The convolution of two real-valued functions is typically denoted with an asterisk ($*$) and it is defined as the integral of the product of the two functions after one is reversed and shifted. However, working with data on a computer, time is usually discretized and it can take only integer values. Thus, if we assume that f and k are two discrete functions defined only on integer n, we can then define the discrete convolution as:

$$s(n) \equiv \sum_{m=-\infty}^{\infty} f(m)k(n-m) = \sum_{m=-\infty}^{\infty} f(n-m)k(m). \tag{2.22}$$

In convolutional network terminology, the first argument to the convolution is often referred to as the *input* (function f) and the second argument as the *filter* or *kernel* (function k). Both of them are multidimensional arrays, or tensors, that are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

The output s can be referred to as the *feature map*, which usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero) [11, ch.9, pp.333-334]. This is because the kernel is usually much smaller than the input image.

The only reason to flip the second argument in Equation 2.22 is to obtain the commutative property. Since in neural networks the kernel is symmetric, commutative property is not usually important and many neural network libraries implement a pseudo-convolution without reversing the kernel, known as *cross-correlation*.

$$s(n) \equiv \sum_{m} f(m)k(m+n). \tag{2.23}$$

It can be easily generalized for a two-dimensional input $F : \mathbb{Z}^2 \to \mathbb{R}$, which probably will be used with a two-dimensional kernel $K : \Omega_r \to \mathbb{R}$, with $\Omega_r = [-r, r]^2 \cap \mathbb{Z}^2$ [46]:

$$S(p) = (F * K)(p) \equiv \sum_{m+n=p} F(m)K(n). \tag{2.24}$$

A CNN (Figure 2.6) can be regarded as a variant of the standard neural network. It is a feedforward network, i.e., each layer receives inputs only from the previous layer, so information is always traveling forward. Its typical architecture is structured as a series of stages. The first few stages consists of alternating so-called convolution and pooling layers, instead of directly using fully connected hidden layers like in RNNs.

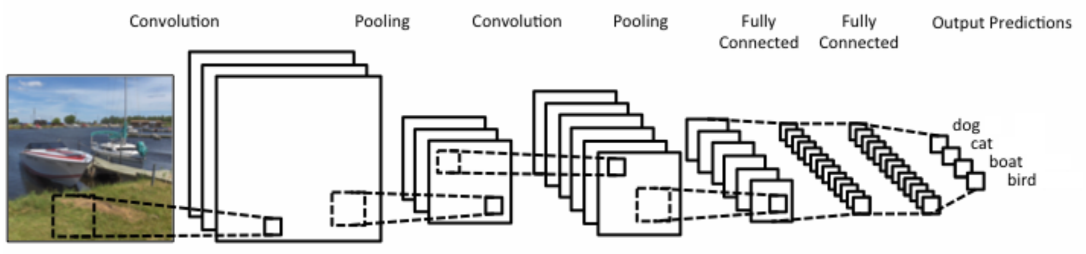


Figure 2.6 *A simple convolutional neural network. (Source: www.clarifai.com).*

CNNs make the explicit assumption that the input data is organized as a number of feature maps. This is a term borrowed from image-processing applications, in which it is intuitive to organize the input as a two-dimensional array (for color images, RGB values can be viewed as three different 2D feature maps). Thus, the layers of a CNN have neurons arranged in three dimensions: width, height and depth. For example, input images in CIFAR-10 are an input volume of activation which has dimensions $32 \times 32 \times 3$ (width, height and depth respectively) as shown in Figure 2.7.

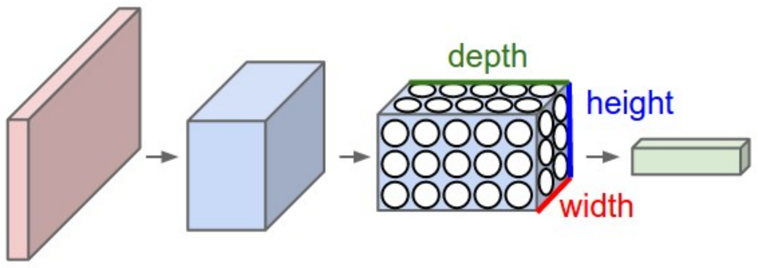


Figure 2.7 *One of the hidden layers show how three dimensions are arranged in a CNN. Every layer transforms the 3D input volume to a 3D output volume of neuron activations through a differentiable function. (Source: cs231n.github.io/convolutional-networks/)*

There are four key concepts behind CNNs that take advantage of the properties of natural signals: local connections, shared weights and biases, pooling and the use of many layers [24]. The idea of stacking many layers up is explained in Section 2.5, introducing the advantages of using deep neural networks.

Local connections. In CNNs not every input sample is connected to every hidden neuron, as well as it is impractical to connect neurons to all neurons in the previous layer. Instead, connections are made in small, localized regions of the input feature map known as *receptive field*. To be more precise, each neuron in the first hidden layer is connected to a small region of the input neurons, say, for example, a 3×3 region as in Figure 2.8. We then slide the local receptive field across the entire input, so for each local receptive field, there is a different hidden neuron in the first hidden layer. We can think of that particular hidden neuron as learning to analyze its particular local receptive field.

Shared weights and biases. Shared weights and bias are often said to define a kernel or filter (different weights led to different filters). Following the example above, each hidden neuron has a bias and 3x3 weights connected to its local receptive field. But this bias and weights are the same for every neuron on each layer. This means that all the neurons in the first hidden layer detect exactly the same

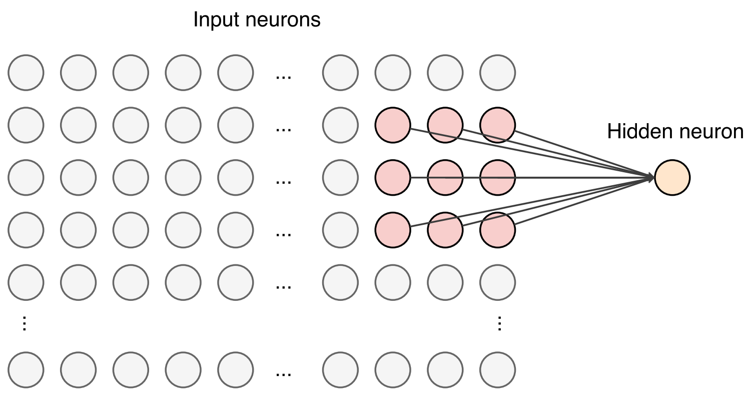


Figure 2.8 *Connections for a particular neuron in the first hidden layer. Its receptive field is highlighted in pink.*

feature, just at different locations in the 2D input array. A big advantage of sharing weights and biases is that it greatly reduces the number of parameters involved in a convolutional network. Despite the runtime of forward propagation remains the same, the storage requirements are vastly reduced.

Pooling. A pooling layer is a form of non-linear downsampling and it is usually used immediately after convolutional layers. Pooling layers condense the information in the output from the convolutional layer by replacing the output of the net at a certain location with a summary statistic of the nearby outputs [11]. As a concrete example, one common procedure for pooling is known as max-pooling where the maximum output within a rectangular neighborhood is reported. Another popular pooling functions is L2, which takes the square root of the sum of the squares of the activations in the region applied.

Dilated Convolution In dense prediction problems such as semantic segmentation or audio generation, working with a large receptive field is an important factor in order to obtain state-of-the art results. In [46], a new convolutional network module that is specifically designed for dense prediction is defined. It is known as *dilated* or *atrous* convolution, a modified version of the standard convolution. Let l be a dilation factor and let $*_l$ be defined as in Equation 2.25 for a two-dimensional input:

$$(F *_l K)(p) \equiv \sum_{m+ln=p} F(m)K(n). \tag{2.25}$$

A dilated convolution is a convolution where the kernel is applied over an area larger than its length by skipping input values with a certain step [42], also called dilation factor. It effectively allows an exponential expansion of the receptive field without loss of resolution or coverage. This is similar to pooling or strided convolutions, but here the output has the same size as the input. Note as a special case, dilated convolution with dilation 1 yields the standard convolution.

2.5 Deep Learning

Deep neural networks (DNNs) have shown significant improvements in several application domains including computer vision and speech recognition [14]. In particular, deep CNNs are one of the most widely used types of deep networks and they have demonstrated state-of-the-art results in object recognition and detection [33, 38].

While the previous century saw several attempts at creating fast NN-specific hardware and at exploiting standard hardware, the new century brought a deep learning breakthrough in form of cheap, multi-processor graphics cards or GPUs. GPUs excel at the fast matrix and vector multiplications required not only for convincing virtual realities but also for NN training, where they can speed up the learning process by a factor of 50 and more [36].

At this point we may ask ourselves: what must a neural network satisfy in order to be called a deep neural network? A straightforward requirement of a DNN follows from its name: it is *deep*. That is, it has multiple, usually more than three, layers of units. This, however, does not fully characterize a deep neural networks. In essence, we often say that a neural network is deep when it has more than three layers and the following two conditions are met [4]:

- The network can be extended by adding layers consisting of multiple units.
- The parameters of each and every layer are trainable.

From these conditions, it should be understood that there is no absolute number of layers that distinguishes deep NNs from shallow ones. The depth grows by a generic procedure of adding and training one or more layers, until it can properly perform a target task with a given dataset [4].

In classic classification tasks, discriminative features are often designed by hand and then used in a general purpose classifier. However, when dealing with complex tasks such as computer vision or natural language processing, good features that are sufficiently expressive are very difficult to design. A deep model has several hidden layers of computations that are used to automatically discover increasingly more complex features and allow their composition. By learning and combining multiple levels of representations, the number of distinguishable regions in a deep architecture grows almost exponentially with the number of parameters, with the potential to generalize to non-local regions unseen in training [32]. Taking the network depicted in Figure 2.6 as an example, the combination of the first four layers work in feature extraction from image and the last fully connected layers in classification.

Nevertheless, DNN are hard to train. We could try to apply stochastic gradient descent by backpropagation algorithm as described in Section 2.3. But there is an intrinsic instability associated to learning by gradient descent in deep networks which tends to result in either the early or the later layers getting stuck during training [30]. In order to avoid that, many factors play an important role for an appropriate train: making good choices of the random weight initialization –a bad initialization can still hamper the learning process–, cost function and activation function [10], applying notably regularization techniques (in order to avoid overfitting) such us dropout and convolutional layers, having a sufficiently large data set and using GPUs.

2.6 Audio generation

Algorithmic music generation is a difficult task that has been actively explored in earlier decades. Many common methods for algorithmic music generation consist of constructing carefully engineered musical features and rely on simple generation schemes, such as hidden Markov models (HMMs) [37]. It captures the musical style of the training data as mathematical models. Following these approaches the resulting pieces usually consist of repetitive musical sequences with a lack thematic structure.

With the increase in computational resources and recent researches in neural network architectures, novel music generation may now be practical for large scale corpuses leading to better results. Models look after a pleasant to hear outcome since it is not easy to find an objective evaluation of the performance of the network.

Extremely good results are obtained with $WaveNet$ model from the paper [42], which works directly at waveform level and uses a very deep dilated convolutional network to generate samples one at a time sampled at 16 KHz. By increasing the amount of dilation at each depth, they are able to capture larger receptive fields and thus, long range dependencies from the audio. Despite the extensive depth, training the network is relatively easy because they treat the generation as a classification problem. It is reduced to classify the generated audio sample into one of 255 values (8 bits encoding).

Nonetheless, many recent studies that work with raw audio databases agree on RNN as the preferred architecture [19, 28, 29] to learn underlying dependencies from music input files. Both works [29] and [19] are based on LSTM networks trained with data in the frequency domain of the audio. This enables a much faster performance because it allows the network to train and predict a group of samples that make up the frequency domain rather than one sample [19].

In practice it is a known problem of these models to not scale well at such a high temporal resolution as is found when generating acoustic signals one sample at a time, e.g., 16000 times per second. That is the reason why enlarging the receptive field [42] is crucial to obtain samples that sound musical.

It may perhaps be considered without straying too far afield from our primary focus some speech synthesis techniques, since it is one of the main areas within audio generation. Conventional approaches typically use decision tree-clustered context-dependent HMMs to represent probability densities of speech parameters given texts [41, 50]. Speech parameters are generated from the probability densities to maximize their output probabilities, then a speech waveform is reconstructed from the generated parameters. This approach has several advantages over the concatenative speech synthesis approach [18], such as the flexibility in changing speaker identities and emotions and its reasonable effectiveness. However, HMMs are inefficient to model complex context dependencies and its naturalness is still far from that of actual human speech.

Inspired by the successful application of deep neural networks to automatic speech recognition, an alternative scheme based on deep NNs has increasingly gained importance applied to speech generation, although it is worth to emphasize that NNs have been used in speech synthesis since the 90s [21]. In the statistical parametric speech synthesis (SPSS) field [49], DNN-based speech synthesis already yields better performance than HMM-based speech synthesis, provided we have a large enough database and under the condition of using a similar number of parameters [47].

Regarding acoustic speech modeling in speech generation, deep learning can also be applied to overcome the limitations from previous approaches. These deep learning approaches can be classified into three categories according to the modeling steps, as well as the relationship between the input and output features represented in the model [26]:

1. Cluster-to-feature mapping using deep generative models. In this approach, the deep learning techniques are applied to the cluster-to-feature mapping step of acoustic modeling for SPSS, i.e., to describe the distribution of acoustic features at each cluster. The input-to-cluster mapping, which determines the clusters from the input features, still uses conventional approaches such as HMM-based speech synthesis [25].

2. Input-to-feature mapping using deep joint models. This approach uses a single deep generative model to achieve the integrated input-to-feature mapping by modeling the joint probability density function (PDF) between the input and output features. In [20], the authors propose an implementation with input features capturing linguistic contexts and output features being acoustic features.

3. Input-to-feature mapping using deep conditional models. Similar to the previous approach, this one predicts acoustic features from inputs using an integrated deep generative model [48]. The difference is that this approach models a conditional probability density function of output acoustic features, given input features instead of their joint PDF.

3. METHOD

This chapter describes the approach studied in this work to predict and generate audio signals based on a deep CNN. The method mainly consist in predicting the value of a sample based on a sequence of previous input samples. We could see the entire system as a black box which receives a bunch of generated waves and outputs a new synthesized audio signal. The model is trained on multiple batches composed of shorter temporal segments from the original signals.

3.1 System overview

In this section the overview of the system is presented with a brief introduction to all the steps in the pipeline. A depiction of the block diagram of the system is shown in Figure 3.1.

The input data set is an ensemble of analog waves that are sampled and then converted to discrete domain by a quantizer that approximates each continuous value sample with a quantized level. The data is divided into three different parts: training, validation and test set. Both training and validation sets are dynamically one-hot encoded, arranged in batches and fed to a deep CNN, which is trained to output the conditional probability for the next sample of every sequence. Once the network has been trained, test signals are selected as different seeds to boost the generation of new ones.

Input data set and its preprocessing to feed the network are explained in Section 3.2; network architecture and its training are explained in Sections 3.3 and 3.4 respectively; audio generation process is detailed in Section 3.5.

3.2 Data format

Waves generated and stored as the input data set are sampled following the Nyquist criterion for an alias-free signal sampling. This is, the sample rate meets the requirement $f_s > 2B$, where B is the bandwidth of the input signal with highest frequency.

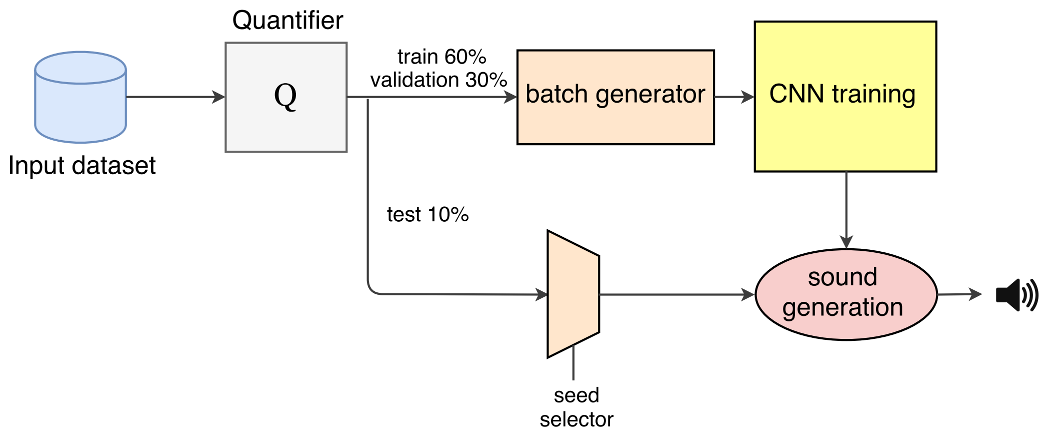


Figure 3.1 *Block diagram depicting an overview of the system.*

Hence, no actual information is lost in the sampling process. Notice that when working with pure sinusoids, the bandwidth is equivalent to the signal's frequency.

The discrete-time version of the original waves is then quantized. A simplified model of the quantizer applied is depicted in Figure 3.2. The value of each input sample is approached by the nearest quantization level Q_i out of $L = 2^b$ possible levels, where b is the number of bits. It is an uniform quantizer since the L output levels and the quantization step Δ are equally spaced. Zero-level is not a possible quantization level, being the quantizer symmetric with $L/2$ positive and $L/2$ negative output values. This characteristic is known as *mid-riser* approach.

To summarize, the uniform quantizer is specified with three parameters: i) the dynamic range $(-v_{sat}, v_{sat})$; ii) the step size Δ; and iii) the number of levels L or, equivalently, the number of bits b. The relation among these three parameters is the following,

$$L\Delta = 2v_{sat}; \quad 2^{b-1}\Delta = v_{sat}. \tag{3.1}$$

By representing a continuous-amplitude signal $a(nT_s)$ with a discrete set of values an error is introduced in the quantized signal $a_q(nT_s)$. We assume a quantization error $e_q(nT_s)$ given by the following equation:

$$e_q(nT_s) = a_q(nT_s) - a(nT_s). \tag{3.2}$$

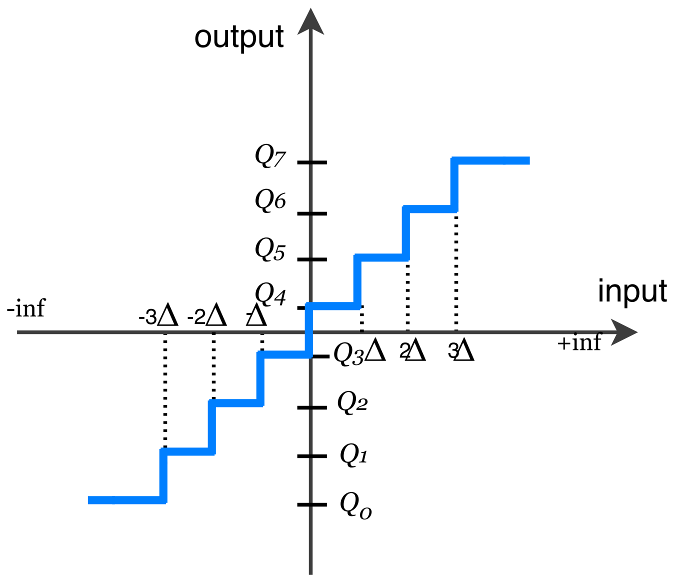


Figure 3.2 *A simple mid-riser quantizer with 8 quantization levels Q_i and uniform quantization step Δ.*

As distance between quantization levels Q_i is constant and equal to the quantization step Δ, i.e., $Q_i - Q_{i\pm1} = \Delta$, we can set a maximum for the error [2] as in Equation 3.3,

$$|e_q| \leq \frac{\Delta}{2} \quad \text{for} \quad |a| < v_{sat}. \tag{3.3}$$

In this section we have introduced the preprocessing applied to each signal in order to make them suitable to feed the network. However, we apply an additional step within batch generator block (see Figure 3.1) to one-hot encode the quantized signals to train the network. This process is detailed later in Section 3.4.

3.3 Neural Network Architecture

We train an artificial NN by showing it thousands of training examples and gradually adjusting the network parameters until it gives the classification we want. The network consists of several stacked layers of artificial neurons. Each wave is fed into the input layer, goes across the hidden layers until eventually the output layer is reached and the network, playing the role of a soft decision decoder, produces an output.

One of the challenges of neural networks is understanding what exactly goes on at each layer. It is known that after training, each layer progressively extracts higher and higher-level features of the input, until the final layer essentially makes a soft decision on what it is (what an image shows, what chord is being played, what is the next sample of a given sequence). The output shapes a vector of probabilities for each class after computing a *softmax* function used to normalize the output, defined by Equation 3.4, such that softmax(x_j) > 0 ∀j and $\sum_m x_m$=1 [32],

$$softmax(x_j) = \frac{e^{x_j}}{\sum_m e^{x_m}}. \tag{3.4}$$

Baseline model. In order to understand the behavior of a deep CNN and to test the best approach to generate new waves, we initially worked with the network architecture depicted in Figure 3.3. Filter weights are uniformly initialized. At this early stage we train the network with pairs of input sequences of length T and its targets which only contain the next sample to the input sequence, i.e., sample *T+1*. The length of the input waves matches the size of the receptive field of the network, which also defines the number of hidden layers according to the following equation,

$$\#hidden\ layers = \log_2(receptive\ field). \tag{3.5}$$

In addition, hidden layers are convolutional layers with stride equal to two, causing output's size is half of input's size. Therefore, taking into account this property and Equation 3.5, the output of the last convolutional layer is a single value. As an example, given an input sequence of 64-samples length, the network has 6 convolutional hidden layers whose intermediate signal's lengths are 32, 16, 8, 4, 2, 1 respectively. Last output is then connected to a dense layer that calculates the output of the network.

ReLU is the activation function of neurons in convolutional layers, while in the dense layer depends on the solution studied. When testing classification performance, *softmax* is applied to calculate the probability of belonging to each output class for the next sample in the input sequence; in this case it can be directly inferred that dense layer has as many output bins as quantization levels. When testing regression, *tanh* is the activation function to output a real value.

Second model. Yet the baseline model proposed works well with short sequences at low frequencies, we need to increase its complexity to handle larger receptive fields. Recent advances in generative models for audio [42] and images [43] have stated the

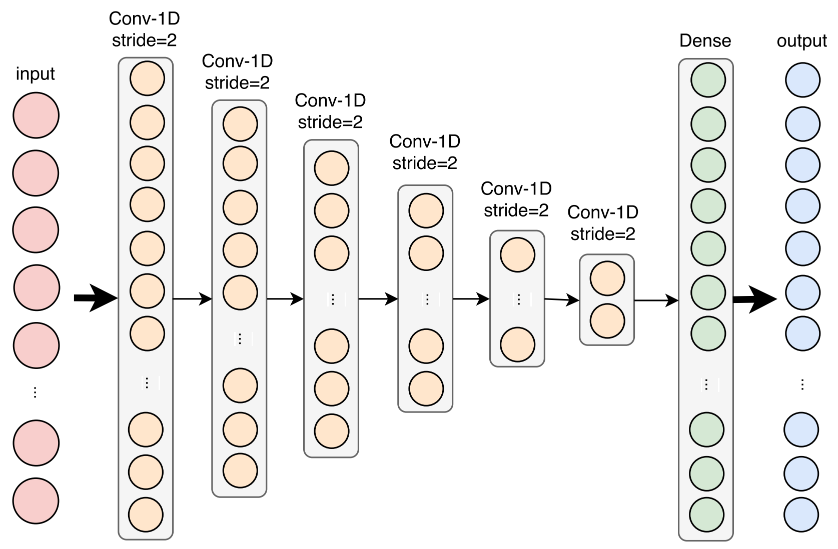


Figure 3.3 *Baseline model of the deep CNN proposed for early studies within this work. The network depicted is an example with 64-length receptive field.*

importance of a large receptive field to achieve a more natural synthesis, especially when working with high temporal resolution tasks such as in raw audio generation.

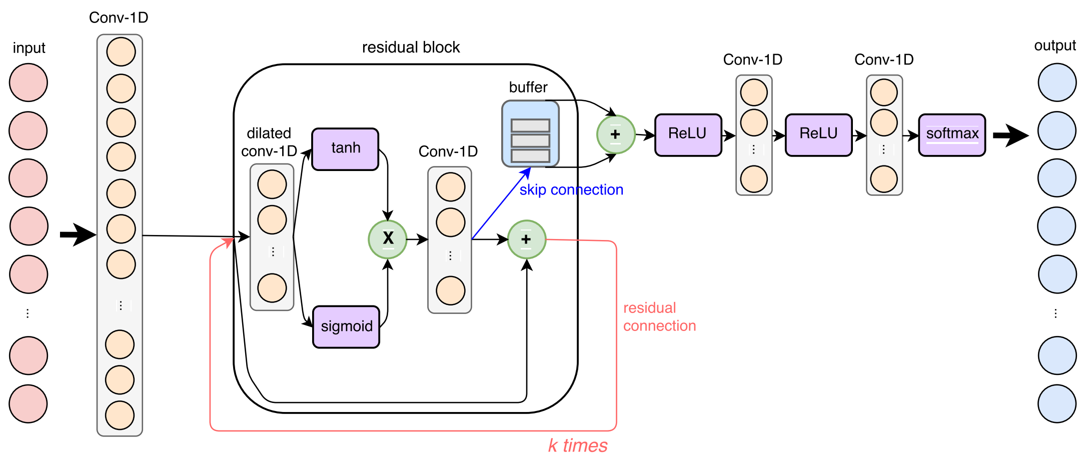


Figure 3.4 *Network architecture based on WaveNet model [42].* Residual block *is stacked k times in the network.* Skip connections *are stored and after k iterations are merged to make the input to the next step in the pipeline. Output keeps the same shape than the original input to the network.*

With this purpose, we implement an adaption of WaveNet architecture presented in [42]. The network topology is based on a deep CNN and presented in Figure 3.4. The main component of the architecture are causal convolutions. By using causal convolutions we make sure the model cannot violate the ordering in which we model the data: the prediction emitted by the model at timestep t does not depend on any of the futures timesteps $t+1, t+2, ...$ [42]. The inclusion of dilated causal convolutions allows an exponential expansion of the receptive field without loss of resolution or coverage [46], which favors long term memory; at the end it leads to a robust wave generation and achieves the synthesis of new waveforms without greatly increasing computational cost. Layers implementing a dilated convolution are defined in Keras; we modified the standard layer to enable *causal* flag following the code from github.com/basveeling/keras#@wavenet as a reference.

The block named *residual block* presents a feedback connection indicated by a red arrow in the diagram, which means that the entire block is stacked k times, or equally, $\log_2(\textit{receptive field})$. The *residual connection* acts as the new input to the block in the next iteration. After k iterations, the *skip connections* that have been stored are merged and continue forward in the pipeline. Unlike with the previous baseline model, now the target keeps the same size than the input segment, which implies that we train the network with pairs of segments [**0, ..., T**] and [**i, ..., T+i**] as input and target respectively.

3.4 Training the network

Quantized signals are split up into three groups as mentioned in Section 3.2. Train and validation sets are the input to a batch generator which selects a certain number of signals to feed the network at every training epoch. Due to memory restrictions we shorten the signals to segments instead of feeding the entire signal at once. The selected format for the training data is one-hot encoded. Figure 3.5 summarizes the steps performed within the batch generator.

A large and deep neural network, with millions of parameters like the one studied in this work, has enough flexibility to properly solve the problem, but will be also very prone to overfit to the training data when this is scarce. For this reason, a vast amount of training data is a key requirement to train a large and robust model. In order to enhance the training process and to be able to generalize to unseen data without a high storage demand, we produce new examples by introducing a variation in the existing ones. Segments are randomly selected within each signal, allowing us to augment the number of training examples seen by the network since even two segments from the same original signal will have a different phase offset. The generator yields batches with both training and target data. Target data is generated from training data in two possible different manners.

Training one. First approach is to feed the network with batches composed of

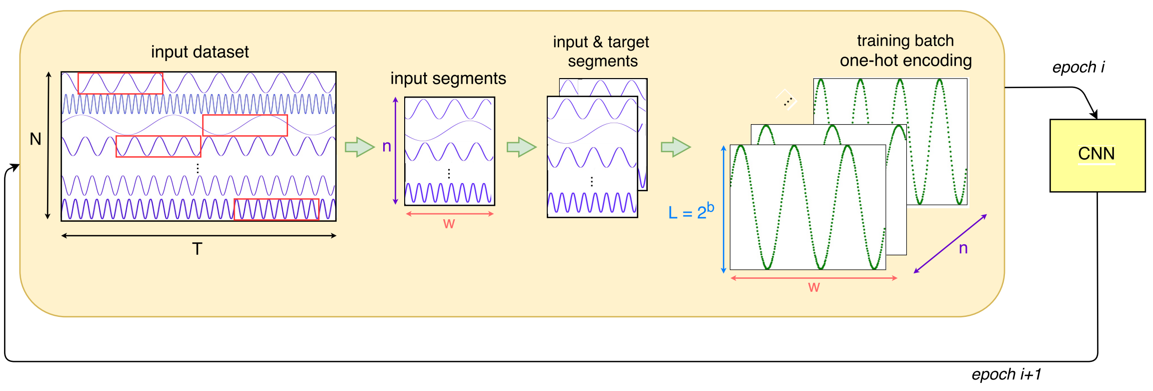


Figure 3.5 *Pipeline of the steps performed in the batch generator. It is called at the beginning of every epoch to generate a new training batch. N is the number of signals in the input set; n is the batch size, with $n < N$; T is the length of the signals in the input set; w is the length of the training segments, with $w < T$; L is the number of quantization levels.*

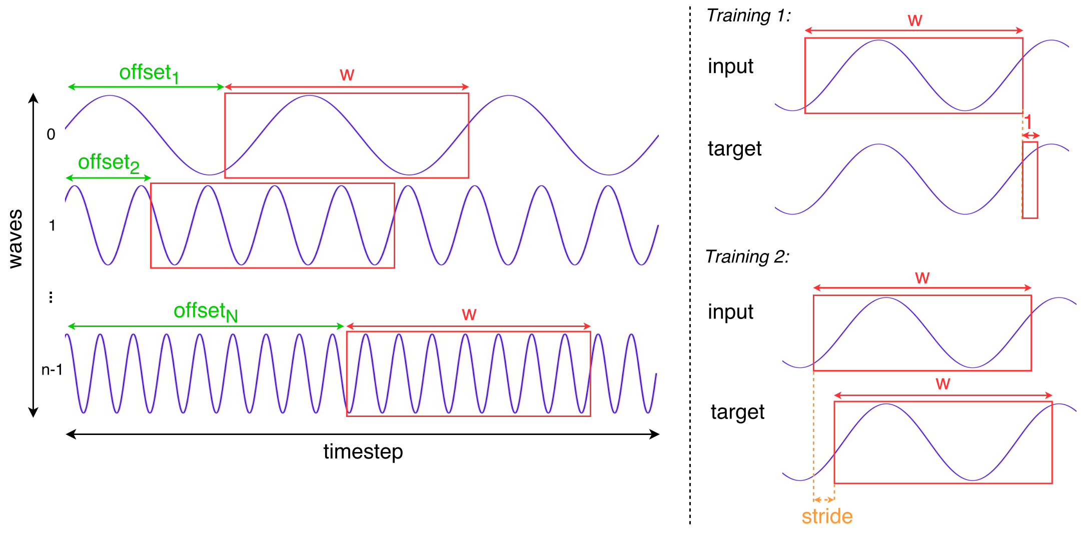


Figure 3.6 *Depiction of how segmentation and target generation work. On the left, there are n signals randomly picked. Within each signal, every $offset$ parameter points the starting sample of every segment of fixed length w. On the right, two training approaches. On top, target is a vector with the one sample encoded, adjacent to the end sample of the input segment. Below, the parameter stride sets the shift -same value for all the segments- from the starting sample of the input segment.*

fixed size segments paired with one sample target. Signals are fed into the network one segment at a time, and it is trained to predict the next sample in the sequence. However, before yielding a batch, the segments are one-hot encoded. Each of them is a matrix with as many columns as the segment size -number of timesteps- and L rows -one per quantization level-. Therefore, it is a zero matrix filled with one number 1 in every column in the corresponding position, as shown in the graphs with green dots from Figure 3.5. Accordingly, target is a vector.

Training two. Both input and target have the same size, but target is shifted a number of samples on time, what we called *stride*, as depicted on the right side of Figure 3.6. Segment length is a design parameter which is carefully studied and affects network performance. We mainly have two variations that distinguish between segment length that matches receptive field size and segment length larger than receptive field; implications of different segment size are explained in Chapter 4. Likewise *training one*, segments are one-hot encoded.

Loss function and optimizer. In both models and training approaches presented above, categorical cross entropy is computed as loss function. How cross entropy performs and why it is a more accurate measure to evaluate the performance of the network when working on classification tasks is explained in Section 2.3. Adam is the selected optimizer, set up with default parameters [22] after verifying it is the configuration that provides better performance.

3.5 Audio generation

Audio generation process starts after having properly trained the neural network. As explained in the previous section, the network is trained with a bunch of tones in the first place. Once the trained architecture is capable to predict correctly pure tones within the training range of frequencies, which does not necessarily mean these tones belong to the training set, we save the network settings and proceed with the generation phase. Since the aim of generation is to synthesize a new waveform, it is advisable to the train the network with non-stationary signals. Thus, it is more difficult to predict the sequential samples and the network has more degrees of freedom to generate a new waveform.

It is a sequential process based on predicting the sample *t+1* for a given sequence of length t. Every time an output value is predicted, it is appended to the input sequence and then fed back to the input of the network to predict the next sample, as depicted on Figure 3.7. The initial sequence is known as *seed* and it belongs to the test set. At this stage, the *seed* and the subsequent network inputs are segments matching the size of the receptive field instead of using larger segments as in the training process. This allows to accelerate the generation procedure.

As we can see in Figure 3.4, the output layer in the network is a softmax function which gives us the conditional probability distribution over the individual audio samples, $p(x_t|x_1, ..., x_{t-1})$ for L output classes. This is, softmax function outputs L probabilities per timestep to model all possible values. Therefore, the predictor determines the new sample after calculating the maximum likelihood.

Then, we append the new sample at the end of the input sequence and shift by one the consequential sequence, i.e., we keep the same segment length by including the new prediction and deleting the first sample, oldest in time. We one-hot encode the sequence and feed the network to make the next prediction. This iterative process is repeated until we have generated the desired number of new samples. It is worth to highlight the fact that the network will be eventually generating new audio samples based on a completely predicted sequence.

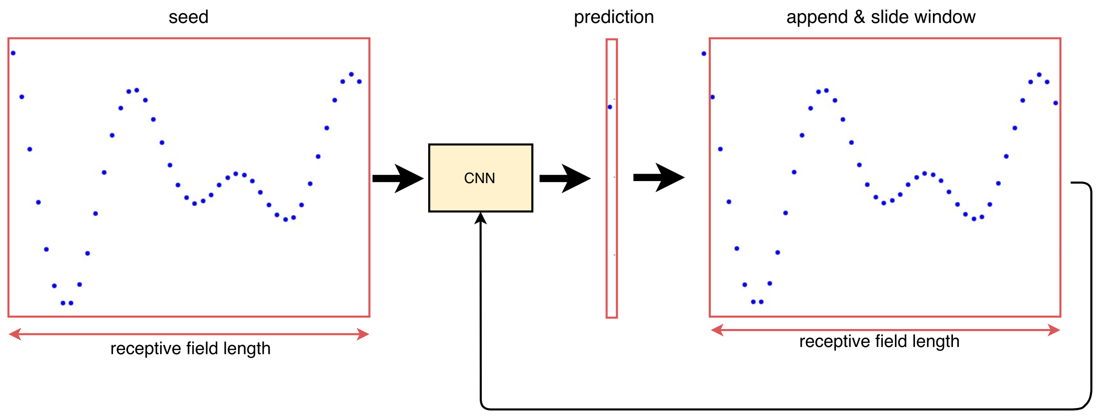


Figure 3.7 *Sound wave generation is an iterative process. Every time an output value is predicted, the prediction is fed back to the input of the network to predict sequentially the next sample.*

4. EVALUATION

Our study takes pure sinusoidal waves, also known as *tones*, as the baseline experiment. The results after training the system with these signals serve us as the reference to evaluate the performance of the system with more complex waveforms. Mathematically, a sinusoidal wave is given as:

$$s(t) = A(t) \cdot \sin(\omega t + \phi) = A(t) \cdot \sin(2\pi f t + \phi), \tag{4.1}$$

where A is the wave amplitude, ω is the wave angular frequency, f is the frequency in Hz and ϕ is the phase offset. Classic modulation techniques are amplitude, frequency and phase modulation that encode information as variations in A, f and ϕ respectively. However, if these parameters remain constant over time it leads to a pure tone. Tones can also be mixed up to produce more complex waveforms.

System development and generation. Data generation, system development, evaluation and post audio generation are entirely based on Python[1]. Design and training of deep CNNs were built on Keras[2], a modular neural network library written in Python that enables fast experimentation.

4.1 Input dataset

In order to measure the performance of different NNs and test the influence of hyperparameter values, we first create a dataset with 1500 pure sinusoids of one second each, whose frequencies belong to an audible range from 100 Hz to 1 KHz. Frequency and initial phase are randomly picked for every sinusoid; amplitude is set to one. Sines are sampled at 8 KHz to lighten memory requirements and quantified with 8 bits as shown in Figure 4.1. From now on, we will refer to this input data set as *set 1*.

A second dataset aimed to achieve generation of new waveforms is created with

[1]url: www.python.org/downloads
[2]url: www.github.com/fchollet/keras

1500 mixture sinusoids of one second each, also sampled at 8 KHz and quantified with 8 bits. Every signal is the result of adding two individual sines of different random frequencies within the range 100 Hz to 1 KHz; its amplitude is normalized to unity. Since this two sines are seldom harmonics, the final wave is not necessarily stationary and it is hard to predict. This is a desirable characteristic to boost generation of new waveforms which does not belong to the training set. On the other hand, we can not objectively quantify the performance of the network even though the sinusoids are deterministic functions because the generator outputs a different waveform, i.e., with different phase, frequency and variable amplitude. As we will see later, it does not predict the samples of the input signal as it is, but generating a new signal. From now on, we will refer to this input data set as *set 2*.

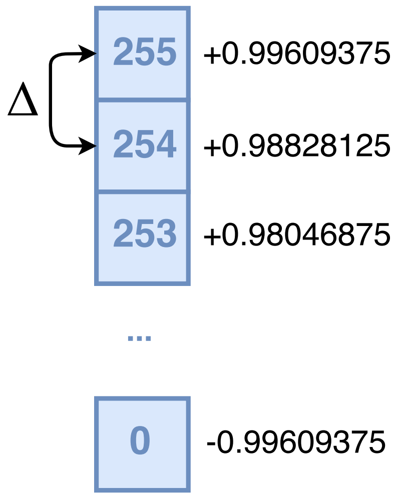


Figure 4.1 *256 quantization levels from 8 bit quantifier. Uniform quantization step* Δ *with seven significant decimal digits. Maximum quantization error is* $\Delta/2 \approx 3.9E - 3$ *according to Equation 3.3.*

In both sets we split up the data into three subgroups, taking into account the importance to have enough data to validate the learning process so network does not fail to overfit the training set. Thus, we have 900 waves to train the network, 450 to validation and 150 to test (60% train, 30% validation and 10% test).

4.2 Evaluation procedure

As explained in Section 2.6, it is not easy to devise an objective measure to quantify an audio generator performance. Neither an evaluation of a multilabel classifier is straightforward. However, the main goal in the latter is well known: to assess how close the network prediction $\hat{\mathbf{y}}$ is to the target $\mathbf{y}$. This is why we decide to undertake an objective evaluation as if our system were a pure classificator, without including the outcome from generation phase.

We firstly train a network architecture with *set 1* and evaluate its performance by means of a loss function. The main advantage of training the network with stationary signals is that they are predictable and thus, we can quantify how well is the network predicting samples over time. After the network is capable to estimate correctly pure tones within both train and test sets, we save the progress made through the learning process and record the elapsed time in training for future comparisons. Then, we proceed to train the same architecture with *set 2*, save weights of the network and load them into the generator (which basically is same network architecture but this time, after the supervised training, predicts one sample every timestep in an unsupervised way, as explained in Section 2.6). Subsequent to the generation phase, we analyze visually and by listening the generated sequence. Since it is a new waveform that does not continue the original seed shape, we can only evaluate and validate it subjectively, provided that it is an audible segment, far from being noisy or squeaky.

Loss function chosen to weigh the distance between the predicted class and the target is categorical cross entropy, presented in Section 2.3 as the most suitable cost function in multiclass classification tasks. We evaluate it in every training epoch based on the ground-truth provided as the target, since first of all we follow a supervised learning technique. To support the results obtained with a second measure, we evaluate it together with MSE (see Equation 2.8). Considering the quantization values from Figure 4.1, we can set a maximum MSE for a single prediction thinking of the worst-case scenario: the real value of the sample is the minimum, i.e. -0.99609375, and the network predicts it is the maximum, i.e. +0.99609375. In other words, it maximizes the numerator of the Equation 2.8,

$$E_{MSE}^{MAX} = \frac{\|-0.99609375 - 0.99609375\|^2}{256} = 0.0155. \tag{4.2}$$

Working with configuration from *training two* -stride parameter always set to one- both categorical cross entropy and MSE are calculated and stored in every epoch. Should we work with a segment length same as the receptive field size, it applies solely to the last sample of each segment from a batch, i.e., the measures only take into account the new sample predicted for the input sequence and compare it with the actual value, which is the last sample of the target segment. This way we skip calculation within the receptive field. Should we train the network with larger segments, measures are computed for all the samples beyond the receptive field.

4.3 First Neural Network approach

Deep NNs have lately seen greater success in classification than in regression tasks [23, 42]. with regard to classification, the problem is to identify to which of a set of defined classes a new observation belongs, while the output in regression tasks is a continuous value.

An initial research in this work is aimed to decide how to tackle the problem of audio generation, as either regression or classification task. Regression is performed with $tanh$ activation function in the output layer yet we observed this nonlinearity worked significantly better than $ReLU$, which was due to the fact that we work with audio signals having negative values. We set up a modest experiment with the baseline network architecture depicted in Figure 3.3 and a variation of *set 1* as input, where segments are shortened to 16 ms (128 timesteps). We tested four combinations, training either directly with raw values or one-hot encoded data for both regression and classification tasks. Whereas in regression the target was the next real value in the sequence, i.e., the value of the sample 129, in classification it was one-hot encoded as shown in Figure 4.2.

Despite classification led to a higher MSE than regression when testing the performance with the test set (an average of 0.046 versus 0.003, according to Equation 2.8), we decided to follow this approach since it performed better results in terms of accuracy of the predicted signal when working with deeper NNs and ease the work with larger sequences, which is essential to model longer term dependencies.

One-hot encoding input data forces us to work with 2D matrices instead of 1D vectors and it has a highly negative impact in computational time required. We observed one-hot encode solution performed four times slower than working directly with real values. Nevertheless, it outperformed the train with raw values as input data in terms of lower MSE for regression and higher accuracy in classification. This fact, in addition to recent successful applications based on one-hot encoded data [3], helped us to opt for an one-hot solution.

4.4 Second Neural Network performance

The NN based on WaveNet architecture (Figure 3.4) was chosen as final implementation after corroborating the key role played by dilated causal convolutions. We

[3]magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn/

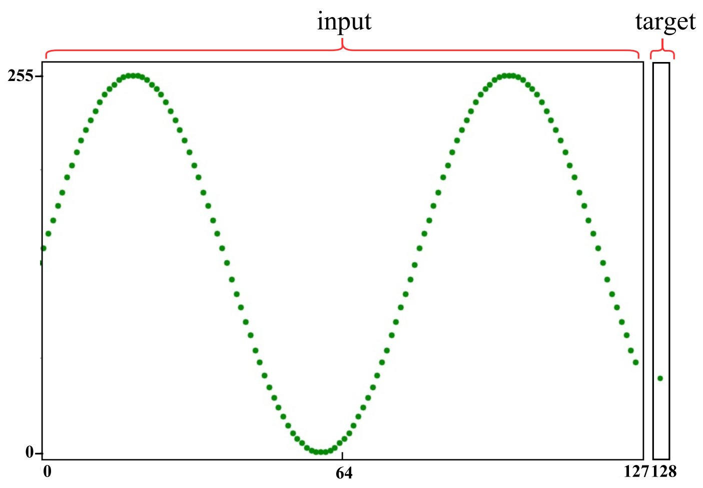


Figure 4.2 *A tone from the training set one-hot encoded and its target.*

analyzed two methods within *training two* set up. **Method 1** trains the network with segments whose length matches the receptive field size of the network. **Method 2** was intended to train the network directly with entire signals of one second. As we came across a memory allocation issue, we reduced the signals to shorter segments of 500 ms randomly placed within the original waves. We tested different settings to study the influence of hyperparameter values such as the receptive field size, batch size, stride and segment length, as well as the computational time needed. Table 4.1 shows the constant parameters fixed after reading documentation [31, 42] and previous test with the baseline network architecture. All experiments were run using one NVIDIA Tesla K40 GPU accelerator card.

# filters:	256
filter length:	2
batch size:	32
epochs:	150
loss function:	categorical crossentropy
optimizer:	ADAM

Table 4.1 *Fixed parameters of the CNN.*

4.4.1 Batch size

Before setting the batch size to 32 segments as a fixed value, we run several tests in order to verify the behavior of the network. At this step we work with signals from *set 1*, a receptive field of 512 and 1024-samples input segments (what we can see as a variant from *Method 2*). The reason of choosing an input segment length that double the receptive field size is because it enhances the learning process compared to *Method 1*, as explained later in Section 4.4.4, and it is more efficient than working with 500 ms segments.

Figure 4.3 illustrates the evolution of cross entropy value when training the network with different amount of examples. The batch generator implemented allows us to train the network with more than one batch every epoch, which provides an additional degree of freedom. This means that training with ten batches of size 32 signals (blue line in Figure 4.3) entails the network sees the same amount of examples every epoch than training with five batches of size 64 (magenta line in Figure 4.3). In both cases the network is trained with 320 different signals every epoch, but arranged in different way. We can conclude that the network performs better with major number of examples per epoch arranged in smaller batches, which also reports a lower MSE (Table 4.2). As a compromise between computational time and performance, we set ten batches of 32 signals (32x10) as baseline for future experiments.

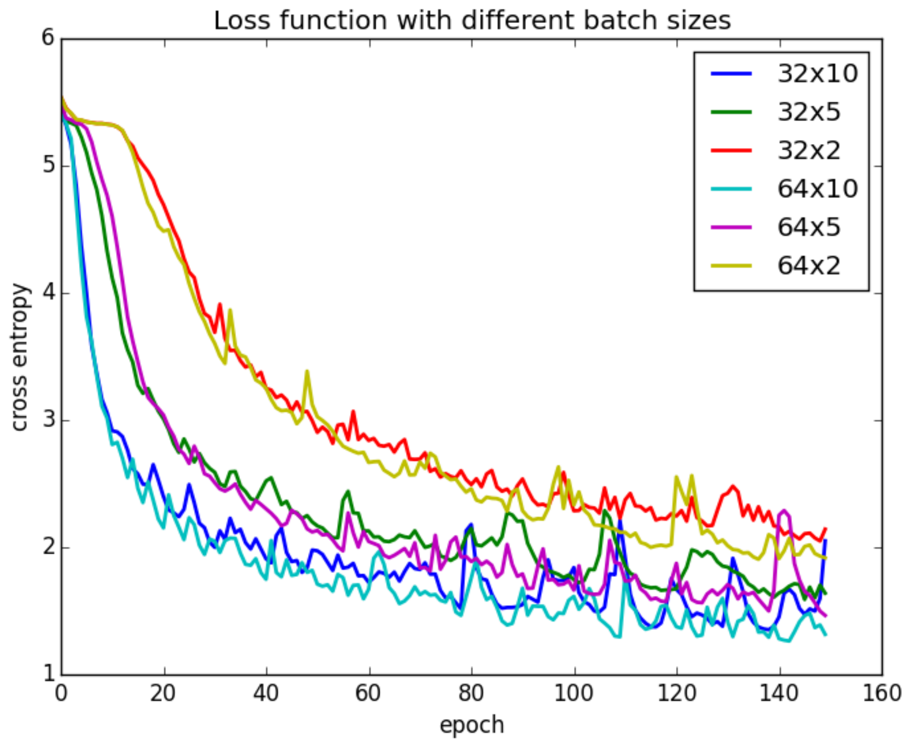


Figure 4.3 *Evolution of the loss function across the training epochs when setting different amount of examples per epoch. The legend shows how the examples are arranged:* 32x10 *references to ten batches of size 32, and so on.*

Batch size	32x10	32x5	32x2	64x10	64x5	64x2
MSE (train set)	0,0025	0,0028	0,0031	0,0024	0,0026	0,0030
MSE (test set)	0,0039	0,0039	0,0039	0,0037	0,0039	0,0039

Table 4.2 *MSE after 150 training epochs. MSE in test barely varies probably due to the small size of the set.*

4.4.2 Segment length

Prediction of the next sample in a given sequence highly depend on the data used to train the network. It does not only depend on the range of frequencies, shape and encoding of the signals but also on its duration. Likewise measuring the effect of batch size, we train same CNN model with a receptive field fixed to 512 and *set 1*. With a configuration of *32x10* batches, we run experiments with different segment lengths as input signals. Considering that the original input dataset was created with one second signals, it would be desirable to test the training feeding directly the entire waves. However, it was not feasible due to computational power restrictions and we decided to select 4000-sample segments, i.e., 500 ms if sampling at 8 KHz, as the largest sequences to train.

Figure 4.4 illustrates the evolution of cross entropy value when training the network with different segment sizes. It can be directly stated that training with segments longer that the receptive field favor the learning process: it converges faster and achieves lower value for the loss function and MSE as exposed in Table 4.3. However, we observe that provided they are larger than the receptive field, enlarge its size does not affect the cross entropy or MSE.

Segment length	512	1024	2048	4000
MSE (train set)	0,0036	0,0025	0,0025	0,0024
MSE (test set)	0,0039	0,0039	0,0039	0,0038

Table 4.3 *MSE after 150 training epochs. MSE in test barely varies probably due to the small size of the set.*

Training with larger sequences also benefit post audio generation. Despite the fact that CNNs were not designed to preserve long term memory, training with segments of longer duration helps to overcome this drawback when trying to model temporal dependent sequences as speech, audio or text sentences. On the other hand, this sort of training highly increment the computational power needed and it was impracticable with the existing technology few years ago.

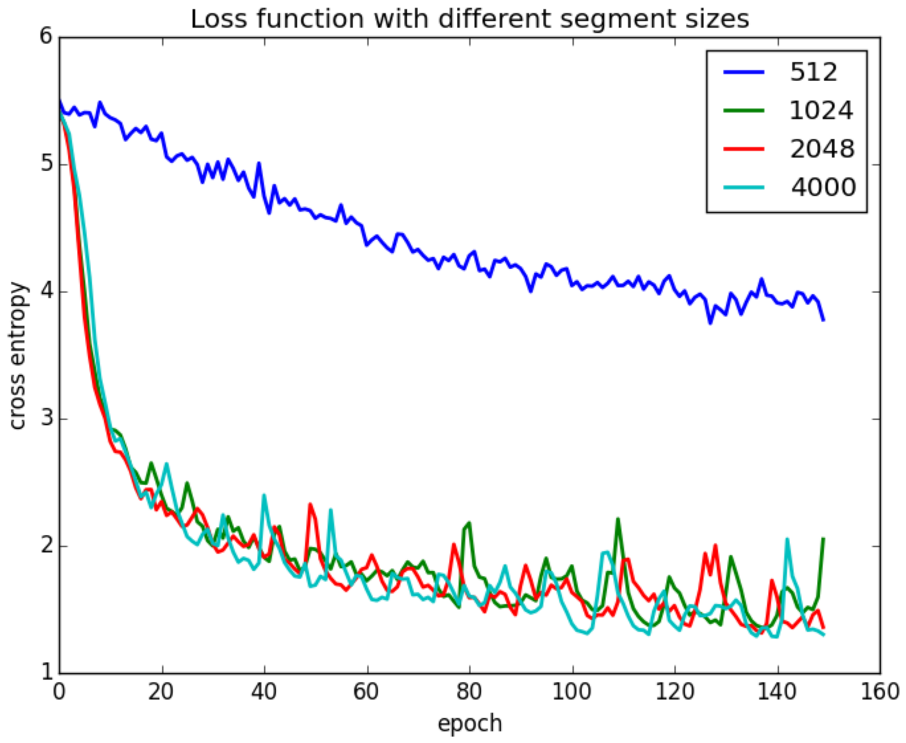


Figure 4.4 *Evolution of the loss function across the training epochs when using different segment lengths as input sequences. All the experiments were run with a receptive field size of 512 and 320 examples per epoch arranged in batches of size 32, i.e., 32x10.*

4.4.3 Receptive field size

One of the key concepts when working with a CNN is the receptive field of the network. As a reminder of Section 2.4, we connect each neuron to only a local region of the input volume, which we call receptive field. It is important to emphasize that connections are local in space (along width and height) but always full along the entire depth of the input volume, which in our case has depth 1. All neurons have the same receptive field size.

Figure 4.5 and 4.6 illustrate the impact of varying the size of the receptive field, training with signals from *set 1*. We analyze it following *Method 1* and *2* exposed above in Section 4.4. When training with the *second method* we set the maximum segment size up to test its behavior since it is the configuration we choose later for generation. From the graph on Figure 4.6 showing results for *Method 2*, we can tell that the performance of the network barely varies with size of the receptive field in terms of loss function. In the same manner, the progress of cross entropy with *Method 1* is similar for all the sizes tested (Figure 4.5). Nonetheless, it is clear that larger sequences equally benefit all the studied configurations. MSE calculated after

150 training epochs is indicated in Table 4.4 and 4.5; it agrees with the behavior shown in graphs and *Method 2* achieves lower values.

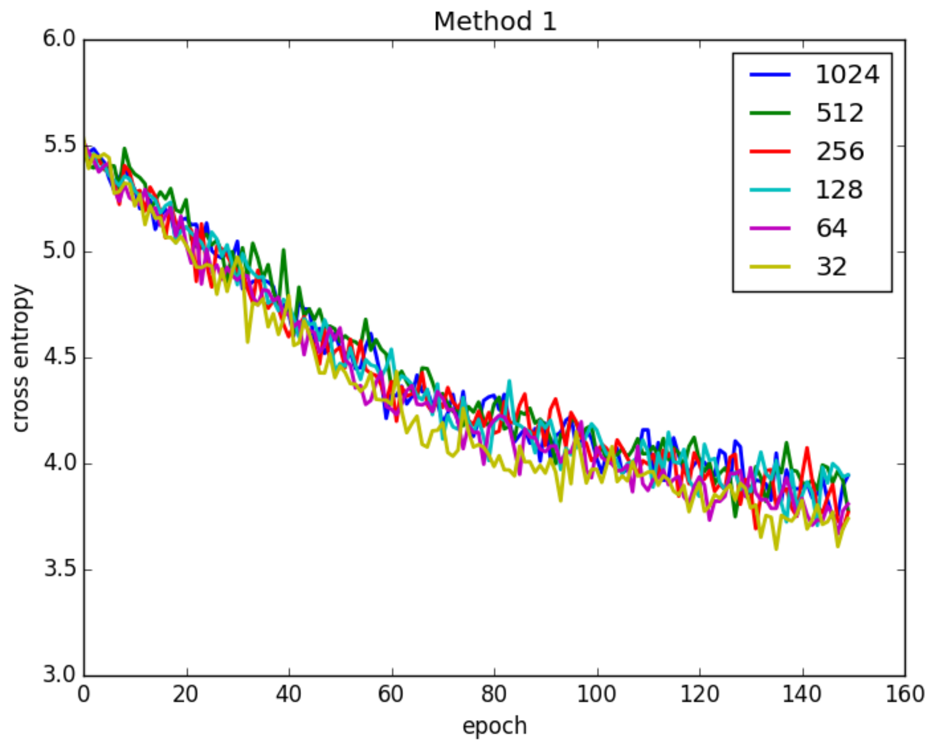


Figure 4.5 *Loss function across the training epochs when using* Method 1. *Legend on top-right corner indicates the size of the receptive field, which matches with input segments length.*

Method 1	32	64	128	256	512	1024
MSE (train set)	0,0039	0,0037	0,0037	0,0036	0,0036	0,0037
MSE (test set)	0,0039	0,0039	0,0038	0,0038	0,0039	0,0039

Table 4.4 *MSE after 150 training epochs. MSE in test barely varies probably due to the small size of the set.*

Method 2	32	64	128	256	512
MSE (train set)	0,0025	0,0026	0,0026	0,0027	0,0024
MSE (test set)	0,0039	0,0039	0,0038	0,0038	0,0038

Table 4.5 *MSE after 150 training epochs. MSE in test barely varies probably due to the small size of the set.*

4.4.4 Computational time

The main consequence of training with entire -or at least larger- signals is to make the processing faster. Yet it does not affect the manner the network learns the

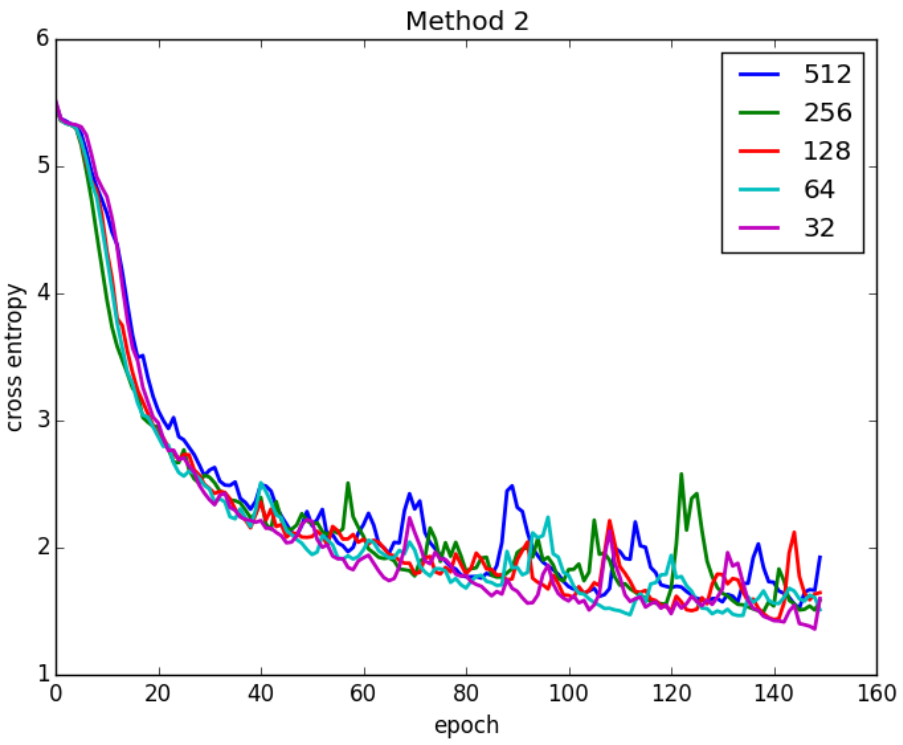


Figure 4.6 *Loss function across the training epochs when using* Method 2. *Input segments length is fixed to 4000-samples. Legend on top-right corner indicates the size of the receptive field.*

underlying structure of the data to predict next samples, it has a big impact on computational time needed for training.

Table 4.6 displays how *Method 2* can be over 500 times faster than the *first method*. The main ingredient that allows to dramatically increase the number of sequences processed per second is the amount of input/output pairs the network sees from a single input segment. In other words, an input/output pair ($\mathbf{x}$,$\mathbf{y}$) corresponds to an input segment and its target with same length as the receptive field of the network. The first method only provides one pair ($\mathbf{x}$,$\mathbf{y}$) because the input segment matches the receptive field size. However, the second method provides many more pairs ($\mathbf{x}$,$\mathbf{y}$) because we feed the network with input segments much larger than the receptive field, and it is the network who automatically generate these sub-segment pairs. For instance, *Method 2* in Table 4.6 (right) provides 3489 input/output pairs since we feed segments of 4000 samples to a network with receptive field 512. Directly, $4000 - 512 + 1 = 3489$ with target sequences that are same as input but shifted by one sample in time. Due to the benefit it implies, we do not need to train the network with so many initial input segments. That is why we halve the number of batches per epoch and still obtain better performance, as we could see in the subsequent results.

Method 1	**Method 2**
receptive field: 512	receptive field: 512
segment length: 512	segment length: 4000
i/o pairs: 1	i/o pairs: 3489
segments seen every epoch: 320	segments seen every epoch: 160
time/epoch: 34 s	time/epoch: 108 s
i/o pairs processed per second: $\lfloor \frac{1\times320}{34} \rfloor = 9$	i/o pairs processed per second: $\lfloor \frac{3489\times160}{108} \rfloor = 5168$

Table 4.6 *Comparison between the two methods studied. Method 2 on the right of the table is 574 times faster than method 1.*

Bar charts from Figure 4.7 and 4.8 depict computational time and amount of network parameters among a range of receptive field sizes training the *second model* network with the first and second method described above. Whereas Figure 4.7 shows the distribution when training with input segments matching the receptive field size, Figure 4.8 corresponds with a train of the network with same settings but larger segments -a fixed segment length of 500ms-.

Since we keep same network configuration in both experiments, the amount of parameters remain invariant and grows linearly with the receptive field. However, analyzing the trend of computational time needed in both cases, we easily verify it is independent of the receptive field size (see Figure 4.8) but follows an exponential dependence with *Method 1* approach (see Figure 4.7). Bar chart depicted in Figure 4.9 shows a linear growth of the elapsed time accordingly to the segment length.

4.4.5 Audio generation

After a thorough study of the deep CNN adopted as the preferred architecture, we select several configurations to proceed with the generation of new waveforms using signals from *set 2*. Best results were obtained with *Method 2*, so we directly consider this as unique to generation. Figure 4.10 serves as a visual verification and justify the fact that *Method 1* is inadequate for synthesis, even working with simple sines from *set 1*. As explained in previous chapters, we lack of an objective measure to evaluate the quality of the new auto-generated waves from *set 2* since the prediction does not continue the original seed shape. It implies we value the quality of generation based on our personal judgment, and, as stated above, we validate the new waveform provided that it is an audible segment without noise artifacts.

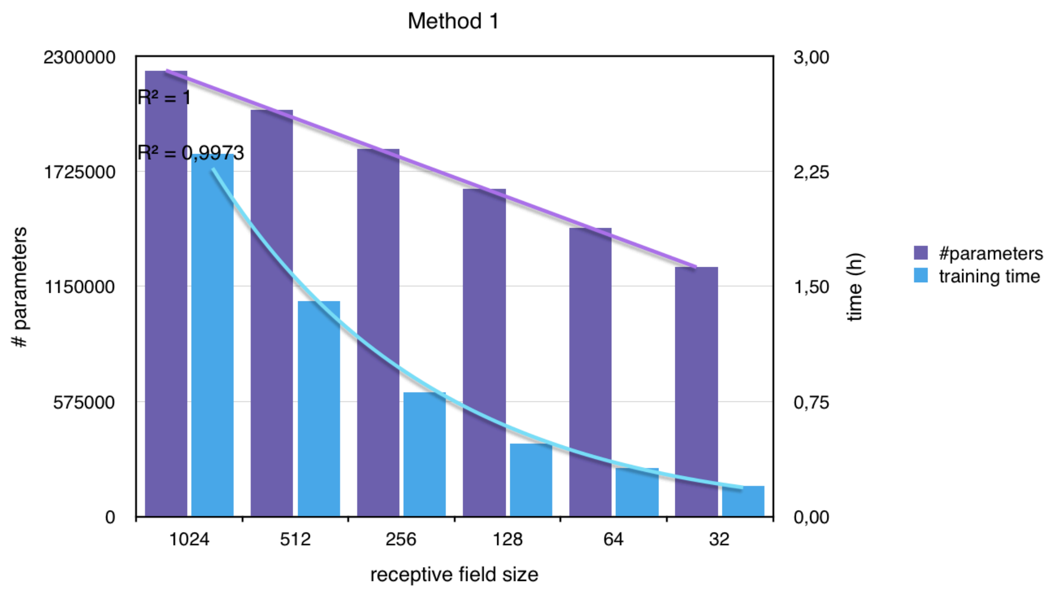


Figure 4.7 *Bar chart depicting amount of parameters (purple), computational training time (blue) for 150 epochs and its trend line with different receptive field sizes. Coefficient of determination (R^2) indicates linear and exponential regression perfectly fit the pertinent data. This experiment was performed with input segments to the network matching the size of its receptive field, batch size: 32x5.*

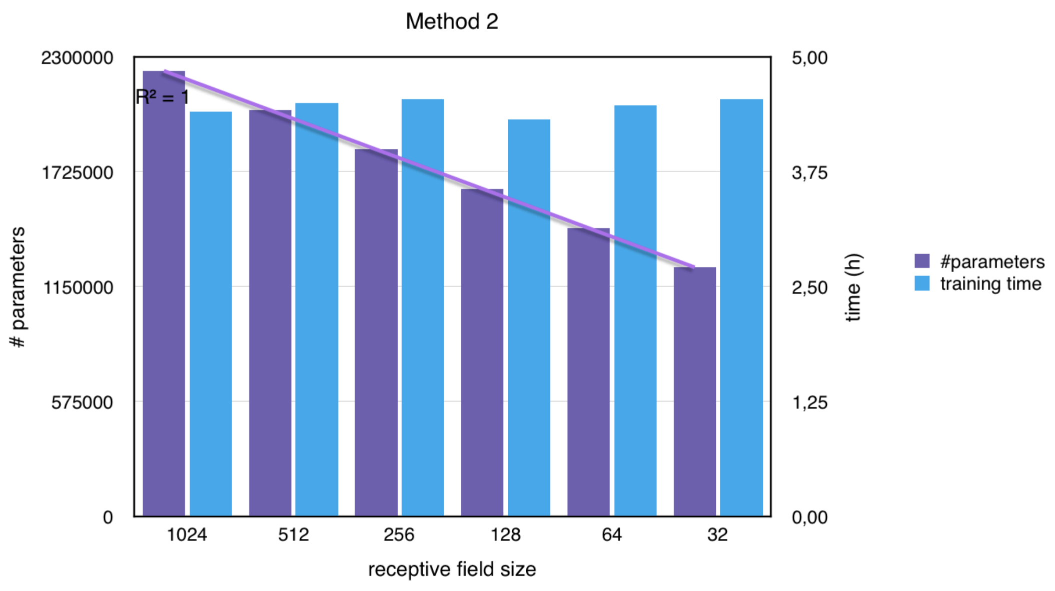


Figure 4.8 *Bar chart depicting amount of parameters (purple) and its trend line and computational training time (blue) for 150 epochs, with different receptive field sizes and batch size 32x5. Coefficient of determination (R^2) indicates linear regression perfectly fit the distribution. Notice that training time is approximately constant and independent of the receptive field size.*

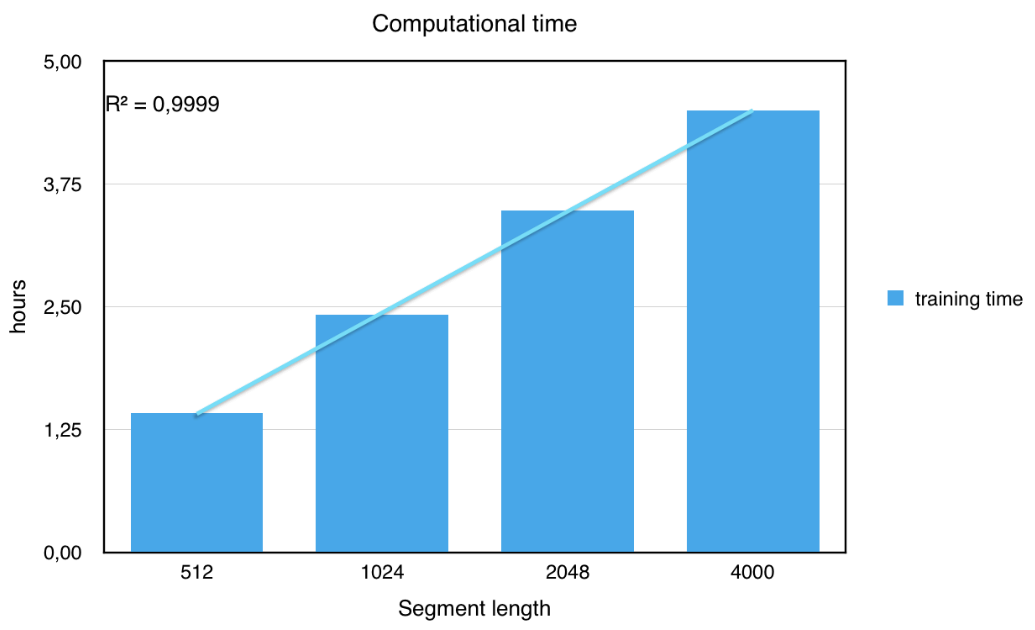


Figure 4.9 *Bar chart depicting computational training time needed for different segment lengths. Coefficient of determination (R^2) indicates linear regression perfectly fit the distribution. Receptive field is set to 512, batch size is 32x5.*

Before training again the model with more complex input data, we verify that it is actually capable to predict any simple sinusoid with a frequency within the training range, i.e., [100, 1000] Hz. We can see some examples in Figure 4.11 for networks with different receptive fields trained with 4000-sample input segments. All of them predict correctly the given sequence.

Nevertheless, network performance is strongly different if training with non stationary signals and the segment length has a bigger impact in the performance. Figure 4.12 show generation with a same network set up but trained with different segment lengths. It is clear the benefit of increasing the length of training sequences, yet we obtain more natural waves without presence of noise artifacts or constant intervals.

The vital importance of a model with large receptive field is shown in Figure 4.13, which supports the use of dilated convolutions to increase even more the receptive field size without greatly increasing computational cost. The generation with a receptive field of 256 results in an inconsistent waveform, where seems to be a concatenation of different pieces: from 0.05 to 0.09 seconds where could try to follow the trend of the original seed, from 0.09 to 0.14 seconds where it becomes more spiky and from 0.14 to the end, where it is just noise. Both networks were trained with 32x10 batches per epoch and 4000-sample length segments.

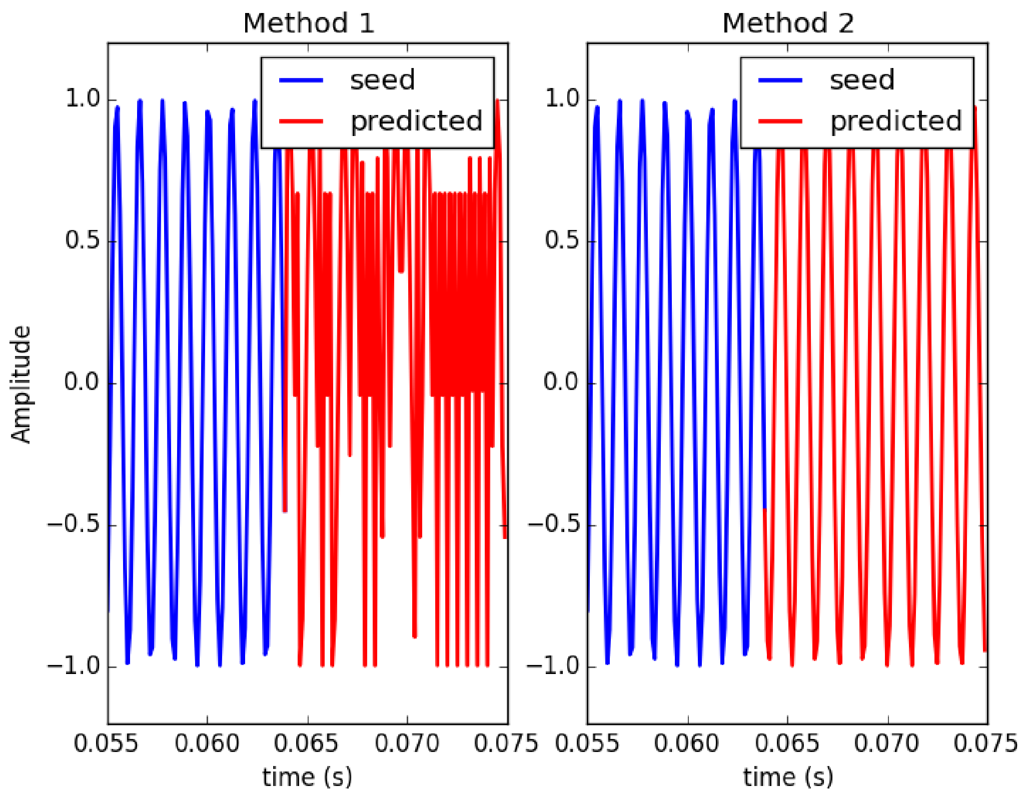


Figure 4.10 *Snippet of a generated sinusoid. The seed used is in blue; the continuation of the wave in red indicates the component predicted by the network. It is obvious that* Method 1 *(on the left) is not valid for generation since the prediction is not even close to a sinusoid. In both situations the network has a receptive field size of 512 and it was trained with 4000-sample segments, 32x10 batches, 150 epochs.*

Accordingly to what stated in Section 4.4.1, the more examples we train a network with, the better results we achieve. It becomes clear when we test generation with same network set up (512 receptive field, 4000-sample segments), but trained with 32x5 and 32x10 examples per epoch. In Figure 4.14 we can check how differ the outcome of the network. While the latter provides a reasonable good generation, the wave obtained after training the network with half of examples is rather unharmonious. Table 4.7 gather what we considered the best settings among all tested for generation of new waveforms, meeting computational requirements and results achieved.

4.5 Discussion

Feeding the network with sequences larger than the receptive field of the network outperform the approach described in *Method 1*. Enlarge the length of the inputs

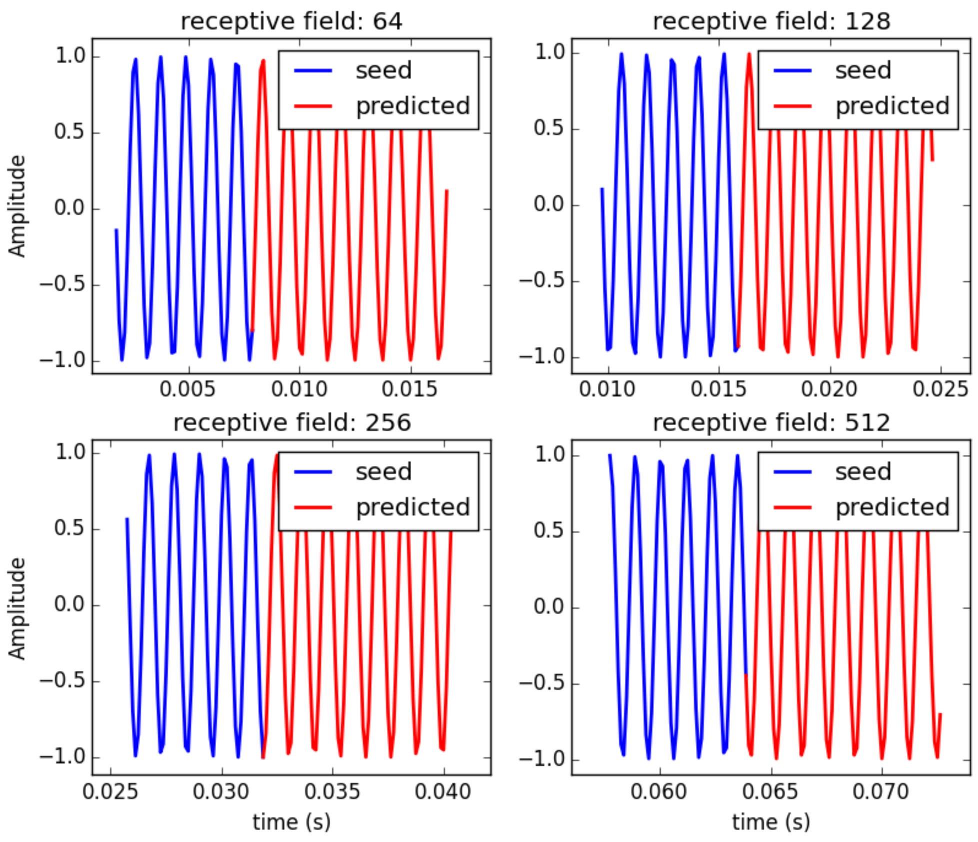


Figure 4.11 *Snippet of generated sinusoids. The seed used is in blue; the continuation of the wave in red indicates the component predicted by the network.*

receptive field size:	512
# filters:	256
filter length:	2
batches:	32x10
epochs:	150
loss function:	categorical crossentropy
optimizer:	ADAM
segment length:	4000 samples

Table 4.7 *Desired configuration of the CNN and input data for generation. Elapsed time in training: 8,5h.*

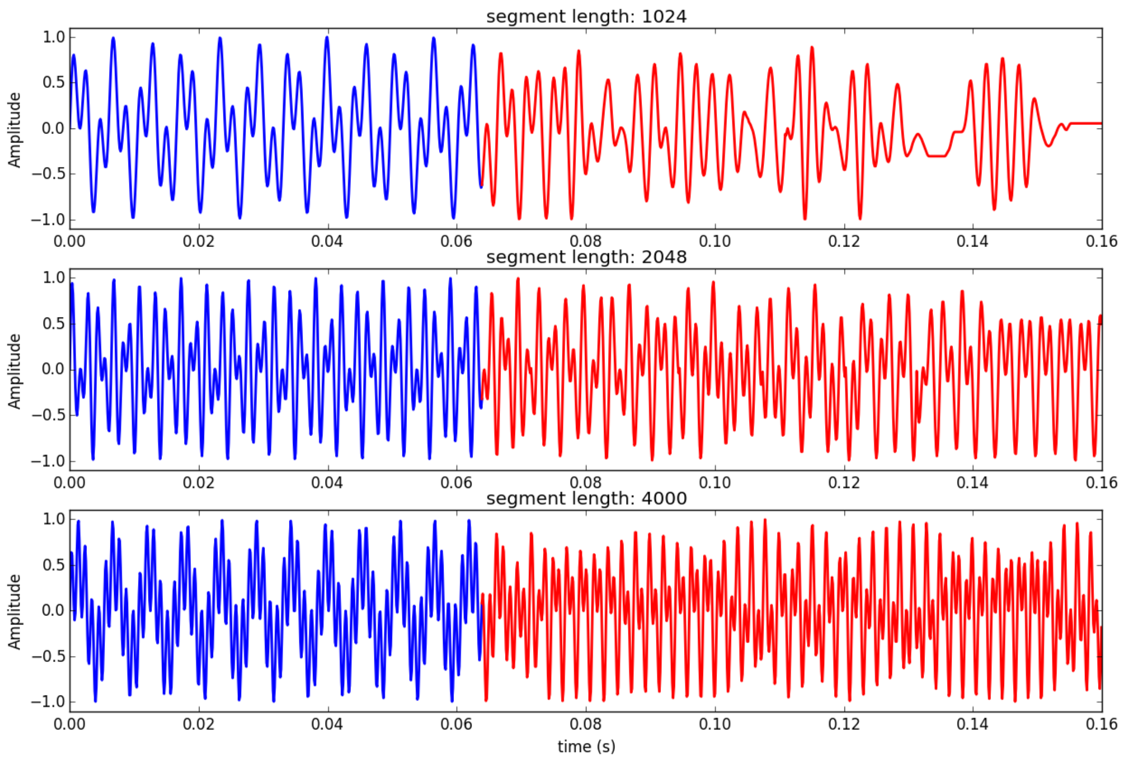


Figure 4.12 *Snippet of generated sinusoids. On top, a wave generated by a network trained with 1024-sample sequences; on the second row, a wave generated by same network trained with 2048-sample sequences; at the bottom, a wave generated by the same network trained with 4000-sample sequences. Different seeds used are depicted in blue; the continuation of the wave in red indicates the component predicted by the network.*

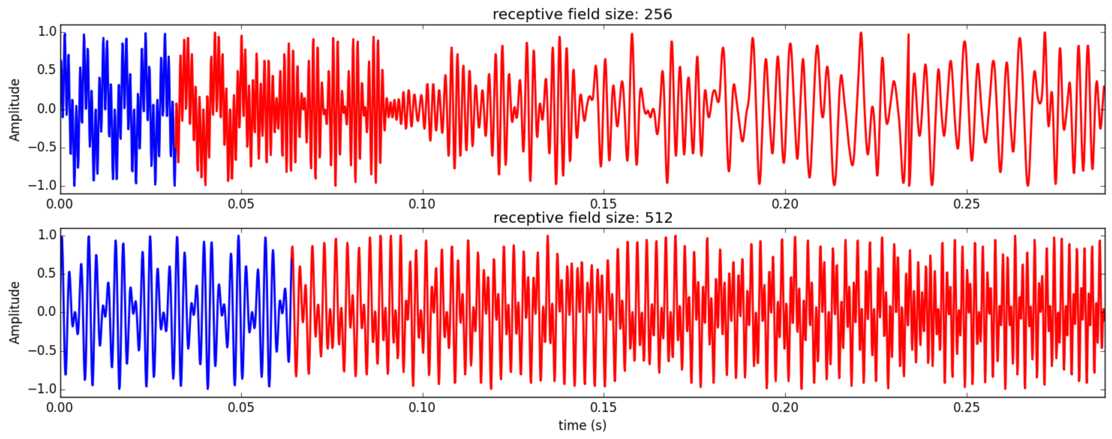


Figure 4.13 *Snippet of generated sinusoids. On top, a wave generated by a network of receptive field 256; below, a wave generated by a network of receptive field 512. Seed used is depicted in blue; the continuation of the wave in red indicates the component predicted by the network.*

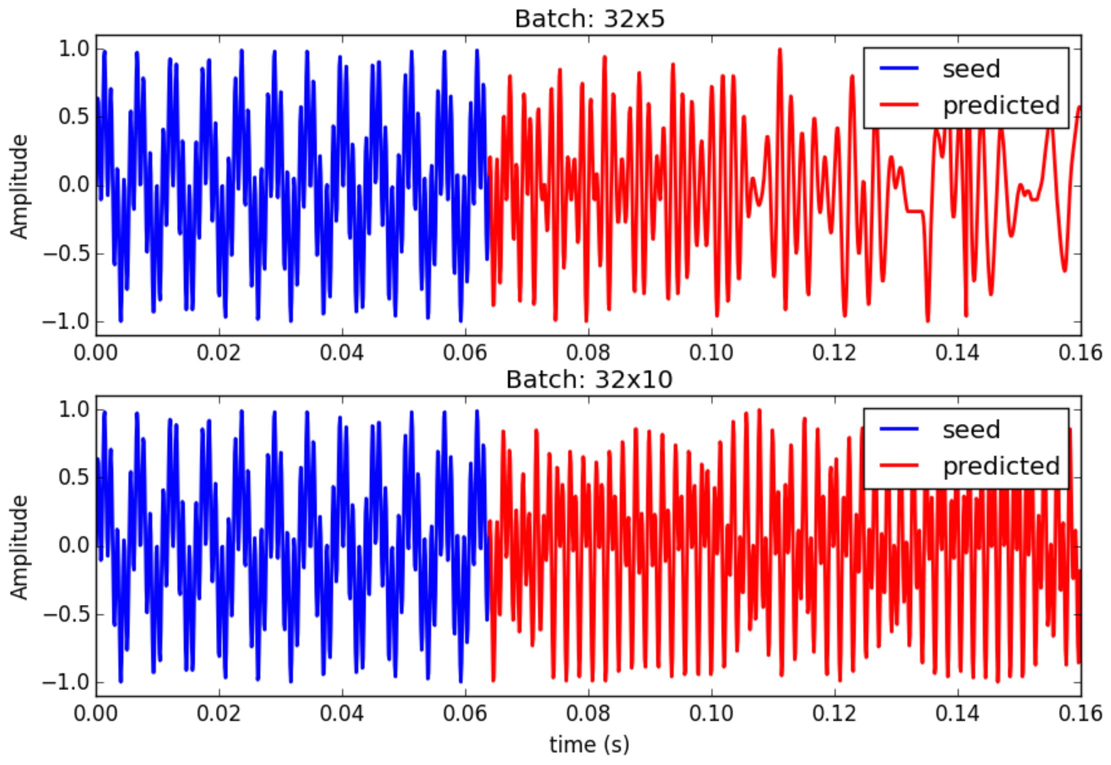


Figure 4.14 *Snippet of generated sinusoids. On top, a wave generated by a network trained with 32x5 batches every epoch; below, a wave generated by the same network trained with 32x10 batches every epoch.*

affects computational cost adversely, with a linear increase of the training time. However, it is evident that it is necessary in order to achieve a more continuous and natural synthesis and get rid of noise artifacts or spikes, although it does not modify the context information that a NN is using to do individual predictions.

In addition, large sequences entail a significant increment of the amount of examples seen by the network every epoch in a more efficient way than simply increasing the batch size, as proved with *Method 2* in Table 4.6. Despite the fact that objective measures reveal little improvement with the growth of input segment lengths once they have doubled the receptive field size, see Figure 4.4, it is an important factor in generation phase. Likewise, enlarging the receptive field of the network leads to a similar behavior. Waves synthesized from networks trained with longer input sequences or bigger receptive fields seem to be reasonably better, while if trained with shorter sequences (or, equally, shorter receptive field) the generation seems to be random, i.e., shape of the waveform at certain time is inconsistent with previous timesteps, and it is more prone to get stuck eventually and output a constant value.

As we are working with a deep network architecture, the size of the input dataset is crucial to achieve a proper training and meaningful results. Likewise, the amount of sequences shown to the network in every epoch has a key role and it has been evidenced either with objective measures and later in generation. Since we studied generation from simple signals constrained to a mixture of two sinusoids, we were able to synthesize new waveforms correctly with a modest database. Notice that in order to generate real audio signals, the input database should be much bigger as well as computational resources. As a reference, *WaveNet* model was trained with MagnaTagATune dataset, which consist of about 200 hours of music audio, and YouTube piano dataset, which consist of about 60 hours of solo piano music.

Overfitting was a persistent issue encountered in training the networks, which was noticeable with test signals from *set 1* mainly. However, our ultimate objective was the generation of new waveforms based on *set 2*, where overfitting did not suppose a major impediment. Including regularization techniques, more data augmentation methods or a bigger amount of sequences shown to the network every training epoch would possibly help to generalize the network performance but increasing overall computational time.

5. CONCLUSIONS

In this work we have studied the use of a deep convolutional neural network for audio generation. We have presented the fundamentals of CNNs and audio generation constrained to a mixture of two sinusoids. We studied an autoregressive model of deep CNN that operates directly at the waveform level.

Despite CNNs were not firstly intended to process temporal sequences due to the lack of feedback connections, which provide recurrent neural networks with past context information, nowadays they are capable of eventually modeling long-range temporal dependencies. The way many problems are approached has changed thanks to the development of a field called *deep learning*. A deep CNN built on dilated convolutions, such as the final architecture adapted from *WaveNet* model studied in this work, possesses a receptive field that exponentially grows with depth of the network. It increases the model capacity to model temporal correlations at longer timescales, which allows the model to generate new waveforms. The baseline CNN studied at the beginning of this work was not deep enough to model the long-range temporal dependencies in audio signals and was not suitable for generation. Therefore, we ended up facing the issue of properly training a deep architecture with the final network studied. We were capable to synthesize new waveforms correctly with a modest database thanks to the nature of input signals, albeit we could not avoid overfitting.

The approach proposed in *Method 2*, i.e., training the network with large input sequences, is more efficient and outperforms the training with segment lengths matching the receptive field size. After testing different training scenarios, we can conclude that the more training examples we feed to the network, the better generation we obtain in terms of a natural and harmonic sound. In addition, it also helps the reduce overfitting.

Due to the lack of an objective method to properly assess the quality of new synthesized signals, we approached the problem of generation as a classification task with pure tones in the first place. After predicting the next sample in a given sequence, we evaluated it and proceeded to either validate that model for generation or directly discard it. A new metric should be implemented in order to get more accurate judgments, equally valid in the previous classification stage and in posterior generation.

Having access to a powerful computational device such as a GPU is crucial to conduct research within deep learning field. Since training a deep network is already a computational expensive task, we should do further studies in order to code the models efficiently. At training time, we take advantage of GPU parallelization of the convolution operations. However, in generation phase, the predictions are sequential: after each sample is predicted, it is fed back into the network to predict the next sample. In [31] an efficient implementation of any generation model based on dilated convolution layers is presented. Their approach removes redundant convolution operations by caching previous calculations, greatly reducing computational complexity without sacrificing space complexity. Its inclusion in this work could be the following step to study a CNN at a deeper level.

Regarding the audio generation results, future work should concentrate on training the network with real audio databases. Another line of work that draws attention is to test the model studied in different applications, such as multi-speaker speech generation.

BIBLIOGRAPHY

[1] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, vol. 22, no. 10, pp. 1533–1545, 2014.

[2] N. Benvenuto and M. Zorzi, *Principles of Communications Networks and Systems*, 1st ed. US: John Wiley and Sons Inc, 2011.

[3] M. Caudill, "Neural networks primer, part i," *AI expert, vol.2, no.12*, pp. 46–52, 1987.

[4] K. Cho, *Foundations and Advances in Deep Learning.* Doctoral Thesis, Aalto University, 2014.

[5] R. Collobert and J. Weston, "A unified architecture for natural language processing: deep neural networks with multitask learning." ACM, 2008, pp. 160–167.

[6] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," 2016.

[7] K.-L. Du and M. N. S. Swamy, *Neural Networks and Statistical Learning*, 2014th ed. London: Springer London, 2014; 2013.

[8] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research, vol. 12*, pp. 2121–2159, 2011.

[9] K. Funahashi, "On the approximate realization of continuous mappings by neural networks," *Neural Networks, vol.2, no.3*, pp. 183–192, 1989.

[10] X. Glorot and Y. Bengio, "Understanding of the difficulty of training deep feedforward neural networks," *AISTATS 9*, pp. 249–256, 2010.

[11] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016, www.deeplearningbook.org.

[12] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *CoRR*, vol. abs/1410.5401, 2014.

[13] L. Grippo, A. Manno, and M. Sciandrone, "Decomposition techniques for multilayer perceptron training," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 11, pp. 2146–2159, 2016.

[14] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[15] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks, vol.2, no.5*, pp. 359–366, 1989.

[18] A. J. Hunt and A. W. Black, "Unit selection in a concatenative speech synthesis system using a large speech database," vol. 1. IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings (ICASSP), 1996, pp. 373–376.

[19] V. Kalingeri and S. Grandhe, "Music generation with deep learning," *CoRR*, vol. abs/1612.04928, 2016.

[20] S.-Y. Kang, X.-J. Qian, and H. Meng, "Multi-distribution deep belief network for speech synthesis." IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings (ICASSP), 2013, p. 80128016.

[21] O. Karaali, G. Corrigan, and I. Gerson, "Speech synthesis with neural networks," *World Congress on Neural Networks*, pp. 45–50, 1996.

[22] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.

[24] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[25] Z.-H. Ling, L. Deng, and D. Yu, "Modeling spectral envelopes using restricted boltzmann machines and deep belief networks for statistical parametric speech synthesis," *IEEE Transactions on Audio, Speech and Language Processing*, vol. 21, no. 10, p. 21292139, 2013.

[26] Z.-H. Ling, S.-Y. Kang, H. Zen, A. Senior, M. Schuster, X.-J. Qian, H. M. Meng, and L. Deng, "Deep learning for acoustic modeling in parametric speech generation: A systematic review of existing techniques and future trends," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 35–52, 2015.

[27] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics, vol.5, Issue 4*, pp. 115–133, December 1943.

[28] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Sample rnn: An unconditional end-to-end neural audio generation model," *CoRR*, vol. abs/1612.07837, 2016.

[29] A. Nayebi and M. Vitelli, "Gruv : Algorithmic music generation using recurrent neural networks," *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*, 2015.

[30] M. A. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015, neuralnetworksanddeeplearning.com.

[31] T. L. Paine, P. Khorrami, S. Chang, Y. Zhang, P. Ramachandran, M. A. Hasegawa-Johnson, and T. S. Huang, "Fast wavenet generation algorithm," *CoRR*, vol. abs/1611.09482, 2016.

[32] G. Parascandolo, *Recurrent neural networks for polyphonic sound event detection.* Master of Science Thesis, Tampere University of Technology, 2015.

[33] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," *CoRR*, vol. abs/1603.05279, 2016.

[34] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review, vol.65, no.6*, pp. 386–408, November 1958.

[35] D. E. Rumelhart and J. L. McClelland, *Parallel distributed processing: Vol. 1, Foundations / explorations in the microstructure of cognition.* Cambridge, MA: MIT Press, 1987.

[36] J. Schmidhuber, “Deep learning in neural networks: an overview,” *Neural networks : the official journal of the International Neural Network Society*, vol. 61, pp. 85–117, 2015; 2014.

[37] W. Schulze and B. van der Merwe, “Music generation with markov models,” *IEEE Multimedia*, vol. 18, no. 3, pp. 78–85, 2011.

[38] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014.

[39] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, “Deepface: Closing the gap to human-level performance in face verification.” IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 1701–1708.

[40] T. Tieleman and G. Hinton, “Lecture 6e: Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*, 2012.

[41] K. Tokuda, Y. Nankaku, T. Toda, H. Zen, J. Yamagishi, and K. Oura, “Speech synthesis based on hidden markov models,” *Proceedings of the IEEE*, vol. 101, no. 5, pp. 1234–1252, 2013.

[42] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “Wavenet: A generative model for raw audio,” *CoRR*, vol. abs/1609.03499, 2016.

[43] A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu, “Conditional image generation with pixelcnn decoders,” *CoRR*, vol. abs/1606.05328, 2016.

[44] P. J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* Doctoral Thesis, Harvard University, 1974.

[45] J. Weston, S. Chopra, and A. Bordes, “Memory networks,” *CoRR*, vol. abs/1410.3916, 2014.

[46] F. Yu and V. Koltun, “Multi-scale context aggregation by dilated convolutions,” *CoRR*, vol. abs/1511.07122, 2015.

[47] H. Ze, A. Senior, and M. Schuster, “Statistical parametric speech synthesis using deep neural networks.” IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings (ICASSP), 2013, pp. 7962–7966.

[48] H. Zen, A. Senior, and M. Schuster, "Statistical parametric speech synthesis using deep neural networks." IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings (ICASSP), 2013, p. 79627966.

[49] H. Zen, K. Tokuda, and A. W. Black, "Statistical parametric speech synthesis," *Speech Communication*, vol. 51, no. 11, pp. 1039–1064, 2009.

[50] H. Zen, K. Tokuda, and T. Kitamura, "Reformulating the hmm as a trajectory model by imposing explicit relationships between static and dynamic feature vector sequences," *Computer Speech & Language*, vol. 21, no. 1, pp. 153–173, 2007.