Audio Synthesis with Long-Short Term Memory Networks

Luca Naterop, Sandro Giacomuzzi

September 17, 2017

Abstract

[LN] Long-Short Term Memory Networks (LSTMs) have proven to be enormously successful for a number of classification problems. Results of that sort suggest that such models are in fact able show almost human-like performance on a range of creative tasks. In the theoretical section of this work, we try to address the deep question of whether Artificial Intelligence (AI) can be creative. We also propose a test that allows one to measure the quality of an AI composer. In a practical section, LSTMs are applied to the task of audio synthesis. We show that for the sort of problem at hand a simple LSTM cannot model the output distribution, therefore we combine the LSTM with a Mixture Density Network. We then train a large LSTM Network on short-time spectrograms which were obtained from works of Johann Sebastian Bach and Frederic Chopin. Spectrograms are then synthesized from the trained network, and a continuous signal is obtained by means of a spectrogram inversion. The resulting audio files show how the network has learned several landmarks which are required for AI composition such as replicating the spectrum of an instrument, learning to compose very simple melodies using the learned instrument, and even some basic well-known harmonic patterns that are at the root of almost every composition. Problems with steady states are met with mixture density networks. Multiple different possibilities for sampling from mixture density networks are proposed. Our model is implemented in tensorflow, and the code is made publicly available.

Blog Post: `https://medium.com/@LucaNaterop/6ac96021253d`

Contents

1 Introduction [LN]

1.1 Us And Them

There are many tasks for which modern-day algorithms unquestionably outperform humans. For instance, when it comes to doing arithmetic computations, then we do not stand a chance. In fact, computers have been able to outperform humans in arithmetic computations almost since the day they were invented. But not only in such trivial tasks have they become better than us. In 1996, Deep Blue, an IBM Computer designed to be good at playing chess [2] was able to beat grandmaster Gary Kasparov in a first. And in October 2015 Alpha Go, a computer developed by Google Deepmind [3] designed to play the board game Go, became the first computer to beat a master Go player without handicap on a full sized 19x19 board. Such victory can be seen as a major milestone in AI research. Most experts believed that an AI capable of beating a Go master is at least 5 years away. Yet by combining Monte Carlo tree search with Deep Neural Networks the researchers from Google Deepmind proved that even here, computers are now outperforming us.

There is a certain set of problems where it can be said that the machines are currently about as good as we are. The problem of safely driving a car from point A to point B falls could be identified as such a problem. Autopilots of recent models from car manufacturer Tesla have, at the time of writing, an accident rate of about 1 accident every 130 million miles. The United States national average is approximately 1 accident every 94 million miles. We, humans, are therefore at this point in time almost equally good as them. But that this is only a temporary situation. In a few years ahead, they will improve, because they can harness more data and more processing power. We, on the other hand, will a hard time improving ourselves.

1.2 Towards Creativity

There is still a wide range of tasks where we are light-years ahead of the machines. Most of such tasks where they are not even close to us have to do with abstract thinking, creativity or both. For example, to this day, no computer or algorithm has ever written something even remotely as revolutionary as Shakespeare's poetry. No computer has been able to deduce new theorems in mathematics or theoretical physics. And no computer has been able to compose music of the sort of which is even close to what the great composers of history were able to accomplish.

Many would argue that such activities which involve creativity or abstract thinking are only limited to humans and that machines will never be able to do such things. The scientific point of view sees things differently, of course. The authors of this work, for instance, would argue that they can in principle do those things. We argue that all these great things were done by real brains, which are in the end just gigantic mechanistic machines. However complex the neural network of our brain can possibly be, there was almost certainly no magic involved when Chopin was composing his works! Real Brains did these things, and since they do not rely on magic, we can build models of brains that would, one day, maybe create similar things.

1.3 AI Music [LN]

In this work, we look at state of the art programs that try to compose music. We also try to push forward the boundary of what has been achieved so far. Therefore, let us ask the central question:

Can computers compose good music; that is, music indistinguishable from man-made music?

What then, do we mean by good music? Isn't it rather subjective after all? Well yes, it certainly is, but in the end, there is an awful lot of people who would agree that J.S. Bach composed good music. And Mr. Bach has written *a lot* of good music. One way to measure the quality of a composer is thus by comparing it to works from a real composer.
Since - in music composition - AI is still enormously far behind human composers it would already be a major milestone if somebody would be able to create an AI composer that would be indistinguishable from a human composer. Therefore, we would like to propose a test which defines a milestone in the development of creative AI.
If you have programmed an AI composer, perform the steps

1. Tell your AI composer to compose music similar to a human composer, e.g. J.S. Bach. In case of a learning algorithm, train your algorithm with Bach's Music.

2. Find people who have some familiarity with Bach Music, but they must not know all music of Bach, otherwise results are biased. Let's say these people should have heard between 10% and 70% of all of J.S. Bach's compositions.

3. Make a double-blind study in which you play randomly compositions from your AI composer and real works of J.S. Bach.

4. If your subjects can't tell AI compositions from real compositions with statistical significance, then the AI composer has passed the test.

We would like to challenge researchers to come up with AI composers designed to pass this test. We haven't been able to accomplish this. In fact, we haven't even come close. A good AI composer might be 5 years or 50 years ahead, but it could also be that somebody does it in a few months. Let us now have a look at our own approach to the problem.

2 Model

The prediction of music, or more generally of digital signals, can be thought of a regression problem, where the explanatory variable is a given sample and the dependent variable corresponds to the next sample. Then the next sample does not only depend on the previous but in general on all the previous samples. This would be a huge amount of explanatory data, which seems to be very difficult to handle, especially because its size would change for every prediction. One way to deal with this problem is to choose only a certain amount of predecessors. However, there are also other approaches, where one tries to learn a mapping from the past data in a memory state of a fixed size.
Another thing we have to think of is the representation of the audio data. There is, of course, the raw audio data as discussed, but since audio signals clearly have a special structure we could perhaps get rid of a lot of redundancy by looking for a suitable transformation. We find that for example the amplitude of the signal is limited and the energy spectrum of short time Fourier Transformations (STFT) are often sparse. Also, the neighboring STFT seem to correlate highly - at least much higher than two sequenced samples in the time domain. So we decided to learn on the spectral representation of the signal.
For further simplicity, we threw away all frequencies above 4000 Hz and decided to get rid of the phase information, such that the reconstructed audio signal is still fine enough to hear if the algorithm actually is learning to compose. In fact, we use an iterative spectrogram inversion approximation algorithm that works very well, since the human ear is essentially phase-insensitive. As for a listener, the frequency resolution gets less important the higher the tone is, we squeezed the frequency scale in a logarithmic way. This kind of simplification is typically called *dimensionality reduction* of the feature space.

2.1 Neural Network

A common model for complex regression or classification problem, having a lot of data compared to the dimension of the feature space and the complexity of the solution space, is the so-called *neural network* inspired by the neurons of the brain. It consists of a cascade of linear and nonlinear mappings. In this way, it can be made arbitrary big and adaptable.

2.1.1 Feed Forward Neural Network (FFNN)

Mathematically the network learns a parameterized function with a parameter vector $\boldsymbol{\theta}$, where for a feed forward network the input argument is an n-dimensional feature-vector $\mathbf{x} = (\mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{n})$. The elements $x_1, \ldots, x_n$ are called features, n being the feature dimension. The function's output is an m-dimensional real valued prediction-vector $\mathbf{F}(\boldsymbol{\theta}, \mathbf{x}) = (\mathrm{f}_1(\mathbf{x}), \ldots, \mathrm{f}_\mathrm{m}(\mathbf{x}))$, one element for each predictor. The network's complexity is characterized by the number and the size of so called *hidden layers*, which basically add parameter, that is, degrees of freedom to the network. The parameters act as weights between two neurons. Between each layer, there is a nonlinear *activation function*, which gives the network the ability to learn non-linear dependencies. Typical activation functions are the sigmoid σ and the hyperbolic tangent *tanh* functions. A schema of a feed forward network with two equally sized hidden layers

is shown in figure 1.

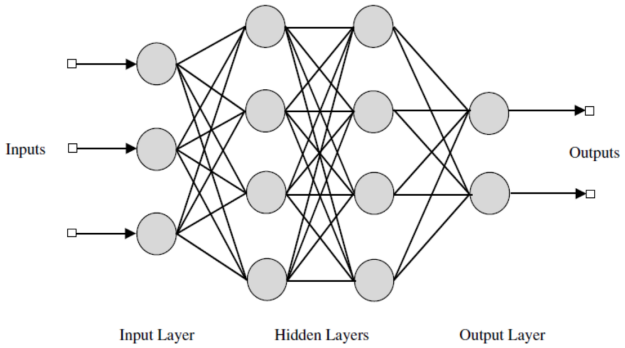


Figure 1: Feed Forward Neural Network with two hidden layers, [5]

The less features we have, the smaller is the network and the faster are the learning iterations. But sometimes the chosen features don't contain enough information to predict well. Or even worse, we can't think of good features at all. Luckily we live in time of fast processors, which allows us to just not care much about the networks' size. So we could try to "blindly" feed every bit of information we have into the network and hope for the best. A big network with a lot of hidden layers is capable of learning complex structures in the feature space and reducing its dimension respectively filtering out redundant information. One also calls this approach *deep learning* . Unfortunately, the more degrees of freedom our model has, the more data we need for not running into the risk of *over fitting* the data set. For our problem this seems not to be a danger, since J.S. Bach gave us quite a huge amount of data.
So a neural network typically contains a ton of parameters which can be modified during the learning process. Finding the best-fitting network function is an optimization problem over the network-parameters for a given ordered set of T samples $\{\mathbf{x_t}\}_{1\leq t\leq T}$. That means, one has to define an error-function as a measure of how well/bad an estimation is, and an algorithm that describes the rule of how to modify the parameters for a given error. We could assume that there exists a true Function $\mathbf{F}(\mathbf{x}) = (\mathrm{f_1}(\mathbf{x}), ..., \mathrm{f_m}(\mathbf{x}))$ describing the predictors perfectly. Let's now add some Gaussian noise for every output $1, ..., m$, which represents the uncertainty, caused by lack of information or other influences that results in randomness. One can then show, that the squared error function gives the best estimation, in the sense that it maximizes the likelihood of the training data. Let $\hat{\mathbf{y}} = (\hat{\mathrm{y}}_1, ..., \hat{\mathrm{y}}_\mathrm{m})$ be the prediction and $\mathbf{y} = (\mathrm{y}_1, ..., \mathrm{y}_\mathrm{m})$ be the true value. The squared error is then given by:

$$e(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{\mathrm{i}=1}^{\mathrm{m}} (\hat{\mathrm{y}}_\mathrm{i} - \mathrm{y}_\mathrm{i})^2 = \|\hat{\mathbf{y}} - \mathbf{y}\|^2$$

2.1.2 Mixture Density Network (MDN)

Minimizing the least squared error can be turned in a equivalent problem of fitting a Gaussian probability density function over the data, such that its mean

and variance maximize the likelihood of the training data.

$$\hat{\theta} = \arg\min_{\theta}\{\left\|\hat{\mathbf{y}}(\theta) - \mathbf{y}\right\|^2\} \hat{=} \arg\max_{\theta}\{\frac{1}{\sqrt{(2\pi\sigma^2)^{\mathrm{m}}}}\exp\big[-\left\|\hat{\mathbf{y}}(\theta) - \mathbf{y}\right\|^2/2\sigma^2\big]\}$$

The estimator for the maximum-likelihood is the mean of the Gaussian. What if the data can't be modeled well by a (single) Gaussian? Let's suppose that the dependent value $\mathbf{y}(\mathbf{x})$ of a given input $\mathbf{x}$ has either high or low values. The best fitting Gaussian would have its mean somewhere intermediate. This means that we can't capture the two level pattern at all with this model and the estimator would be very bad! To overcome this problem one can introduce a mixture density network, which uses a mixture of K Gaussians as a model. Such a model was first introduced by Bishop [6]. For an m-dimensional problem, treating each dimension independently, there are $3 * K * m$ parameter needed to describe the mixture distribution since every dimension need K weighting coefficients π, K means μ and K variances σ^2. To overcome this massive increase in the networks output, we use parameter sharing, where the variances and the weights are the same for all dimensions, resulting in $2*K+K*m$ output units. This is equivalent to a m-dimensional Gaussian, mixture model of only K multivariate Gaussian each having its own weight π_k and a diagonal covariance matrix of the form $diag(\sigma_k)$. Let $M \equiv \{\pi_k, \sigma_k, \boldsymbol{\mu_k}\}_{\mathrm{k=1}}^{\mathrm{K}}$ be the set of the mixture-parameters, then

$$P(\mathbf{y}|\mathrm{M}(\mathbf{x})) = \sum_{\mathrm{k=1}}^{\mathrm{K}} \pi_{\mathrm{k}}(\mathbf{x}) \frac{1}{\sqrt{(2\pi\sigma_{\mathrm{k}}^2(\mathbf{x}))^{\mathrm{m}}}}\exp\big[-\left\|\boldsymbol{\mu_k}(\mathbf{x}) - \mathbf{y}\right\|^2/2\sigma_{\mathrm{k}}^2(\mathbf{x})\big]$$

is the likelihood of a given data point $\mathbf{y}$. The mixture coefficients π_k and the corresponding means and variances $\boldsymbol{\mu}_k$, σ_k need to be learned by the model. The likelihood of the whole training data set $\mathbf{Y} = (\mathbf{y_1}, ..., \mathbf{y_N})$ for independent data points is given by

$$P(\mathbf{Y}|\mathrm{M}(\mathbf{X})) = \prod_{\mathrm{n=1}}^{\mathrm{N}} \mathrm{P}(\mathbf{y_n}|\mathrm{M}(\mathbf{x_n}))$$

The parameter M of the mixture model can be found by the neural network in a maximum-likelihood approach by maximizing $P(\mathbf{Y}|\mathrm{M}(\mathbf{X}))$. Since it's common to use a minimization algorithms for the training process, the corresponding error function is of an "easy" form by taking the negative logarithm of the likelihood.

$$e(M, \mathbf{y}) = -\ln(\mathrm{P}(\mathbf{y}|\mathrm{M})$$

Therefore the total error is given by

$$E(M, \mathbf{Y}) = -\ln(\mathrm{P}(\mathbf{Y}|\mathrm{M})) = \sum_{\mathrm{n=1}}^{\mathrm{N}} \mathrm{e}(\mathrm{M}, \mathbf{y_n})$$

The trained mixture model can now capture clusters like the ones in our two level example.
In such a mixture model we can think of multiple possibilities to estimate the dependent variable of a new (unseen) data point:

1. **Greedy Maximum Likelihood:** If the means of the Gaussian mixtures are not too close to each other compared to their variances, the maximum likelihood corresponds to the mean of the Gaussian with a maximum mixture coefficient, divided by their standard deviation (since the Gaussian distribution is reciprocally proportional to it). We will call the resulting mixture coefficient after division and normalization the standardized mixture coefficients $\hat{\pi}_k$, i.e.

$$\hat{\pi}_k = \frac{\pi_k/\sigma_k}{\sum_{i=1}^{K} \pi_i/\sigma_i}$$

The greedy maximum likelihood sample rule for a new data point $\mathbf{x_{n+1}}$ is:

$$\mathbf{y_{n+1}} = \boldsymbol{\mu}_{\mathrm{i}}$$
$$i = \arg\max_k \left\{ \hat{\pi}_k(\mathbf{x_{n+1}}) \right\}$$

2. **Exact Maximum Likelihood:** With a bit more computational effort we could estimate the maximizer also for the cases when some Gaussian are close to each other. Formally we have

$$\mathbf{y_{n+1}} = \arg\max_{\mathbf{y}} \left\{ \mathrm{P}(\mathbf{y}|\mathrm{M}(\mathbf{x_{n+1}})) \right\}$$

3. **Tempered Sampling:** In a game theoretic setting we would bet on the most probable event. But in some cases it is more interesting to bet on more realistic events. Let's assume that in the two level example from above the higher value is more probable than the lower. If we want to estimate a sequence of ten (independent) data points, the most probable estimator would be the value "high" for every data point. But this seems like a very unrealistic outcome and one could without problems distinguish the artificially generated estimator from a randomly chosen set of ten values from the training data since we expect to have not only points with a high value but also some with a low value. In words lent from information theory we could state that such a sequence is not typical with respect to its entropy. To generate a more realistic estimator we could define a certain subset S of the means as possible outcomes. We may introduce a parameter $\tau \in [0, 1]$ describing the amount of probability contained in the largest $|S|$ standardized mixture coefficients.
Let I be an index function, such that $\hat{\pi}_{I(1)} \geq \hat{\pi}_{I(2)} \geq ... \geq \hat{\pi}_{I(K)}$. We can then define the tempered subset S as

$$S = \left\{ \boldsymbol{\mu}_{I(1)} \right\} \cup \left\{ \boldsymbol{\mu}_{I(1)}, \boldsymbol{\mu}_{I(2)} ..., \boldsymbol{\mu}_{I(|S|)} : \tau \geq \sum_{i=1}^{|S|} \hat{\pi}_{I(i)} \right\}$$

We can now sample from S with probability proportional to the corresponding standardized mixture coefficients. Let $s = 1, 2, ..., |S|$, then

$$P(\boldsymbol{\mu}_{I(s)}) = \frac{\hat{\pi}_{I(s)}}{\sum_{i=1}^{|S|} \hat{\pi}_{I(i)}},$$

The estimator $\mathbf{y_{n+1}}$ is therefore drawn from a discrete distribution with $|S|$ possible outcomes, i.e.

$$\mathbf{y_{n+1,s}} = \boldsymbol{\mu}_{\mathrm{s}}(\mathbf{x_{n+1}})$$
$$P(\mathbf{y_{n+1,s}}) = \mathrm{P}(\boldsymbol{\mu}_{\mathrm{s}})$$

Tempered sampling enables the network to explore less obvious solutions and make it easier to find transitions between learned patterns. In our opinion this seems to be an important step towards artificial creativity. Note that setting $\tau = 0$ is equivalent to greedy maximum likelihood sampling.

4. **Sample from probability distribution**
   Of course if we believe in the probability distribution inferred from the data we should sample directly from that distribution to get the most realistic result, i.e. draw $\mathbf{y_{n+1}}$ from $P(\mathbf{y}|\mathrm{M}(\mathbf{x_{n+1}}))$.

2.1.3 Recurrent Neural Network (RNN)

How could we modify our network model if we want to predict an ordered series of sample-vectors $\mathbf{X_T} = (\mathbf{x_1}, \ldots, \mathbf{x_t}, \ldots, \mathbf{x_T})$, where $\mathbf{X}$ denotes the $n \times T$-sample-matrix?
According to our problem we associate the series with samples at different time-stamps $t \in \{1, \ldots, T\}$. As long as the prediction of the sample at time t is independent of the others, meaning that reordering the sample-matrix would have no influence of the net's performance, we don't have to change our feed forward Neural Network at all. But if we want to give our model the option to learn dependencies of previous samples of the past, we could come up with the so called *recurrent neural network*. For this approach the network receives a memory-vector $\mathbf{h_t}$, a so called *hidden state*. In every time step a sub-network maps the current sample $\mathbf{x_t}$ and the previous hidden state $\mathbf{h_{t-1}}$ to the new hidden state $\mathbf{h_t}$ in a recurrent way as shown in figure 2. Let's call that mapping-function $\mathbf{A}(\mathbf{x_t}, \mathbf{h_{t-1}}) = \mathbf{h_t}$. Another sub-network $\mathbf{F}(\cdot)$ maps the current hidden state $\mathbf{h_t}$ to the prediction-vector $\hat{\mathbf{y}}_\mathbf{t}$. Mathematically spoken we want our network to be capable to learn any conditioned predictive function of the form:

$$\hat{\mathbf{y}}_\mathbf{t}|\mathbf{X_{t-1}} \;\widehat{=}\; \mathbf{F}(\boldsymbol{\theta}, \mathbf{h_t}),\ \forall\, \mathrm{t} \in \{1, \ldots, \mathrm{T}\}$$

with the recursive rule:

$$\mathbf{h_t} = \mathbf{A}(\mathbf{x_t}, \mathbf{h_{t-1}})$$

It is straight forward to build a multi layer RNN, where L different recurrent networks (the A-boxes of figure 2) are cascaded before they map on the output. The predictive function becomes:

$$\hat{\mathbf{y}}_\mathbf{t}|\mathbf{X_{t-1}} \;\widehat{=}\; \mathbf{F}(\boldsymbol{\theta}, \mathbf{h_{L,t}}),\ \forall\, \mathrm{t} \in \{1, \ldots, \mathrm{T}\}$$

with the corresponding recursive rule:

$$\mathbf{h_{1,t}} = \mathbf{A}(\mathbf{x_t}, \mathbf{h_{1,t-1}})$$
$$\mathbf{h_{l+1,t}} = \mathbf{A_l}(\mathbf{h_{l-1,t}}, \mathbf{h_{l,t-1}}),\ \forall\ \in \{2, \ldots, \mathrm{L}-1\}$$

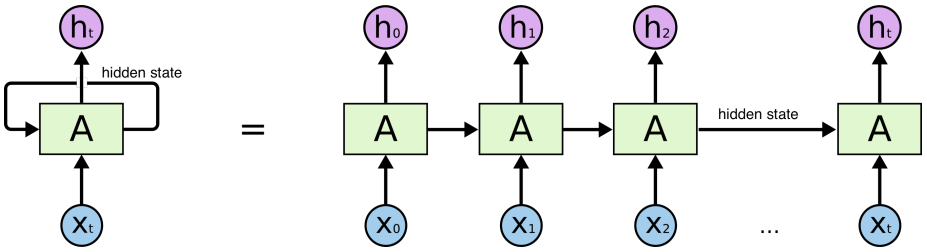


Figure 2: Schema of a 1-layer RNN unrolled in time, [7]

2.1.4 Long Short-Term Memory (LSTM)

For our model we have chosen a special kind of RNN. It is called *long short-term memory* and was proposed in the 1997 by Jürgen Schmidhuber. A LSTM outperforms a vanilla RNN significantly if it the Data has long time dependencies. The recursive function is extended by a third argument, the so called *cell state* c_t, which is of the same size as the hidden state and acts like a passive filter on the hidden state. There is a clever arrangement of three gates, controlling the modification and the filtering operation of the cell state. The *forget gate* gives the cell state the possibility to forget unnecessary information, while the *input gate* controls the addition of new information to the cell state. Finally, an *output gate* controls the effective filtering of the hidden state by the cell state, figure 3. Every gate is trainable by a sub-network. We note that the recursive formulation of a LSTM is extended to:

$$[\mathbf{h_t}, \mathbf{c_t}] = \mathbf{A}(\mathbf{x_t}, \mathbf{h_{t-1}}, \mathbf{c_{t-1}})$$

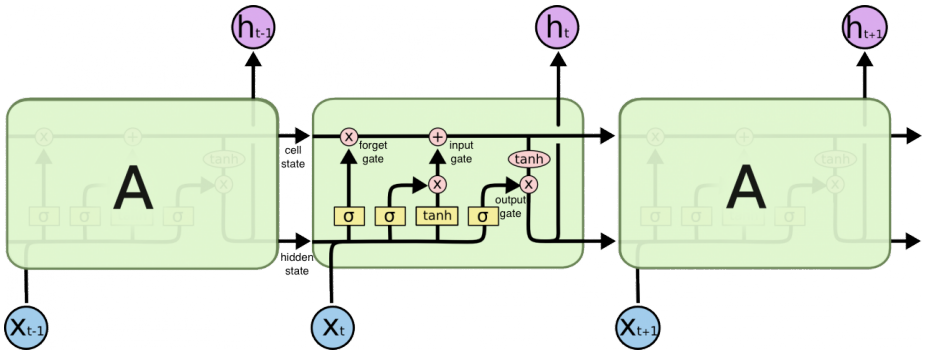


Figure 3: Schema of a LSTM recursion, [7]

2.2 Generative model

Lets assume having a sample-matrix $\mathbf{X_{T-1}}$. We want to estimate the next sample-vector $\hat{\mathbf{x}}_\mathbf{T}$ under the condition of $\mathbf{X_{T-1}}$. By choosing the dimension of the prediction-vector the same as the feature dimension (n=m), we can reinterpret the output of the RNN model as the new sample-vector. Once the network

is trained, we can generate sound recursively with the formula:

$$\hat{\mathbf{y}}_\mathbf{t}|\mathbf{X}_{\mathbf{t-1}} \;\widehat{=}\; \hat{\mathbf{x}}_\mathbf{t} = \mathbf{F}(\boldsymbol{\theta}, \mathbf{h}_\mathbf{t}),$$

given an initial conditions $(\mathbf{x_0}, \mathbf{h_0}, \mathbf{c_0})$.

2.3 Gradient descent

The most common way to optimize the parameters of a neural network is called *gradient descent*, where the gradient of the error function's output with respect to the parameter is calculated analytically and small changes on the parameters, proportional to the corresponding differential, are applied. Because we kind a follow the path of the error back to the responsible parameter by excessively applying the chain rule, one call this process *back propagation*. However, dealing with a RNN, simple back propagation through the network is not enough. Since the error is depending on all the sample-vectors of the past, the RNN can be thought as unrolled in time and the gradient is then calculated with back propagation through time. Let $e(\mathbf{x_{t+1}}, \hat{\mathbf{x}}_\mathbf{t+1})$ be the error function of the true next sample $\mathbf{x_{t+1}}$ and the estimating output $\hat{\mathbf{x}}_\mathbf{t+1} = \mathbf{F}(\boldsymbol{\theta}, \mathbf{h_t}(\hat{\mathbf{x}}_\mathbf{t}, \mathbf{h_{t-1}}))$, where $\boldsymbol{\theta}$ is the parameter-vector of the network and $\hat{\mathbf{x}}_t$ is calculated with the recursion formula $\hat{\mathbf{x}}_\mathbf{t} = \mathbf{F}(\boldsymbol{\theta}, \mathbf{h_{t-1}}(\hat{\mathbf{x}}_\mathbf{t-1}, \mathbf{h_{t-2}}))$. The gradient is then given as:

$$\nabla_{\boldsymbol{\theta}} e(\mathbf{x_{t+1}}, \hat{\mathbf{x}}_\mathbf{t+1}) \;\widehat{=}\; (\mathrm{e}_{\theta_1}, \ldots, \mathrm{e}_{\theta_\mathrm{M}})$$

where M is the amount of parameters in the network. An optimization step with gradient descent over the whole epoch of T samples is given as:

$$\boldsymbol{\theta_{new}} = \boldsymbol{\theta_{old}} - \alpha \sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} e(\mathbf{x_t}, \hat{\mathbf{x}}_\mathbf{t})$$

where α is a positive real number controlling the step size. There are plenty modifications of gradient based optimization algorithm such as stochastic gradient descent (SGD), where only a subset of the training data is used in every training step. However, in practice it is too time-consuming to unroll the network all the way back to $t = 1$ and usually this is also useless, since there are no such ultra long term dependencies. Therefore one applies the back propagation through time only for a certain amount of time steps $\hat{T}$. As a consequence the network can't learn sample dependencies longer than $\hat{T}$. Since for every data point of the training set, there must be given at least the $\hat{T} - 1$ predecessors, the smallest possible decomposition of the training set for SGD is the set of $T - \hat{T} + 1$ different data batches, each consists of $\hat{T}$ serial points:

$$\left\{\mathbf{x_t}\right\}_{\mathbf{t=1}}^{\mathbf{T}} = \biguplus_{\mathbf{t=1}}^{\mathbf{T-\hat{T}+1}} \left\{\mathbf{x_i}\right\}_{\mathbf{i=t}}^{\mathbf{t+\hat{T}-1}}$$

A gradient descent step over the whole epoch of T samples can be written as:

$$\boldsymbol{\theta_{new}} = \boldsymbol{\theta_{old}} - \alpha \sum_{t=\hat{T}}^{T} \nabla_{\boldsymbol{\theta}} e\big(\mathbf{x_t}, \hat{\mathbf{x}}_\mathbf{t} | \left\{\mathbf{x_i}\right\}_{\mathbf{i=t-\hat{T}+1}}^{\mathbf{t}}\big)$$

2.4 Final Model

2.4.1 Preprocessing

The goal of data preprocessing is to get rid of redundant information and find a data representation, that result in good pattern for the given learning problem. As a first step of dimensionality reduction, we downsampled the signal to a sample rate of 8 kHz.
Since we are working on audio signals, we must take psycho acoustic effects in to account. A human ear is less sensitive to the amplitude of low frequencies than for high frequencies. To capture this behavior, we applied an A-weighting filter to the signal. Next we made a short-term Fourier Transformation with 75% overlap and 32 ms time resolution. For further dimensionality reduction we threw the phase information away, which results in an amplitude spectrogram. In a last step, we used the fact that a human ear looses its absolute frequency sensitivity towards higher frequencies in a logarithmic fashion, such that we can build larger frequency bins towards higher frequencies in which we average the corresponding amplitudes. In the end, we have reduced the Data size by over 90%.
The t-th column of the reduced amplitude spectrum is associated with a feature-vector $\mathbf{x_t}$ and is in that case 100 dimensional.

2.4.2 Learning

For the learning task we mounted a MDN with Gaussian kernels on a multi layer LSTM neural network. Such a network has basically three hyper-parameters: The number of LSTM-layers, the number of mixture components and the hidden state size in the LSTM. We experimented with networks that have around 5 layers, 10 mixture components and a hidden state of around 100 dimensions. For the network optimization we need to choose another three important hyper-parameters: Step size of the gradient descent, batch size for SGD and the maximum back-propagation time. With these hyper-parameter we played around in all orders of magnitude, depending on the performance of our network.

2.4.3 Post-processing

The post-processing step is basically the inversion of the preprocessing. The most difficult part is the pseudo inversion of the amplitude spectrogram. The true inversion does not exists, since we threw the phase information away and therefore the resulting mapping is not bijective anymore. However, by claiming that the audio signal is smooth, there are algorithms looking for such solutions. We found an algorithm, written by Malcolm Slaney, Microsoft Research (July 2014) [8]. The Idea there is to find iteratively the best time delay between the summed short time inverse Fourier transformations an the next frame, such that those transitions are consistent.

3 Implementation and Execution [LN]

As a starting point for our implementation we were inspired by Andrej Karpathy's implementation of his LSTM for character language models (which can be found under `https://github.com/karpathy/char-rnn`). His work inspired us to use a similar network architecture, but we used different computational tools: Tensorflow as a framework for doing scientific computations, and python as the language that provides tensorflow bindings. Originally, we started working on our own implementation in torch of recurrent neural networks for regression problems, which we would then transform into our own variant of an LSTM implementation. But we found our own implementation to be rather slow and non-trivial to parallelize. Parallelization was important since we did not intend to train the network on GPUs but instead on many CPUs in parallel. We thus decided to use tensorflow with would give us rather good parallelization for free.

The code which produced the results is open-source available and can be found (without the data corpus) on github under

`https://github.com/LucNaterop/cmc-rnn`.

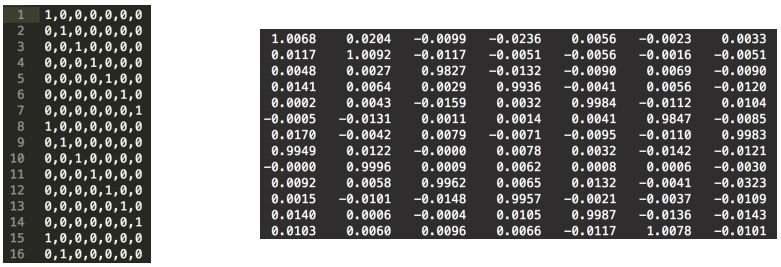
In the readme file on the github repo a description is given for how to run the code with your own data corpus.

We executed the code on ETHs EULER cluster. For the main simulation on the large Bach music corpus, we used 16 Intel Xeon E7-8837 CPUs (with 8 cores) and trained the network for approximately 10 days. Unfortunately, we were not able to train the net with GPUs, since the cluster's old GPUs have been decommissioned.

4 Results

4.1 Results from Dummy Data

If we think of it as a black-box, the network works like this: It takes a vector, given as a line of a csv file, as it's input and tries to predict the next vector in the sequence; i.e. the next line of the csv file. Line i in the csv file therefore corresponds to the vector at time t. In order to test our network, it was trained on simple data. First, it was trained on a simple "cyclic" data set from which an excerpt is given in figure 4 on the left. Then, we've let the network synthesize from it's trained state a sequence of vectors. This produced the output in figure 4 on the right. We observe that after only a short time the network has learned the patterns behind the data very well, with deviations around $\Delta \approx 0.01$.

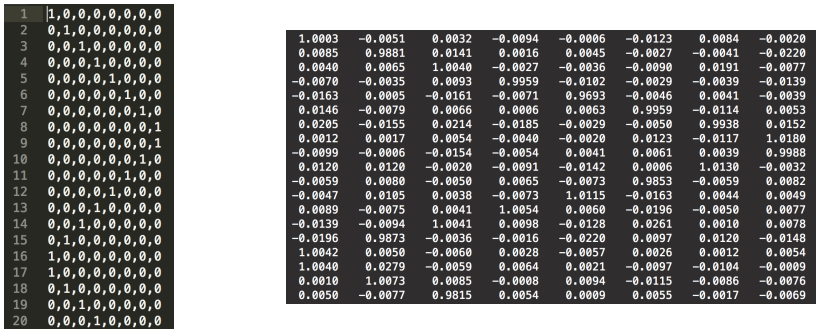


```
1,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0
0,0,1,0,0,0,0,0
0,0,0,1,0,0,0,0
0,0,0,0,1,0,0,0
0,0,0,0,0,1,0,0
0,0,0,0,0,0,1,0
1,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0
0,0,1,0,0,0,0,0
0,0,0,1,0,0,0,0
0,0,0,0,1,0,0,0
0,0,0,0,0,1,0,0
0,0,0,0,0,0,1,0
1,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0
```

```
 1.0068   0.0204  -0.0099  -0.0236   0.0056  -0.0023   0.0033
 0.0117   1.0092  -0.0117  -0.0051  -0.0056  -0.0016  -0.0051
 0.0048   0.0027   0.9827  -0.0132  -0.0090   0.0069  -0.0090
 0.0141   0.0064   0.0029   0.9936  -0.0041   0.0056  -0.0120
 0.0002   0.0043  -0.0159   0.0032   0.9984  -0.0112   0.0104
-0.0005  -0.0131   0.0011   0.0014   0.0041   0.9847  -0.0085
 0.0170  -0.0042   0.0079  -0.0071  -0.0095  -0.0110   0.9983
 0.9949   0.0122  -0.0000   0.0078   0.0032  -0.0142  -0.0121
-0.0000   0.9996   0.0009   0.0062   0.0008   0.0006  -0.0030
 0.0092   0.0058   0.9962   0.0065   0.0132  -0.0041  -0.0323
 0.0015  -0.0101  -0.0148   0.9957  -0.0021  -0.0037  -0.0109
 0.0140   0.0006  -0.0004   0.0105   0.9987  -0.0136  -0.0143
 0.0103   0.0060   0.0096   0.0066  -0.0117   1.0078  -0.0101
```

Figure 4: Training data on the right and a few synthesized vectors on the right for "cyclic" data set (no network memory required).

The network was also trained with another test data set. This time, the data is a little bit different. Instead of going from left to right in a cyclic manner, this time the 1's are bounced off at the border. Such training data can be seen in figure 5 on the left. Again, once having trained for long enough, we've let the network synthesize from it's trained state. This produced the output seen in figure 5 on the right hand side. Once again, we observe that the network has learned the (granted simple) pattern rather well. So why have we tested the network for both data sets? Please have a look at the discussion in section 4 for an explanation.



```
1,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0
0,0,1,0,0,0,0,0
0,0,0,1,0,0,0,0
0,0,0,0,1,0,0,0
0,0,0,0,0,1,0,0
0,0,0,0,0,0,1,0
0,0,0,0,0,0,0,1
0,0,0,0,0,0,1,0
0,0,0,0,0,1,0,0
0,0,0,0,1,0,0,0
0,0,0,1,0,0,0,0
0,0,1,0,0,0,0,0
0,1,0,0,0,0,0,0
1,0,0,0,0,0,0,0
1,0,0,0,0,0,0,0
0,1,0,0,0,0,0,0
0,0,1,0,0,0,0,0
0,0,0,1,0,0,0,0
```

```
 1.0003  -0.0051   0.0032  -0.0094  -0.0006  -0.0123   0.0084  -0.0020
 0.0085   0.9881   0.0141   0.0016   0.0045  -0.0027  -0.0041  -0.0220
 0.0040   0.0065   1.0040  -0.0027  -0.0036  -0.0090   0.0191  -0.0077
-0.0070  -0.0035   0.0093   0.9959  -0.0102  -0.0029  -0.0039  -0.0139
-0.0163   0.0005  -0.0161  -0.0071   0.9693  -0.0046   0.0041  -0.0039
 0.0146  -0.0079   0.0066   0.0006   0.0063   0.9959  -0.0114   0.0053
 0.0205  -0.0155   0.0214  -0.0185  -0.0029  -0.0050   0.9938   0.0152
 0.0012   0.0017   0.0054  -0.0040  -0.0020   0.0123  -0.0117   1.0180
-0.0099  -0.0006  -0.0154  -0.0054   0.0041   0.0061   0.0039   0.9988
 0.0120   0.0120  -0.0020  -0.0091  -0.0142   0.0006   1.0130  -0.0032
-0.0059   0.0080  -0.0050   0.0065  -0.0073   0.9853  -0.0059   0.0082
-0.0047   0.0105   0.0038  -0.0073   1.0115  -0.0163   0.0044   0.0049
 0.0089  -0.0075   0.0041   1.0054   0.0060  -0.0196  -0.0050   0.0077
-0.0139  -0.0094   1.0041   0.0098  -0.0128   0.0261   0.0010   0.0078
-0.0196   0.9873  -0.0036  -0.0016  -0.0220   0.0097   0.0120  -0.0148
 1.0042   0.0050  -0.0060   0.0028  -0.0057   0.0026   0.0012   0.0054
 1.0040   0.0279  -0.0059   0.0064   0.0021  -0.0097  -0.0104  -0.0009
 0.0010   1.0073   0.0085  -0.0008   0.0094  -0.0115  -0.0086  -0.0076
 0.0050  -0.0077   0.9815   0.0054   0.0009   0.0055  -0.0017  -0.0069
```

Figure 5: Training data on the right and a few synthesized vectors on the right for "bounce" data set. Proofs that network has learned a meaningful memory.

4.2 Results from training on real music

First, the network without mixture model was trained on a large corpus of Bach music (about 3GB in size). Results can be listened on the blog post of this work unter `https://medium.com/@LucaNaterop/6ac96021253d`. The network was also trained with a single passage from a bach piece where it was overfitted on purpose. This was done to see if the network can in principle learn complex melodies and long-term dependencies. Generated audio from this process can also be listened to on the github repo. In figure 6 a few spectrograms are shown.

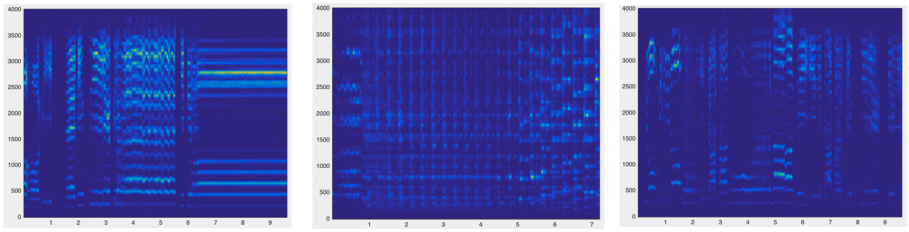


Figure 6: A few spectrograms of generated audio. Steady state (left), sample from network trained with Bach without steady state (middle), and sample from network trained with singer (right)

We observe that in the first spectrogram the music is rather dynamic in the beginning, but at a certain point in time it suddenly becomes steady state.
The generated samples here indicate that the network is far less likely to end up in steady state when using a Mixture Density Network. In some cases, the network can find a way out of the steady state after some time. In other cases, the problem of steady-state has vanished completely over a long period of time, as can be heard when listening to the audio files.

5 Discussion

5.1 Meaning of the tests

Our network was able to learn the cyclic pattern shown in figure 4 rather easily. It is important to realize that this data does not require memory to learn. Why? Because from line i of the data it is always possible to predict line $i + 1$ once the pattern has been learned. And the pattern is rather trivial, we just shift the place where the 1 appears one place to the right with each successive line, and when we've reached the end, then we start over all again. The next vector (or line) at time $t + 1$ is a pure function of the previous vector at time t and it does not depend on any other vectors at $t - 2, t - 3, ..$ and so on. Therefore the network need not know any previous vectors. Thus, this pattern could be learned by a simple feed-forward neural network where we feed it the current vector and it predicts the next one.

This is not the case with the "bounce" data set from figure 5. We know there is a simple pattern, but there is a particular place in the data where it is not enough to know the current vector if one would like to predict the next one. Take a look at lines 8-10 from the input data in figure 5. For example, we can see that the vector on line 8 is

$$(0, 0, 0, 0, 0, 0, 0, 1)$$

and it has the following vector (line 9).

$$(0, 0, 0, 0, 0, 0, 0, 1)$$

But in the next step, we take this one as an input and the output should be (line 10)

$$(0, 0, 0, 0, 0, 0, 1, 0)$$

It is therefore not always possible to predict the next vector only based on the current one. A predictor must know what happened two timesteps ago in order to predict the next one. In other words, the vector at time $t + 1$ is not a pure function of the vector at time t but it depends also on previous timesteps. Thus, a simple feed-forward neural network cannot learn such data. Instead, a network with memory is required. Since our implementation of the network was able to learn the data very well, it means that it does in fact have memory. This is why we tested the network on the additional "bounce" dataset.

5.2 The problem of steady-state

A phenomenon that occurs often with RNNs applied to regression problems is that the network arrives in a steady state where it's predictions converge to a certain vector. Once convergence has been established the output vector is the same as the input vector, only possibly scaled by an arbitrary factor. When the neural network is seen as a mapping which maps the input vector to an output vector, then this vector (that is not changed by the mapping but only stretched) is an eigenvector of the mapping. Therefore by re-applying the output again and again it can converge to it's eigenvector. For linear mappings, this

is a known method called the power method, and the method always converges to the eigenvector with the largest corresponding eigenvalue [14]. The same method can also be used for non-linear mappings, but here it is to day not at all clear when it converges and when it doesn't, and finding eigenvectors of large nonlinear systems is a delicate task. As a reminder, we would like to point out that if somebody were to find a theorem that says when the nonlinear power method converges and when not, then - in light of what has been said here - such a result can immediately be applied to the convergence behavior of neural networks. Our results suggest that the problem of steady-state predictions can be handled very effectively using Mixture Density Networks. Why that is we do not know. It might be attributed to the multi-modal nature of the output distribution. On the other hand, it might be a consequence of the fact that the mixture density network does not represent a pure regression, but it contains a classification - depending on the method of sampling. It could also be a combination of the two. But whatever the reason, it seems that mixture density networks help fighting the problems with steady state, and can therefore be recommended to use in similar tasks.

5.3 Representation of music

In this work, we've represented music on a very low level. By music we mean here a signal from which we take short time windows and compute their short-time Fourier Transforms. Our representation of music is thus a series of spectrograms. Any AI composer which is based on this representation of music must try to clip together spectrograms and form a piece of music in this manner. This is a low-level representation of music, and that has major advantages. First of all, we cannot only represent music in this manner, but any signal. This is very general. If successful, such an AI could be trained on other audio which is not music. For instance, one could train the AI on laughter of large groups of humans. Such laughter is used often in television series. Clearly it would be cheaper to have an AI create laughter for a new series than bringing dozens of people into a room and recording their laughter. Applications are not even restricted to audio, because we can represent any signal in terms of it's Short-Time Fourier Transforms.
But there is a big disadvantage of such a low-level representation. It is very expensive. If we think about it, clipping together spectrograms in order to compose music is a task that is more difficult than that of a human composer. A human composer works on a higher level representation. A human composer writes (mostly) a series of notes on a musical score. In this representation, every single note represents many different spectrograms. If the task is not the general one of modelling signals, but the more specific task of music composition, then we could in principle have a network try to synthesize musical score. But a better representation would be MIDI. In this representation - similarly to musical scores - there is simply information on when to play which note with what volume and so on. Because MIDI would be far cheaper than spectrograms, one would probably be able to generate better music if working with the same computing power and/or data set. One would simply synthesize music, and then use virtual instruments to play them along. Such an AI composer would not need to learn how to produce the sound of an instrument, it could focus on learning harmonic and rhythmic patterns.

MIDI as a high-level representation of music would thus be advantageous if the task at hand is to compose music. But it does not come free: One loses the generality of low-level representations such as spectrograms.

6 Outlook [LN & SG]

6.1 Proposals for further improvements

The ground goal of our work was to find a possible method to learn implicitly the characteristic of sound and to reproduce sound of the same style. We spent about 90% of our time for theory, literature research, analysis and implementation work until we finally came up with the presented approach. Unfortunately there was not much time left to perfect the network to finally find real awesome artificial music. All we did is showing that it is possible to produce acceptable music and that there must be a huge potential in our regression model. Now it is up to you, if you like our makeup, to further improve it. Here are some proposals:

1. **Invariances of error function:** Most human have a so called relative pitch. As an example they can't distinguish if a certain song is played at its original pitch or if it was modulated by a whole step. In contrast our network has an absolute pitch, i.e. it learns the tonal relations for certain (absolute) frequencies only. If we want to teach our network, that it is also beautiful to modulate some cadences to other pitches, one method would be to extend the training set by pitched versions of itself.

2. **Optimization of the networks hyper parameters:** Since the parallelized training of a rather small network is around one week (on 30 CPU cores), we didn't have the time to look systematically for a good set of hyper parameters. We are convinced that much better results are possible by choosing a better set of hyper parameters only.

3. **Much more training data:** We trained our networks on rather small training data corresponding to around four hour of music. The first rule of machine learning is that more training data never hurts and almost always improves the result. In our case the improvement is obvious and for most musical genres hundreds of hours of music are available.

4. **Avoid overfitting:** If the degree of freedom of the network is large enough compared to the information in the training data it is a fact, that such a network is able to learn the music almost perfectly "by heart". We have tried that by training on 5 minutes of Bach music. Of course simple reproduction is not our goal, since we want to have the effect of creating something new. Since there is access to a huge amount of training data, it is not expensive to divide the data in a training and a validation set. By evaluating the error of the network on the unseen validation set at different training states, we can identify the critical point where the validation error starts to grow again. As a further approach one could add a regularization term to the error function, such as $+\frac{\lambda}{2}\|\theta\|^2$. The corresponding hyper parameter λ controls the parameters of the model such that they are more likely to stay small. This may be statistically interpreted as a Bayesian prior belief that the "true" parameters are likely to be small.
   In this sense we may make a statement what creativity could possibly be, namely ideas of a non-overfitted but well trained brain. If you want to be able to reproduce ideas of other people perfectly, in example perfect a

piece of classic piano music, try to train your brain or muscles on this specific task as much as possible. But if you want to come up with something new, try to capture the main concepts and relations of as many other good ideas as you can but in a rather vague way.

6.2 The future of creative AI

Although results generated in this work are not particularly overwhelming, we are still thrilled about the future of creative AI. As mentioned before, we speculate that Artificial Intelligence can in principle compose music as well as humans can do it. We have also proposed a method of testing such an AI. Let's for a moment think about an AI taking such a test. And let's imagine that the result would be such that humans would - in case we take, for instance, the J.S. Bach version of the test - classify *more* AI compositions as Bach music than actual composition from J.S. Bach himself. What would that mean? In our view, such a result would have very deep implications on the nature of Bach's compositions. It would mean that Bach has not created the ultimate music in the particular kind of space that he explored. Instead, he created something that, just maybe, is only the tip of the iceberg of what the ultimate Bach music would be. It would mean that there is plenty of room for even better compositions in the space of possibilities that Bach himself explored, but was not able to find. We speculate that an AI composer can maybe explore further this space, and find compositions that remind us of Bach but are in fact far better than Bach's own compositions.

But of course, it does not need to stop there. An AI composer does not need to be limited to the spaces that we humans try to explore. We speculate that AI can explore spaces of possibilities that are beyond reach for any human, and therefore compose music of a totally unseen style. Such music might sound unlike anything we've ever heard. It might make us break out in tears because it could be so abnormally beautiful; and all that in a musical style that we haven't heard before. And then, their music might also sound completely alien to us, and we wouldn't be able to relate to their music at all. But in the end we must ask ourselves: Who are we to judge them?

We shall see what the future holds for us and them.

References

[1] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. *A neural algorithm of artistic style*. In arXiv: 1508.06576, 2015.

[2] Hsu, Feng-hsiung. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, ISBN 0-691-09065-3, 2002.

[3] Silver, David; Huang, Aja; Maddison, Chris J.; Guez, Arthur; Sifre, Laurent; Driessche, George van den; Schrittwieser, Julian; Antonoglou, Ioannis; Panneershelvam, Veda. *Mastering the game of Go with deep neural networks and tree search* Nature. 529 (7587): 484–489. doi:10.1038/nature16961, 1997.

[4] Sepp Hochreiter; Jürgen Schmidhuber. *Long short-term memory* Neural Computation. 9 (8): 1735–1780. doi:10.1162/neco.1997.9.8.1735. PMID 9377276, 2016.

[5] National Mission of Education through ICT. *Artificial Neural Networks Virtual Lab* http://cse22-iiith.vlabs.ac.in/exp4/index.html, 2016.

[6] Neural Computing Research Group Report: NCRG/94/0041 *Mixture Density Networks* http://www.ncrg.aston.ac.uk/, 1994.

[7] Christopher Olah. *Understanding LSTM Networks* http://colah.github.io/posts/2015-08-Understanding-LSTMs, 2015.

[8] Microsoft *Spectrogram inversion toolbox* https://www.microsoft.com/en-us/download/details.aspx?id=52505, 2014.

[9] L. Hiller, and L. Isaacson. *Musical composition with a high-speed digital computer* Machine Models of Music .9-21. Cambridge, Mass.: The MIT Press, 1993.

[10] S.M. Schwanauer, and D.A. Levitt *MachineModels ofMusic* Cambridge, Mass. The MIT Press, 1993

[11] G.M. Rader *A method for composing simple traditional music by computer.* Reprinted inSchwanauer, S.M., and Levitt D.A., ed. Machine Models of Music.243-260. Cambridge, Mass. The MIT Press. 1993.

[12] D. Cope *Experiments in Music Intelligence.* A-R Editions 1996.

[13] D. Cope *Pattern Matching as an engine for the computer simulation of musical style.* In Proceedings of the 1990 International Computer Music Conference. San Francisco, Calif.: International Computer Music Association, 1990.

[14] Richard von Mises *Praktische Verfahren der Gleichungsauflösung.* ZAMM - Zeitschrift für Angewandte Mathematik und Mechanik 9, 152-164 (1929).

[15] M. L. Johnson *An expert system for the articulation of Bach fugue melodies. In Readings in Computer Generated Music.* ed. D. L. Baggi, 41-51. Los-Alamitos, Calif.: IEEEPress. 1992.

[16] C. Fry *Flavors Band: A language for specifying musical style (1984).* Reprinted in Schwanauer, S.M., and Levitt, D.A., ed.1993. Machine Models of Music. 427-451. Cambridge, Mass. The MIT Press.