MNG Rämibühl
Mathematisch-Naturwissenschaftliches Gymnasium

Classical Music Generation with Neural Networks

Matura Project by

Author
Curriculum Informatik
Supervisor
Date 03.01.2022

Abstract

Historically music was only created intentionally by humans, but since computers were invented a variety of methods have been proposed to generate music algorithmically. The challenge of designing an algorithm capable of competing with human musicians is yet to be overcome, but we are closer than ever to finding a solution. This paper will cover two neural networks and try them at this challenge. The first architecture that will be investigated is a variant of the simple feed forward neural network called an autoencoder(AE). The autoencoders structure gives access to the defining dimensions in which a given dataset lies. The second architecture is the long short term memory network (LSTM) which is a more advanced form of the classical recurrent neural network (RNN). Its ability to learn long time dependencies makes it suitable for this task. An overview of the theory and math needed to understand these two algorithms is provided as well. It was found that the LSTM overshadows the autoencoder but both are indeed capable of generating convincing pieces.

Contents

1 Introduction

There is nothing new about the idea of composing music using algorithms. Neural networks back in the day weren't very effective however, since computation wasn't as readily available as it is today. Now even an individual can train a neural network on their computer at home to generate something recognizable as music. Leading this new wave of deep learning algorithms are companies that produce open-source libraries for easy computation of neural networks such as Tensorflow or OpenNN. These technological advances have led to a lot of projects and research being done. "Deep Learning Techniques for Music Generation" (Briot, Hadjeres & Pachet 2017) is a survey done in 2017 on how neural networks have been applied to the task of music generation. The LSTM and autoencoder network make their début in a couple of papers. The video "Computer evolves to generate baroque music!" made by Cary Huang on youtube showcases how LSTMs can be used to learn baroque music and generate tunes on its own. Magentas MusicVAE (Roberts, Engel, Raffel, Hawthorne & Eck 2018) is an example for an autoencoder used to generate melodies by interpolating between the latent vectors of two melodies.

1.1 Motivation

The motivation behind this paper was to find a way to apply neural networks to an artistic process, somehow bringing mathematics and art closer together. Art and mathematics don't often interact with each other especially in processes such as music composition where it is usually not thought of to apply formulas to an emotional piece. As some people say, art is a form of expressing emotions and mathematics is pure logic, which might be the reason why the two subjects are separated. But by getting neural networks to learn and generate musical pieces, this might be a step towards combining mathematics and artistic processes, thus exhibiting the beauty of mathematics. Although it is not possible to know exactly what goes on inside a neural network, exploring their capabilities is still quite fascinating.

1.2 Goal

The goal therefore is, by utilization of various structures and architectures, to create a neural network that can produce it's own music. This project will focus on a model to produce baroque/classical music, since this type of music has properties that help with learning. A similar approach could probably be applied to any type of music.

1.3 Concept Declaration

1.3.1 Music

Music is part of the creative arts. To create music, notes of various lengths and pitches are ordered in succession. Music can either be polyphonic, multiple notes get played at the same time, or monodic, a single note is played at a time (Mangal, Modak & Joshi 2019). One wouldn't consider every random succession of notes to be music. There are many aspects that make up music such as harmony, rhythm and melody to name a few. Each genre has it's own qualities, which makes it more or less suitable for certain machine learning techniques. In this paper only classical/baroque piano music was used for simplicity. More details on the choice of music are in section 3.2.1.

1.3.2 Artificial Intelligence

The term artificial intelligence dates all the way back to 1955 where it was coined by Professor John McCarthy. The main goal of the field of artificial intelligence was at first to reproduce something close to human intelligence. It was theorized that maybe one could simulate human intelligence in a machine such as a computer. Since, according to Herbert Simon and Allen Newell, computers and human minds are "species of the same genus", they are namely both symbolic information processing machines (Dick 2019). The consequences of this statement would be that any task done by humans could just as well be done by a machine, tasks such as composing music.

1.3.3 Neural Networks

Artificial neural networks(ANN) loose models of the human brain. They consist of many artificial neurons that are connected via weights, just like neurons are connected via synapses in our brain. Input signals are propagated from one layer of neurons to the next. If not stated otherwise, the term "neural network" will refer to an artificial neural network as opposed to a biological neural network. A more detailed explanation of neural networks is presented in section 2.1.

2 Theory

2.1 Neural Network

The predecessor to neural networks is the perceptron. The perceptron was first proposed in 1957 by Frank Rosenblatt as an algorithm for pattern recognition. This very simple model was criticized by Marvin Minsky and Seymour Papert due to it's inability to classify nonlinearly separable domains. To solve this problem the idea of adding hidden layers and non linear activation functions was introduced. This led to the reappearance of neural networks along with the invention of the backpropagation algorithm which was used to train such models. (Briot, Hadjeres & Pachet 2017)

2.1.1 Intuition behind Neural Networks

Neural networks as the name suggests are loose attempts at modelling biological neural networks such as our brain. Although it is still debated whether they are an accurate representation, neural networks have achieved a lot of success in recent years. Models are trained on large amounts of data and tasked with pattern recognition.

2.1.2 Theory

A neural network consists of layers of perceptrons connected to each other via weights. A perceptron is essentially a parameterized mathematical function which takes in a set of inputs as a vector $\boldsymbol{x} \in \mathbb{R}^n$ and outputs a single value $\hat{y}$. Each element $x_i \in \boldsymbol{x}$ is multiplied by its corresponding weight parameter $w_i \in \boldsymbol{w}$ and then the products are all added together with an extra bias b. The result of this addition is passed through a non-linear activation function $\sigma(z)$ which then produces the output $\hat{y}$. Each weight w_i is initialized to a random value. There are many different techniques for initializing weights, such as Xavier initialization used in this paper (Glorot & Bengio 2010). The final equation is,

$$\hat{y} = \sigma\left(b + \sum_{i=1}^{n} x_i \cdot w_i\right) = \sigma(b + \boldsymbol{x}^\intercal \cdot \boldsymbol{w})$$

, as a diagram:

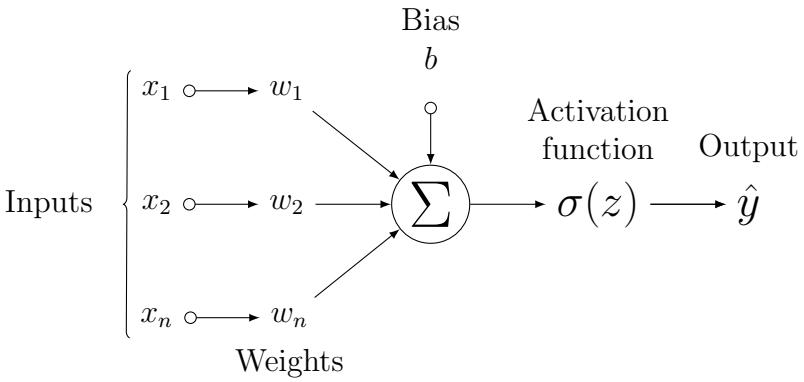


Figure 1: Perceptron Neuron

There are many activation functions to choose from, each one has its own advantages and disadvantages. Here are a couple of the most common activation functions (Sigmoid / ReLU / Hyperbolic Tangent):

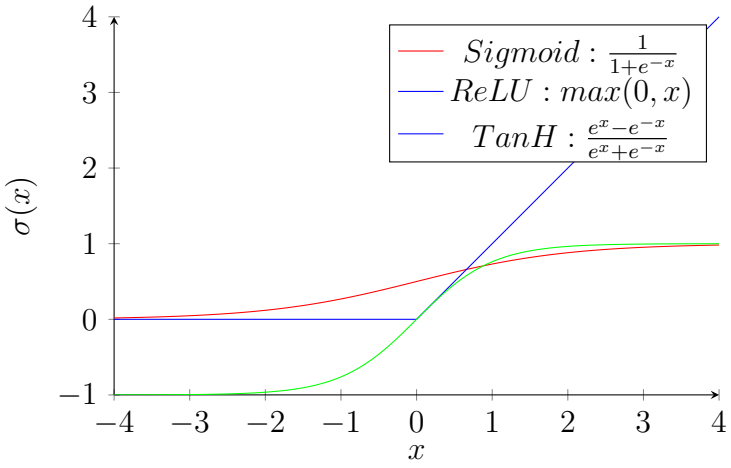


Figure 2: Activation functions

With an understanding of how perceptrons work we can now stack them to create a dense neural network. A layer in a neural network is a collection of perceptrons. These layers get stacked on top of each other. In a dense neural network each perceptron in a layer is connected to each perceptron in the previous layer so that the activations of the previous layers multiplied by their weight matrix result in the activations for the next layer of perceptrons. Perceptrons in the same layer are not connected. Since the first layer, the input layer, has no preceding layer the values of the input layer are set directly to the values of an input vector $\boldsymbol{i}$ from some dataset. The activation function of the input layer is usually a linear function. From there the vector from the data gets passed from layer to layer via weights as described above. The last layer, the output layer, outputs the final vector $\boldsymbol{o}$. Each layer between input and output layer is called a hidden layer. Figure 3 shows a 3-layer neural network with n input neurons m hidden neurons and l output neurons.

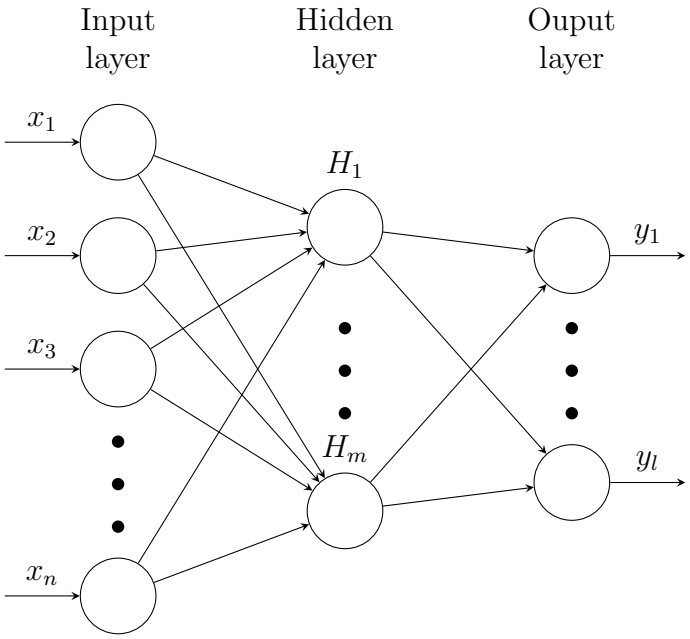


Figure 3: Fully connected Neural Network

$$f(\boldsymbol{x}) = \sigma(\alpha(\boldsymbol{x} \cdot \boldsymbol{W}_1 + \boldsymbol{b}_1) \cdot \boldsymbol{W}_2 + \boldsymbol{b}_2) = \boldsymbol{y}$$

2.1.3 Gradient Descent

After initializing a neural network there needs to be a way for the network to learn from the dataset. This is done by adjusting the parameters of the neural network. The proposed algorithm for this is gradient descent. There are many variations of this algorithm such as Adam or RMSprop which are all referred to as optimizers. To begin a loss function $L(\boldsymbol{x}, \boldsymbol{y})$ is introduced with $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^N$. $\boldsymbol{x}$ is the output vector of the neural network and $\boldsymbol{y}$ is the target vector. The goal of gradient descent is to minimize the output of $L(\boldsymbol{x}, \boldsymbol{y})$. This is done by updating the parameters in the opposite direction of the gradient of the objective function $\nabla L(\boldsymbol{x}, \boldsymbol{y})$ with respect to the parameters. The amount that the parameters are adjusted is determined by what learning rate η s et. With each adjustment a step is taken in the downhill direction of the slope until the network reaches a local minima (Ruder 2016). Figure 4 shows a visualization of the gradient descent algorithm where the x and y components are the parameters of the neural network and the z component is accumulated loss over all training samples. For network with more than 2 parameters it is impossible to draw this graph. A common loss function for example is mean squared error(MSE) which was used for training the models in this paper.

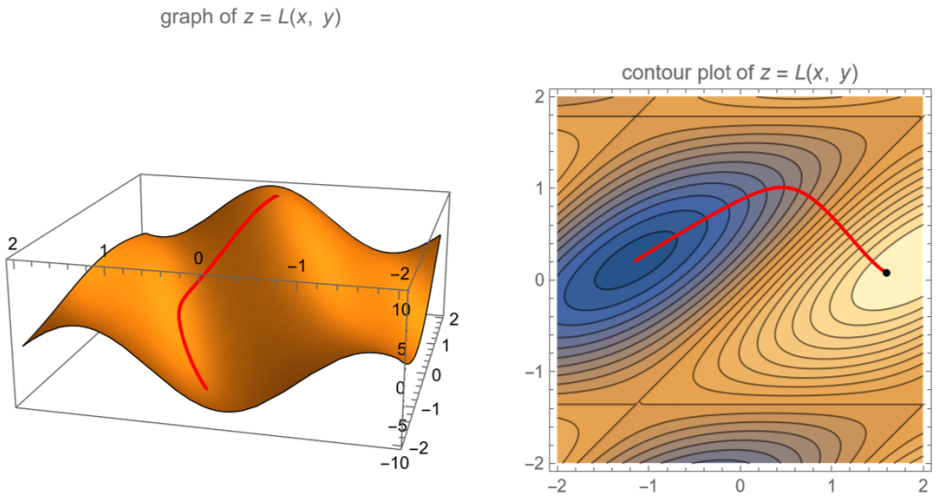


Figure 4: Visualization of Gradient Descent Algorithm

$$L(\boldsymbol{y}, f(\boldsymbol{x})) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f(\boldsymbol{x})_i)^2$$

$$[0, 1, 0, 0, 1...0, 1, 1]$$

$$[0, 0, 1, 1, 0...0, 0, 1]$$

$$[1, 0, 0, 1, 0...1, 0, 0]$$

2.2 Autoencoder

An autoencoder is an unsupervised learning technique for data compression and feature extraction implemented with neural networks.

2.2.1 Dimensionality Reduction

In statistical pattern recognition one often has to deal with high dimensional data, which can lead to the curse of dimensionality. Data high in dimensionality comes with some difficulties: Analysing it is hard, interpreting is difficult, visualization of any data with more than 3 dimensions is hard and storing the data vectors can be computationally expensive, slowing down the entire process. For these reasons there is incentive to find an accurate representation of a dataset in lower dimensions. Dimensionality reduction techniques exploit certain properties in the dataset to achieve this goal. For example, many dimensions are often redundant and can be expressed as combinations of other dimensions. Furthermore, dimensions are often correlated so that the data posses an intrinsic lower-dimensional structure that can be exploited (Deisenroth, Faisal & Ong 2020). Traditionally mainly linear dimensionality reduction techniques were used such as principal component analysis. These techniques are very useful, yet they fail to process complex non-linear data. In recent years though many non-linear techniques have been proposed. Here is a list of examples for both linear and non-linear techniques: Factor analysis, Kernel PCA, autoencoders, isomaps, maximum variance unfolding etc. (Van Der Maaten, Postma, Van den Herik et al. 2009)

2.2.2 Intuition behind Autoencoders

Autoencoders are algorithms for dimensionality reduction. The idea behind autoencoders is that a set of high dimensional data can often be projected onto a lower dimensional space with minimal loss of information. The autoencoder algorithm is implemented using neural networks and aims to learn a compressed representation for an input through minimizing its reconstruction error (Wang, Huang, Wang & Wang 2014). An autoencoder can be divided into two networks, the encoder and the decoder network. These networks do not have to be symmetric. Figure 5 shows an example for how an autoencoder could look like.

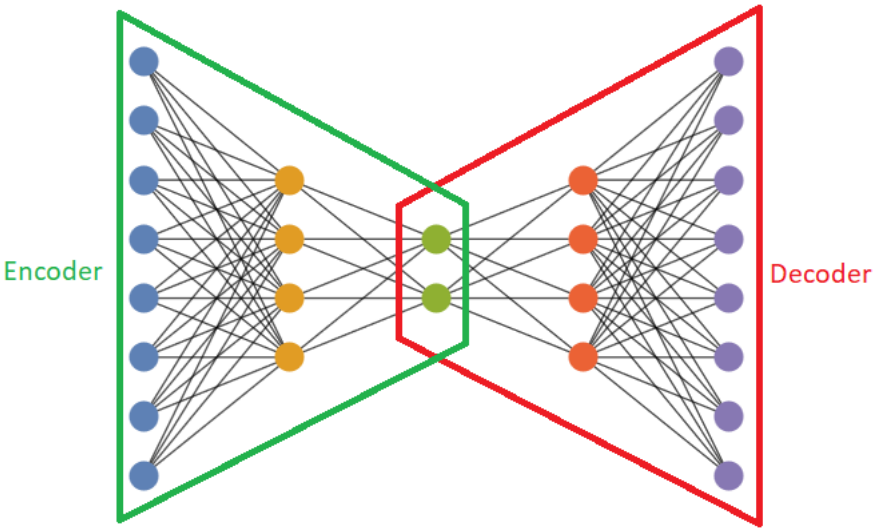


Figure 5: Autencoder Network

The encoder network projects high dimensional input data onto a lower dimensional latent space and can be written as a function $g(\boldsymbol{x})$ parametrized by a weight matrix $\boldsymbol{\psi}$. The decoder network reconstructs the original data from the output of the encoder and can also be written as a function $f(\boldsymbol{z})$ parametrized by a weight matrix $\boldsymbol{\theta}$. The lower dimensional representation for input $\boldsymbol{x}$ in the bottleneck layer is $\boldsymbol{z} = g_\psi(\boldsymbol{x})$. The reconstructed input is $\boldsymbol{x}' = f_\theta(g_\psi(\boldsymbol{x}))$. The reconstruction error for a dataset with n samples is (Weng 2018)

$$e_i(\boldsymbol{\psi}, \boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} (\boldsymbol{x}_i - f_\theta(g_\psi(\boldsymbol{x}_i)))$$

2.2.3 Principal Component Analysis

Looking at other dimensionality reduction algorithms can help gain further insight into the intricacies of autoencoders. Specifically principal component analysis(PCA) bears significant resemblance to autoencoders. (Abdi & Williams 2010). This section gives a very brief description of PCA and it's similarities to autoencoders.

Given a data matrix $\boldsymbol{X} \in \mathbb{R}^{m \times n}$, where each row represents a different observation and each column represents a variable describing that observation, PCA looks to find a new orthogonal basis that is a linear combination of the original basis in which the data was captured. The goal being to express the data $\boldsymbol{X}$ in a different basis. The axes of the new basis are ordered by variability of the data along each axes. This already imposes a restriction, PCA can only re-express the data as a linear combination of its basis vectors (Shlens 2014). Both PCA and autoencoders try to find the intrinsic dimensions of a data set by manipulating the data

through space. Despite their similarities there are still two major differences. Firstly PCA is a linear transformation while autoencoders, when using a non-linear activation function, are not. Secondly the axes retrieved from PCA are ordered according to their representational power, whereas in autoencoders this is not usually the case. This demonstrates some of the disadvantages with autoencoders. While autoencoders can find very flexible and powerful latent spaces it is difficult and sometimes impossible to interpret their meaning. Finding the right latent space dimension is not trivial, the autoencoder will use all the space it can get and not leave any dimension empty. (Ladjal, Newson & Pham 2019)

2.3 Recurrent Neural Networks

Recurrent neural networks (RNN) are sequential modelling and processing algorithms that have had a lot of success in areas such as language modelling, speech recognition, image captioning etc. RNNs are feedforward neural networks, but they have the extra capability to store information about past events using loops in the network. This allow them to learn series of items, such as a melody sequence of nots. RNNs can be expressed as a weighted function $\hat{\boldsymbol{y}}_t = f_{\boldsymbol{W}}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$ parametrized by a weight matrix $\boldsymbol{W}$ where $\boldsymbol{x}_t$ represents the input vector, $\hat{\boldsymbol{y}}_t$ the output vector and $\boldsymbol{h}_{t-1}$ the hidden state at time t. Figure 6 shows two ways of representing an RNN.

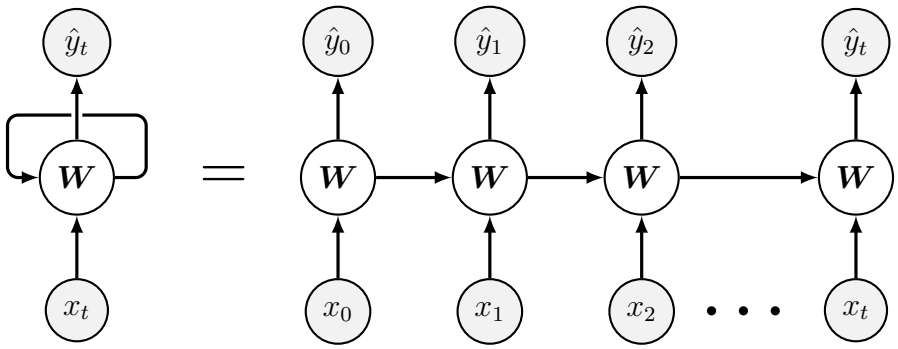


Figure 6: Recurrent Neural Network

Standard RNNs suffer from two very big problems, namely the exploding gradient and the vanishing gradient. With backpropagation through time (BPTT) error signals flowing backwards in time can either blow up or vanish. The evolution of the backpropagated error depends exponentially on the size of the weights. If the gradients explode the weights can oscillate between extremes whereas if they become exponentially small progress is practically halted. To solve this problem long short term memory cells (LSTM) were introduced. (Hochreiter & Schmidhuber 1997)

2.3.1 Intuition behind LSTMs

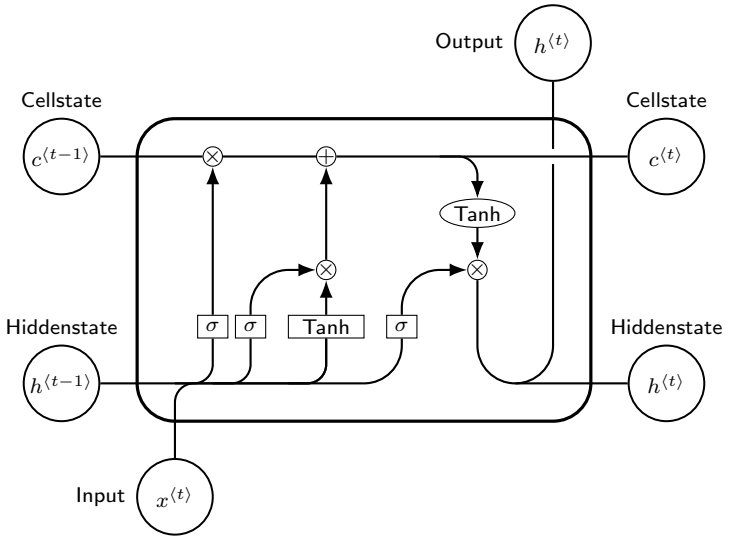


Figure 7: Long Short Term Memory Cell

The horizontal line running through the top of Figure 7 is the cellstate. Information can flow mostly unchanged through the entire chain. To put it into practical terms, the cellstate is the long term memory of our model. The LSTM does have the ability to add or remove information from the cellstate using the forget and update gates. This can be interpreted as such: The sigmoid function outputs a number between 0 and 1. This describes how much of each component should be let through. 0 meaning nothing at all and 1 meaning everything. There are three of these gates in an LSTM cell. The mathematical implementation is described below.

2.4 LSTM

There are many variations of the LSTM model that each have their own perks and disabilities. In this paper the "vanilla" LSTM is used without the peep-holes. Shown here is a step-by-step description of a single forward pass through an LSTM layer of size N which takes in an input vector $x_t \in \mathbb{R}^M$. These are our following weights:

Input weights: $\boldsymbol{W}_f, \boldsymbol{W}_i, \boldsymbol{W}_C, \boldsymbol{W}_o \in \mathbb{R}^{N \times M}$
Recurrent weights: $\boldsymbol{R}_f, \boldsymbol{R}_i, \boldsymbol{R}_C, \boldsymbol{R}_o \in \mathbb{R}^{N \times M}$
Bias weights: $\boldsymbol{b}_f, \boldsymbol{b}_i, \boldsymbol{b}_C, \boldsymbol{b}_o \in \mathbb{R}^N$

- Forget Gate

  This gate gives the network the ability to forget things. It takes into consideration the input $\boldsymbol{x}_t$ and hidden state $\boldsymbol{h}_{t-1}$ and outputs a number between 0 and 1 for each element in the cell state C_{t-1}, due to the sigmoid function. 0 meaning forget this variable and 1 meaning keep this variable.

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f \cdot \boldsymbol{x}_t + \boldsymbol{R}_f \cdot \boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$$

- Update Gate

  The update gate decides which information gets added to the cell state.

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \cdot \boldsymbol{x}_t + \boldsymbol{R}_i \cdot \boldsymbol{h}_{t-1} + \boldsymbol{b}_c)$$

$$\widetilde{\boldsymbol{C}}_t = \tanh(\boldsymbol{W}_C \cdot \boldsymbol{x}_t + \boldsymbol{R}_C \cdot \boldsymbol{h}_{t-1} + \boldsymbol{b}_C)$$

- Update cell state

  In this step the cell state is updated by first multiplying by the forget vector $\boldsymbol{f}_t$ and then adding the update vector $\boldsymbol{i}_t$.

$$\boldsymbol{C}_t = \boldsymbol{f}_t \odot \boldsymbol{C}_{t-1} + \boldsymbol{i}_t \odot \widetilde{\boldsymbol{C}}_t$$

- Output Gate

  As a final step the hidden state $\boldsymbol{h}_{t-1}$ is calculated. It has to be decided which parts of the cell state get outputted. Finally the cell state gets pushed through a *tanh* function to scale all values to between 1 and -1.

$$\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \cdot \boldsymbol{x}_t + \boldsymbol{R}_o \cdot \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{C}_t)$$

(Greff, Srivastava, Koutník, Steunebrink & Schmidhuber 2016) (Olah 2015)

3 Practical Part

3.1 Software

Everything in this project was done using python, the above QR code is linked to the github repository with all the code, which is also accessible under https://github.com/DireWolf117/KoeTF. After first attempts with C++ and the C++ neural network library OpenNN failed, due to the development stage of OpenNN, all the code had to be re-written with python, but the code is still there under https://github.com/DireWolf117/Koe. For development the following libraries were used:

- Tensorflow keras for training and handling neural networks along with tensorboard for visualization
- Matplotlib and numpy for vector operations and graphing
- Libfmp for handling MIDI files and visualization
- Tkinter for user interface
- Selenium for scraping MIDI files from the internet

3.2 Dataset

There are some variables that have to calibrated to ensure an optimal dataset. Firstly the size of the dataset has to be determined, too big and training time becomes very long, too small and overfitting will probably occur. In the end the dataset for this project was put together out of 294'656 measures of piano pieces, of which 18'688 were used for testing. The data also has to be correlated otherwise the network won't be able to extract any patterns which is also a reason why only classical piano pieces were chosen.

To collect songs various MIDI files were downloaded from online-sites. More about how the MIDI files were formatted in section 3.2.2. This approach led to some difficulties however. Having to download over 2000 MIDI files manually is very time consuming. To solve this problem a custom python script was written to scrape any given page for .mid files. Next certain song segments had to be picked out manually for training, because not all songs could be used for reasons explained in section 3.2.2.

3.2.1 Choice of Music

For this project only classical musical pieces were chosen to train all models. In comparison to modern music(Pop), which generally has a very clear melodic and chord structure, classical music is more free flowing with few restrictions concerning what notes have to be played next. This allows the network more freedom and makes it's output less prone to perceived error. Although clear patterns, as found in popular music, are easier for a network to classify generating them would be more difficult.

3.2.2 Formatting

To train the models on music there needs to be a way to convert songs into purely numerical vectors that can be fed into a neural network. In the end it was decided to represent songs as many-hot-vectors in pianoroll format from MIDI files. The downloaded MIDI files were converted to .txt files with the program midicsv.exe and then processed with a script CSVtoTXT.cpp. The format of the resulting file is as follows:

- Each of the pianos 88 keys are converted to their respective ascii character, from !(33) to x(120). If a note is played during a beat, then the respective ascii character gets printed. For example: Middle C is the 40th key, which corresponds to the ascii representation "H(72)". A middle C held for 4 beats looks like: H H H H.
- Notes that are printed simultaneously during a beat are printed next to each other. For Example: If the notes C, E and G are played together for 4 beats it looks like: HJL HJL HJL HJL
- Beats are separated by a " " character. Each beat that does not play a note does not print anything, therefore multiple spaces are printed each after another.

Using this kind of approach has a couple of disadvantages. Tempo change and note volume cannot be represented in this format. This was very noticeable when training models on certain datasets. Although the loss value was very low, the generated songs did not sound anything like the original pieces, this counts especially for the autoencoders. Upon further analysis it was apparent that these models were trained on datasets with a lot of tempo change in the original songs. Consequently these songs had to be filtered out.

3.2.3 Representation

As mentioned above, neural networks require numerical vectors as inputs. From the files outputted by CSVtoTXT.cpp fixed size portions of the song were extracted as vectors for the autoencoders or as matrices for the LSTM. A single beat is represented as a vector v of size 88 where each component $v_1 \rightarrow v_{88}$ maps to a key on the piano. The mapping occurs as follows: If a note is played then the corresponding element in v is set to 1 otherwise 0(many-hot-encoding). The vectors of consecutive beats are concatenated to form a full segment of a song. The main advantage of this is that polyphonic melodies can be represented without sacrificing any notes though this comes at a very high cost to resources. In the future one might consider looking for a better method to encode these vectors. Autoencoders could be suitable for this task.

3.3 Experiment

The goal of this paper is to train a neural network model to generate classical piano music. The criteria for success is whether or not the model can produce an original piece that shows the same properties as the dataset. This will be attempted by using two different neural network architectures: Autoencoders and LSTMs.

3.4 Autoencoder

The first strategy to generate new music is by using an autoencoder. Autoencoders can extract features that characterize a dataset, in this case those features would be the characteristic properties of music. To generate music these features can be used as an input interface to parametrize the generation process, which has a similar intuition to reversing the PCA algorithm. If one were to apply the PCA algorithm to a dataset, one could choose a random point in the principal component space and then translate this point back into the original space by reversing the PCA process. This point would then then lie in the common distribution of the dataset. For autoencoders the process would look like the following:

- A vector $s \in \mathbb{R}^n$ is chosen as a seed where n is the dimension of the latent space.
- s is inserted into the bottleneck layer.
- And finally s is fed through the decoder.

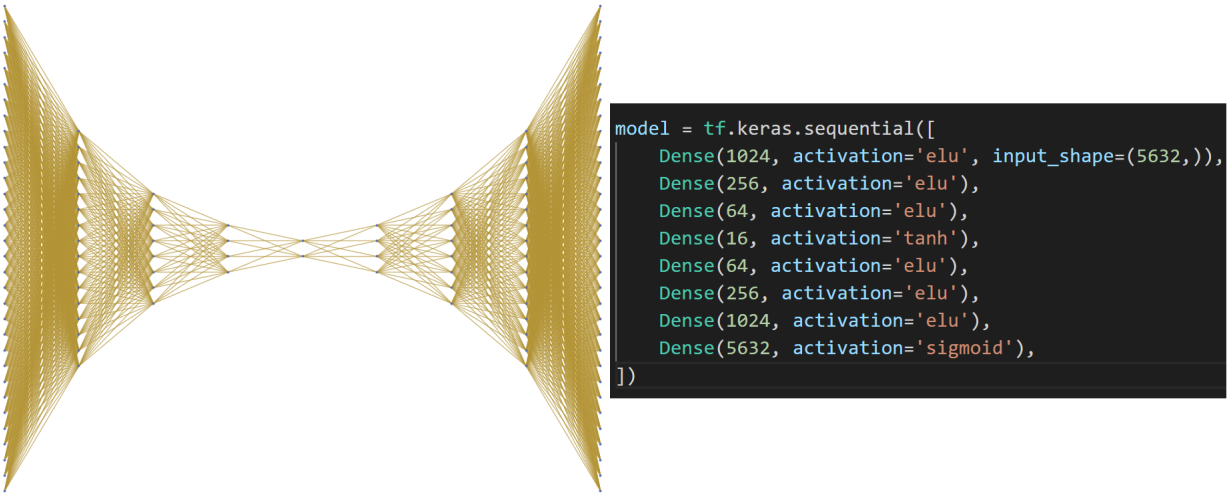
If trained correctly each song in the dataset should have its own unique feature vector. Inputting this feature vector into the decoder network would then in theory output the corresponding song segment. This makes it possible to interpolate between songs by interpolating between their feature vectors but it is also possible to generate new pieces by randomly seeding the latent space.

3.4.1 Design Choices

As already mentioned in section 2.2, it is a bit of a hit-or-miss situation when designing an autoencoder. If the latent space is too big time is wasted on training an unnecessarily big model and the model won't be as efficient as it could be. On the other hand if the latent space is too small the model will underfit. Many models were tested to reach the models described below, the biggest problem being training time. AE1 described below has $6'055'056$ adjustable parameters and some models had even more. Training such a model can take many days and early design mistakes will nullify the results. With this in mind finding the right model is a time consuming task, which can only be achieved by trial and error.

The first models were too small. These models weren't capable of generating anything meaningful. Every model always produced the note C4, which by chance was the most common note in the dataset. After these failed attempts to create my own model I drew inspiration from already existing models.

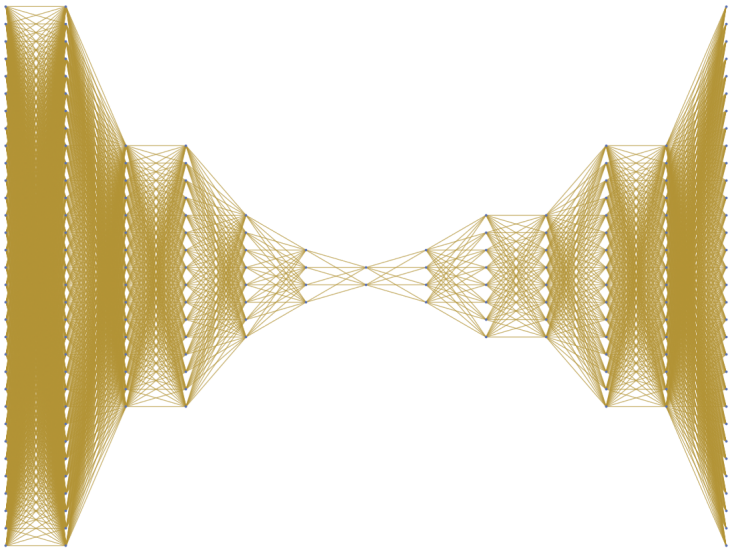
The first working model is a 4-layer stacked autoencoder which is referred to as AE1. The data comes in a multi-one-hot encoding with each 16th of a measure represented as a vector $v \in \mathbb{R}^{88}$. This model has 5632 input and output neurons, allowing for song segments of 4 measures in size ($4 \cdot 16 \cdot 88 = 5632$). The number of hidden neurons decreases down to the bottleneck layer with 16 neurons which allows for 16 degrees of freedom to generate pieces. The activation functions are shown in the code snippet in figure 8. TanH in the bottleneck layer keeps the activations between -1 and 1 making sure the values don't explode and sigmoid in the last layer let's the result be interpreted as probabilities. The number of neurons per layer was derived from the DeepHear music generation system by Sun (Sun 2017). Figure 8 is a vague diagram of how the model would look like, the number of neurons per layer is shown in the code snippet.



```
model = tf.keras.sequential([
    Dense(1024, activation='elu', input_shape=(5632,)),
    Dense(256, activation='elu'),
    Dense(64, activation='elu'),
    Dense(16, activation='tanh'),
    Dense(64, activation='elu'),
    Dense(256, activation='elu'),
    Dense(1024, activation='elu'),
    Dense(5632, activation='sigmoid'),
])
```

Figure 8: Autoencoder 1 / AE1

The second model, which will be referred to as AE2, is very similar to the first model. It's structure is essentially the same except that 4 dropout layers were added between the layers. In Figure 9 the 2nd, 4th, 10th and 12th layers represent the dropout layers. Dropout layers were added to the structure of AE1 because AE1 was overfitting. In short dropout layers randomly "kill" neurons in each forward pass according to some probability. This forces the network to learn more robust patterns since it is trained to not rely on all neurons being active. Please refer to the paper "Dropout: a simple way to prevent neural network from overfitting" for more information (Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov 2014). The activation functions are all the same.



```
model = tf.keras.sequential([
    Dense(1024, activation='elu', input_shape=(5632,)),
    Dropout(0.3),
    Dense(256, activation='elu'),
    Dropout(0.3),
    Dense(64, activation='elu'),
    Dense(16, activation='tanh'),
    Dense(64, activation='elu'),
    Dense(256, activation='elu'),
    Dropout(0.3),
    Dense(1024, activation='elu'),
    Dropout(0.1),
    Dense(5632, activation='sigmoid'),
])
```

Figure 9: Autoencoder 2 / AE2

3.4.2 Training Process

In this section the training process for AE1 and AE2 is described in detail. As mentioned both models were trained using tensorflow keras in python because tensorflow has many features that help with training models.

```
model = mod.autoencoder()
adam = tf.keras.optimizers.Adam(learning_rate=0.001)
model.compile(optimizer=adam, loss=tf.keras.losses.MeanSquaredError(), metrics=['binary_accuracy'])
```

Figure 10: Settings for training autoencoders

Training was done by having the autoencoder reconstruct it's input vector with mean squared error as the loss function and Adam for optimization. Figure 11 shows the loss and accuracy graphs during training of AE1. From epoch 0 to 250 training is smooth and the loss decreases gradually down to about $9.4 \cdot 10^{-3}$ while the accuracy goes up to around 0.99. After epoch 250 the loss starts to increase again. Behaviour where the loss increases is caused by the learning rate being too high. The second half of training suffers from more loss spikes than the first, meaning that the network is overshooting it's target. Gradients are too large and the model starts to deteriorate. Changing the learning rate fixes this problem, but a loss of $9.4 \cdot 10^{-3}$ is already very low.

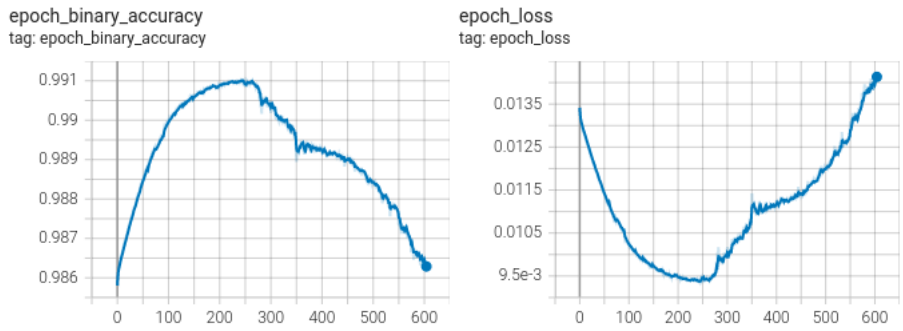


Figure 11: Loss and Accuracy vs. epochs for AE1

After each epoch AE1 was evaluated on a test set, the loss and accuracy was recorded on a graph as shown in figure 12. It is evident that loss increases over time while accuracy decreases, which is not a good sign. This means that the model is overfitting which is true for the first 250 epochs but after epoch 250 training loss and accuracy also decrease (as depicted in figure 11), which leads to an overall worse trained model.

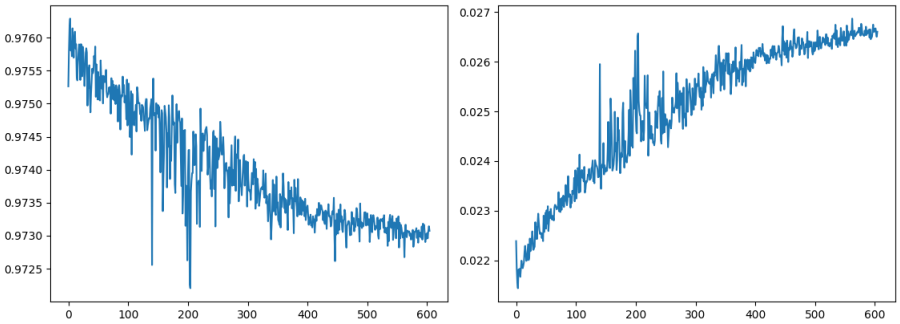


Figure 12: Test Loss and Accuracy vs. epochs for AE1

Even though it is impossible to visualize exactly what big neural networks are doing in the background, since they usually do operations in higher dimensions, one can still get a rough idea by looking at how the parameters evolve over time. In figure 13 the distributions of the models weights with respect to epochs are shown (E = Encoder, B = Bottleneck, D = Decoder), graphs were generated using tensorboard. Because all parameters were initialized with the Xavier normal distribution (Glorot & Bengio 2010), there is a spike at the beginning of each histogram with a mean of 0.

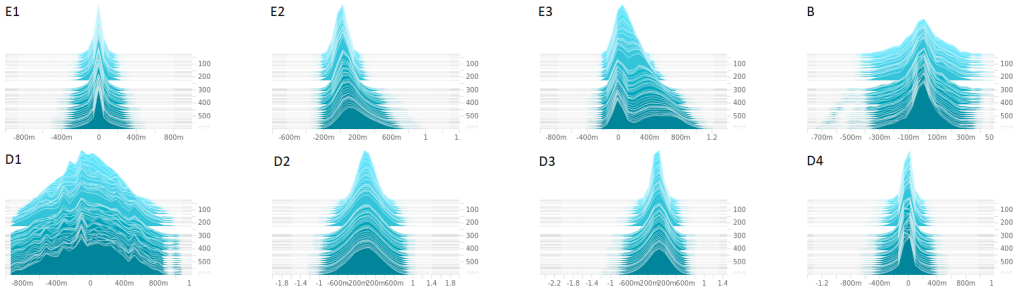


Figure 13: Weights Histogram AE1

After analysing the training of AE1 it is clear that AE1 suffers from overfitting. In an attempt to reduce overfitting, dropout layers were added to AE1 which resulted in the new model AE2. Figure 14 shows the loss and accuracy graphs during training of AE2. Training was done with the same settings as for AE1 except that the learning rate is lower than for AE1.

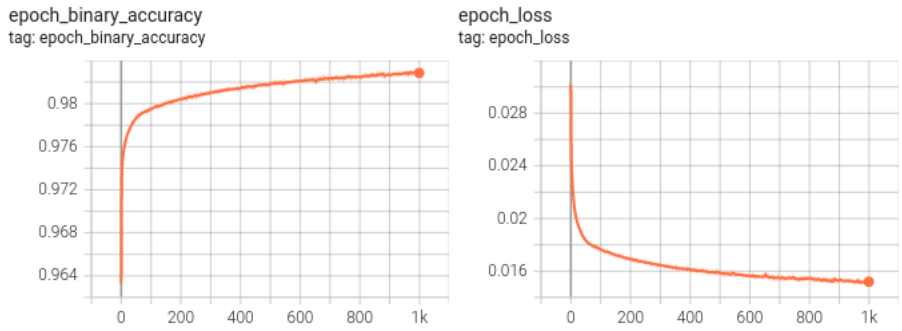


Figure 14: Training Loss and Accuracy vs. epochs for AE2

The evaluation graphs on test data show the effects of the dropout layers. AE1 was trained for 600 epochs and had a maximum test loss of around 0.0265 while AE2 was trained for 1000 epochs and only reached a maximum test loss of around 0.0225. The ideal situation is if the model does not show any signs of overfitting, which means that it is able to find a general pattern that fits each element in the dataset. AE1 achieved a lower training loss than AE2 but AE2 is better at generalizing it's output than AE1.

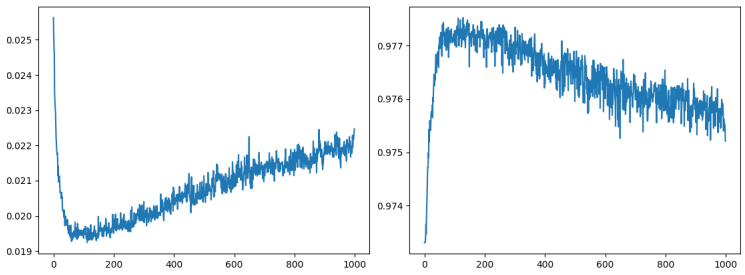


Figure 15: Test Loss and Accuracy vs. epochs for AE2

3.4.3 Results

After training AE1 for 600 epochs and AE2 for 1000 epochs these are the final results. For AE1 the model after 250 epochs was used as the final model, since after epoch 250 the model deteriorated. AE1 has a minimum training loss of $9.4 \cdot 10^{-3}$ and a test loss of 0.0245. AE2 has a final training loss of 0.0157 and a test loss of 0.0221. After each epoch, both models were given a random song out of the testset to recreate. There is not much difference in the respective evolution of the models, except that AE2 progressed slower than AE1, which is why only the evolution of AE2 will be described.

After each epoch a random song was fed through the network. The network outputs a vector which should be in the same one-hot-encoding as the input vector, since it is trying to recreate the input vector, but because of the sigmoid activation the output vector can have values ranging from 0 - 1. These values can be seen as probabilities of whether a note should be played or not. If the chance that a note would be played was higher than a certain threshold then the note gets played. During the first 10 epochs the model was not confident enough and

did not play a single note with a threshold value of 0.1. Then after a couple more epochs the model began to play the note C4 which gets represented as the ascii character H. The middle C is the most common note in the dataset which is why the model learnt to play this note first. After some more iterations the model started to build chords around the C4 note and in the end is able to replace the C4 note for other notes and chords. From this point on melodies and chord progressions start to appear. Even though both models were able to learn a substantial amount, they still have many errors . Figure 16 for example shows the reconstructed segment and the original segment side by side. The upper half is almost fully represented whereas the bottom is not played at all. The autoencoder thus still has many faults.

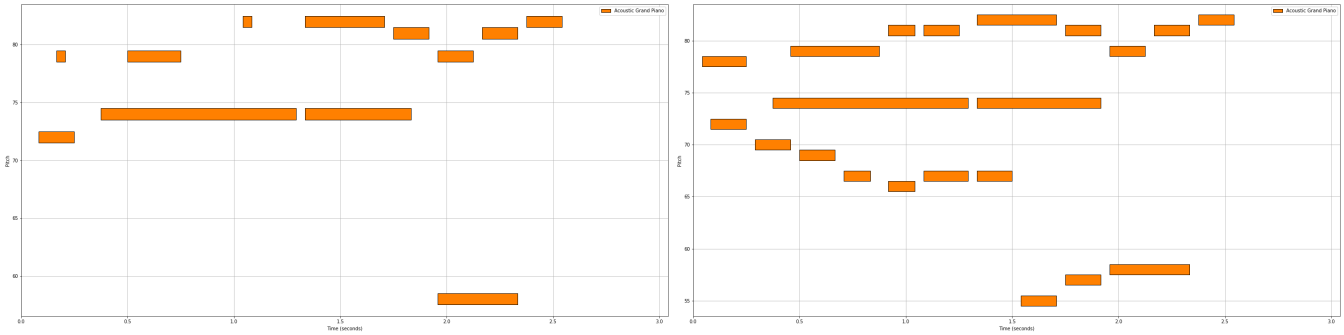


Figure 16: Reconstructed vs. Original Piece

To generate songs with the autoencoders, the trained network is divided into two parts, the encoder and the decoder network as mentioned in section 3.4. The decoder network for both AE1 and AE2 takes in a vector with 16 elements. Each element has to be between -1 and 1 because of the TanH activation function. This vector can either be generated manually or retrieved from the dataset by feeding a song segment from the dataset through the encoder and retrieving it's output. Since the goal is to generate new music, the encoders output can be tweaked manually to create a similar but still unique music piece. The first 16 adjustable sliders in figure 17 represent the activations of the latent vector, the sliders can be adjusted to reach any point in the latent space.

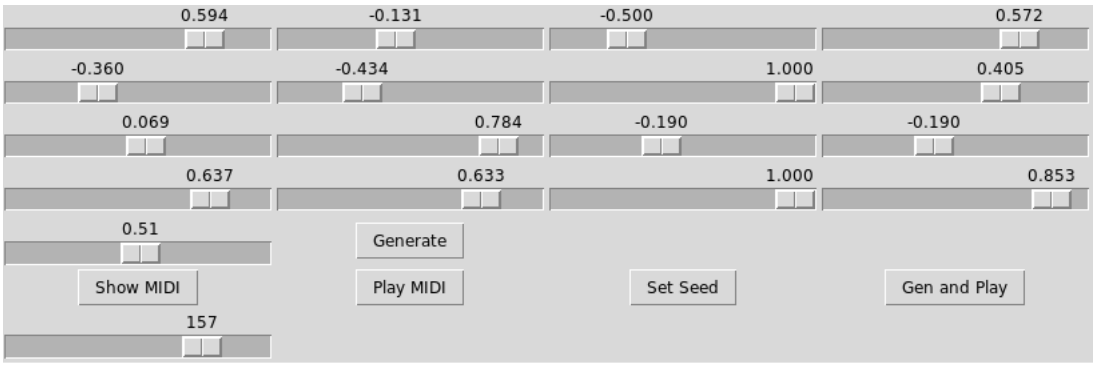


Figure 17: Latent Vector from Dataset

The main differences between AE1 and AE2 come into play during generation. While AE1 is very much capable of producing pieces that sound like music, there is not much playroom for the variables in the latent space. On the other hand the latent space for AE2 is much more customizable. Sadly not every combination of latent variables evokes good sounding music. Sometimes the notes do not harmonize at all and other times only one single note gets played

over the entire duration. Probably one of the biggest faults of the autoencoders is that many notes are too short and don't sound good as a result. If the autoencoder were to make the same notes longer the pieces would sound better. Interpreting the effect of each slider was not possible in this case, they don't seem to show any obvious patterns. The variability of each latent component is quite low. Interesting to see is that when taking the average over all latent spaces in the dataset and decoding the resulting vector, the network outputs the note C4.

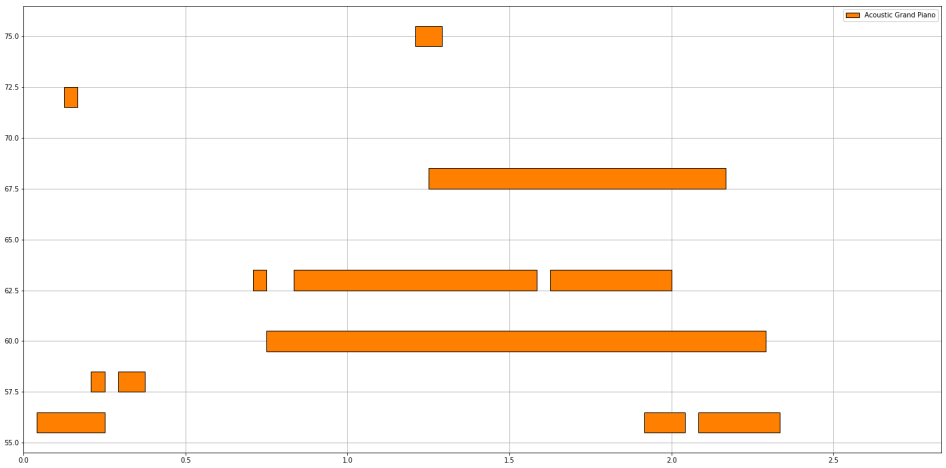


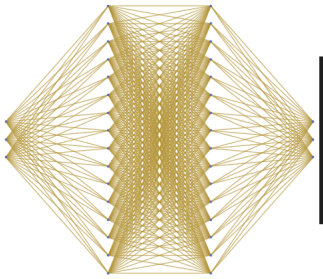
Figure 18: Segment produced with AE2

3.5 LSTM

LSTMs have the capability to capture long time dependencies which comes in handy for long sequences such as music. To generate from an LSTM a one-hot-vector is created, either manually or sampled from the data, which is used to begin the piece. The vector gets fed through the network and creates a new vector. The values of this new vector can be interpreted as probabilities just like the autoencoder. If the value is high then the corresponding note has a higher chance of being played. Simply feeding the output right back into the network, repeating the process a couple of times and then playing the outputted notes that are over a certain threshold is one way of generating. Sometimes however this leads to the network "running out" of notes to play if the piece gets very long. To avoid this an element of randomness is added to the process. This is done by taking the probability for each note and doing a random binary "toss" to see whether that note gets played or not with respect to it's probability. For example if element 25 in the output vector has a value of 0.3 then the chance that the corresponding note, in this case A2, will be played is 30%. This is of course only possible if the activation function outputs a number between 0 and 1 but allows for longer pieces to be generated.

3.5.1 Design Choices

The following model will be referred to as LSTM1. LSTM1 takes in a vector with 88 elements. It has two layers with 500 LSTM cells and one dense layer with 88 cells as shown in figure 19. The activation for the final layer is sigmoid. The activation functions for the LSTM layers are described in section 2.4. A diagram of the unfolded network could not be drawn, please refer to figure 6 to see an unfolded RNN. The process of finding the right network took a lot of time and tuning, however it seems beneficial to have a larger network that can analyse the input vector in detail. It is almost the opposite to the autoencoder where the input is compressed. Here the input is projected onto a higher dimension before being translated back into the original space.



```
model = tf.keras.sequential([
    LSTM(500, input_dim=88,
    return_sequences=True),
    LSTM(500, return_sequences=True),
    Dense(88, activation='sigmoid'),
])
```

Figure 19: LSTM1 Diagram

3.5.2 Training Process

The same collection of measures that was used to train the autoencoders was used for training the LSTM. For training the model was given a measure and tasked to generate the next one while the loss was recorded. Then the output was fed back into the network and the process repeated 128 times before resetting the LSTM. After playing around with different optimizers RMSprop showed the best results along with mean squared error as a loss function.

```
model = mod.lstm()
RMSprop = tf.keras.optimizers.RMSprop(learning_rate=0.001)
model.compile(optimizer=RMSprop, loss=tf.keras.losses.MeanSquaredError(), metrics=['binary_accuracy'])
```

Figure 20: Setting for training LSTM

Many model were tested until finally LSTM1 was found. Other models were able to reach the same loss and accuracy as LSTM1, but their predictive capabilities fell short most of the time for unknown reasons. When trying to produce new content the majority of models either stopped generating notes after very few seconds, or started randomly generating excessive amounts of notes. LSTM1 achieved a test loss of below 0.008 after 200 epochs and 12 hours as can be seen in figure 21. LSTM1 was selected simply because of it's predictive abilities.

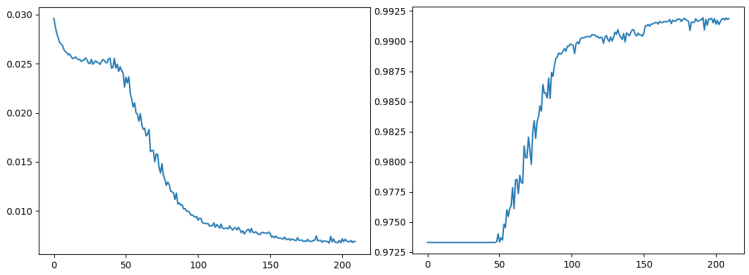


Figure 21: Test Loss and Accuracy vs. epochs for LSTM1

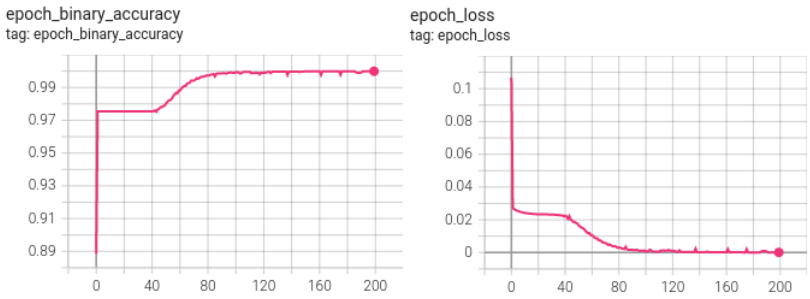


Figure 22: Training Loss and Accuracy vs. epochs for LSTM1

3.5.3 Results

The evolution of LSTM1 was quite opposite to AE1 and AE2. LSTM1 first started with many random notes placed all over the place, randomly guessing and making many mistakes. The values in the output array were around 0.3, which is very low but still higher than AE1 and AE2. Over time LSTM1 learned how to "filter out" unnecessary notes. The gap between high and low values in the output vector was becoming larger, meaning that the network learnt which notes were relevant. In the end the network was able to filter out unnecessary notes completely and produce recognizable musical content. Figure 23 illustrates a piece created by LSTM1. The very first chord was given to the network manually as a starting point, it is part of an C#9b5 chord which is a rare chord. The network transitions to a C#dim7 and then into an unknown chord into a B7 chord finishing of with a B chord whilst adding various melodic notes. It is worth pointing out that the LSTM reached a far lower loss than both autoencoders in both test and training. Which leads to the assumption that LSTMs are better suited for classical music than autoencoders.

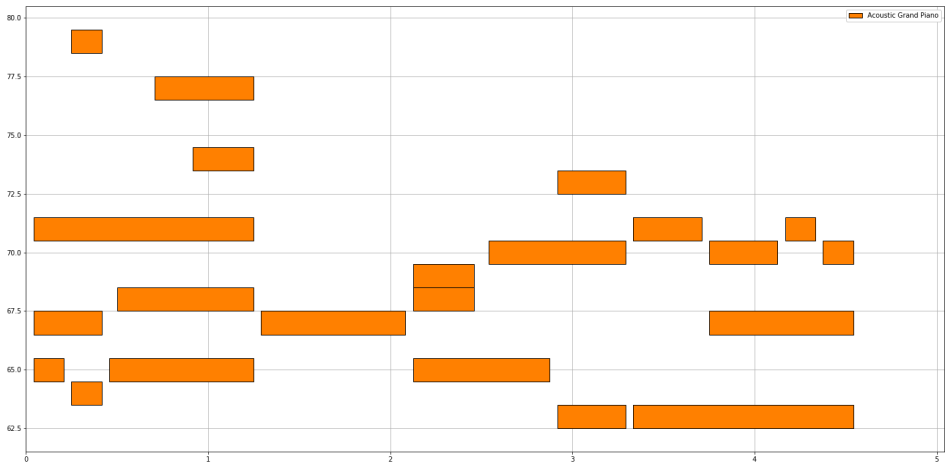


Figure 23: Piece by LSTM1 after 200 Epochs

4 Discussion

This work has shown how machine learning algorithms are capable of generating classical music. Furthermore this study suggest that LSTMs are superior to autoencoders in terms of classical music generation. Whether this is true or not would require further testing and if this statement still holds for other genres is a completely different matter. LSTM1 is able to produce a piece from almost any given starting vector. The autoencoders are able to reconstruct parts

of songs from the test set but are still lacking in some areas. Tweaking the values in the latent spaces results in new pieces that sound good on some occasions.

Neural networks are very much capable of creating music as the results demonstrate. Although the algorithms of today are still a ways off from competing with human composers it is not unreasonable to assume that someday machines will reach a level equal to that of a human musician. This does not mean that humans will be replaced by machines, far more it opens a whole new set of tools for humans to work and produce with. Algorithms similar to the autoencoders presented in this project could be used for altering existing music or for mixing two melodies together by interpolating between their latent spaces. Just like painters mix colors together musicians could mix melodies or chords together. LSTMs might help inspire an artist or bridge the gaps between music segments making it easier for musicians to connect ideas together. Better models than the ones in this work have already been developed using more sophisticated algorithms such as bidirectional LSTM. Portability is not an issue since the only step that requires big computers is the training phase and even that is getting more efficient by the day.

5 Acknowledgements

I would like to thank Prof. for supervising and reviewing this project and giving supportive feedback whenever necessary. I would also like to thank my family and especially my father and grandfather for help with editing and always motivating me to become a better programmer.

6 References

Abdi, H. & Williams, L. J. (2010), 'Principal component analysis', *Wiley interdisciplinary reviews: computational statistics* **2**(4), 433–459.

Briot, J., Hadjeres, G. & Pachet, F. (2017), 'Deep learning techniques for music generation - A survey', *CoRR* **abs/1709.01620**.
URL: *http://arxiv.org/abs/1709.01620*

Deisenroth, M. P., Faisal, A. A. & Ong, C. S. (2020), *Mathematics for machine learning*, Cambridge University Press.

Dick, S. (2019), 'Artificial intelligence'.

Glorot, X. & Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* 'Proceedings of the thirteenth international conference on artificial intelligence and statistics', JMLR Workshop and Conference Proceedings, pp. 249–256.

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R. & Schmidhuber, J. (2016), 'Lstm: A search space odyssey', *IEEE transactions on neural networks and learning systems* **28**(10), 2222–2232.

Hochreiter, S. & Schmidhuber, J. (1997), 'Long short-term memory', *Neural computation* **9**(8), 1735–1780.

Ladjal, S., Newson, A. & Pham, C.-H. (2019), 'A pca-like autoencoder', *arXiv preprint arXiv:1904.01277* .

Mangal, S., Modak, R. & Joshi, P. (2019), 'Lstm based music generation system', *arXiv preprint arXiv:1908.01080* .

Olah, C. (2015), 'Understanding lstm networks'.
URL: *http://colah.github.io/posts/2015-08-Understanding-LSTMs/*

Roberts, A., Engel, J., Raffel, C., Hawthorne, C. & Eck, D. (2018), A hierarchical latent vector model for learning long-term structure in music, *in* 'International conference on machine learning', PMLR, pp. 4364–4373.

Ruder, S. (2016), 'An overview of gradient descent optimization algorithms', *arXiv preprint arXiv:1609.04747* .

Shlens, J. (2014), 'A tutorial on principal component analysis', *arXiv preprint arXiv:1404.1100* .

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014), 'Dropout: a simple way to prevent neural networks from overfitting', *The journal of machine learning research* **15**(1), 1929–1958.

Sun, F. (2017), 'Deephear-composing and harmonizing music with neural networks', *URL: https://fephsun. github. io/2015/09/01/neural-music. html* .

Van Der Maaten, L., Postma, E., Van den Herik, J. et al. (2009), 'Dimensionality reduction: a comparative', *J Mach Learn Res* **10**(66-71), 13.

Wang, W., Huang, Y., Wang, Y. & Wang, L. (2014), Generalized autoencoder: A neural network framework for dimensionality reduction, *in* 'Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops'.

Weng, L. (2018), 'From autoencoder to beta-vae'.
URL: *https://lilianweng.github.io/lil-log/2018/08/12/from-autoencoder-to-beta-vae.html*

7 Selbständigkeitserklärung

Ich, , bestätige mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

Unterschrift:

Zürich, 3.Januar.2022