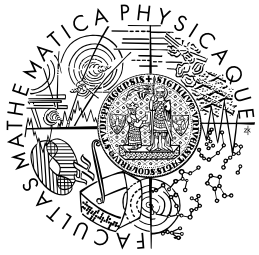


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Eliška Janásková

Statistical machine learning with application in music

Department of Probability and Mathematical Statistics

Supervisor of the master thesis: RNDr. Jan Večeř, Ph.D.
Study programme: Mathematics
Study branch: PMSE

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to express my gratitude to my supervisor Jan Večeř for support and helpful comments. His suggestions led me through the whole process of writing. Besides, I wish to thank Matúš Maciak for essential remarks and the guidance. Also, I would like to thank my boyfriend, who patiently helped me to overcome technical issues when learning new programming language.

Title: Statistical machine learning with application in music

Author: Eliška Janásková

Department: Department of Probability and Mathematical Statistics

Supervisor: RNDr. Jan Večeř, Ph.D., Department of Probability and Mathematical Statistics

Abstract: The aim of this thesis is to train a computer on Beatles' songs using the research project Magenta from the Google Brain Team to produce its own music, to derive backpropagation formulas for recurrent neural networks with LSTM cells used in the Magenta music composing model, to overview machine learning techniques and discuss its similarities with methods of mathematical statistics. In order to explore the qualities of the artificially composed music more thoroughly, we restrict ourselves to monophonic melodies only. We train three deep learning models with three different configurations (Basic, Lookback, and Attention) and compare generated results. Even though the artificially composed music is not as interesting as the original Beatles, it is quite likeable. According to our analysis based on musically informed metrics, artificial melodies differ from the original ones especially in lengths of notes and in pitch differences between consecutive notes. The artificially composed melodies tend to use shorter notes and higher pitch differences.

Keywords: machine learning, music composition, neural networks with LSTM, Tensorflow, evaluation of music

Contents

Introduction

Artificial and human intelligence compete with each other in various tasks. Computer programs perform very well especially when it comes to games with defined rules. In 1997, Deep Blue from IBM beat the human grandmaster Garry Kasparov at chess [Campbell, Hoane Jr., and Hsu, 2002]. In 2011, the question-answering computer system IBM Watson [Ferrucci, 2012] won the first prize on the quiz show Jeopardy! against the legendary champions Brad Rutter and Ken Jennings. In 2015, Google's DeepMind system Alpha Go defeated the European reigning Go Champion Fan Hui [Silver et al., 2016]. In 2017, the artificial intelligence Libratus developed at Carnegie Mellon University defeated four top professional poker players in no-limit Texas Hold'em [Brown and Sandholm, 2017].

Tasks requiring creativity, on the other hand, can be very challenging for computers [A. Boden, 1998]. Despite that fact, generative systems already managed to write poetry [Yan, 2016] or paint images [Gregor et al., 2015]. A music composition is undoubtedly a very complicated and powerful field of art, as music can express emotions and inspire people all around the world.

> *"Music gives a soul to the universe, wings to the mind, flight to the imagination and life to everything."* — Plato

> *"How is it that music can, without words, evoke our laughter, our fears, our highest aspirations?"* — Jane Swan

An artificial music composition is challenging for various reasons. One of them is the inability to automatically evaluate the generated results. Another is to capture a long-term structure, which many musical masterpieces undeniably have.

This thesis is organized as follows: In the first chapter, we present the ideas of machine learning as well as the most common algorithms (a linear regression, support vector machines, k-nearest neighbors, regression trees and random forests, a principal component analysis and k-means) and introduce neural networks. In the second chapter, we compare the feedforward neural network with the generalized linear model, describe recurrent neural networks with LSTM cells, which are suitable for music representation, derive backpropagation formulas used in Magenta model and review the important milestones in artificial music composition.

Finally, in the third chapter, we train three recurrent neural networks with LSTM cells, which were introduced by Google's Brain Team Magenta [Waite, 2016], on Beatles' songs. In order to avoid overfitting, we choose an appropriate number of iterations of neural network training for each of the three different configurations. As we restricted ourselves to monophonic melodies to be able to explore generated music more comprehensively, we provide a script in Python capable of reducing polyphonic melodies into monophonic ones. In addition, generated pieces of music are evaluated both objectively using musically informed metrics [Yang and Lerch, 2018] and subjectively.

1. Machine Learning: Overview

Machine learning is a form of applied statistics, which enables us to solve problems, that are too difficult to be solved by fixed programs without any learning abilities. In this chapter, we introduce the basic concepts of machine learning and the most frequently used algorithms alongside with the neural networks.

At the very beginning, I would like to point out the difference between machine learning models and statistical models, as some terms (such as a linear regression, see the section 1.1.1) may sound similar, but they are slightly different. Generally speaking, machine learning models are not based on a probability theory. The machine learning model is usually trained on a subsample of the data set and we do not know how well the model will perform until we test it on the additional data set that was not present during training. On the other hand, statistical models are specified under certain assumptions, which enable us to derive the properties of the parameter estimations. There is no need for the testing and the evaluating dataset. The more detailed comparison of the statistical generalized linear model and the machine learning simple feed forward neural network can be found in the section 2.1.

Both approaches can be useful based on the nature of the problem. Machine learning models are suitable when it comes to predicting numerical values (regression problems) or categories (classification problems) based on other variables, because they are able to capture complex nonlinear relationships. Nevertheless, they may be hard to interpret and can be used as a blackbox. On the contrary, statistical models enable us to make inference about parameters and distinguish which variables actually effect the dependent variable through submodel testing. No such thing is possible using machine learning algorithms.

The first attempt to develop a machine that would imitate a living creature in performance goes back to the beginning of the 1930s when Thomass Ross [1938] started his work on automata maze-solver called Robot Rat. His machine was inspired by a familiar test of animal intelligence in finding the way out of a maze. The automatic learning feature of the Robot Rat was based on a rotating disk, so-called Memory Disk. He used a system of toy-train tracks, so his maze-solver could learn the way out by trial and error.

In 1959, Arthur Samuel, who is considered to be a pioneer in the field of artificial intelligence, established the term *Machine Learning* and claimed: *"Programming computers to learn from experience should eventually eliminate the need for much of this detailed programming effort"* [Samuel, 1959]. That quote was slightly changed and is now frequently used to define *Machine Learning* as:

> *"Field of study that gives computers the ability to learn without being explicitly programmed."*

In 1997, a more engineering-oriented definition was stated by Mitchell [1997]:

> *"A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

Mitchell's definition covers all three necessary parts of a machine learning algorithm. **Task T** is usually described in terms of how the machine learning algorithm should process an observation, which is given as a set of features. Mathematically, we usually represent one observation as a vector $\boldsymbol{X} \in \mathbb{R}^p$, where for each $i = \{1, ..., p\}$, X_i describes one feature. The most common tasks T are classification and regression, examples of other tasks can be a transcription, a machine translation, an anomaly detection, an imputation of missing values, a denoising, a density estimation, etc. The process of learning is usually done by improving the approximation of certain parameters. Nevertheless, some parameters often need to be fixed before the training process, e.g. the degree of a polynomial in linear regression. These parameters are called *hyperparameters*.

Performance measure P is chosen to evaluate the quality of the model in terms of the task T. In case of a classification, it is reasonable to measure the *accuracy* of the model (a proportion of correctly classified observations) or the *error rate* (a proportion of incorrectly classified observations). The usual way of measuring the performance is done by various loss functions L. The accuracy or the error-rate are referred to as the 0-1 loss, as one specific observation is classified either correctly (0) or incorrectly (1). On the other hand, it is more reasonable to use continuous loss functions L for tasks such as a regression or a density estimation. The learning process is then associated with minimization of the loss function L with respect to the parameters that aim to be estimated.

Based on **experience E**, machine learning algorithms can be broadly categorized as *supervised* (each input observation is associated with a label or a target value) or *unsupervised* (no output annotation provided). Supervised algorithms learn to understand the structure of examples with respect to provided outputs or labels. Unsupervised algorithms experience features of a dataset and learn useful properties of the structure without being told about desired results. Nevertheless, supervised and unsupervised learning are not completely separable concepts, as it is sometimes hard to distinguish between features and labels.

In practice, we want the machine learning model to perform well in general, not only on the training dataset. It is possible to design an extremely complex model that would fit perfectly to the provided training examples but failed in case of the unobserved ones. This issue is called **overfitting**. To avoid overfitting, it is recommended to randomly divide the dataset into the training set and the evaluating set. Firstly, we use the training set to optimize the model parameters and then, after the training part is complete, we evaluate the model on the test set. The model able to generalize should perform similarly on both sets.

1.1 Supervised Learning Algorithms

1.1.1 Linear Regression

One of the basic supervised learning algorithm is the linear regression. Suppose we have data $(Y_i, \boldsymbol{X}_i^T)^T$, $i = 1, \ldots, n$, $Y_i \in \mathbb{R}$, $\boldsymbol{X}_i \in \mathbb{R}^p$. The goal is to predict the outcome Y_i from $\boldsymbol{X}_i$.

Let us denote

$$\mathbb{X} = \begin{pmatrix} \boldsymbol{X}_1^T \\ \vdots \\ \boldsymbol{X}_n^T \end{pmatrix}, \boldsymbol{Y} = (Y_1, \ldots, Y_n)^T,$$

the input data matrix $\mathbb{X}$ with included intercepts and $\boldsymbol{Y}$ the output vector.

In the statistical approach, we often work under the assumption of normal linear model specified as $\boldsymbol{Y}|\mathbb{X} \sim \mathcal{N}_n(\mathbb{X}\boldsymbol{\beta}, \sigma^2\mathbb{I}_n)$, where $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_{p-1})^T \in \mathbb{R}^p$ and $0 < \sigma^2 < \infty$ are unknown parameters. This enables us to derive properties of parameter estimations, such as their conditional distribution etc.

In machine learning approach, on the other hand, there are no assumptions about conditional distribution $\boldsymbol{Y}|\mathbb{X}$, nor conditional expected value $\mathbb{E}(\boldsymbol{Y}|\mathbb{X})$, nor conditional variance $\text{var}(\boldsymbol{Y}|\mathbb{X})$. The goal is to build a model, that takes vector $\boldsymbol{X}_i \in \mathbb{R}^p$ with included intercept as an input and predict a value of a scalar Y_i as the output in a form of a linear function of the input. Let us denote predicted value of Y_i as $\hat{Y}_i = \boldsymbol{w}^T\boldsymbol{X}_i$, where $\boldsymbol{w} \in \mathbb{R}^p$ is a parameter to be estimated. Task T in this case is to predict Y_i from $\boldsymbol{X}_i$ by computing $\hat{Y}_i = \boldsymbol{w}^T\boldsymbol{X}_i$. As a perfomance measure P we can choose mean square error loss function L_{MSE}, which is given by:

$$L_{MSE}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{w}^T\boldsymbol{X}_i - Y_i)^2.$$

To improve perfomance, we want to minimize MSE with respect to $\boldsymbol{w}$, which can be done by calculating gradient and setting it equal to 0. As a result we obtain system of normal equations for parameter $\boldsymbol{w}$: $\boldsymbol{w} = (\mathbb{X}^T\mathbb{X})^-\mathbb{X}^T\boldsymbol{Y}$. Nevertheless, we obtain the same results due to solving the same set of normal equations in both approaches. Although, in machine learning approach we can only verify quality of parameter $\boldsymbol{w}$ if we split dataset into training and evaluating part and compare their L_{MSE} for all observations $\boldsymbol{X}_i$.

1.1.2 Support Vector Machine

Support vector machine (SVM) is one of the most influential approaches to supervised learning [Cortes and Vapnik, 1995],[Boser, Guyon, and N. Vapnik, 1996]. The goal is to predict one of two categories 0 or 1 for observations $\boldsymbol{X}_i \in \mathbb{R}^{p-1}$, $i = 1, ..., n$, with no itercept term included. Based on the linear function $\boldsymbol{w}^T\boldsymbol{X}_i + b$, the SVM predicts class 1 classification if $\boldsymbol{w}^T\boldsymbol{X}_i + b$ is positive and class 0 classification otherwise.

Thus we need to find a $p-1$ dimensional hyperplane, specified as the set of points $\boldsymbol{x} \in \mathbb{R}^{p-1}$ satisfying $\boldsymbol{w}^T\boldsymbol{x} + b = 0$, where $\boldsymbol{w} \in \mathbb{R}^{p-1}$ is an uknown normal vector to hyperplane, that would separate observations, and $b \in \mathbb{R}$ is an unknown constant. As there might be many hyperplanes that classify the data, it is reasonable to choose the one that represents the largest separation, e.g. we can choose such hyperplane so that the distance from it to the nearest data point on each size is maximized.

A linear function describing a hyperplane can be rewritten as

$$\boldsymbol{w}^T\boldsymbol{x} + b = \sum_{i=1}^{n}\alpha_i\boldsymbol{x}^T\boldsymbol{X}_i + b,$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^T \in \mathbb{R}^n$, $b \in \mathbb{R}$ are parameters to be estimated. This way, we can optimize parameters while taking observations into account.

In case of non-linearly separable points, we can make predictions based on a sign of a function $f : \mathbb{R}^{p-1} \to \mathbb{R}$,

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}, \boldsymbol{X}_i) + b,$$

where $k(\boldsymbol{x}, \boldsymbol{X}_i) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{X}_i)$ is called a kernel function and $\phi : \mathbb{R}^{p-1} \to \mathbb{R}^d$, $d > 0$ is a given feature function. Thanks to this so-called *kernel trick*, we can separate points, which was not linearly separable before transormation ϕ.

Even though a function $f(\boldsymbol{x})$ is non-linear with respect to $\boldsymbol{x}$, it is linear with respect to $\phi(\boldsymbol{x})$ and α. A model based on the kernel-based function is equivalent to the model we would get if we firstly preprocessed the original data by applying the function $\phi(\boldsymbol{x})$ and then learned a model in the newly transformed space.

Common kernels include:

$$k_{polynomial}(\boldsymbol{x}, \boldsymbol{X}_i) = (\boldsymbol{x}^T \boldsymbol{X}_i)^d,$$

$$k_{radial}(\boldsymbol{x}, \boldsymbol{X}_i) = e^{-\gamma \|\boldsymbol{x} - \boldsymbol{X}_i\|^2}, \gamma > 0,$$

$$k_{tanh}(\boldsymbol{x}, \boldsymbol{X}_i) = tanh(\kappa \boldsymbol{x}^T \boldsymbol{X}_i + c), \kappa > 0, c < 0.$$

1.1.3 k-Nearest Neighbours

k-nearest neighbours (k-NN) is a family of simple algorithms that can be used for both a classification and a regression. In both cases, the first step is to find k closest observations to chosen observation $\boldsymbol{X} \in \mathbb{R}^p$, where $k \in \mathbb{N}$ is typically small.

In a k-NN classification, we simply assign the object $\boldsymbol{X}$ to the class most common among its k closest neighbours. In a k-NN regression, the output is averaged output value of its k closest neighbours. For $k = 1$, the algorithm just uses the nearest neighbour value. k-NN is a type of instance-based learning, which means that instead of performing an explicit generation, it simply compares new instances with instances stored in memory. To determine the k-nearest neighbors, one needs to choose a distance metric. Euclidean distance is often used for continuous variables, Hamming distance for categorical variables. There are many generalizations of this concept, such as assigning a weight to the contribution of the neighbors, so that the nearest neighbors contribute more to the average than the distant ones.

1.1.4 Regression Trees and Random Forests

Regression trees are based on a recursive partitioning. In the first step, the sample population $\boldsymbol{X}_i \in \mathbb{R}^p, i = 1, \ldots, n$ is split into two groups based on a *splitting value* of a *splitting variable*. The leftward and rightward groups are then again recursively split until they reach their terminal nodes. Various algorithmic rules can be applied to determine which splitting variable and splitting value to choose.

We will describe the most common method at the step k of the algorithm. Suppose we have set of observations $(Y_i, \boldsymbol{X}_i^T)^T, i = 1, \ldots, n, \boldsymbol{X}_i \in \mathbb{R}^p, Y_i \in \mathbb{R}$. Let us denote I_k group of N_k cases remaining to be split, $m_k = \sum_{i \in \mathbb{I}_k} \frac{Y_i}{N_k}$ mean of their outputs and $S_k^2 = \sum_{i \in \mathbb{I}_k} (Y_i - m_k)^2$. Dividing I_k into $I_{k,left}$ and $I_{k,right}$ gives us respective values of $m_{k,left}$ and $m_{k,right}$ and $S_{k,left}^2$ and $S_{k,right}^2$. The algorithm then choose the splitting variable $\boldsymbol{X}_k$ with the splitting value v_k such that sum $S_{k,left}^2 + S_{k,right}^2$ is minimized. This way, we get two groups that are as different from each other as possible.

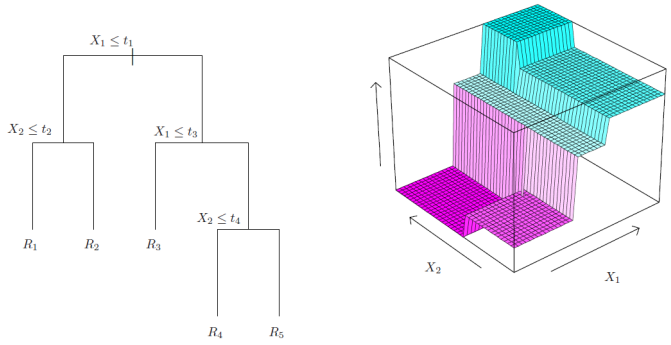


Figure 1.1: Regression tree ([Efron and Hastie, 2016])

Regression trees are high-variance estimators. The bushier they are, the higher is the variance. To overcome this issue, *random forests* were introduced. The idea is to grow many trees and reduce the variance by averaging. As individual trees should not be too correlated, randomness is injected into the tree-growing process by bootstrap resampling and split variable randomization. For more details please refer to the book by Efron and Hastie [2016, section 17.1].

1.2 Unsupervised Learning Algorithms

1.2.1 Principal Component Analysis

A principal component analysis (PCA) provides orthogonal data transformation into a lower dimension while capturing as much variance as possible. It can be viewed as an unsupervised learning algorithm that aims to reduce a dimensionality and a correlation of data. PCA learns a linear orthogonal data transformation which project input vector $\boldsymbol{X} \in \mathbb{R}^p$ into reduced uncorrelated representation $\boldsymbol{Z} \in \mathbb{R}^k$, where $k \ll p$. The PC transformation of a random vector $\boldsymbol{X}$ with a covariance matrix $\boldsymbol{\Sigma}$ and mean $\boldsymbol{\mu}$ is defined as $\boldsymbol{Y} = \boldsymbol{\Gamma}^T(\boldsymbol{X} - \boldsymbol{\mu})$, where $\boldsymbol{\Gamma}$ comes from spectral decomposition of $\boldsymbol{\Sigma}$: $\boldsymbol{\Sigma} = \boldsymbol{\Gamma}\boldsymbol{\Lambda}\boldsymbol{\Gamma}^T$. All elements of $\boldsymbol{Y}$ are mutually uncorrelated and $\mathrm{Var}(Y_i) = \lambda_i, i = 1, \ldots, p$, where λ_i are eigen values of matrix $\boldsymbol{\Sigma}$ ordered by size in diagonal matrix $\boldsymbol{\Gamma}$. After specifying k reasonably with respect to the captured variance, we denote $\boldsymbol{Z}$ as first k components of $\boldsymbol{Y}$.

1.2.2 k-Means Clustering

A k-means clustering is a simple algorithm that divides the training set consisting of n samples $\boldsymbol{X}_i \in \mathbb{R}^p, i = 1, ..., n$, into k different clusters $C_1, \ldots, C_k$, such that observations in each cluster are similar to each other. At the very beginning, k-means algorithm initialize random values to k centroids $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$, $\boldsymbol{\mu}_i \in \mathbb{R}^p$. Then following two steps iterate until convergence:

(1) Each training sample $\boldsymbol{X}_i$ is assigned to the cluster C_i, where i is the index of the closest centroid $\boldsymbol{\mu}_i$.

(2) Values of centroids $\{\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k\}$ are updated to the means of all training examples $\boldsymbol{X}_i$ assigned to respective clusters $C_1, \ldots, C_k$.

Similarly to k-NN, an Euclidean distance is often used for continuous variables. Modifications of the algorithm include k-mode algorithm for categorical data, with simple-matching metric and modes (instead of means) for centroid updates, and k-prototype algorithm for mixed-type data, which combines both k-means and k-modes.

The main drawback of k-means algorithm is that k must be fixed ahead. To determine suitable k, one can use *Elbow method*: Total within sum of squares $\sum_{i=1}^{k} \sum_{j \in C_i} (\boldsymbol{X}_j - \boldsymbol{\mu}_i)^2$ are determined for the different number of clusters and usually the smallest k with reasonably low total within sum of squares is chosen.

1.3 Basic Principles of Neural Networks

Neural networks were evolved from the Perceptron, which was proposed by Rosenblatt [1958], and form an important subfield of machine learning. The Perceptron was originally designed for image recognition, but after a promising start, it was proved that it was only capable of learning linearly separable patterns. After that, neural networks reappeared in the 1980s. This time, the linear separability limitation was overcome by introducing multiple hidden layers joint with nonlinear units [Werbos, 1982]. As the loss function was not convex after such generalization and parameters optimization became far more complicated, backpropagation algorithms using gradient descent were introduced [E. Rumelhart, E. Hinton, and J. Williams, 1986]. This section is based mainly on work of Goodfellow et al. [2016].

Efron and Hastie [2016] describe neural networks as:

> *"Highly parametrized model, inspired by the architecture of the human brain, what was widely promoted as an universal approximator – a machine that with enough data could learn any smooth predictive relationship."*

1.3.1 Motivational Example

At first, we demonstrate the limitations of a linear model and introduce a useful generalization, which will lead us to neural networks.

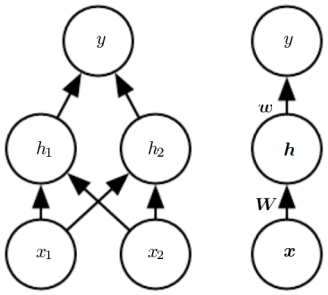


Figure 1.2: Simple feedforward neural network with two hidden units [Goodfellow et al., 2016]

Consider XOR function $f^* : \mathbb{R}^2 \to \mathbb{R}$

$$f^*(x_1, x_2) = \begin{cases} 1 & x_1 = 0, x_2 = 1, \text{ or } x_1 = 1, x_2 = 0, \\ 0 & \text{otherwise.} \end{cases}$$

It is impossible to model the XOR function f^* without the nonlinear approach. We want our model to perform correctly on the four observations:

$$\mathbb{X} = (\boldsymbol{X}_1^T, \boldsymbol{X}_2^T, \boldsymbol{X}_3^T, \boldsymbol{X}_4^T)^T = ((0,0),(0,1),(1,0),(1,1))^T.$$

Goodfellow, Bengio, and Courville [2016] showed, that if we try the linear regression approach with the model $f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T\boldsymbol{w} + b$ and minimize MSE loss function $L_{MSE}(\boldsymbol{w}, b) = \frac{1}{4}\sum_{i=1}^{4}(f^*(\boldsymbol{X}_i) - f(\boldsymbol{X}_i; \boldsymbol{w}, b))^2$ with respect to the parameters $\boldsymbol{w} \in \mathbb{R}^2$, $b \in \mathbb{R}$ using the normal equations, we obtain results $\boldsymbol{w} = \boldsymbol{0}$ and $b = 1/2$. Which means that the linear model simply outputs the value $1/2$ for all the input values.

To solve this issue, we introduce a simple feedforward network with one hidden layer containing two hidden units (figure 1.2), which consists of two chained functions $f^{(1)} : \mathbb{R}^2 \to \mathbb{R}^2$ and $f^{(2)} : \mathbb{R}^2 \to \mathbb{R}$.

A vector $\boldsymbol{h} \in \mathbb{R}^2$, called hidden units, is computed by function $f^{(1)}(\boldsymbol{x}; \mathbb{W}, \boldsymbol{c})$, where $\mathbb{W}_{2\times 2}$ is weight matrix of linear transformation, $\boldsymbol{c} \in \mathbb{R}^2$ is the bias vector, and then used as the input for second layer. Output of second layer is still linear regression model, but this time applied to $\boldsymbol{h}$ instead of $\boldsymbol{x}$. The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}, \mathbb{W}, \boldsymbol{c})$, $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$, with the complete model $f(\boldsymbol{x}; \mathbb{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

To capture nonlinearity features of the function f^*, we clearly need the nonlinear function $f^{(1)}$, otherwise the compound function f would stay linear.

We define $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \mathbb{W}, \boldsymbol{c}) = g(\mathbb{W}^T\boldsymbol{x} + \boldsymbol{c})$. Function g, so-called an *activation function*, is set to be $g(z) = max\{0, z\}$. Thus, complete model can be specified as $f(\boldsymbol{x}; \mathbb{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T max\{\boldsymbol{0}, \mathbb{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$, where $\mathbb{W}$, $\boldsymbol{w}$, $\boldsymbol{c}$ and b are the parameters to be learned. One solution, which can be found by gradient descent algorithm (see section 1.3.3) is

$$\mathbb{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \boldsymbol{c} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \boldsymbol{w} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0.$$

Then for

$$\mathbb{X} = (\boldsymbol{X}_1^T, \boldsymbol{X}_2^T, \boldsymbol{X}_3^T, \boldsymbol{X}_4^T)^T = ((0,0),(0,1),(1,0),(1,1))^T,$$
$$\boldsymbol{Y} = (Y_1, Y_2, Y_3, Y_4)^T = (0,1,1,0)^T,$$

we get that $Y_i = f(\boldsymbol{X}_i) = f^*(\boldsymbol{X}_i), i = 1, \ldots, 4$.

1.3.2 Feedforward Neural Networks

Feedforward neural networks, also called multilayer perceptrons, represent the first and the simplest type of artificial neural networks. Later on, convolutional, recurrent and recursive neural networks were derived from this base. The core ideas behind modern feedforwards networks are still based on the same backpropagation algorithm and gradient descent approach like in the 1980s, but the availability of larger datasets and more powerful computers significantly improve their performance.

The goal is to approximate a function f^* with a function f, which is formed by chaining several functions, e.g. $f(\boldsymbol{x}) = f^{(n)} \circ ... \circ f^{(1)}(\boldsymbol{x})$. A function $f^{(1)}$ is called the **input layer**, functions $f^{(2)}, \ldots, f^{(n-1)}$ represent **hidden layers**, a function $f^{(n)}$ is called the **output layer**. The natural number n captures the overall length of the chain and gives us the depth of the neural network. A feedforward neural network with more then two hidden layers is called a deep feedforward neural network, which arose the name *deep learning*. The name **feedforward** came from the fact, that information captured in $\boldsymbol{x}$ goes straight from $f^{(1)}$ to $f^{(n)}$ and there are no feedback connections, unlike in recurrent neural networks (see section 2.1).

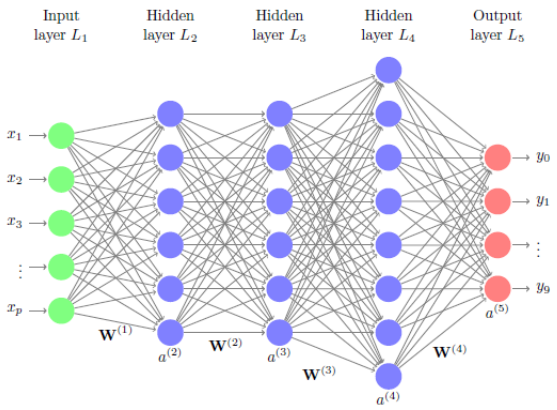


Figure 1.3: Feedforward neural network [Efron and Hastie, 2016]

In fact, a neural network is just a nonlinear model, not too different from many other generalizations of a linear model. Each cell, called a **neuron**, computes a linear combination of its inputs and puts the result into an activation function g. The output of the j-th neuron in the i-th layer is given by $f_j^{(i)}(\boldsymbol{h}^{(i-1)}; \boldsymbol{w}_j^{(i)}) = g(\boldsymbol{w}_j^{(i)T}\boldsymbol{h}^{(i-1)})$, where $\boldsymbol{w}_j^{(i)} \in \mathbb{R}^p$ is a specific parameter vector of weights belonging to j-th neuron in the i-th layer which needs to be learned,

$\boldsymbol{h}^{(i-1)} \in \mathbb{R}^p$ is output from previous layer and g is a nonlinear activation function. Usually, we also include a bias term $w_0 \in \mathbb{R}$, then $f_j^{(i)}(\boldsymbol{h}^{(i-1)}; \tilde{\boldsymbol{w}}_j^{(i)}) = g(\boldsymbol{w}_j^{(i)T}\boldsymbol{h}^{(i-1)} + w_0)$, where $\tilde{\boldsymbol{w}}_j^{(i)} = (w_0, \boldsymbol{w}_j^{(i)T})^T$.

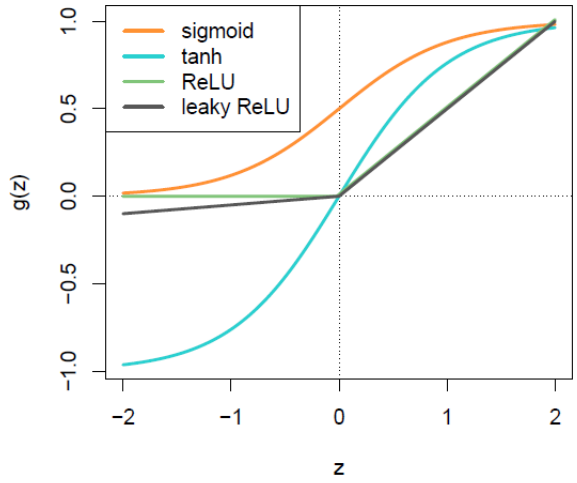


Figure 1.4: Feedforward neural network [Efron and Hastie, 2016]

The most often used activation function are sigmoid, tanh, reLU, leaky reLU [Efron and Hastie, 2016].

Sigmoid $g(z) = \frac{e^z}{e^z+1} \in (0, 1)$.

Hyperbolic tangens tanh $g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \in (-1, 1)$.

Rectified linear unit ReLU $g(z) = max\{0, z\} = z_+ \in \mathbb{R}_0^+$, also called a positive-part function, with advantage of making gradient computations cheaper to compute.

Leaky rectified linear unit leaky ReLU $g(z) = z_+ - \alpha z_- \in \mathbb{R}$, where α is non-negative and close to zero. Introducing α reduces flat spots for negative numbers and helps avoid zero gradients.

Finally, we need to specify a loss function. An example of a loss function commonly used for regression is the mean square error:

$$L_{MSE}(\hat{\boldsymbol{Y}}, \boldsymbol{Y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2,$$

where $\hat{\boldsymbol{Y}} \in \mathbb{R}^n$ are estimated values of $\boldsymbol{Y} \in \mathbb{R}^n$.

A cross entropy loss is a widely used loss function for classification tasks with K categories $Y_1, \ldots, Y_K$. At first, we use a softmax function, which takes an output vector $\hat{\boldsymbol{Y}} \in \mathbb{R}^K$ estimating $\boldsymbol{Y} \in \mathbb{R}^K$ and transforms it into a vector $\boldsymbol{p} \in \mathbb{R}^K$ of probabilites

$$p_i = softmax(\hat{\boldsymbol{Y}})_i = \frac{e^{\hat{Y}_i}}{\sum_{j=1}^{K} e^{\hat{Y}_j}},$$

then we can apply a cross entropy loss as:

$$L_{CE}(\boldsymbol{p}, \boldsymbol{Y}) = -\sum_{i=1}^{K} Y_i log(p_i).$$

1.3.3 Backpropagation and Gradient Descent Algorithm

In this section, we provide a brief overview of backpropagation and gradient descent algorithm. Detailed derivation of backpropagation in recurrent neural network, which was used for artificial music composition, follows in section 2.3. During a training phase of a neural network, backpropagation and gradient descent algorithms are used to the find optimal values of the parameter $\boldsymbol{w}$. After feeding the input $\boldsymbol{X}$ into the feedforward neural network, we obtain output $\hat{\boldsymbol{Y}}$ and respective scalar cost $L(\boldsymbol{w})$. The backpropagation algorithm then lets the information from the cost flow backward through the network to calculate the gradient. Then, the gradient descent algorithm is used to perform learning using this gradient.

As a neural network consists of many chained functions, the backpropagation algorithm computes derivatives of compound functions using the chain rule. Analytically, this is straightforward, but evaluating gradient numerically can be computationally expensive and through iterations, the same subexpressions are computing over and over again. A detailed description of various modifications using different approaches to those subexpressions can be found in Goodfellow et al. [2016, section 6.5.3].

Having the gradient calculated, one can use the gradient descent algorithm to iteratively compute optimal values of $\boldsymbol{w}$. For a training set $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ with the targets $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$ and the loss function L, the gradient $\boldsymbol{g}$ of the parameter $\boldsymbol{w}$ is calculated and values of $\boldsymbol{w}$ are updated:

$$\boldsymbol{g} = \frac{1}{n} \nabla_w \sum_{i=1}^{n} L(f(\boldsymbol{X}_i; \boldsymbol{w}), \boldsymbol{Y}_i),$$

$$\boldsymbol{w}^{new} = \boldsymbol{w}^{old} - \epsilon_k \boldsymbol{g},$$

where ϵ_k is prespecified learning rate, which is gradually decreasing over time.

As computing gradient on all available data can be very slow in practice, **stochastic gradient descent** calculating a gradient only with a batch of B examples is used instead. Similarly as with any other optimization algorithm, many modifications were introduced to make a gradient descent more effective and robust against a local minima, for more details see Goodfellow et al. [2016, chapter 9].

2. Mathematical and Statistical Theory

In this chapter, we show the similiarities between neural networks and generalized linear model. Then, we introduce recurrent neural networks, illustrate complications with long-term dependecies and derive the expressions for the backpropagation in recurrent neural networks with LSTM cells used for the artificial music composition in the third chapter. Finally, we provide an overview of important milestones in artifical music composition

2.1 Neural networks and generalized linear model

Suppose we have the data $(Y_i, \boldsymbol{X}_i^T)^T$, $i = 1, \ldots, n$, $Y_i \in \mathbb{R}, \boldsymbol{X}_i \in \mathbb{R}^p$. The goal is to predict the outcome Y_i from $\boldsymbol{X}_i$. In case of continous variable $Y_i \in \mathbb{R}$, this is a regression task, in case of discrete $Y_i \in \mathbb{Z}$ with K values, $K \in \mathbb{N}$, this is a classification task. We will apply both neural network and generalized linear model approach to point out differences and similarities. We will show, that a logistic regression and a feedforward neural network with one hidden layer and the sigmoid activation function with a cross entropy lost function L_{CE} for two categories lead to the same results.

2.1.1 Generalized Linear Model Approach

Let us recall what holds for the data $(Y_i, \boldsymbol{X}_i), Y_i \in \mathbb{R}, \boldsymbol{X}_i \in \mathbb{R}^p$, under the assumptions of the generalized linear model [Nelder and Wedderburn, 1972]:

1. Let $Y_1, \ldots, Y_n$ be independent with the distribution of Y_i depending on $\boldsymbol{X}_i$ through regression parameters $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)^T$
2. The conditional distribution of Y_i given $\boldsymbol{X}_i$ has the form

$$f(y; \theta_i, \varphi) = exp\left\{ \frac{y\theta_i - b(\theta_i)}{\varphi} + c(y, \varphi) \right\}$$

   (is of exponential type), where $b(\cdot)$ is a known twice continuously differentiable function, θ_i depends on $\boldsymbol{X}_i$ and $\boldsymbol{\beta}$, φ is known or unknown constant
3. The parameter θ_i depends on $\boldsymbol{X}_i$ and $\boldsymbol{\beta}$ through linear predictor $\eta_i := \boldsymbol{X}_i^T \boldsymbol{\beta}$
4. There exists a known strictly monotone, twice continuously differentiable link function g such that $g(\mu_i) = \eta_i$.

To compare the cross-entropy loss function results with the logistic regression, consider independent variables $Y_{ij}^* \sim Alt(\pi_i)$, $\pi \in (0, 1)$, $i = 1, \ldots, n$, $j = 1, \ldots, m_i$, where for a fixed i, $Y_{i1}^*, \ldots, Y_{im_i}^*$ are identically distributed. The total number of observations is $N = \sum_{i=1}^n m_i$.

Let us choose a canonical logistic link $g(\mu_i) = (b')^{-1}(\mu_i) = log\frac{\mu_i}{1-\mu_i}$, where $\mu_i = \frac{exp\{\boldsymbol{X}_i^T\beta\}}{1+exp\{\boldsymbol{X}_i^T\beta\}}$ which further implies $\theta_i = log\frac{\pi_i}{1-\pi_i}, b(\theta_i) = log(1 + exp\{\theta_i\})$.
This lead us to the likelihood

$$L(\boldsymbol{\beta}|\boldsymbol{Y}) = \prod_{i=1}^{n}\prod_{j=1}^{m_i}\left[\pi_i^{Y_{ij}^*}(1-\pi_i)^{1-Y_{ij}^*}\right],$$

if we denote $Y_i = \sum_{j=1}^{m_i} Y_{ij}^*$, we get loglikelihood in a form

$$l(\boldsymbol{\beta}|\boldsymbol{Y}) = \sum_{i=1}^{n}\left[Y_i\,\log\frac{\pi_i}{1-\pi_i} - m_i\,\log\frac{1}{1-\pi_i}\right].$$

For $m_i = 1$, meaning we have only one response Y_i for each covariate vector $\boldsymbol{X}_i$, which happens when at least one covariate is continuous, we obtain this equation to maximize with respect to $\boldsymbol{\beta}$:

$$l(\boldsymbol{\beta}|\boldsymbol{Y}) = \sum_{i=1}^{n}\left[Y_i\,\log\frac{\pi_i}{1-\pi_i} - \log\frac{1}{1-\pi_i}\right]. \tag{2.1}$$

We will show, that we obtain exactly the same expression to minimize with respect to the parameters in case of feedforward neural network with one hidden layer and the sigmoid activation function when we use the cross-entropy loss function L_{CE} for 2 categories.

2.1.2 Neural Network Approach

At first, we present the universal approximation theorem [Cybenko, 1989], which states that a feedforward neural network f with one hidden layer and nonconstant, bounded and continous activation funcion g can approximate any continuous function f^* on I_p, p-dimensional unit hypercube $[0,1]^p$. The theorem holds even if we replace I_p with any compact subset of $\mathbb{R}^p$.

Theorem 1. *Let $g : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded and continous function (called the activation function). Let I_p denote p-dimensional unit hypercube $[0,1]^p$. The space of real-valued continuous functions on I_m is denoted by $C(I_p)$. Then, given any $\epsilon > 0$ and any function $f^* \in C(I_p)$, there exist $n \in \mathbb{N}$, $v_i, b_i \in \mathbb{R}$, $\boldsymbol{w}_i \in \mathbb{R}^p, i = 1,\ldots,n$, such that we may define: $f(\boldsymbol{x}) = \sum_{i=1}^{n} v_i g(\boldsymbol{w}_i^T\boldsymbol{x} + b_i)$ as an approximate realization of the function f^*, that is $|f(\boldsymbol{x}) - f^*(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in I_p$.*

In order to predict the outcome Y_i from $\boldsymbol{X}_i$, we assume there exists a function $f^* : \mathbb{R}^p \to \mathbb{R}$, such that $Y_i = f^*(\boldsymbol{X}_i)$. We aim to use a feedforward neural network with one hidden layer and a sigmoid activation function $g(z) = \frac{e^z}{e^z+1}$ to estimate the function f^* by the function f, specified as:

$$f(\boldsymbol{x}) = \frac{exp\{\boldsymbol{w}^T\boldsymbol{x} + b\}}{1 + exp\{\boldsymbol{w}^T\boldsymbol{x} + b\}},$$

where $\boldsymbol{w} \in \mathbb{R}^p, b \in \mathbb{R}$ are parameters to be estimated. Note that the assumptions of the universal approximation theorem are satisfied, so it should be possible to approximate the function f^* with function f.

Choice of the loss function depends on the nature of the variable Y_i.

For the continuous variable Y_i, we choose usually the MSE loss function, which leads to $\boldsymbol{w}, b$ that minimize

$$L_{MSE}(\boldsymbol{w}, b) = \sum_{i=1}^{n} \left(\frac{exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}}{1 + exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}} - Y_i \right)^2.$$

For the categorical variable Y_i, we choose typically the cross-entropy function. For the special case of two categories $K = 2$, the cross-entropy loss function is equivalent to the log-likelihood of an alternative distribution:

$$\begin{aligned} L_{CE}(\boldsymbol{w}, b) &= -\sum_{i=1}^{n} \left[Y_i \log \frac{exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}}{1 + exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}} + (1 - Y_i) \log \frac{1}{1 + exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}} \right] = \\ &= -\sum_{i=1}^{n} \left[Y_i \log \pi_i + (1 - Y_i) \log(1 - \pi_i) \right] = -\sum_{i=1}^{n} \left[Y_i \log \frac{\pi_i}{1 - \pi_i} - \log \frac{1}{1 - \pi_i} \right], \end{aligned}$$

where $\pi_i = \frac{exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}}{1 + exp\{\boldsymbol{w}^T \boldsymbol{X}_i + b\}}$.

This leads to the same optimization problem for the parameters as in logistic regression:

$$L_{CE}(\boldsymbol{w}, b) = -l(\boldsymbol{\beta}|\boldsymbol{Y}) = -\sum_{i=1}^{n} \left[Y_i \log \frac{\pi_i}{1 - \pi_i} - \log \frac{1}{1 - \pi_i} \right].$$

The major difference is, that under the assumptions of generalized linear model, we have asymptotic results about estimated parameter

$$\hat{\boldsymbol{\beta}} = argmax\ \mathrm{l}(\boldsymbol{\beta}|\boldsymbol{Y}),$$

thus we can perform statistical inference etc. In case of feedforward neural network, the only thing we know from universal approximation theorem is that function f approximates f^*. Nevertheless, there is no guarantee that such function $Y_i = f^*(\boldsymbol{X}_i)$ really exists. Again, we would need to split dataset into training and evaluating part to see the actual performance.

2.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks for processing sequential data, such as sequences of values $\boldsymbol{X}_i^{(1)}, ..., \boldsymbol{X}_i^{(T)}, i = 1, \ldots, n, T > 0$. Recurrent networks can deal with much longer sequences than would be practical for networks without sequence-based specialization. The idea behind recurrent neural networks is to include cycles, which represent the influence of the current state of a variable on its own future state. Therefore, the state of a hidden unit $\boldsymbol{h}$ at time t can be expressed as

$$\boldsymbol{h}^{(t)} = g^{(t)}(\boldsymbol{X}^{(t)}, \boldsymbol{X}^{(t-1)}, \ldots, \boldsymbol{X}^{(1)}) = f(\boldsymbol{h}^{(t-1)}; \boldsymbol{X}^{(t)}, \boldsymbol{w}).$$

All previous steps are taken into account.

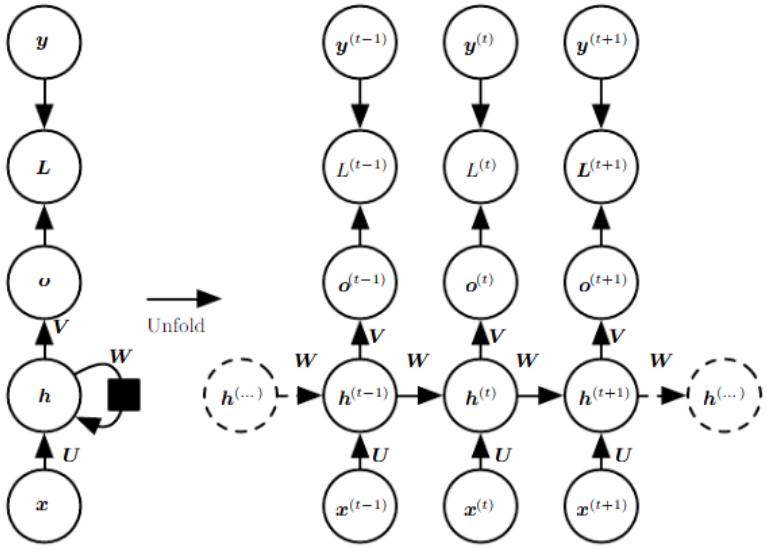


Figure 2.1: Unfolded recurrent neural network [Goodfellow et al., 2016]

Recurrent network with one hidden layer that maps an input sequence to an output sequence of the same length on figure 2.1 can be specified as follows:

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \mathbb{W}\boldsymbol{h}^{(t-1)} + \mathbb{U}\boldsymbol{X}^{(t)}, \\
\boldsymbol{h}^{(t)} &= tanh(\boldsymbol{a}^{(t)}), \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \mathbb{V}\boldsymbol{h}^{(t)}, \\
\hat{\boldsymbol{Y}}^{(t)} &= softmax(\boldsymbol{o}^{(t)}),
\end{aligned}$$

where the parameters are the bias vectors $\boldsymbol{b}$, $\boldsymbol{c}$ and the weight matrices $\mathbb{U}$, $\mathbb{V}$ and $\mathbb{W}$, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. The total loss for a given sequence of $\boldsymbol{X}^{(t)}, t = 1, \ldots, T$ values paired with a sequence of $\boldsymbol{Y}^{(t)}, t = 1, \ldots, T$ values would be the sum of the losses over all the time steps:

$$L(\boldsymbol{X}^{(1)}, \ldots, \boldsymbol{X}^{(T)}, \boldsymbol{Y}^{(1)}, \ldots, \boldsymbol{Y}^{(T)}) = \sum_{t=1}^{T} L(\boldsymbol{X}^{(t)}, \boldsymbol{Y}^{(t)}).$$

2.2.1 Long-Term Dependencies and LSTM Cells

Recurrent neural networks aim to capture all previous information to know the right context in the right time for predicting future values. When trying to capture such long term dependencies, one needs recurrent networks with many hidden layers. The main issue with deep recurrent networks is that gradients propagated through many stages tend to either vanish (often) or explode (rarely).

Let us consider a very simple recurrent neural network defined as

$$\boldsymbol{h}^{(t)} = \mathbb{W}\boldsymbol{h}^{(t-1)} = \mathbb{W}^t\boldsymbol{h}^0$$

without any activation function and without inputs. We can apply an eigendecomposition on $\mathbb{W}$ of the form $\mathbb{W} = \mathbb{Q}\boldsymbol{\Lambda}\mathbb{Q}^T$, with $\mathbb{Q}$ orthogonal, $\boldsymbol{\Lambda}$ diagonal and

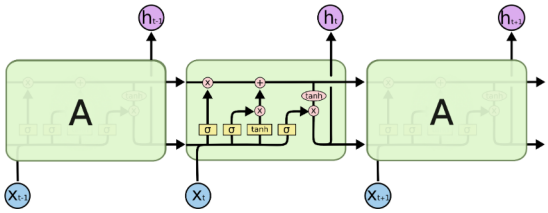


Figure 2.2: Overview of LSTM cell [Olah, 2015]

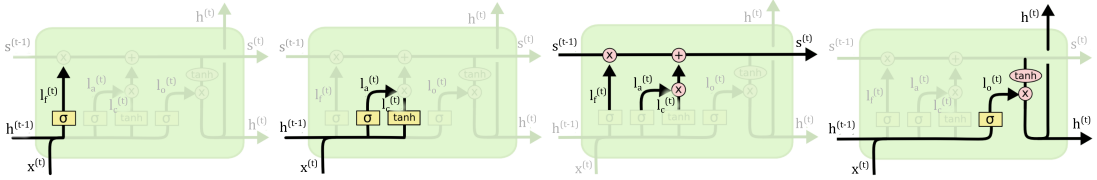


Figure 2.3: Step 1 Figure 2.4: Step 2 Figure 2.5: Step 3 Figure 2.6: Step 4

we can write $\boldsymbol{h}^{(t)} = \mathbb{Q}\boldsymbol{\Lambda}^t\mathbb{Q}^T\boldsymbol{h}^0$. Eigenvalues are thus raised to the power of t. That causes eigenvalues with magnitude greater then one to explode and eigenvalues with magnitude less then one go to zero for large values of t. This problem particular to recurrent networks was first discovered by Bengio, Frasconi, and Simard [1993].

The best currently used approach to overcome this issue is to use special hidden units called gated RNNs. These include **long short-term memory** (LSTM) and **gated recurent units** (GRU). Nayebi and Vitelli [2015] from Stanford compared their performance and their results indicate that outputs of the LSTM network are more musically plausible. As we use LSTM cell in our artificial music composition, we provide a detailed look.

LSTM cells were firstly introduced in 1997 by Hochreiter and Schmidhuber [1997]. The core idea was to add the vector $\boldsymbol{s}^{(t)}, t = 1, \ldots, T$, calles a cell state, which acts as long-term memory. Overview of a LSTM cell can be seen in figure 2.2.

In the first step (figure 2.3), LSTM chooses which information from the previous cell state $\boldsymbol{s}^{(t-1)}$ will be kept and which will be thrown away. The input is $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{X}^{(t)}$, the output is number between 0 and 1 for each number in the cell state $\boldsymbol{s}^{(t-1)}$, where 1 represents information worth keeping and 0 stands for useless information. This sigmoid layer $\boldsymbol{l}_f$ is called the **forget gate layer**:

$$\boldsymbol{l}_f^{(t)} = \boldsymbol{l}_f^{(t)}(\mathbb{W}_f, \mathbb{U}_f, \boldsymbol{b}_f) = \sigma(\mathbb{W}_f\boldsymbol{h}^{(t-1)} + \mathbb{U}_f\boldsymbol{X}^{(t)} + \boldsymbol{b}_f),$$

where $\mathbb{W}_f, \mathbb{U}_f$ and $\boldsymbol{b}_f$ are parameters to be learned.

In the second step (figure 2.4), LSTM decides in two parts what new information will be added to the cell state. Firstly, a sigmoid layer $\boldsymbol{l}_c$ called the **input gate layer** chooses which values will be updated. Secondly, a tanh layer $\boldsymbol{l}_a$ creates vectors of potentially new values $\boldsymbol{l}_a^{(t)}$, that could be added to the cell state:

$$\boldsymbol{l}_c^{(t)} = \boldsymbol{l}_c^{(t)}(\mathbb{W}_c, \mathbb{U}_c, \boldsymbol{b}_c) = \sigma(\mathbb{W}_c\boldsymbol{h}^{(t-1)} + \mathbb{U}_c\boldsymbol{X}^{(t)} + \boldsymbol{b}_c),$$
$$\boldsymbol{l}_a^{(t)} = \boldsymbol{l}_a^{(t)}(\mathbb{W}_a, \mathbb{U}_a, \boldsymbol{b}_a) = tanh(\mathbb{W}_a\boldsymbol{h}^{(t-1)} + \mathbb{U}_a\boldsymbol{X}^{(t)} + \boldsymbol{b}_a),$$

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \in (-1, 1),$$

where $\mathbb{W}_c, \mathbb{U}_c, \boldsymbol{b}_c, \mathbb{W}_a, \mathbb{U}_a$ and $\boldsymbol{b}_a$ are parameters to be learned.

In the third step (figure 2.5), we update the cell state $\boldsymbol{s}^{(t-1)}$ into the new cell state $\boldsymbol{s}^{(t)}$ using values from previous steps:

$$\boldsymbol{s}^{(t)} = \boldsymbol{l}_f^{(t)} * \boldsymbol{s}^{(t-1)} + \boldsymbol{l}_c^{(t)} * \boldsymbol{l}_a^{(t-1)},$$

where * stands for bit-wise multiplication operator.

In the last step (figure 2.6), LSTM produces the output $\boldsymbol{h}^{(t)}$. Firstly, we apply hyperbolic tangens to the cell state $\boldsymbol{s}^{(t)}$ and then we multiply it by the output of sigmoid gate to filter it. Last step is called the **output gate layer**:

$$\boldsymbol{l}_o^{(t)} = \boldsymbol{l}_o^{(t)}(\mathbb{W}_o, \mathbb{U}_o, \boldsymbol{b}_o) = \sigma(\mathbb{W}_o \boldsymbol{h}^{(t-1)} + \mathbb{U}_o \boldsymbol{X}^{(t)} + \boldsymbol{b}_o),$$

where $\mathbb{W}_o, \mathbb{U}_o$ and $\boldsymbol{b}_o$ are parameters to be learned.

Finally, the output $\boldsymbol{h}^{(t)}$ is produced as a combination of results from output layer $\boldsymbol{l}_o^{(t)}$ and new cell state $\boldsymbol{s}^{(t)}$:

$$\boldsymbol{h}^{(t)} = \boldsymbol{l}_o^{(t)} * tanh(\boldsymbol{s}^{(t)}).$$

2.3 Backpropagation

In this section, we describe the specific architecture of a recurrent neural network used for an artificial music composition. As a music is a sequence of tones, a recurrent neural network represents a perfect way to model it. We will use a neural network with one hidden layer with LSTM cells and the cross-entropy loss function with the combination of the softmax transformation, because future note prediction is a classification problem with K categories.

Recall that LSTM cells are specified by the following functions:

$$\boldsymbol{l}_f^{(t)} = \boldsymbol{l}_f^{(t)}(\mathbb{W}_f, \mathbb{U}_f, \boldsymbol{b}_f) = \sigma(\mathbb{W}_f \boldsymbol{h}^{(t-1)} + \mathbb{U}_f \boldsymbol{X}^{(t)} + \boldsymbol{b}_f), \tag{2.2}$$

$$\boldsymbol{l}_c^{(t)} = \boldsymbol{l}_c^{(t)}(\mathbb{W}_c, \mathbb{U}_c, \boldsymbol{b}_c) = \sigma(\mathbb{W}_c \boldsymbol{h}^{(t-1)} + \mathbb{U}_c \boldsymbol{X}^{(t)} + \boldsymbol{b}_c), \tag{2.3}$$

$$\boldsymbol{l}_a^{(t)} = \boldsymbol{l}_a^{(t)}(\mathbb{W}_a, \mathbb{U}_a, \boldsymbol{b}_a) = tanh(\mathbb{W}_a \boldsymbol{h}^{(t-1)} + \mathbb{U}_a \boldsymbol{X}^{(t)} + \boldsymbol{b}_a), \tag{2.4}$$

$$\boldsymbol{l}_o^{(t)} = \boldsymbol{l}_o^{(t)}(\mathbb{W}_o, \mathbb{U}_o, \boldsymbol{b}_o) = \sigma(\mathbb{W}_o \boldsymbol{h}^{(t-1)} + \mathbb{U}_o \boldsymbol{X}^{(t)} + \boldsymbol{b}_o), \tag{2.5}$$

$$\boldsymbol{s}^{(t)} = \boldsymbol{l}_f^{(t)} * \boldsymbol{s}^{(t-1)} + \boldsymbol{l}_c^{(t)} * \boldsymbol{l}_a^{(t)}, \tag{2.6}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{l}_o^{(t)} * tanh(\boldsymbol{s}^{(t)}), \tag{2.7}$$

where $t = 1, \ldots, T$. Before applying the cross-entropy loss function, we need to transform the output vector $\boldsymbol{h}^{(t)} \in \mathbb{R}^K$ into the vector of probabilities. To achieve that, we use the softmax function:

$$p_k^{(t)} = softmax(\boldsymbol{h}^{(t)})_k = \frac{e^{h_k^{(t)}}}{\sum_{j=1}^{K} e^{h_j^{(t)}}}. \tag{2.8}$$

Now, we can compare the output probability vector $\boldsymbol{p}^{(t)} = (p_1^{(t)}, \ldots, p_K^{(t)}) \in \mathbb{R}^K$ with the real value $\boldsymbol{Y}^{(t)} \in \mathbb{R}^K$ in each time event t using the cross-entropy loss function:

$$L_{CE}^{(t)}(\boldsymbol{p}^{(t)}, \boldsymbol{Y}^{(t)}) = -\sum_{k=1}^{K} Y_k^{(t)} \log p_k^{(t)}. \tag{2.9}$$

To find the optimal values of the parameter matrices $\mathbb{W}_f, \mathbb{W}_c, \mathbb{W}_a, \mathbb{W}_o, \mathbb{U}_f$, $\mathbb{U}_c$, $\mathbb{U}_a$, $\mathbb{U}_o \in \mathbb{R}^{K\times K}$, and the parameter vectors $\boldsymbol{b}_f$, $\boldsymbol{b}_c, \boldsymbol{b}_a, \boldsymbol{b}_o \in \mathbb{R}^K$, we need to minimize the loss function $L_{CE}^{(t)}$ summed over all the time events t:

$$L_{total} = \sum_{t=1}^{T} L_{CE}^{(t)}. \tag{2.10}$$

As we will use the gradient descent algorithm, we need to calculate the gradients of the loss function L_{total} with respect to all parameters. The loss function L_{total} is a compound function, thus we have to use a chain rule.

At first, we calculate the partial derivatives of $p_k^{(t)}$ defined in (2.8) with respect to the j-th component of the output vector $\boldsymbol{h}^{(t)}$ at the time t. We will distinguish two cases $k = j$ and $k \neq j$:

$$k = j : \frac{\partial p_k^{(t)}}{\partial h_j^{(t)}} = \frac{\partial \frac{e^{h_k^{(t)}}}{\sum_{j=1}^{K} e^{h_j^{(t)}}}}{\partial h_j^{(t)}} = \frac{e^{h_k^{(t)}} \sum_{j=1}^{K} e^{h_j^{(t)}} - e^{h_k^{(t)}} e^{h_j^{(t)}}}{(\sum_{j=1}^{K} e^{h_j^{(t)}})^2} = p_j^{(t)}(1 - p_j^{(t)}), \tag{2.11}$$

$$k \neq j : \frac{\partial p_k^{(t)}}{\partial h_j^{(t)}} = \frac{\partial \frac{e^{h_j^{(t)}}}{\sum_{j=1}^{K} e^{h_j^{(t)}}}}{\partial h_j^{(t)}} = -p_k^{(t)} p_j^{(t)}. \tag{2.12}$$

Afterwards, we calculate derivatives of the cross-entropy loss function $L_{CE}^{(t)}$ with respect to the $h_j^{(t)}$:

$$\begin{aligned} \frac{\partial L_{CE}^{(t)}}{\partial h_j^{(t)}} &= -\sum_{k=1}^{K} y_k^{(t)} \frac{\partial log p_k^{(t)}}{\partial h_j^{(t)}} = -\sum_{k=1}^{K} y_k^{(t)} \frac{1}{p_k^{(t)}} \frac{\partial p_k^{(t)}}{\partial h_j^{(t)}} = \\ &= -y_k(1 - p_k) - \sum_{k\neq j} y_k^{(t)} \frac{1}{p_k^{(t)}} (-p_k^{(t)} p_j^{(t)}) = \\ &= -y_j^{(t)}(1 - p_j^{(t)}) + \sum_{k\neq j} y_k^{(t)} p_j^{(t)} = p_j^{(t)} \sum_{k=1}^{K} y_k^{(t)} - y_j^{(t)} = p_j^{(t)} - y_j^{(t)}. \end{aligned} \tag{2.13}$$

When calculating the derivatives of $L_{CE}^{(t)}$ with respect to all parameters $\mathbb{W}_f$, $\mathbb{W}_c$, $\mathbb{W}_a$, $\mathbb{W}_o$, $\mathbb{U}_f, \mathbb{U}_c, \mathbb{U}_a, \mathbb{U}_o, \boldsymbol{b}_f$, $\boldsymbol{b}_c, \boldsymbol{b}_a, \boldsymbol{b}_o$ one needs to take into account all previous time events, because all parameters are involved in compound functions from the very beginning at the time $t = 1$. To keep everything simple at first, we start at the time $t = 1$.

Now we need to calculate the derivatives of $L_{CE}^{(1)}$ with respect to all parameters $\mathbb{W}_f, \mathbb{W}_c, \mathbb{W}_a, \mathbb{W}_o, \mathbb{U}_f, \mathbb{U}_c, \mathbb{U}_a, \mathbb{U}_o, \boldsymbol{b}_f$, $\boldsymbol{b}_c, \boldsymbol{b}_a, \boldsymbol{b}_o$. At first, we calculate the derivatives

of $L_{CE}^{(1)}$ with respect to the parameters from the output-gate layer $\mathbb{W}_f$, $\mathbb{U}_f$ and $\boldsymbol{b}_f$. As the loss function $L_{CE}^{(1)}$ is a compound function, we have to use the chain rule:

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{W}_f} = \frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} * \frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} * \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f}. \tag{2.14}$$

To calculate $\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} \in \mathbb{R}^K$ we use 2.13. To express the derivatives with respect to the vector $\boldsymbol{h}^{(1)} \in \mathbb{R}^K$, we can calculate the derivatives component-wise as $\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} = \left(\frac{\partial L_{CE}^{(1)}}{\partial h_1^{(1)}}, \ldots, \frac{\partial L_{CE}^{(1)}}{\partial h_K^{(1)}}\right)$.

$$\left[\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}\right]_j = \left[\frac{\partial L_{CE}^{(1)}}{\partial h_j^{(1)}}\right] = p_j^{(1)} - y_j^{(1)} \implies \frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} = (\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)}). \tag{2.15}$$

Next, we continue with $\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} \in \mathbb{R}^K$:

$$\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} = \frac{\partial(\boldsymbol{l}_o^{(1)} * tanh(\boldsymbol{s}^{(1)}))}{\partial \boldsymbol{s}^{(1)}} = \boldsymbol{l}_o^{(1)} * (\boldsymbol{1} - tanh^2(\boldsymbol{s}^{(1)}), \tag{2.16}$$

as the derivatives of $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ are

$$\begin{aligned}\frac{\mathrm{d}tanh(x)}{\mathrm{d}x} &= \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)' = \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} = \\ &= 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 = 1 - tanh^2(x).\end{aligned}$$

Then, we calculate $\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} \in \mathbb{R}^K$:

$$\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} = \frac{\partial \boldsymbol{l}_f^{(1)} * \boldsymbol{s}^{(0)} + \boldsymbol{l}_c^{(1)} * \boldsymbol{l}_a^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} = \boldsymbol{s}^{(0)}. \tag{2.17}$$

Finally, we calculate $\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f} \in \mathbb{R}^{K \times K}$. To express the partial derivatives of $L_{CE}^{(1)}$ with respect to matrix $\mathbb{W}_f \in \mathbb{R}^{K \times K}$, we can again calculate derivatives component-wise as $\left[\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f}\right]_{ij} = \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial w_{f_{ij}}}, i, j = 1, \ldots, K$ where $w_{f_{ij}}$ is $(i,j)th$ element of $\mathbb{W}_f$, which can be written with use of the outer product $\otimes$.

$$\begin{aligned}\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f} &= \frac{\partial \sigma(\mathbb{W}_f \boldsymbol{h}^{(t-1)} + \mathbb{U}_f \boldsymbol{X}^{(t)} + \boldsymbol{b}_f)}{\partial \mathbb{W}_f} = \\ &= \sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f) * (\boldsymbol{1} - \sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f)) \otimes \boldsymbol{h}^{(0)},\end{aligned}$$

as the derivatives of $\sigma(x) = \frac{e^x}{1+e^x}$ are

$$\frac{\mathrm{d}\sigma(x)}{\mathrm{d}x} = \left(\frac{e^x}{1+e^x}\right)' = \frac{e^x(1+e^x) - e^x e^x}{(1+e^x)(1+e^{x)}} = \frac{e^x}{(1+e^x)}\frac{1}{(1+e^x)} = \sigma(x)(1-\sigma(x)).$$

Altogether, we get

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{W}_f} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{s}^{(0)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}}}$$
$$* \underbrace{\sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f) * (\mathbf{1} - \sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f)) \otimes \boldsymbol{h}^{(0)}}_{\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f}}.$$

Similarly, we can calculate derivatives of $L_{CE}^{(1)}$ with respect to the parameters $\mathbb{U}_f$ and $\boldsymbol{b}_f$.

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{U}_f} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{s}^{(0)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}}}$$
$$* \underbrace{\sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f) * (\mathbf{1} - \sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f)) \otimes \boldsymbol{X}^{(1)}}_{\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{U}_f}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{b}_f} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{s}^{(0)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}}}$$
$$* \underbrace{\sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f) * (\mathbf{1} - \sigma(\mathbb{W}_f \boldsymbol{h}^{(0)} + \mathbb{U}_f \boldsymbol{X}^{(1)} + \boldsymbol{b}_f))}_{\frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \boldsymbol{b}_f}}.$$

Analogously, we calculate the derivatives of $L_{CE}^{(1)}$ with respect to the parameters from the input gate layer $\mathbb{W}_c$, $\mathbb{U}_c$ and $\boldsymbol{b}_c$.

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{W}_c} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_a^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_c^{(1)}}}$$
$$* \underbrace{\sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c) * (1 - \sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c)) \otimes \boldsymbol{h}^{(0)}}_{\frac{\partial \boldsymbol{l}_c^{(1)}}{\partial \mathbb{W}_c}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{U}_c} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_a^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_c^{(1)}}} * \underbrace{\sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c) * (\mathbf{1} - \sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c)) \otimes \boldsymbol{X}^{(1)}}_{\frac{\partial \boldsymbol{l}_c^{(1)}}{\partial \mathbb{U}_c}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{b}_c} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_a^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_c^{(1)}}} * \underbrace{\sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c) * (\mathbf{1} - \sigma(\mathbb{W}_c \boldsymbol{h}^{(0)} + \mathbb{U}_c \boldsymbol{X}^{(1)} + \boldsymbol{b}_c))}_{\frac{\partial \boldsymbol{l}_c^{(1)}}{\partial \boldsymbol{b}_c}}.$$

Then, we continue with the derivatives of $L_{CE}^{(1)}$ with respect to the parameters $\mathbb{W}_a$, $\mathbb{U}_a$ and $\boldsymbol{b}_a$.

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{W}_a} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_c^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_a^{(1)}}} * \underbrace{(\mathbf{1} - tanh^2(\mathbb{W}_a \boldsymbol{h}^{(0)} + \mathbb{U}_a \boldsymbol{X}^{(1)} + \boldsymbol{b}_a)) \otimes \boldsymbol{h}^{(0)}}_{\frac{\partial \boldsymbol{l}_a^{(1)}}{\partial \mathbb{W}_a}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{U}_a} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_c^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_a^{(1)}}} * \underbrace{(\mathbf{1} - tanh^2(\mathbb{W}_a \boldsymbol{h}^{(0)} + \mathbb{U}_a \boldsymbol{X}^{(1)} + \boldsymbol{b}_a)) \otimes \boldsymbol{X}^{(1)}}_{\frac{\partial \boldsymbol{l}_a^{(1)}}{\partial \mathbb{U}_a}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{b}_a} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{\boldsymbol{l}_o^{(1)} (\mathbf{1} - tanh^2(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}}} * \underbrace{\boldsymbol{l}_c^{(1)}}_{\frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_a^{(1)}}} * \underbrace{(\mathbf{1} - tanh^2(\mathbb{W}_a \boldsymbol{h}^{(0)} + \mathbb{U}_a \boldsymbol{X}^{(1)} + \boldsymbol{b}_a))}_{\frac{\partial \boldsymbol{l}_a^{(1)}}{\partial \boldsymbol{b}_a}}.$$

Finally, we calculate the derivatives of $L_{CE}^{(1)}$ with respect to the parameters from the output-gate layer $\mathbb{W}_o$, $\mathbb{U}_o$ and $\boldsymbol{b}_o$.

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{W}_o} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{tanh(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{l}_o^{(1)}}} * \underbrace{\sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o) * (\boldsymbol{1} - \sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o)) \otimes \boldsymbol{h}^{(0)}}_{\frac{\partial \boldsymbol{l}_o^{(1)}}{\partial \mathbb{W}_o}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \mathbb{U}_o} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{tanh(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{l}_o^{(1)}}} * \underbrace{\sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o) * (\boldsymbol{1} - \sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o)) \otimes \boldsymbol{X}^{(1)}}_{\frac{\partial \boldsymbol{l}_o^{(1)}}{\partial \mathbb{U}_o}}.$$

$$\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{b}_o} = \underbrace{(\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)})}_{\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}}} * \underbrace{tanh(\boldsymbol{s}^{(1)})}_{\frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{l}_o^{(1)}}} * \underbrace{\sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o) * (\boldsymbol{1} - \sigma(\mathbb{W}_o \boldsymbol{h}^{(0)} + \mathbb{U}_o \boldsymbol{X}^{(1)} + \boldsymbol{b}_o))}_{\frac{\partial \boldsymbol{l}_o^{(1)}}{\partial \boldsymbol{b}_o}}.$$

Having calculated all that, we know the derivatives of $L_{CE}^{(1)}$ with respect to all parameters $\mathbb{W}_f, \mathbb{W}_c, \mathbb{W}_a, \mathbb{W}_o, \mathbb{U}_f, \mathbb{U}_c, \mathbb{U}_a, \mathbb{U}_o, \boldsymbol{b}_f, \boldsymbol{b}_c, \boldsymbol{b}_a, \boldsymbol{b}_o$. If the final time T was $T = 1$, we could update all parameters, calculate new values of $L_{CE}^{(1)}$ and continue until convergence.

For the final time T, $T > 1$, we have to continue with calculating the derivatives of $L_{CE}^{(2)}$. Again, we start with the derivatives with respect to $\mathbb{W}_f$. This time, the function $L_{CE}^{(2)}$ contains the parameter $\mathbb{W}_f$ through $\boldsymbol{l}_f^{(2)}$ (2.18), but also through $\boldsymbol{l}_f^{(1)}$, which is included in $\boldsymbol{h}^{(1)}$, which is included in four functions $\boldsymbol{l}_f^{(2)}$ (2.19), $\boldsymbol{l}_c^{(2)}$ (2.20), $\boldsymbol{l}_a^{(2)}$ (2.21) and $\boldsymbol{l}_o^{(2)}$ (2.22), thus, we need to apply the chain rule for multiple variable functions to express the partial derivatives of $L_{CE}^{(2)}$ with respect to $\mathbb{W}_f$:

$$\frac{\partial L_{CE}^{(2)}}{\partial \mathbb{W}_f} = \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_f^{(2)}} * \frac{\partial \boldsymbol{l}_f^{(2)}}{\partial \mathbb{W}_f} + \tag{2.18}$$

$$+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_f^{(2)}} * \frac{\partial \boldsymbol{l}_f^{(2)}}{\partial \boldsymbol{h}^{(1)}} * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} * \frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} * \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f} + \tag{2.19}$$

$$+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_c^{(2)}} * \frac{\partial \boldsymbol{l}_c^{(2)}}{\partial \boldsymbol{h}^{(1)}} * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} * \frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} * \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f} + \tag{2.20}$$

$$+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_a^{(2)}} * \frac{\partial \boldsymbol{l}_a^{(2)}}{\partial \boldsymbol{h}^{(1)}} * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} * \frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} * \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f} + \tag{2.21}$$

$$+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{l}_o^{(2)}} * \frac{\partial \boldsymbol{l}_o^{(2)}}{\partial \boldsymbol{h}^{(1)}} * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} * \frac{\partial \boldsymbol{s}^{(1)}}{\partial \boldsymbol{l}_f^{(1)}} * \frac{\partial \boldsymbol{l}_f^{(1)}}{\partial \mathbb{W}_f}. \tag{2.22}$$

Analogously, we can calculate the partial derivatives $L_{CE}^{(2)}$ with respect to all the other parameters. For the final time T, $T > 2$, the derivatives of $L_{CE}^{(3)}$ would consist of even more parts. Even though many terms keep repeating, it is quite complicated and a bit confusing to continue this way.

Fortunately, there is a more convenient and more elegant way of calculating desired updates of the parameteres. Instead of starting at the time $t = 1$, we begin at the final time $t = T$ and go backwards in time. At first, we will demonstrate the idea for the overall time $T = 2$ and generalize it later on.

At the final time $T = 2$, we take into account only information from the last layer. Thus we calculate the partial derivatives only with respect to the final layer, as there were no other previous layers at all. As we will chain a lot, we write the partial derivatives in a suitable form to keep expressions more clear:

$$\delta \boldsymbol{h}^{(2)} = \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} = (\boldsymbol{p}^{(2)} - \boldsymbol{Y}^{(2)}),$$
$$\delta \boldsymbol{s}^{(2)} = \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{s}^{(2)}} = \delta \boldsymbol{h}^{(2)} * \boldsymbol{l}_o^{(2)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(2)})).$$

We will denote

$$\delta \boldsymbol{l}_f^{(2)} = \delta \boldsymbol{s}^{(2)} * \boldsymbol{s}^{(1)} * \boldsymbol{l}_f^{(2)} * (\mathbf{1} - \boldsymbol{l}_f^{(2)}),$$

even though it does not hold $\delta \boldsymbol{l}_f^{(2)} = \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{l}_f^{(2)}}$. Including the derivatives of the function $\sigma(x)$ into $\delta \boldsymbol{l}_f^{(2)}$ enables us to write the derivatives of $L_{CE}^{(2)}$ with respect to e.g. $\mathbb{W}_f$ in a form of $\delta \boldsymbol{l}_f^{(2)} \otimes \boldsymbol{h}^{(1)}$ etc.

Analogously, we denote

$$\delta \boldsymbol{l}_c^{(2)} = \delta \boldsymbol{s}^{(2)} * \boldsymbol{l}_a^{(2)} * \boldsymbol{l}_c^{(2)} * (\mathbf{1} - (\boldsymbol{l}_c^{(2)})),$$
$$\delta \boldsymbol{l}_a^{(2)} = \delta \boldsymbol{s}^{(2)} * \boldsymbol{l}_c^{(2)} * (\mathbf{1} - (\boldsymbol{l}_a^{(2)})^2),$$
$$\delta \boldsymbol{l}_o^{(2)} = \delta \boldsymbol{s}^{(2)} * tanh(\boldsymbol{s}^{(2)}) * \boldsymbol{l}_o^{(2)} * (\mathbf{1} - \boldsymbol{l}_o^{(2)}).$$

Now, we can go backwards to the time t = 1. When calculating the parameter updates at the time t = 1, we also need to take into account the time t = 2, as all parameters from layer 1 are also included in layer 2. Thus we have to calculate all derivatives with respect to the sum $L_{CE}^{(1)} + L_{CE}^{(2)}$.

We start with the derivatives of $L_{CE}^{(1)} + L_{CE}^{(2)}$ with respect to $\boldsymbol{h}^{(1)}$, because this term will be included in all other expressions.

$$\begin{aligned}
\delta\boldsymbol{h}^{(1)} &= \frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_f^{(2)}} * \frac{\partial \boldsymbol{l}_f^{(2)}}{\partial \boldsymbol{h}^{(1)}} + \\
&+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_c^{(2)}} * \frac{\partial \boldsymbol{l}_c^{(2)}}{\partial \boldsymbol{h}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{l}_a^{(2)}} * \frac{\partial \boldsymbol{l}_a^{(2)}}{\partial \boldsymbol{h}^{(1)}} + \\
&+ \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(2)}} * \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{l}_o^{(2)}} * \frac{\partial \boldsymbol{l}_o^{(2)}}{\partial \boldsymbol{h}^{(1)}} = \\
&= (\boldsymbol{p}^{(1)} - \boldsymbol{Y}^{(1)}) + \mathbb{W}_f \delta \boldsymbol{l}_f^{(2)} + \mathbb{W}_c \delta \boldsymbol{l}_c^{(2)} + \mathbb{W}_a \delta \boldsymbol{l}_a^{(2)} + \mathbb{W}_o \delta \boldsymbol{l}_o^{(2)}.
\end{aligned}$$

Then, we continue with the derivatives of $L_{CE}^{(1)} + L_{CE}^{(2)}$ with respect to $\boldsymbol{s}^{(1)}$.

$$\begin{aligned}
\delta\boldsymbol{s}^{(1)} &= \frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{s}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{s}^{(1)}} = \left(\frac{\partial L_{CE}^{(1)}}{\partial \boldsymbol{h}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \right) * \frac{\partial \boldsymbol{h}^{(1)}}{\partial \boldsymbol{s}^{(1)}} + \frac{\partial L_{CE}^{(2)}}{\partial \boldsymbol{s}^{(2)}} * \frac{\partial \boldsymbol{s}^{(2)}}{\partial \boldsymbol{s}^{(1)}} = \\
&= \delta\boldsymbol{h}^{(1)} * \boldsymbol{l}_o^{(1)} * (1 - tanh^2(\boldsymbol{s}^{(1)})) + \delta\boldsymbol{s}^{(2)} * \boldsymbol{l}_f^{(2)}.
\end{aligned}$$

Now, we directly obtain

$$\begin{aligned}
\delta\boldsymbol{l}_f^{(1)} &= \delta\boldsymbol{s}^{(1)} * \boldsymbol{s}^{(1)} * \boldsymbol{l}_f^{(1)} * (\mathbf{1} - \boldsymbol{l}_f^{(1)}), \\
\delta\boldsymbol{l}_c^{(1)} &= \delta\boldsymbol{s}^{(1)} * \boldsymbol{l}_a^{(1)} * \boldsymbol{l}_c^{(1)} * (\mathbf{1} - (\boldsymbol{l}_c^{(1)})) \\
\delta\boldsymbol{l}_a^{(1)} &= \delta\boldsymbol{s}^{(1)} * \boldsymbol{l}_c^{(1)} * (\mathbf{1} - (\boldsymbol{l}_a^{(1)})^2) \\
\delta\boldsymbol{l}_o^{(1)} &= \delta\boldsymbol{s}^{(1)} * tanh(\boldsymbol{s}^{(1)}) * \boldsymbol{l}_o^{(1)} * (\mathbf{1} - \boldsymbol{l}_o^{(1)}).
\end{aligned}$$

In the same way, we can start at the final time $T > 2$, chain the updates and go back in time from $t = T$ to $t = 1$. We need to calculate the derivatives with respect to the total loss in each time event t, let us denote:

$$L_{total}^{(t)} = \sum_{i=t}^{T} L_{CE}^{(i)}.$$

Then $L_{total}^{(t)} = L_{CE}^{(t)} + L_{total}^{(t+1)}$ and we can use above expressions for general t. For general $t, t = 1, \ldots, T$ we thus get the following relationships, which en-

ables us to update parameters after backpropagation is completed at time $t = 1$.

$$
\begin{aligned}
&t = T: \\
&\delta \boldsymbol{h}^{(t)} = \frac{\partial L_{total}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial L_{CE}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = (\boldsymbol{p}^{(t)} - \boldsymbol{Y}^{(t)}), \\
&\delta \boldsymbol{s}^{(t)} = \frac{\partial L_{total}^{(t)}}{\partial \boldsymbol{s}^{(t)}} = \frac{\partial L_{CE}^{(t)}}{\partial \boldsymbol{s}^{(t)}} = \delta \boldsymbol{h}^{(t)} * \boldsymbol{l}_o^{(t)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(t)})), \\
&t < T: \\
&\delta \boldsymbol{h}^{(t)} = \frac{\partial L_{total}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = (\boldsymbol{p}^{(t)} - \boldsymbol{Y}^{(t)}) + \mathbb{W}_f \delta \boldsymbol{l}_f^{(t+1)} + \mathbb{W}_c \delta \boldsymbol{l}_c^{(t+1)} + \mathbb{W}_a \delta \boldsymbol{l}_a^{(t+1)} + \mathbb{W}_o \delta \boldsymbol{l}_o^{(t+1)}, \\
&\delta \boldsymbol{s}^{(t)} = \frac{\partial L_{total}^{(t)}}{\partial \boldsymbol{s}^{(t)}} = \delta \boldsymbol{h}^{(t)} * \boldsymbol{l}_o^{(t)} * (\mathbf{1} - tanh^2(\boldsymbol{s}^{(t)})) + \delta \boldsymbol{s}^{(t+1)} * \boldsymbol{l}_f^{(t+1)}.
\end{aligned}
$$

The terms can be express as follows:

$$
\begin{aligned}
\delta \boldsymbol{l}_f^{(t)} &= \delta \boldsymbol{s}^{(t)} * \boldsymbol{s}^{(t-1)} * \boldsymbol{l}_f^{(t)} * (\mathbf{1} - \boldsymbol{l}_f^{(t)}), \\
\delta \boldsymbol{l}_c^{(t)} &= \delta \boldsymbol{s}^{(t)} * \boldsymbol{l}_a^{(t)} * \boldsymbol{l}_c^{(t)} * (\mathbf{1} - (\boldsymbol{l}_c^{(t)})), \\
\delta \boldsymbol{l}_a^{(t)} &= \delta \boldsymbol{s}^{(t)} * \boldsymbol{l}_c^{(t)} * (\mathbf{1} - (\boldsymbol{l}_a^{(t)})^2), \\
\delta \boldsymbol{l}_o^{(t)} &= \delta \boldsymbol{s}^{(t)} * tanh(\boldsymbol{s}^{(t)}) * \boldsymbol{l}_o^{(t)} * (\mathbf{1} - \boldsymbol{l}_o^{(t)}).
\end{aligned}
$$

This approach is suitable also for implementation purposes. Parameters are upgraded after each completed backpropagation process from $t = T$ to $t = 1$ in the following way. At first, we need to calculate overall change of all parameters:

$$
\Delta \mathbb{W}_f = \sum_{t=1}^{T} \delta \boldsymbol{l}_f^{(t)} \otimes \boldsymbol{h}^{(t-1)}, \Delta \mathbb{U}_f = \sum_{t=1}^{T} \delta \boldsymbol{l}_f^{(t)} \otimes \boldsymbol{X}^{(t)}, \Delta \boldsymbol{b}_f = \sum_{t=1}^{T} \delta \boldsymbol{l}_f^{(t)}.
$$

Analogously, we can calculate updates for other parameters as well. Then, we use calculated gradients to update the old values of the parameters with the known learning rate ϵ.

$$
\mathbb{W}_f^{new} = \mathbb{W}_f^{old} - \epsilon \Delta \mathbb{W}_f, \mathbb{U}_f^{new} = \mathbb{U}_f^{old} - \epsilon \Delta \mathbb{U}_f, \boldsymbol{b}_f^{new} = \boldsymbol{b}_f^{old} - \epsilon \Delta \boldsymbol{b}_f.
$$

This way, we keep iterating forward and back through the recurrent neural network until convergence.

2.4 Algorithmic Implementation

Implementation of machine learning is possible in both R and Python. When it comes to machine learning, there is a wide range of useful packages, we state the most famous ones.

In R, there is a package `CARAT` Kuhn [2009] which refers to classification and regression training. The parameters are optimized over intergation of several functions to calculate the overall performance of a given model by the grid search method, which finds the best combinations. Another popular machine learning package is `randomForest` Liaw and Wiener [2002], which implement random

forests for both classification and regression tasks, as well as for imputing missing values and detecting outliers. For deep learning, there is a package called `neuralnet` Fritsch et al. [2019] which allows custom-choice of loss functions and activation functions.

In Python, popular machine learning library is a `Sci-kit Learn` Pedregosa et al. [2011], which provides a range of supervised and unsupervised learning algorithms, but its deep learning functionality is quite limited. On the other hand `TensorFlow` Abadi et al. [2015] enables to build complex deep learning models from scratch instead of having a pre-defined off-the-shelf algorithms, but its syntax is quite complicated. Also, `TensorFlow` allows using GPUs for more efficient training. `Keras` Chollet [2015] is a high level API built on `TensorFlow`, which was introduced to make `TensorFlow` more user-friendly even for building very elaborate models.

2.5 Related Work

In 1957, Illiac Suite (also known as String Quartet No. 4), which is generally agreed to be the first algorithmic musical composition, was created. This famous example of the early efforts was composed by the ILLIAC I computer at the University of Illinois at Urbana-Champaign, which was programmed by composers and professors Lejaren Hiller and Leonard Isaacson. It consists of four movements, corresponding to four various ruled-based experiments (see [Sandred, Laurson, and Kuuskankare, 2009] for its modern reconstruction). In the late 1980s, a system named Experiments in Musical Intelligence(EMI) designed by David Cope [2000] extended handcrafted rules approach with the capacity to learn from a corpus of scores of a composer to create its own grammar and database of rules.

Through time, music generation moved from rule-based systems, generative grammars and Markov chains to using various types of neural networks. As early approaches often relied on specific rules created by musical experts, the results were quite limited.

Comprehensive survey and analysis of different ways of using deep artificial neural networks to generate musical content were conducted by Briot et al. [2018], alongside with different representations of music with suitable architectures followed by respective challenges. Fernandez and Vico [2014] provided a detailed description of all methods used for algorithmic composition such as grammars, knowledge-based systems, Markov chains, artifical neural networks, and evolutionary methods.

Generally, there are two main approaches when it comes to generating music. The first one is to synthesize a raw audio signal using a direct waveform approach, whereas the second is based on a symbolic discrete representation of tones. The advantage of the first approach is that it considers the raw material untransformed with its full resolution. The main drawback is in the computational loads, as it is demanding in terms of processing and memory. The latest example of this approach is represented by WaveNet [Van Den Oord et al., 2016], which performs amazing results when it comes to text-to-speech tasks. On the other hand, it seems that for music generation raw signals are too complex to be captured by current neural networks.

Nowadays, the second approach is more preferable for several reasons [Briot et al., 2018]:

- the grand majority of the current deep learning systems for music generation are symbolic
- artists compose music via symbolic representation and it might be reasonable to follow their compositional process
- it is easier to perform harmonic analysis etc. on the symbolic representation of music

In the past, few difficulties had to be overcome to make music generation by recurrent neural networks possible. Mozer [1994] attempted to compose music with RNNs and noticed their inability to capture global music structure and rhythm. Eck and Schmidhuber [2002] overcame this limitation, proposed using Long-Short Term Memory (LSTM) cells in RNN and created a system that was able to learn and compose pleasing blues music. An alternative to LSTM is using Gated Recurrent Unit (GRU). Nevertheless, Nayebi and Vitelli [2015] from Stanford compare their performance and their results indicate, that outputs of the LSTM network are more musically plausible.

Various compound architectures are often used to solve specific issues. Bidirectional RNNs were introduced by Schuster and K. Paliwal [1997] for cases when the prediction depends also on the next elements (not just the previous ones), as it is for example with speech recognition. RNN Encoder-Decoder [Cho et al., 2014] enables to encode a variable-length sequence produced by a recurrent network into another variable-length sequence produced by another recurrent network. Boulanger-Lewandowski, Bengio, and Vincent [2012] introduced an approach that combines a restricted Boltzmann machine with RNN to compose polyphonic music. This approach enables them to work with the fact, that the occurrence of particular note at the particular time changes the probability with which other notes may occur at the same time.

The field of music generation is rapidly improving. Recently, Mao, Shin, and Cottrell [2018] introduced DeepJ, a generative model with tunable parameters, which enables to control the style of generated music. Also, many sources are open to the public. Flow Composer [Papadopoulos, Roy, and Pachet, 2016] is a web application that helps generate musical lead sheets. Users can specify the style of the lead sheet by choosing a corpus of existing lead sheets. Chord sequences and melodies are based on Markov and Meter models. OpenMusic [Assayag et al., 1999] is a visual programming environment for music composition. It is based on programming language Common Lisp. Many specialized libraries are involved, which extend its functionality into areas like spectral music, minimalist music or sound synthesis.

Finally, Magenta [Waite, 2016] research project from Google Brain Team based on Python machine learning library TensorFlow [Géron, 2017], [Müller and Guido, 2016] offers various configuration for training and generating monophonic music as well as polyphonic music with dynamics.

3. Application: Artificial Music Composition

In this chapter, we train three recurrent neural networks with LSTM cells on the Beatles' songs and discuss qualities of artificial melodies. Firstly, we describe the basics of music theory and the structure of a musical format called MIDI, which is used as an input into deep learning models. Then, we describe three Magenta's configuration (Basic, Lookback, and Attention), which we use for artifical music composition. As all configurations are designed for monophonic melodies, we provide a script able to transfer polyphonic music into monophonic. In order to avoid overfitting, we split Beatles' songs into training and evaluating datasets and choose appropriate number of iterations, so that models perform similar results on both datasets. Finally, we generate new melodies from all three configurations and evaluate them both objectively using musically informed objective metrics [Yang and Lerch, 2018] and subjectively.

3.1 Music Representation

Music can be viewed as a sequence of tones organized in time. A written form of a tone is called a note. Four main attributes of tones are a pitch, a length, a timbre, and a velocity. The way our artificial music is composed, as well as conducted melody analysis, take into account only a pitch and a length.

We work with music in a MIDI (Musical Instrument Digital Interface) format, which was designed for sharing musical information between computer devices. Unlike audio file formats (MP3, WAV or FLAC), MIDI files do not contain actual raw audio data and are thus smaller in size.

A MIDI file is a stream of *event messages*, including a note's notation, a pitch, and a velocity. Each note begins with a *note-on* message and ends with a *note-off* message. In addition, each of those messages is specified with time (in internal units), a pitch on a scale $21 - 108$ representing a standard twelve-tone tuning from A0 to C8, a channel, and a volume. An example of a musical piece and its MIDI representation is shown in figure 3.1.

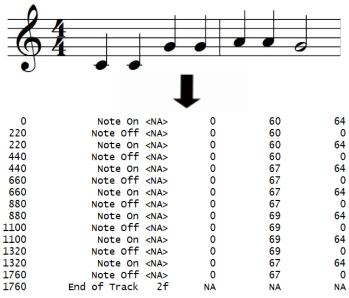
Metadata messages are often included and can specify a tempo, a key signature or a text information about the composer and the interpreter.

A considerable disadvantage of MIDI files is that the event representation is quite elementary and for example does not capture all aspects of a tone color. Nevertheless, they serve perfectly for our purposes.

There are many collections of MIDI files available online, such as Nottingham collection or Yamaha e-piano Competition dataset, which are widely used for machine learning tasks.

3.2 Magenta Algorithm

Magenta is a research project released by Google Brain Team. It is built on machine learning library `TensorFlow` in Python and provides scripts for training



0	Note On <NA>	0	60	64
220	Note Off <NA>	0	60	0
220	Note On <NA>	0	60	64
440	Note Off <NA>	0	60	0
440	Note On <NA>	0	67	64
660	Note Off <NA>	0	67	0
660	Note On <NA>	0	67	64
880	Note Off <NA>	0	67	0
880	Note On <NA>	0	69	64
1100	Note Off <NA>	0	69	0
1100	Note On <NA>	0	69	64
1320	Note Off <NA>	0	69	0
1320	Note On <NA>	0	67	64
1760	Note Off <NA>	0	67	0
1760	End of Track 2f	NA	NA	NA

Figure 3.1: Melody in traditional notation and its MIDI representation. The pitch number 60 stands for C4

deep learning models based on real-world music. In this section, we describe three Magenta's configurations (Basic, Attention, and Lookback), which we use for artificial music composition based on training samples from the Beatles. All configurations process MIDI files as an input and an output.

In all configurations, each bar is divided into 16 steps. According to configuration, each step is encoded as an input vector in a slightly different way using a MIDI representation of events. Those vectors are then used as the input to the recurrent neural network with LSTM cells. In each step, the target is to determine the next step, also encoded as a vector. This way, the neural network aims to learn patterns in what notes follow each other. All notes should be in a range between 48 and 83, which corresponds to C3 and B5, otherwise, they are transposed.

3.2.1 Basic RNN

The Basic configuration is the simplest configuration. Each step $\boldsymbol{X}^{(t)} \in \mathbb{R}^{38}$, $t = 1, \ldots, T$ is encoded as a one-hot vector of a length 38. The first position is an indicator of a *note-off* event, the second position is an indicator of no event (previous note continues playing or remaining silence). Positions 3 to 38 indicate a *note-on* event for each pitch in range 48 to 83.

3.2.2 Lookback RNN

The Lookback configuration aims to capture the long-term structure in music. Each step $\boldsymbol{X}^{(t)} \in \mathbb{R}^{121}, t = 1, \ldots, T$ is encoded as a vector of a length 121 consisting of values -1, 0 and 1. Positions from 1 to 38 correspond to a one-hot vector from the Basic configuration for the current step. Positions from 39 to 76 correspond to a one-hot vector from the Basic configuration for step in the previous bar on current position increased by one, positions from 77 to 114 correspond to a one-hot vector from the Basic configuration for the step two bars ago on the current position increased by one. That means that if the current step is the first one in a bar, the algorithm takes a look at the second step in the previous bar and the bar before that.

Positions from 115 to 119 capture the current position of a step within the measure (number between 1 and 16) coded as a binary step clock, but with values -1 instead of 0. Step 1 is than coded as $[-1, -1, -1, -1, 1]$, step 2 as $[-1, -1, -1, 1, -1]$, step 3 as $[-1, -1, -1, 1, 1]$ etc.

Finally, the position 120, resp. 121 indicates whether the current step one-hot vector from the Basic configuration is equal to the one-hot vector from the Basic configuration for the same position one, resp. two bars ago. For example, if the third bar of the melody is completely repeating the first bar, all vectors created for steps in the third bar would be equal to 1 on position 121.

This allows the model to more easily repeat 1 or 2 bar phrases and recognize new patterns from scratch.

3.2.3 Attention RNN

The Attention configuration aims to capture even longer long-term structure. Each step $\boldsymbol{X}^{(t)} \in \mathbb{R}^{74}, t = 1, \ldots, T$ is encoded as a 74 long vector consisting of values -1, 0, 1. Positions 1 to 38 again correspond to a one-hot vector from the Basic configuration for the current step. The position 39 indicates if there is a note playing (note-on event or note playing from previous steps). The position 40 indicates whether the melody is currently ascending (1) or descending (-1). Positions 41 and 42 correspond to lookback distances, as positions 120 and 121 in Lookback configuration. Positions 43 to 49 capture the current position of the step within a measure (number between 1 and 16) coded as a binary step clock, but with values -1 instead of 0, this time with 6 values. The step 1 is than coded as $[-1, -1, -1, -1, -1, 1]$, the step 2 as $[-1, -1, -1, -1, 1, -1]$, the step 3 as $[-1, -1, -1, -1, 1, 1]$ etc. The position 50 indicates if the next step is the start of a bar (and thus current step is the last one in a bar). Positions 51 to 62 consist of a one-hot vector representing 12 pitches classes, where 1 is on the position of the pitch class of the currently playing note. Positions 63 to 74 consist of a one-hot vector representing 12 pitches classes, where 1 is on the position of the pitch class of the last three different playing notes.

Furthermore, weighted previous outputs are also taken into account using the attention mask $\boldsymbol{a}^{(t)} \in \mathbb{R}^n$ [Bahdanau et al., 2014]. Specifically:

$$\boldsymbol{u}^{(t)} = \boldsymbol{v}^T tanh(\mathbb{W}_1 \boldsymbol{h}^{(t)} + \mathbb{W}_2 \boldsymbol{s}^{(t)}), \boldsymbol{a}^{(t)} = softmax(\boldsymbol{u}^{(t)}), \boldsymbol{h}^{(t)} = \sum_{i=t-40}^{t-1} a_i^{(t)} \boldsymbol{h}^{(i)},$$

where $\boldsymbol{v}, \mathbb{W}_1$ and $\mathbb{W}_2$ are learnable parameters of the model, $\boldsymbol{h}^{(i)}$ are the RNN outputs from previous 40 steps $(\boldsymbol{h}^{(t-40)}, \ldots, \boldsymbol{h}^{(t-1)})$ and $\boldsymbol{a}^{(t)}$ is called attention mask.

3.3 Artificial Music Composition and Evaluation

3.3.1 Data Preprocessing

All configurations are intended for composing monophonic music, therefore all chords had to be removed. We provide a script in Python, which we designed to transform polyphonic melodies into monophonic melodies. Based on general knowledge of music theory, we decided to remove lower notes from chords and keep the highest one, as it plays the most important part in melody specification.

Furthermore, when a new note starts playing, but a previous note has not stopped yet, we force it to end prematurely.

As a training dataset, we used the famous Beatles' songs. Our main goal was to generate interesting and likable melodies similar to the Beatles, so we extracted key melody parts from the chorus, the bridge and the verse of each song. Altogether, we obtained 124 pieces of melody.

3.3.2 Artificial Music Composition

The common issue with neural networks is overfitting. In order to avoid it, we split the set of examples into training and evaluating subsets with evaluation ratio 0.3 and run training and evaluating at the same time. Based on results (Figure 3.2), we chose the number of iterations as 9500 for Basic, 2140 for Lookback and 620 for Attention. Hyperparameters were set as follows: The batch size for stochastic gradient descent was set up to 16, learning error ϵ was set up to 0.001. A recurrent neural network with LSTM cells consists of two layers as suggested by creators of Magenta. A number of neurons in both layers was set up to 16.

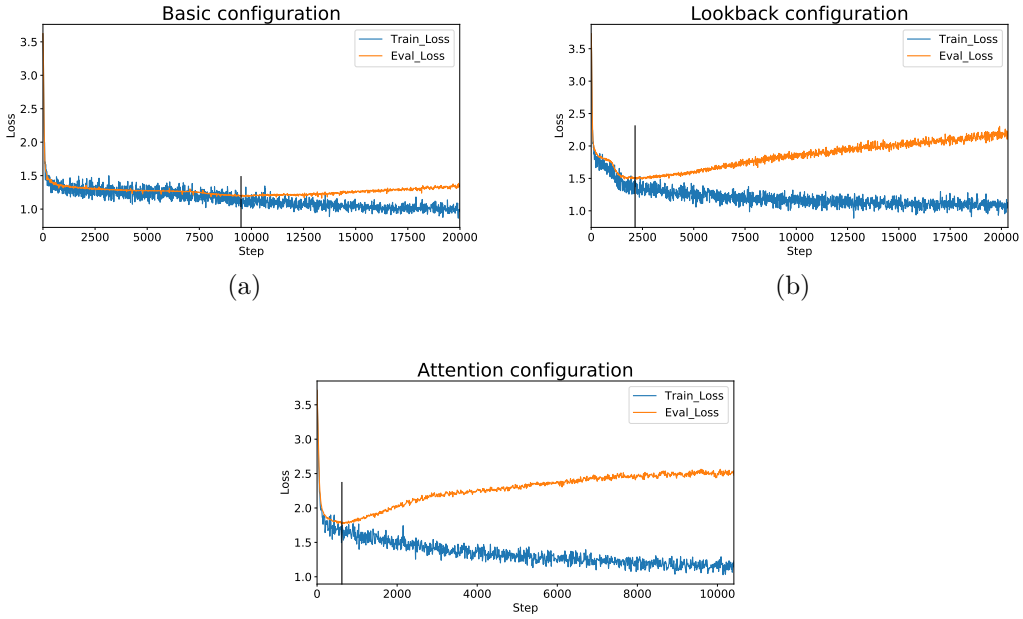


Figure 3.2: Loss of all three configuration on both training and evaluating datasets

After training all three configurations, 124 pieces of music were composed for each of them. The composing process started with one prespecified note. The individual first notes were chosen the same as the starting notes in training pieces. At each step $\boldsymbol{X}^{(t)}, t = 1, \ldots, T$, we want to predict the next step $\boldsymbol{X}^{(t+1)}$ in the form of one-hot vector of a length 38 from Basic configuration. This prediction can be viewed as a classification problem with 38 categories. The trained recurrent neural network output for each step $\boldsymbol{X}^{(t)}$ vector of probabilities $\boldsymbol{p}^{(t)} \in \mathbb{R}^{38}$ for the following step $\boldsymbol{X}^{(t+1)}$ (for more details see 2.3). The step $\boldsymbol{X}^{(t+1)}$ is then randomly chosen based on $\boldsymbol{p}^{(t)}$ as it was a conditional probability distribution over 38 categories. Generated melodies are on average 37 notes long, as well as the original melodies.

3.3.3 Artificial Music Evaluation

In this section, we are going to evaluate artificial music and compare it with the original Beatles. Objective analysis is based on musically informed objective metrics for evaluating and comparing the outputs of music generative systems proposed by Yang and Lerch [2018]. The majority of those metrics are based on note pitches, the rest of them describes rhythm features.

Pitch-Based Features

The pitch measures the degree of highness or lowness of a tone based on frequency. There exists 12 semitones (C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, B) in one pattern block which altogether forms an octave.

- **Pitch count (PC):** The pitch count specifies the number of different pitches within one sample. The output is a scalar for each sample.
- **Pitch count histogram (PCH):** The pitch class histogram has one bin for each of the 12 octave-independent representation of the pitch. That means that C♯1 = C♯2. The output is a vector of length 12 for each sample.
- **Pitch class transition matrix (PCTM):** The pitch class transition matrix is a two-dimensional representation of pitch class transitions for each ordered pair of notes. The output is 12×12 matrix for each sample.
- **Pitch range (PR)** The pitch range represents the difference between the highest and lowest pitch in semitones. The output is a scalar for each sample.
- **Average pitch interval (API)** The average pitch interval is calculated as the average value of the interval between two consecutive pitches in semitones. The output is a scalar for each sample.
- **Average non-decreasing sequence (ANDS), average non-increasing sequence (ANIS)** The average non-decreasing sequence, resp. the average non-increasing sequence captures the average length of all non-increasing, resp. non-decreasing sequences of at least two notes. Both features are calculated independently on each other, thus multiple following notes with the same pitch would be classified as part of both, non-increasing as well as non-decreasing sequence.
- **Pitch distance transition matrix (PDTM)** The pitch distance transition matrix is a two-dimensional representation of pitch distance transitions between consecutive notes. The distance can be either positive (melody is increasing), or negative (melody is decreasing). Distances with absolute values larger than 6 were united to category 6+, resp. -6+. The output is 13×13 matrix for each sample.
- **Pitch distance histogram (PDH)** The pitch distance histogram has one bin for each of the 26 absolute distances in pitch between two consecutive notes. The output is a vector of length 26 for each sample.

Rhythm-Based Features

The rhythm is a music pattern based on the length of notes. **The length** is measured in units of musical time called beats. Categories of duration are fractions of a beat, usually in powers of 2. Terminology goes as follows: a 4-beats note is called a *whole note*, a 2-beats note is a *half note*, a 1-beat note is a *quarter note*, etc. To extend a note for another half of its duration, a dot symbol is used, e.g. a *half dot note* last 3-beats. Pieces of silence are called *rests* and are measured in the same way as notes.

- **Note count (NC)** The note count specifies the number of different lengths of notes within a sample. The output is a scalar for each sample.
- **Average rest length (ARL)** The average rest length is calculated by averaging all rests included in one sample. The output is a scalar for each sample.
- **Note length histogram (NLH)** The note length histogram has one bin for each of the 9 beat length classes [full, dot half, half, dot quarter, quarter, dot eighth, eighth, dot sixteenth, sixteenth]. The classification of each note length is performed by finding the closest length category. The output is a vector of length 9 for each sample.
- **Rest length histogram (RLH)** The rest length histogram has one bin for each of the 9 beat length classes [full, dot half, half, dot quarter, quarter, dot eighth, eighth, dot sixteenth, sixteenth]. The classification of each rest length is performed by finding the closest length category. The output is a vector of length 9 for each sample.
- **Note length transition matrix (NLTM)** The note length transition matrix is a two-dimensional representation of length class transitions for each ordered pair of notes. The output is 9×9 matrix for each sample.

3.3.4 Results

We compare and evaluate four sets of 124 examples - one original Beatles used for training and three generated with different configuration - Basic, Lookback, and Attention. For various features, intra-set distances and inter-set distances are calculated to compare the distance of the features within and between sets of musical examples, then pairwise cross-validation is performed. In each cross-validation step, one sample is chosen and the Euclidean distance to each of the other samples is computed. In case of cross-validation within one data set, we talk about intra-set distances. If we compare each sample of one set with all samples of the other set, we talk about inter-set distances. As a result, we obtain the histogram of distances for each feature, thus we can calculate the sample mean and the sample standard deviance (table 3.1).

Furthermore, we calculate absolute measures for pitch count, pitch range, average pitch interval, average length of non-decresing and non-increasing sequences, note count, and average rest length (table 3.2).

Pitch-Based Features Results

Generally speaking, artificially generated pieces of music contain a wider range of note pitches and higher pitch differences in semitones of consecutive notes.

The average pitch range of the original Beatles per piece is nearly 11, whereas generated melodies have the range on average 15 notes (Basic), almost 14.5 (Lookback) and almost 15.5 (Attention). The closely linked average number of different pitches follows the same trend, as the original Beatles usually consist of nearly 7 different pitches and generated music of almost 8 in case Attention and even more than 8 in the other two cases. We also observe considerable differences between original and generated music for average pitch interval. Consecutive notes in the Beatles' songs usually differ by 2 pitches, but the difference is at least 2 times higher for generated melodies with 4 pitches for Basic and more than 5 pitches for Lookback and Attention. Furthermore, higher lengths of monotonic sequences in case of original music also indicate that generated music is less coherent and contains higher jumps and shorter monotonous parts (table 3.2).

All those conclusions are echoed by summative transition matrices and histograms. Pitch difference transition matrices show, that in the original music the same notes are repeated after each other in 14 % of cases (figure 3.4), whilst in case of generated music two consecutive jumps higher than 6 semitones occur the most often. This is more than consistent with previous findings of the wider spread of consecutive notes of generated melodies. In addition, 60.2 % of two pitch consecutive differences in a row are less or equal to 2 when it comes to Beatles, but only 20.6 % in case of Basic, only 11.3 % in case of Lookback and 11.4 % in case of Attention. It is also worth noting, that pitch difference +1 semitone is never followed by pitch difference +2 semitones and the other way around with -1 semitone followed by -2 semitones in case of Beatles and Lookback. The most common pitch difference between two consecutive notes is 2 in all cases (figure 3.6), but all generated pieces reach a maximum pitch difference between two consecutive notes at least 24 which means two octaves, whereas a maximum pitch difference between two consecutive notes of Beatles is 16.

Intra-set distances of pitch based features are slightly higher for original music, which indicates that generated pieces are more similar to each other. Inter-set distances comparing directly original and generated music are often slightly higher then intra-set distances, as can be expected regarding the nature of example sets (table 3.1). The biggest change between inter-set and intra-set distances occurs for pitch range and average pitch interval which mirrors conclusions from previous paragraphs. In all four cases, natural notes (such as C) are used more often than the raised ones (such as $C^{\sharp}$) as can be seen in figure 3.5.

Rhythm-Based Features Results

First of all, we observe that all three generative models tend to use a lot of 16th notes (at least in 25 % cases), which are the shortest notes those models are capable of. On the contrary, the most frequently used note length in the original Beatles is dotted 8th (around 25 % of all cases) as can be seen in figure 3.8a, whereas 16th notes were around 5 % of all cases. The more sophisticated model, the shorter notes were used most. This could be related to the fact, that the most complicated model Attention was training with the lowest number of iteration.

	Beatles		Basic		Lookback		Attention		Beatles & Bas.		Beatles & Look.		Beatles & Att.	
	Intra-set		Intra-set		Intra-set		Intra-set		Inter-set		Inter-set		Inter-set	
	Mean[1]	SD[2]	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Pitch count	2.08	1.63	1.80	1.40	1.61	1.31	2.10	1.66	2.24	1.70	2.20	1.64	2.25	1.73
Pitch class histogram	0.53	0.15	0.42	0.15	0.42	0.15	0.46	0.15	0.49	0.13	0.49	0.13	0.50	0.15
Pitch class trasition matrix	0.47	0.11	0.39	0.06	0.40	0.07	0.40	0.07	0.44	0.09	0.45	0.09	0.45	0.09
Pitch range	3.70	3.03	4.43	3.43	3.19	2.59	4.77	3.77	5.22	3.98	4.43	3.42	5.52	4.34
Average pitch interval	0.76	0.56	1.18	0.96	1.22	0.93	1.37	1.09	2.00	1.22	3.00	1.26	3.19	1.40
Note count	1.29	1.05	0.87	0.74	0.86	0.76	0.92	0.78	1.60	1.13	1.42	1.08	1.27	1.00
Average rest length	0.28	0.21	0.14	0.13	0.15	0.13	0.14	0.11	0.26	0.21	0.26	0.20	0.26	0.21
Note length histogram	0.61	0.24	0.28	0.09	0.29	0.09	0.27	0.10	0.56	0.15	0.60	0.15	0.67	0.15
Note length transition matrix	0.54	0.16	0.33	0.05	0.33	0.06	0.32	0.09	0.49	0.11	0.50	0.11	0.53	0.11

Note: [1] Sample mean [2] Sample standard deviation

Table 3.1: Relative measures for pitch-based and rhythm-based features

	Beatles		Basic		Lookback		Attention	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
Pitch count	6.95	1.86	8.30	1.61	8.33	1.46	7.97	1.88
Pitch range	10.99	3.36	15.02	3.94	14.41	2.89	15.41	4.28
Average pitch interval	2.03	0.67	4.01	1.07	5.04	1.08	5.22	1.23
Average non-increasing sequence	3.97	1.44	2.75	0.52	2.66	0.46	2.65	0.41
Average non-decreasing sequence	3.66	1.16	2.77	0.44	2.64	0.34	2.63	0.39
Note count	4.95	1.17	6.30	0.80	6.02	0.80	5.68	0.85
Average rest length	0.43	0.25	0.27	0.13	0.29	0.14	0.26	0.14

Table 3.2: Absolute measures for pitch-based and rhythm-based features

Nevertheless, the average number of notes with different lengths is nearly 5 in case of original Beatles which is around 1 note lower than in case of generated music, but with higher variability among samples (table 3.2). The average length of rest is 0.43 for of original music, which is considerably higher than 0.27 for Basic, 0.29 for Lookback and 0.26 for Attention. The trend of longer notes and longer rests is also reflected in table 3.1, as inter-set values for rhythm-based features exceed intra-set values for generated samples. This also indicates that original samples are more diverse in rhythmic structure, whereas generated samples tend to be more homogenous.

Subjective Evaluation

As the ultimate judge of creative output is the human [Yang and Lerch, 2018], we provide also subjective evaluation of generated music. How interesting, how real and how pleasant do new melodies appear? Nearly no melody does sound z-of-tune. However, they definitely do not reach the Beatles' qualities. We observed that generated melodies really do sound more pleasing with the increasing number of iterations, but overfitting caused the model to repeat the same melody pattern all over again. At first, we attempted to evaluate each melody with a number between 1 and 100 to see if there are any hidden treasures, but we perceived that all melodies are on a similar level of likability. Furthermore, is nearly impossible to distinguish between three different configurations. Unfortunately, we do not found generated music as real and pleasant as we hoped.

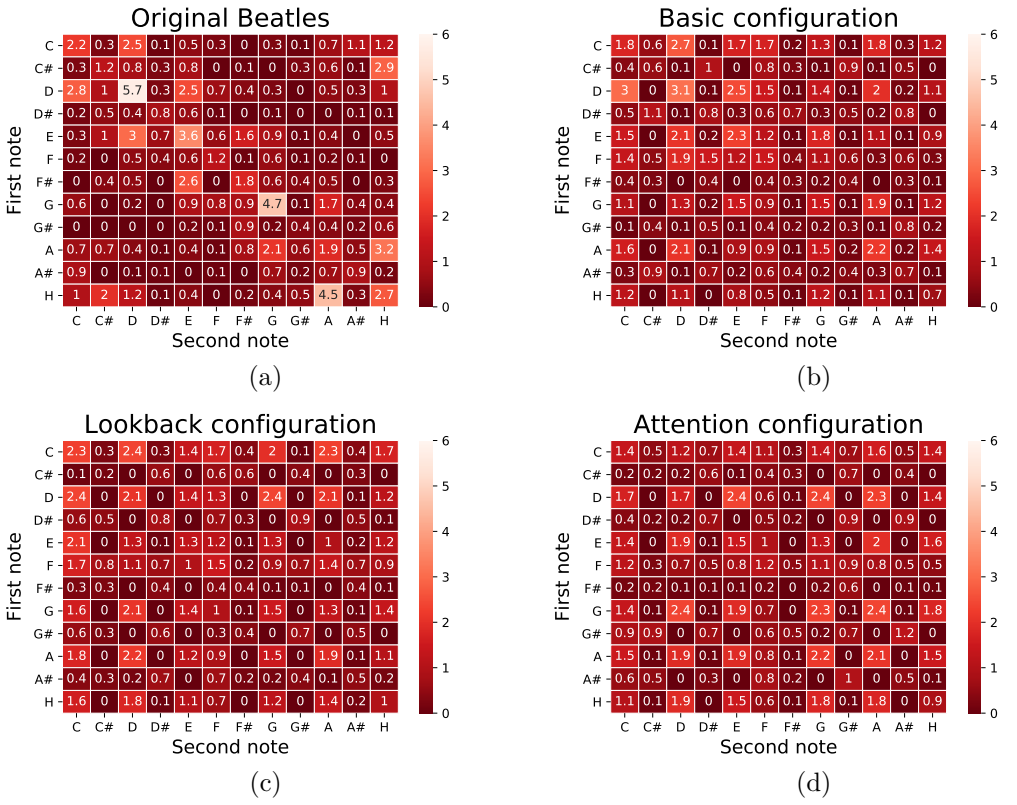


Figure 3.3: Pitch class transition matrices in percentage for all four datasets

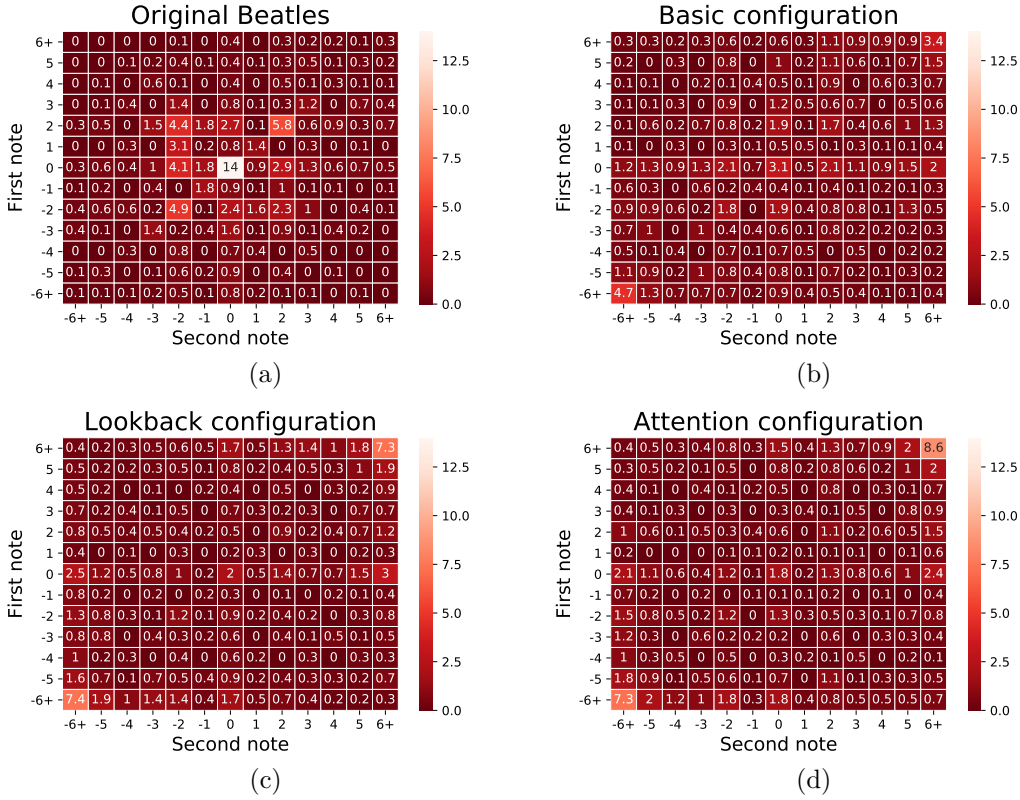


Figure 3.4: Pitch difference transition matrices in percentage for all four datasets

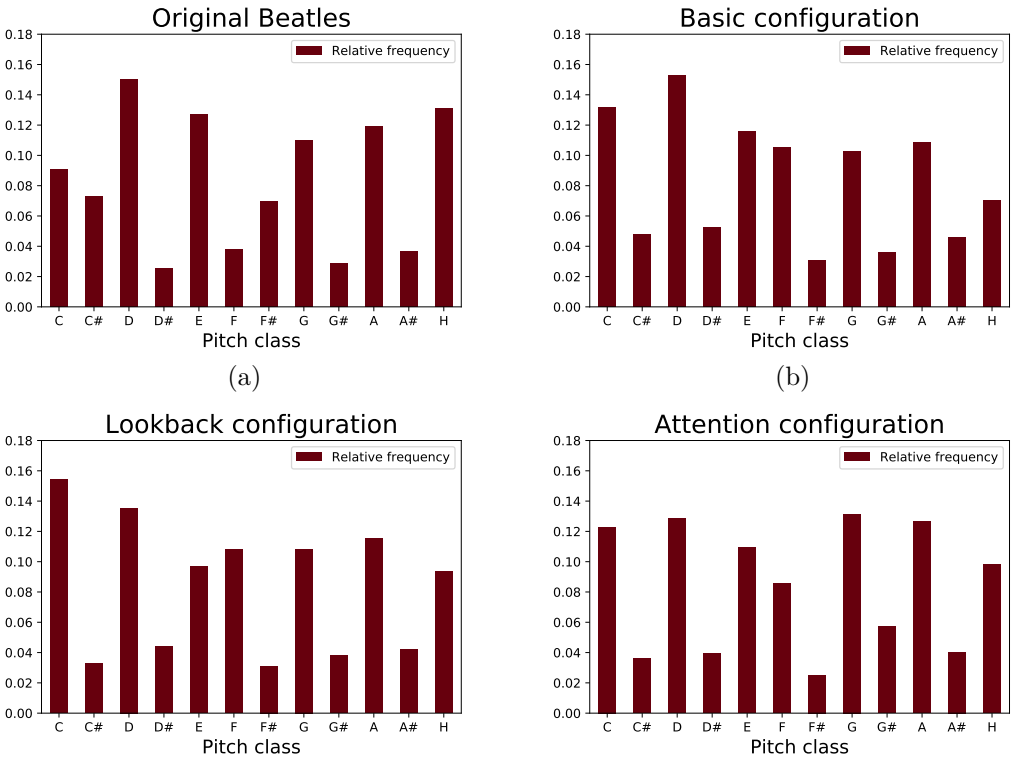


Figure 3.5: Pitch class histograms for all four datasets

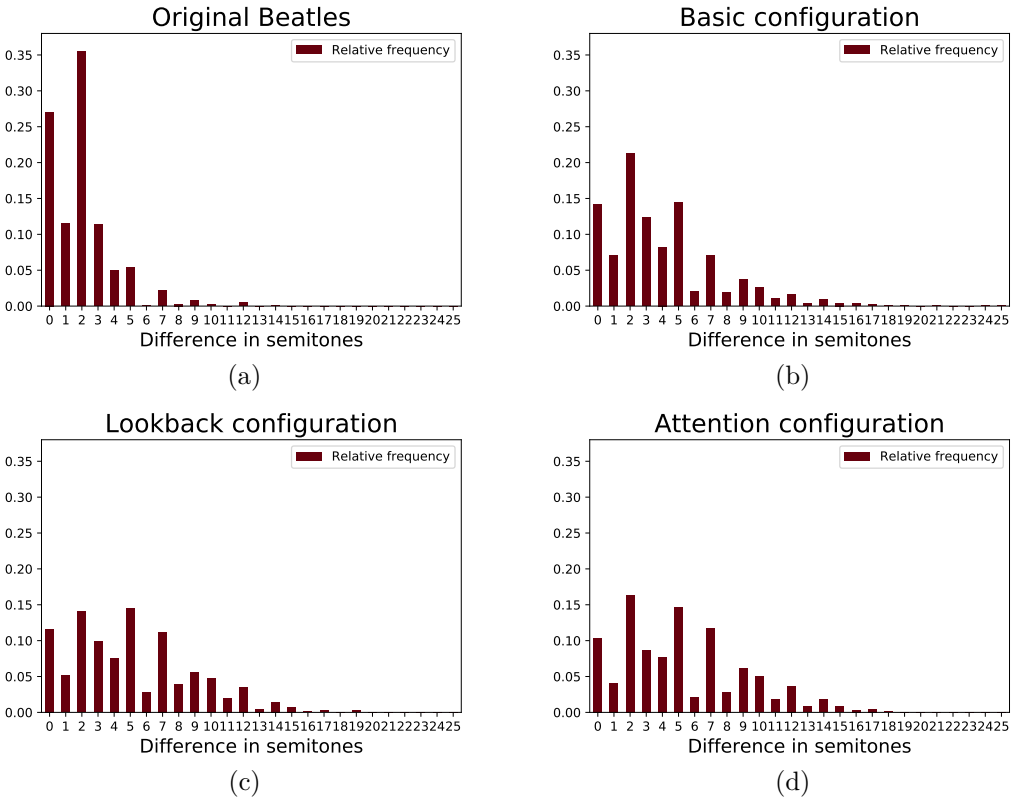


Figure 3.6: Pitch difference histograms for all four datasets

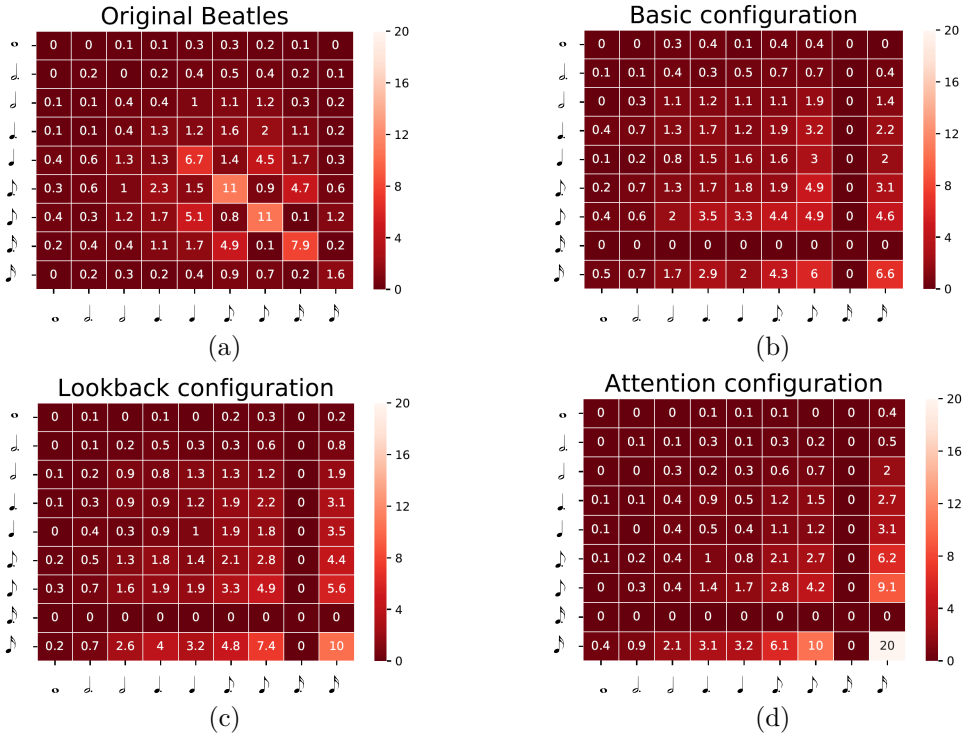


Figure 3.7: Note length transition matrix in percentage for all four datasets

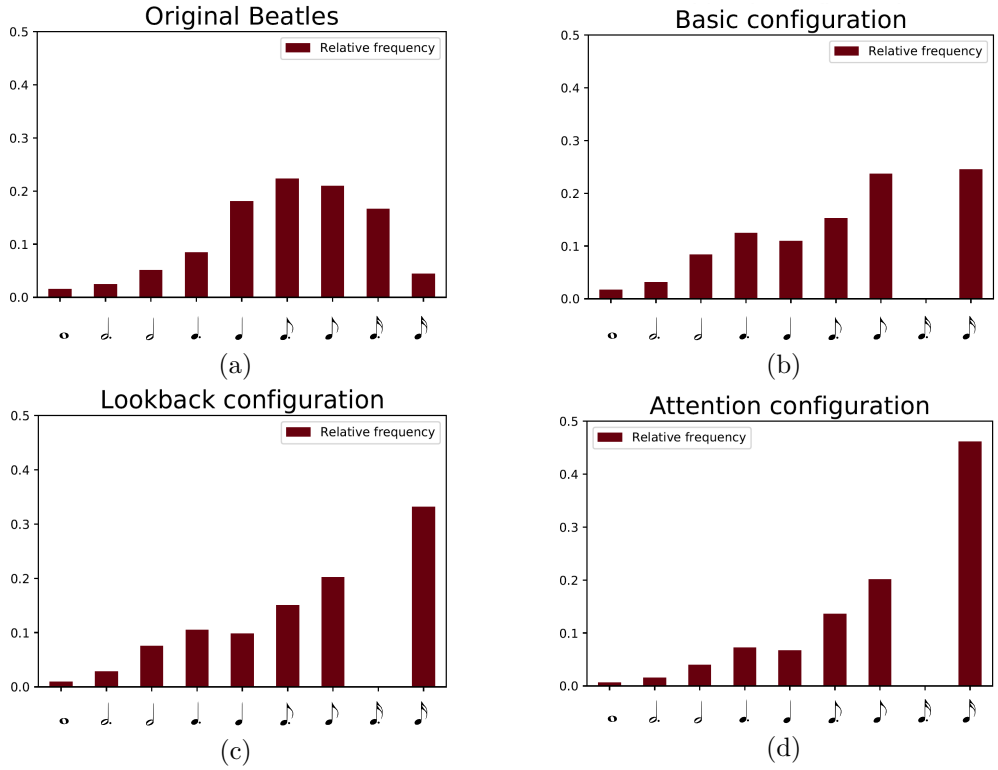


Figure 3.8: Note length histograms for all four datasets

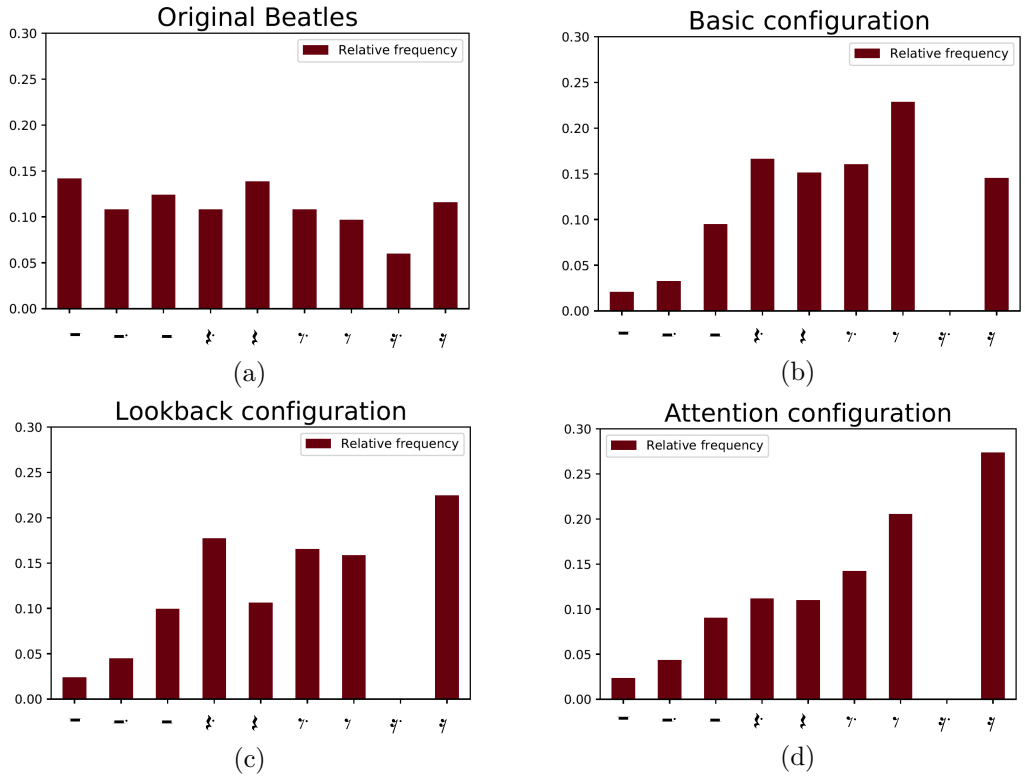


Figure 3.9: Rest length histograms for all four datasets

Conclusion

The aim of this thesis was to train a computer on Beatles' songs using research project Magenta from the Google Brain Team to produce its own music, to derive backpropagation formulas for recurrent neural networks with LSTM cells used in the Magenta music composing model, to overview machine learning techniques and discuss its similarities with methods of mathematical statistics.

At first, we presented the idea of machine learning and described the most common algorithms in both supervised and unsupervised learning (a linear regression, support vector machines, k-nearest neighbors, regression trees and random forests, a principal component analysis and k-means). After that, we generalized the concept of the linear model into neural networks with non-linear hidden units. We showed similarity between the feedforward neural network with one hidden layer and the sigmoid activation function and the generalized linear model of a logistic regression. We introduced recurrent neural networks with LSTM cells, as they represent a way of generating music with long-term structure and derived backpropagation formulas used in the Magenta music composing model. Furthermore, we provided an overview of important milestones in artifical music composition and presented several other approaches of generating music, including open software available to the public.

In practically oriented third chapter, we summarized the basics of a music theory and described the structure of MIDI files, which were used as an input for training generative models. We presented three different configurations (Basic, Lookback, and Attention) of Magenta, a research project based on Python's machine learning library Tensorflow, which was recently released by Google Brain Team. We decided to restrict ourselves to monophonic melodies only, as we wanted to see if generative systems can capture and creates interesting patterns. We wrote a script able to transform polyphonic melodies into monophonic ones. We preprocessed 124 monophonic melodies from Beatles and determined a suitable number of iterations for each configuration in order to avoid overfitting.

After training three generative models, we composed 124 samples from each of them with the same starting notes as training samples do. We evaluated them using musically informed objective metrics [Yang and Lerch, 2018] and compared them with original music. We discovered that generated samples tend to use a wider range of shorter notes with higher pitch differences between consecutive notes. Even though the generated pieces did not sound out-of-tune, they definitely did not reach the Beatles' qualities.

It is plausible, that we would obtain better results if we imputed the same Beatles samples multiple times after various transpositions. We did not use this option as several musically informed objective criteria depend on pitch classes and they would become pointless after such action. Regarding various representations of three different configurations and results with much higher pitch differences, it could be interesting to represent individual steps in terms of pitch differences between consecutive notes in order to learn model to keep notes closer together, as we believe that melodies with notes closely tight together sound better.

Bibliography

Margaret A. Boden. Creativity and artificial intelligence. *Artificial Intelligence*, 103(1-2):347–356, August 1998. URL https://www.sciencedirect.com/science/article/pii/S0004370298000551.

Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer-assisted composition at IRCAM: From patchwork to openmusic. *Computer Music Journal*, 23(3):59–72, September 1999. URL http://dx.doi.org/10.1162/014892699559896.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv:1409.0473v7*, 1409, September 2014. URL https://arxiv.org/pdf/1409.0473.pdf.

Yoshua Bengio, Paolo Frasconi, and Patrice Simard. The problem of learning long-term dependencies in recurrent networks. In *IEEE International Conference on Neural Networks*, volume 3, pages 1183–1188, March 1993. URL https://ieeexplore.ieee.org/abstract/document/298725/authors.

Bernhard Boser, Isabelle Guyon, and Vladimir N. Vapnik. A training algorithm for optimal margin classifier. *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, 5, August 1996. URL http://www.svms.org/training/BOGV92.pdf.

Nicolas Boulanger-Lewandowski, Yoshua Bengio, and Pascal Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *Proceedings of the 29th International Conference on Machine Learning*, 2, June 2012. URL http://www-etud.iro.umontreal.ca/~boulanni/ICML2012.pdf.

Jean-Pierre Briot, Gaetan Hadjeres, and Francois Pachet. Deep learning techniques for music generation - a survey. *arXiv:1709.01620v2*, November 2018. URL https://arxiv.org/abs/1709.01620.

Noam Brown and Tuomas Sandholm. Libratus: The superhuman AI for no-limit poker. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 2017. URL https://www.cs.cmu.edu/~noamb/papers/17-IJCAI-Libratus.pdf.

Murray Campbell, Joseph A. Hoane Jr., and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, January 2002. URL https://www.sciencedirect.com/science/article/pii/S0004370201001291.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Fethi Bougares, Dzmitry Bahdanau, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv:1406.1078v3*, June 2014. URL https://arxiv.org/pdf/1406.1078.pdf.

François Chollet. keras. https://github.com/fchollet/keras, 2015.

David Cope. *The Algorithmic Composer (Computer Music & Digital Audio Series)*. A-R Editions, June 2000.

Corinna Cortes and Vladimir Vapnik. Support vector networks. *Machine Learning*, 20:273–297, January 1995. URL https://link.springer.com/article/10.1007/BF00994018.

T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inf. Theor.*, 13(1):21–27, September 2006. ISSN 0018-9448. doi: 10.1109/TIT.1967.1053964. URL https://doi.org/10.1109/TIT.1967.1053964.

G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 1989.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back propagating errors. *Nature*, 323:533–536, October 1986. URL https://www.researchgate.net/publication/229091480_Learning_Representations_by_Back_Propagating_Errors.

Douglas Eck and Jurgen Schmidhuber. A first look at music composition using LSTM recurrent neural networks. Technical report, Instituto Dalle Molle di studi sull' intelligenza artificiale, April 2002. URL http://people.idsia.ch/~juergen/blues/IDSIA-07-02.pdf.

Bradley Efron and Trevor Hastie. *Computer Age Statistical Inference, Algorithms, Evidence, and Data Science*. Cambridge University Press, 2016. ISBN 978-1107149892.

Jose Fernandez and Francisco Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48, February 2014. URL http://www.svms.org/training/BOGV92.pdf.

D.A. Ferrucci. Introduction to "this is watson". *IBM Journal of Research and Development*, 56:1:1–1:15, 05 2012. doi: 10.1147/JRD.2012.2184356.

Stefan Fritsch, Frauke Guenther, Marvin Wright, Marc Suling, and Sebastian M. Mueller. *neuralnet: Training of Neural Networks*, 2019. R package version 1.44.2.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org.

Karol Gregor, Ivo Danihelka, Ales Graves, Danilo Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv:1502.04623v2*, May 2015. URL https://arxiv.org/pdf/1502.04623.pdf.

Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, March 2017. ISBN 978-1-491-96229-9.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, December 1997. URL https://www.bioinf.jku.at/publications/older/2604.pdf.

Max Kuhn. The caret package, 2009.

Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R News*, 2(3):18–22, 2002. URL https://CRAN.R-project.org/doc/Rnews/.

Huanru Henry Mao, Taylor Shin, and Garrison Cottrell. DeepJ: Style-specific music generation. *arXiv:1801.00887v1*, pages 377–382, January 2018. URL https://www.researchgate.net/publication/324957605_DeepJ_Style-Specific_Music_Generation.

Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc. New York, NY, first edition, March 1997. ISBN 0070428077.

Andreas C. Müller and Sarah Guido. *Introduction to Machine Learning with Python*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, first edition, October 2016. ISBN 978-1-449-36941-5.

Michael Mozer. Neural network music composition by prediction: Exploring the benefits of psychoacoustic constraints and multi-scale processing. *Connection Science - CONNECTION*, 6:247–280, January 1994. URL http://www.cs.colorado.edu/~mozer/Research/Selected%20Publications/reprints/Mozer1994.pdf.

Aran Nayebi and Matt Vitelli. Gruv : Algorithmic music generation using recurrent neural networks. 2015. URL https://cs224d.stanford.edu/reports/NayebiAran.pdf.

J. A. Nelder and R. W. M. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, 135(3):370–384, 1972.

Christopher Olah. *Understanding LSTM networks*, 2015. URL http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Alexandre Papadopoulos, Pierre Roy, and Francois Pachet. Assisted lead sheet composition using flowcomposer. In *Principles and Practice of Constraint Programming*, volume 9892, September 2016. URL https://www.researchgate.net/publication/304490458_Assisted_Lead_Sheet_Composition_Using_FlowComposer.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386 – 408, 1958. URL `https://psycnet.apa.org/record/1959-09865-001`.

Thomas Ross. The synthesis of intelligence—its implications. *Psychological Review*, 45(2):185–189, March 1938. URL `https://psycnet.apa.org/record/1938-02877-001`.

Arthur Lee Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, July 1959. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.2254&rep=rep1&type=pdf`.

Örjan Sandred, Mikael Laurson, and Mika Kuuskankare. Revisiting the illiac suite - a rule-based approach to stochastic processes. *Sonic Ideas/Ideas Sonicas*, 2:42–46, January 2009. URL `http://sandred.com/texts/Revisiting_the_Illiac_Suite.pdf`.

Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45:2673 – 2681, December 1997. URL `https://www.researchgate.net/publication/3316656_Bidirectional_recurrent_neural_networks`.

D. Silver, A Huang, Ch. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, January 2016.

Aaron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv:1609.03499*, September 2016. URL `https://arxiv.org/abs/1609.03499`.

Elliot Waite. Generating long-term structure in songs and stories. July 2016. URL `https://magenta.tensorflow.org/2016/07/15/lookback-rnn-attention-rnn`.

Paul Werbos. *Lecture Notes in Control and Information Sciences*, volume 38, chapter Applications of advances in nonlinear sensitivity analysis, pages 762–770. National Science Foundation, 1982.

Rui Yan. i, Poet: Automatic poetry composition through recurrent neural networks with iterative polishing schema. *Proceedings*

of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16), 2016. URL https://pdfs.semanticscholar.org/b97a/29712ad698387509bf91a68181a9758034e7.pdf.

Li-Chia Yang and Alexander Lerch. On the evaluation of generative models in music. *Neural Computing and Applications*, November 2018. URL https://doi.org/10.1007/s00521-018-3849-7.

List of Figures