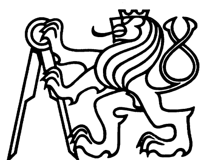
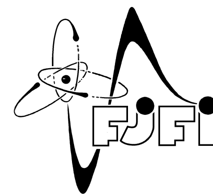


CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Nuclear Sciences and Physical Engineering



Signal Event List Generation Using Neural Networks

Generování seznamu událostí v signálu pomocí neuronových sítí

Master's Thesis

Author: **Bc. Martin Kovanda**

Supervisor: **Ing. Milan Chlada, Ph.D.**

Consultant: **doc. Ing. Tomáš Hobza, Ph.D.**

Academic year: 2021/2022

Katedra: matematiky | Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Martin Kovanda

Studijní program: Aplikované matematicko-stochastické metody

Název práce (česky): Generování seznamu událostí v signálu pomocí neuronových sítí

Název práce (anglicky): Signal Event List Generation Using Neural Networks

Pokyny pro vypracování:

1) Udělejte rešerši současných poznatků ohledně možností zpracování digitálních signálů pomocí neuronových sítí. Specifikace vhodných architektur pro analýzu spojitých zvukových záznamů bez nutnosti předzpracování do časo-frekvenční formy.
2) Proveďte sběr experimentálních dat, např. při buzení různých zdrojů akustické emise. Příprava kódu pro generování datových sad pro učení neuronových sítí.
3) Využijte autoencoderu pro odhad podobnosti signálů, porovnání s jinými metrikami.
4) Testujte neuronové sítě pro potlačení nežádoucích ruchových složek signálu a udělejte dekompozici signálů podle typů jejich zdrojů.
5) Verifikujte schopnosti sítí identifikovat události v signálu v závislosti na jejich podobnosti a časové separovatelnosti.

Doporučená literatura:

1) S. Pal, A. Gulli, Deep Learning with Keras, Packt Publishing Ltd., ISBN 9781787128422, 2017.

2) F. Chollet, Deep Learning with Python, Manning Publications Co., ISBN 9781617294433, 2017.

3) J. L. Rose, Ultrasonic Waves in Solid Media. Cambridge University Press, 2004.

4) I. Fawaz et al., InceptionTime: Finding AlexNet for time series classification, Data Mining and Knowledge Discovery volume 34, 2020, 1936–1962.

5) J. Proakis, D. Manolakis, Digital signal processing, Pearson Prentice Hall, ISBN 9780131873742, 2007.

Jméno a pracoviště vedoucího diplomové práce:

Ing. Milan Chlada, Ph.D.
Ústav termomechaniky AV ČR, Dolejškova 1402/5, 182 00 Praha 8

Jméno a pracoviště konzultanta:

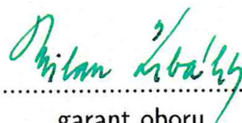
doc. Ing. Tomáš Hobza, Ph.D.
Katedra matematiky FJFI ČVUT v Praze, Trojanova 13, 120 00 Praha 2

Datum zadání diplomové práce: 31.10.2021

Datum odevzdání diplomové práce: 2.5.2022

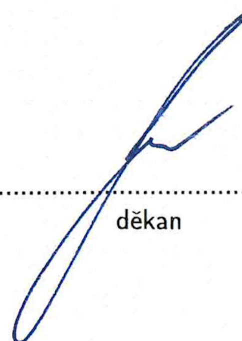
Doba platnosti zadání je dva roky od data zadání.

V Praze dne 01.11.2021



garant oboru

vedoucí katedry



děkan

Acknowledgment:

I would like to thank Ing. Milan Chlada, Ph.D. for his expert guidance.

Author's declaration:

I declare that this Master's Thesis is entirely my own work and I have listed all the used sources in the bibliography.

Prague, May 2, 2022

Martin Kovanda

Název práce:

Generování seznamu událostí v signálu pomocí neuronových sítí

Autor: Bc. Martin Kovanda

Program: Aplikované matematicko-stochastické metody

Druh práce: Diplomová práce

Vedoucí práce: Ing. Milan Chlada, Ph.D.,
Ústav termomechaniky AV ČR, v. v. i., Dolejškova 1402/5, 182 00 Praha 8

Konzultant: doc. Ing. Tomáš Hobza, Ph.D.,
Katedra matematiky FJFI ČVUT v Praze, Trojanova 13, 120 00 Praha 2

Abstrakt: Cílem práce je rešerše metod hlubokého učení a následná aplikace shrnutých poznatků pro dekompozici signálu do jednotlivých zdrojů a detekci událostí v signálu. V první části je zkoumána možnost aplikace modifikovaných existujících architektur na spojitý signál za účelem odhadu původních jeho komponent příslušných jednotlivým zdrojům. Ve druhé části je pak zkoumána strategie postupu u detekce jednotlivých událostí různých typů ve vstupním signálu. Potenciál těchto metod je demonstrován jak na hudebním datasetu, tak u praskavé akustické emise pocházející z únavových zkoušek materiálu. Naučené neuronové sítě mohou být použity pro automatizaci analýzy ultrazvukových signálů, např. k online rozpoznávání emisních událostí.

Klíčová slova: akustická emise, dekompozice signálů, generování seznamu událostí, hluboké učení, klasifikace časových řad, klasifikace obrazu, strojové učení

Title:

Signal Event List Generation Using Neural Networks

Author: Bc. Martin Kovanda

Abstract: The goal of this paper is to summarize deep learning methods and to apply the gained knowledge on signal decomposition and signal event detection tasks. In the first part several architectures are applied to a continuous signal in order to estimate its original components corresponding to the individual sources. Several metrics are used in order to evaluate the final models. In the second part the strategy for signal events detection in the input signal is discussed. The potential of these methods is demonstrated both on a musical dataset and on burst acoustic emission originating from material fatigue tests. The trained neural networks can be used to automate the analysis of ultrasonic signals, e.g. for real-time detection of emission events.

Key words: acoustic emission, deep learning, event list generation, image classification, machine learning, signal decomposition, time series classification

Contents

Introduction **11**

1 Theory **13**

1.1 Feed Forward Neural Networks 13
1.2 Convolutional neural networks 13
1.3 Activation functions 14
1.4 Loss functions 15
1.5 Weight initialization 16
1.6 Backpropagation 18
1.7 Optimizer strategies 19
1.8 Optimizer algorithms 19
1.9 Data augmentation 21
1.10 Time-frequency transformations 22
1.11 Autoencoders 23

2 Neural Network Architectures **27**

2.1 Development summary 27
2.2 Architecture visualization 29
2.3 Early development 29
2.4 Inception 31
2.5 ResNet 34
2.6 Inception-v4, Inception-ResNet-v2 36
2.7 ResNeXt-50 37
2.8 EfficientNets 38
2.9 InceptionTime 40
2.10 U-Net 42

3 Signal Decomposition Task **45**

3.1 Problem definition . . . 46
3.2 Dataset . . . 47
3.3 Metrics . . . 48
3.4 Autoencoders . . . 48
3.5 Training strategy . . . 49
3.6 Signal to signal networks . . . 50
3.7 STFT to STFT networks . . . 51
3.8 Separate instrument comparison . . . 52

4 Event List Generation **55**

4.1 Problem definition . . . 55
4.2 Approach . . . 56
4.3 Dataset - Tones . . . 56
4.4 Evaluation . . . 57
4.5 Raw signal networks . . . 57
4.6 STFT networks . . . 59
4.7 Identification of acoustic emission bursts . . . 61

Conclusion **65**

Introduction

In the last decade there was a big progress in deep learning algorithms. This progress accelerated by increasing hardware capabilities affected many research fields and in a short time became a part of our lives. As an example, many security systems use some kind of face recognition algorithm based on deep learning. Many various algorithms are being used for advertisement recommendation but also for helping doctors to analyze ECG signal or X-ray images.

The concept of neural networks began already in the 20th century. However, that time it was not popular since these networks had a high computational and memory demand. Training deep convolutional networks also requires a lot of data which became easier recently mostly because of the spread of the internet. Soon many recognition tasks were created in order to make research teams from all the world compete on the same data to make the models comparable. One of these tasks is called the ImageNet Classification task. Here the dataset consists of over a million images taken from 1000 classes. This challenging task became one of the most popular benchmarks for image recognition models.

From the time of AlexNet in 2012 all the winning architectures were based on Convolutional Neural Networks (ConvNets). Soon this technology started to spread into many other fields. One of them is the Signal Decomposition task in which a signal is split into its original individual sources. This can be useful for many problems, e.g. noise canceling or a separation of instrumental sources in audio records. This problem is challenging because many sources produce signals containing the same frequency range and there is no unique solution how to separate them. Neural networks inspired by those from Image Classification and Segmentation can be used for this task as they are able to find crucial patterns in the signal or the STFT representation. Finding these patterns appears to be necessary for building a well performing network.

Another goal of this paper is to create a deep learning model capable of detecting individual events in the signal. This can be used in many areas, e.g. acoustic emission analysis or piano tones detection. In this paper we explore both of them in order to find out which models are performing better and which strategy to use. Resulting network can be used in automating the burst detection in acoustic emission without any need for preprocessing.

All the code used in this paper will be available on *https://github.com/AIKovanda/signalai*.

Chapter 1

Theory

1.1 Feed Forward Neural Networks

There are many types of neural networks (NN) used for many different tasks. The basic type is the Feed Forward Neural Network (FFNN) which is still used these days, often together with more complex methods such as convolutional networks etc. FFNN is a group of neurons placed in several connected layers. All neurons from each layer are connected to all neurons from the previous and following layers. This architecture can be understood as a series of non-linear parametric transformations. The goal is a to find optimal parameters in the process of learning, i.e. iteratively changing parameters using gradient descent methods.

In the FFNN all the structure is based on Layer, which is composed of individual neurons. A neuron can be understood as a simple processing unit which takes the weighted sum of its inputs and transform it using a predefined activation function. All the neurons between two adjacent layers are connected to each other using a connection containing a weight parameter. This parameter is initialized randomly and is adjusted during the training. For a simple network with m layers the overall behavior can be described using equation

$$\mathbf{x}_{n+1} = f(\mathbf{W}^{(n)}\mathbf{x}_n), \tag{1.1}$$

where $\mathbf{x}_n$ is the input vector of the n-th layer, f represents the activation function, $\mathbf{W}^{(n)}$ stands for the weight matrix of the n-th layer, $\mathbf{x}_1$ is the input vector and $\mathbf{x}_m$ the desired output vector. The value $W_{ij}^{(n)}$ represents the weight between the i-th neuron in the n-th layer and the j-th neuron from the $(n-1)$-st layer.

1.2 Convolutional neural networks

FFNN has some problems when using multi-dimensional input, e.g. an image. Moving image by just some pixels is not a big change, however, a vectorized version of that image would differ dramatically. For this reason, training FFNN for image classification or other similar tasks would be very inefficient. Another problem is that the image contains many spacial features that would be lost if the tensor is vectorized.

For this reason, another type of neural networks called the Convolutional Neural Network (ConvNets) are used. Here the layers are composed of convolutional cores instead of simple neurons to extract features from the input tensor.

Each convolutional layer consists of D convolutional cores $\mathbf{A}^{(i)} \in \mathbb{R}^{w_a,h_a,c}$, $\forall i \in \{1,...,D\}$. When they are applied to the input tensor $\mathbf{T} \in \mathbb{R}^{w_t,h_t,c}$, the output tensor is created as

$$o_{ijn} = \sum_{k=1}^{w_a}\sum_{l=1}^{h_a}\sum_{m=1}^{c} a_{klm}^{(n)} t_{si+k,sj+l,m}\ , \quad \forall i \in \{1,...,(w_t - w_a)/s+1\}, \quad \forall j \in \{1,...,(h_t - h_a)/s+1\}, \tag{1.2}$$

where s stands for stride. Since $(w_t - w_a)/s + 1 \leqslant w_t$ and $(h_t - h_a)/s + 1 \leqslant h_t$, sometimes a zero padding is added to the input tensor so that the output tensor would keep the same size of the first two dimensions.

In many cases there is a need of reducing dimensionality in the data. This contributes to reduction of noise and redundant information present in the input data. For this reason the pooling layer of shape (w_p, h_p) may be applied as

$$o_{ijn} = f\big(\{t_{si+k,sj+l,n},\ \forall k \in \{1,...,w_p\},\ \forall l \in \{1,...,h_p\}\}\big)\ , \quad \begin{array}{l}\forall i \in \{1,...,(w_t - w_p)/s+1\},\\ \forall j \in \{1,...,(h_t - h_p)/s+1\},\end{array} \tag{1.3}$$

where f represents the core pooling function. Most used functions are max and average, which leads to the naming convention *max-pooling* and *avg-pooling*. In practice usually with $h_p = w_p = s = 2$ which leads to the reduction of data with a factor of 4.

1.3 Activation functions

A combination of multiple linear transformation is a linear transformation. Since the goal of neural networks is to create a non-linear transformation, a non-linear activation function is usually added after some (or all) the layers. There are many functions used in the modern architectures, such as a sigmoid function $\sigma(x) := \frac{1}{1+\mathrm{e}^{-x}}$ which is used especially in the last layer of a classification model since it maps input x to the range of $[0,1]$. In the early literature the ReLU function $\mathrm{ReLU}(x) := \max(0,x)$ replaced Sigmoid and Tanh functions in the hidden layers because of a good performance while reducing the computational demand.

There are many hand-designed modifications for the ReLU activation which try to enhance the networks by allowing the information to flow even for $x < 0$. These improvements can increase the model performance with the cost of bigger computational demand. These functions are the **Leaky ReLU** defined as

$$\mathrm{LReLU}(x) := \begin{cases} x & \text{if } x \geqslant 0, \\ \alpha x & \text{if } x < 0, \end{cases} \tag{1.4}$$

where $\alpha = 0.01$. Unlike ReLU, LReLU enables information to flow even for $x < 0$. An alternative is the **Parametric ReLU, PReLU**, in which the α parameter is trainable. Another often used function is the **Scaled Exponential Linear Unit, SELU** defined as

$$\mathrm{SELU}(x) := \lambda \begin{cases} x & \text{if } x \geqslant 0, \\ \alpha\big(\exp(x) - 1\big) & \text{if } x < 0, \end{cases} \tag{1.5}$$

where $\alpha \approx 1.6733$ and $\lambda \approx 1.0507$.

However, most of them are not consistent on improving the neural networks. In 2017, Ramachandran et al. came with another approach. Instead of designing a new activation manually, the appropriate activation was searched using a reinforcement learning method[1]. The result of their paper is an activation called the Swish function defined as

$$\mathrm{swish}(x) := x \cdot \mathrm{sigmoid}(\beta x), \tag{1.6}$$

where β is a constant or trainable parameter. The experiments made by Ramachandran et al. showed that switching to the Swish activation leads to the improvement of around 0.6%-0.9% among some popular architectures [1]. A big difference from the ReLU is that the Swish function is not monotone around zero. This feature can be controlled using the β parameter.

In 2020 Mishra proposed a new self-regularized non-monotonic activation function called the Mish function [2] defined as

$$\operatorname{mish}(x,\beta) := x\tanh\left(\frac{1}{\beta}\log\left(1+\exp(\beta x)\right)\right), \tag{1.7}$$

where β is a parameter. Mish is a self-regularized non-monotonic activation function. Together with SELU they are both continuously differentiable which enhances the gradient-based optimization of the network [2]. Mishra showed that on CIFAR-10 dataset (dataset of small images divided to 10 classes) and on COCO object detection dataset the Mish activation increased the validation accuracy over the other activations. This was shown across various neural network architectures from both image classification and object detection tasks. Most of the mentioned activation functions are depicted in Figure 1.1.

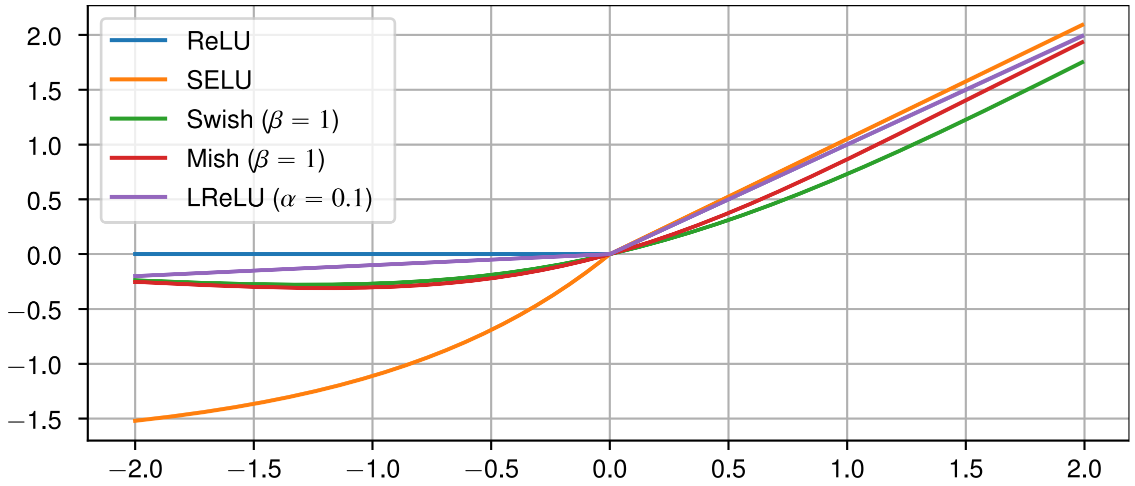


Figure 1.1: Comparison of chosen activation functions.

1.4 Loss functions

In order to update network parameters in a good direction, a proper loss function (cost function) must be used at the learning. There are many functions used in many various tasks. One of them is a simple L_1 loss defined as

$$\mathrm{L}_1(\hat{\mathbf{y}}_1,...,\hat{\mathbf{y}}_N,\mathbf{y}_1,...,\mathbf{y}_N) := \frac{1}{N}\sum_{i=1}^{N}\|\hat{\mathbf{y}}_i-\mathbf{y}_i\|_{L_1}, \tag{1.8}$$

where N stands for a batch size, $\mathbf{y}_i \in \mathbb{R}^m$ represents the i-th ground truth value, $\hat{\mathbf{y}}_i \in \mathbb{R}^m$ stands for the i-th prediction of $\mathbf{y}$, and $\|\cdot\|_{L_1}$ represents a standard L1 norm for vectors. The batch size is chosen by the optimizer strategy and is $N \in \{1,...,n\}$ for a dataset with n samples. Similarly, the L2 loss can be defined as

$$\mathrm{L}_2(\hat{\mathbf{y}}_1,...,\hat{\mathbf{y}}_N,\mathbf{y}_1,...,\mathbf{y}_N) := \frac{1}{N}\sum_{i=1}^{N}\|\hat{\mathbf{y}}_i-\mathbf{y}_i\|_{L_2}^2, \tag{1.9}$$

where $\|\cdot\|_{L_2}$ represents a standard L2 norm for vectors. The L_1 loss is more immune to outliers, however, L_2 loss more penalize big value differences.

Another loss function in this paper is the **Binary Cross-Entropy** defined as

$$\mathrm{BCE}(\widehat{\mathbf{y}}_1, ..., \widehat{\mathbf{y}}_N, \mathbf{y}_1, ..., \mathbf{y}_N) := -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m}[y_{ij}\log\widehat{y}_{ij} + (1-y_{ij})\log(1-\widehat{y}_{ij})]. \tag{1.10}$$

This loss is valid for $y, \widehat{y} \in [0, 1]$ and is used for multi-label classification since it enables to train the network using maximum likelihood estimation.

1.5 Weight initialization

Before the beginning of training a neural network, all the parameters (weights and biases) have to be initialized. There are several rules that need to be followed. First, the weights cannot be initialized constantly. The network would become symmetric and the learning would not be possible because of a symmetric gradient.

Furthermore, the absolute value of weights cannot be too small or too high because of the **vanishing gradient problem**. This problem occurs when the gradient of a used activation function gets close to zero. In that case the network does not learn properly anymore.

Another problem can appear when all the network weights are initialized using the same distribution function. In that case some layers may learn faster than the other ones and this instability can highly reduce the benefit of using many layers.

There are several techniques how to properly initialize a neural network. Let $\mathbf{W}$ be the weight matrix in one particular layer in a feed forward neural network. The number of neurons in the previous layers will be denoted as fan_in and the number of neurons in the following layers as fan_out. As for 2022, there are two main approaches used for weight initialization.

Xavier (Glorot) initialization

In 2010, on of the commonly used initialization was [3] a uniform distribution

$$W_{ij} \sim \mathcal{U}\Big[-\frac{1}{\sqrt{fan_in}}, \frac{1}{\sqrt{fan_in}}\Big]. \tag{1.11}$$

Xavier Glorot and Yoshua Bengio showed that using this initialization for layers with sigmoid or TanH activations leads to poor quality networks [3]. The networks tend to learn with different speed in different layers, see Figure 1.2.

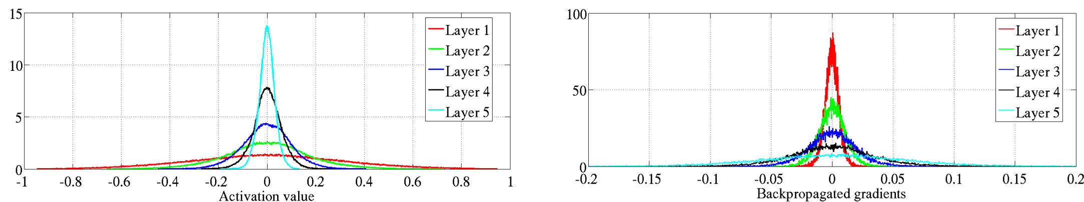


Figure 1.2: Activation value and backpropagation gradients in different layers using a wrong initialization [3].

Let $x^{(i)}$ denote the result vector coming from the i-th layer and $s^{(i)}$ the argument of corresponding activation function f, $f(s^{(i)}) = x^{(i)}$. The way how to solve this problem is to try to keep the variance of all the weights and the variance of back-propagation matrices the same, i.e.

$$\mathrm{Var}[x^{(i)}] = \mathrm{Var}[x^{(j)}], \qquad \mathrm{Var}\Big[\frac{\partial L}{\partial s^{(i)}}\Big] = \mathrm{Var}\Big[\frac{\partial L}{\partial s^{(j)}}\Big], \qquad \forall i, j \in \{1, ..., n\}, \tag{1.12}$$

for a neural network with n layers using a loss function L. Glorot and Bengio showed that the resulting distribution should have a mean value of $\mu = 0$ and a variance of $\sigma^2 = \frac{2}{fan_in+fan_out}$ which gives

$$W_{ij} \sim \mathcal{U}\Big[-\frac{\sqrt{6}}{\sqrt{fan_in + fan_out}}, \frac{\sqrt{6}}{\sqrt{fan_in + fan_out}}\Big] \tag{1.13}$$

and

$$W_{ij} \sim \mathcal{N}\Big(0, \frac{2}{fan_in + fan_out}\Big), \tag{1.14}$$

for **Xavier Uniform** and **Xavier Normal** distributions respectively. With this initialization the distribution of activation values and backpropagation gradients are more similar, see Figure 1.3. This allows training much deeper neural networks since all those layers will be trained evenly.

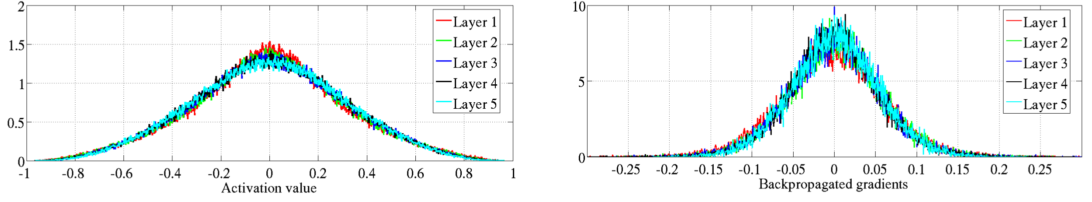


Figure 1.3: Activation value and backpropagation gradients in different layers using Xavier initialization [3].

Kaiming (He) initialization

The Xavier initialization assumes using a linear activation function. This may not be a problem for Sigmoid and TanH activations since they both can be linearly approximated near zero. However, for ReLU-like activations this approach does not work properly [4]. The problem here is that $\mathrm{E}x^{(i)} \neq 0$. He et al. showed that the variance should be scaled using some factor [4] (e.g. 2 for ReLU). Furthermore, it was discussed that there is no need for using both fan_in and fan_out and that is it sufficient to only use fan_in. The resulting initializations are then defined as

$$W_{ij} \sim \mathcal{U}\Big[-\frac{\sqrt{6}}{\sqrt{fan_in}}, \frac{\sqrt{6}}{\sqrt{fan_in}}\Big] \quad \text{and} \quad W_{ij} \sim \mathcal{N}\Big(0, \frac{2}{fan_in}\Big), \tag{1.15}$$

for **Kaiming Uniform** and **Kaiming Normal** distributions, respectively. In this paper all neural networks will be initialized using Kaiming Uniform distribution.

Analogically, this all process can be applied for the convolutional layers by defining

$$fan_in := kernel_height \cdot kernel_width \cdot num_input_channels \tag{1.16}$$

and

$$fan_out := kernel_height \cdot kernel_width \cdot num_output_channels. \tag{1.17}$$

1.6 Backpropagation

After all the weights are initialized, the network is prepared for training. The weights are being updated according to their gradient based on the input tensor X and a chosen loss function L. The method for calculating gradient is called **backpropagation**.

Every neural network consists of layers connected to a computational acyclic graph. Each i-th layer can be defined as

$$f_i(X^{(i)}) = Q^{(i)}, \quad f_i(X_1^{(i)}, X_2^{(i)}) = Q^{(i)} \quad \text{or} \quad f_i(X^{(i)}, W^{(i)}) = Q^{(i)}, \tag{1.18}$$

where $X_j^{(i)}$ stands for j-th input, $W^{(i)}$ stands for the weight parameter, f represents a function (add, multiply, convolution, activation, etc.) and $Q^{(i)}$ stands for the output of i-th layer. Some layers may only have one input, e.g. layers containing solely an activation function. The graph determines how to calculate forward propagation, i.e. to calculate the overall output $\widehat{Y}$. The goal is then to determine $\frac{\partial L}{\partial W^{(i)}}$ for all trainable weights $W^{(i)}$. For this calculation the chain rule

$$\frac{\partial L}{\partial X_j^{(i)}} = \frac{\partial Q^{(i)}}{\partial X_j^{(i)}} \frac{\partial L}{\partial Q^{(i)}} \tag{1.19}$$

will be applied inside each layer.

In the forward pass, each $Q^{(i)}$ value is calculated simply by its definition. At the same time, all the input tensors are saved into the memory for the backward propagation. After calculating all the layers inside the neural network, the final output value $\widehat{Y}$ is computed and evaluated by the loss function. Then the gradient value $\frac{\partial L}{\partial \widehat{Y}}$ is calculated and used as a starting point for the backward propagation.

In the backward propagation each layer starts with a gradient vector $\frac{\partial L}{\partial \widehat{Q}^{(i)}}$. Vector $\frac{\partial L}{\partial X_j^{(i)}}$ is calculated using the equation 1.19 for each input $X_j^{(i)}$ (or a weight $W^{(i)}$). These values are then used in each input layer as the $\frac{\partial L}{\partial \widehat{Q}}$. In case that the $Q^{(i)}$ vector is used in multiple places in the computational graph, all the incoming gradient vectors $\frac{\partial L}{\partial \widehat{Q}^{(i)}}$ are averaged.

This algorithm allows to calculate gradient in all the layers even for very complex graphs. In case of using max value, i.e. $Q^{(i)} = \max(X_1^{(i)}, X_2^{(i)})$, the bigger input $X_b^{(i)}$ gets $\frac{\partial L}{\partial X_b^{(i)}} = \frac{\partial L}{\partial Q^{(i)}}$, whereas the smaller one gets a zero gradient. For this reason, only the relevant input is modified.

This approach works on various operations, such as a dense layer defined as $Q^{(k)} = W^{(k)} X^{(k)}$. Here the partial derivations of $X_j^{(k)}$ and $W_{ij}^{(k)}$ are calculated as

$$\frac{\partial L}{\partial X_j^{(k)}} = \frac{\partial Q_i^{(k)}}{\partial X_j^{(k)}} \frac{\partial L}{\partial Q_i^{(k)}} = W_{ij}^{(k)} \frac{\partial L}{\partial Q_i^{(k)}}, \qquad \frac{\partial L}{\partial W_{ij}^{(k)}} = \frac{\partial Q_i^{(k)}}{\partial W_{ij}^{(k)}} \frac{\partial L}{\partial Q_i^{(k)}} = X_j^{(k)} \frac{\partial L}{\partial Q_i^{(k)}}. \tag{1.20}$$

A similar situation can be shown at the convolutional layer defined as $Q^{(k)} = \mathrm{Conv}(X^{(k)}, F^{(k)})$, where $X^{(k)}$ stands for the input tensor and $F^{(k)}$ represents the filter in the k-th layer. However, it was shown [5] that here the gradients can be expressed as a convolution between the input tensor and the incoming gradient,

$$\frac{\partial L}{\partial F^{(k)}} = \mathrm{Conv}\Big(X^{(k)}, \frac{\partial L}{\partial Q^{(k)}}\Big) \qquad \frac{\partial L}{\partial X^{(k)}} = \mathrm{Conv}_{\mathrm{full}}\Big(F_{rot}^{(k)}, \frac{\partial L}{\partial Q^{(k)}}\Big), \tag{1.21}$$

where $\mathrm{Conv}_{\mathrm{full}}$ stands for a full convolution and $F_{rot}^{(k)}$ represents the filter $F^{(k)}$ rotated by 180°.

1.7 Optimizer strategies

After the backpropagation is calculated using a proper loss function, the weights θ are updated using an optimizer. There are several strategies how to apply the optimizer [6]. First, in the **batch gradient descend (BGD)** the weights are updated using all the samples from the dataset. For a loss function $L(\theta_n)$ and learning rate of ν the update is made as

$$\theta_{n+1} = \theta_n - \nu \nabla_\theta L(\theta_n), \tag{1.22}$$

where θ_n stands for weights in the n-th iteration, ∇_θ represents the mean gradient from all the training samples. The problem of this strategy is that calculating gradient descend for all the training data can be very slow and usually the memory requirements exceed the RAM capacity. This strategy also does not count with using generators instead of specific training data, e.g. with data augmentation. On the other hand, it is proven that BGD will always converge to the global minimum for convex surfaces and to the local minimum for non-convex surfaces [6].

Large datasets usually contain some similar data for which the gradient has a similar value. Calculating all of them therefore is not necessary. Instead, the gradient can be calculated on each sample individually. This strategy is called the **Stochastic gradient descend (SGD)**. One more benefit of using SGD is that it can be used for the online training, i.e. when training data are created additionally. The convergence is not guaranteed as of BGD, however, experiments show that with decreasing learning rate those minima are usually reached anyway [6]. On the other hand, the optimization suffers from high fluctuations as the weights are often updated incorrectly.

Finally, the **mini-batch gradient descent** (also called **SGD**) takes a subset of the training data and updates weights according to the mean value of their gradient. This strategy leads to a more stable and efficient way how to update weights [6]. It is worth mentioning that the convergence is still not guaranteed and there may be a problem with choosing a proper learning rate. Small learning rate can lead to slow convergence while too big rate can cause oscillations around the local minimum.

1.8 Optimizer algorithms

On the surfaces which curves more rapidly in some dimensions then the other ones, the SGD struggles to find the minimum easily [6], see figure 1.4. For this reason, the momentum with a parameter of γ can be added to the algorithm as

$$u_n = \gamma u_{n-1} + \nu \nabla_\theta L(\theta_n) \tag{1.23}$$

$$\theta_{n+1} = \theta_n - u_n, \tag{1.24}$$

where u_n stands for the weight update in the n-th step. In practice, the parameter gamma is usually set to $\gamma \approx 0.9$ [6]. Using the momentum often accelerates the convergence and reduces the oscillations.

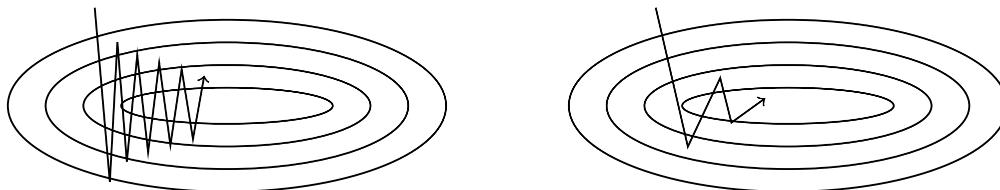


Figure 1.4: Left: SGD without momentum. Right: SGD using momentum.

On the other hand, with momentum the update does not slow down before it reaches the minimum and can therefore pass around it. One solution to this problem is the **Nesterov Accelerated Gradient (NAG)** which uses the approximation of the next weight value $\theta_n - \gamma u_{n-1}$ to calculate the weight update using

$$u_n = \gamma u_{n-1} + \nu \nabla_\theta L(\theta_n - \gamma u_{n-1}) \tag{1.25}$$

$$\theta_{n+1} = \theta_n - u_n. \tag{1.26}$$

Thus, the gradient is not calculated with respect to the current weights θ_n but to the approximation of their future value. This technique enhance especially the training of Recurrent Neural Networks but it applicable also in other architectures.

In the previous methods the learning rate is set the same for all the parameters and usually remains constant in time. The next generation of optimizers therefore tries to solve these problems by adapting the learning rate in the process of training. First, the **Adagrad** optimizer was introduced in order to help training on sparse data where some parameters were more important then the other ones. The update is done as

$$g_n^{(i)} := \nabla_\theta L(\theta_n) \tag{1.27}$$

$$\theta_{n+1}^{(i)} = \theta_n^{(i)} - \frac{\nu}{\sqrt{G_n^{(i)} + \epsilon}} g_n^{(i)}, \quad G_n^{(i)} := \sum_{j=1}^{n} \left(g_j^{(i)}\right)^2 \tag{1.28}$$

where $\epsilon \approx 10^{-8}$ prevents division by zero. Here the problem is that the $\sqrt{G_n^{(i)} + \epsilon}$ is monotone increasing and therefore the learning rate is decreasing and can stop learning some parameters completely. There is a modification of Adagrad called the **Adadelta**, where a decay of ρ is added to the sum of squared gradients making the learning rate decay less aggressive. The algorithm uses equations

$$g_n^{(i)} := \nabla_\theta L(\theta_n^{(i)}), \tag{1.29}$$

$$v_n^{(i)} = \rho v_{n-1}^{(i)} + (g_n^{(i)})^2 (1 - \rho), \quad v_0^{(i)} := 0, \tag{1.30}$$

$$\Delta\theta_n^{(i)} = \frac{\sqrt{u_{n-1}^{(i)} + \epsilon}}{\sqrt{v_n^{(i)} + \epsilon}} g_n^{(i)}, \tag{1.31}$$

$$u_n^{(i)} = u_{n-1}^{(i)} \rho + (1 - \rho)(\Delta\theta_n^{(i)})^2, \quad u_0^{(i)} := 0, \tag{1.32}$$

$$\theta_{n+1}^{(i)} = \theta_n^{(i)} - \gamma \Delta\theta_n^{(i)}. \tag{1.33}$$

Similarly to the Adadelta algorithm, **RMSprop** proposed by G. Hinton also tries to deal with the vanishing learning rate problem. The update is done by

$$g_n^{(i)} := \nabla_\theta L(\theta_n^{(i)}), \tag{1.34}$$

$$v_n^{(i)} = \alpha v_{n-1}^{(i)} + (1 - \alpha)(g_n^{(i)})^2, \quad v_0^{(i)} := 0, \tag{1.35}$$

$$\theta_{n+1}^{(i)} = \theta_n^{(i)} - \gamma \frac{g_n^{(i)}}{\sqrt{v_n^{(i)} + \epsilon}}, \tag{1.36}$$

where α plays the same role as ρ in Adadelta algorithm.

Finally, the **Adaptive Moment Estimation (Adam)** algorithm adds a modified momentum strategy to the Adadelta or RMSprop algorithms. In addition to the ρ parameter of Adadelta (labeled as β_2), the momentum parameter of decay is labeled as β_1. The m_n and v_n defined as

$$m_n^{(i)} = \beta_1 m_{n-1}^{(i)} + (1-\beta_1)(g_n^{(i)}), \quad v_0^{(i)} := 0, \tag{1.37}$$

$$v_n^{(i)} = \beta_2 v_{n-1}^{(i)} + (1-\beta_2)(g_n^{(i)})^2, \quad v_0^{(i)} := 0, \tag{1.38}$$

$$\tag{1.39}$$

can be considered as the first and second uncentered momenta respectively. The update is then made as

$$\theta_{n+1}^{(i)} = \theta_n^{(i)} - \frac{\nu \widetilde{m}_n^{(i)}}{\sqrt{\widetilde{v}_n^{(i)}} + \epsilon}, \quad \widetilde{m}_n^{(i)} := \frac{m_n^{(i)}}{1-\beta_1^n}, \ \widetilde{v}_n^{(i)} := \frac{v_n^{(i)}}{1-\beta_2^n}. \tag{1.40}$$

The authors of Adam recommend default values as $\beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

1.9 Data augmentation

The quality of dataset is crucial to train a well performing neural network. However, in practice the datasets are usually small and expensive to create. The performance quality of neural networks can be improved by training them on multiple different representations of the same training data. The process of creating new representations is called **data augmentation**.

On the image data a common augmentation technique is the rotation, color modification, warping, cropping, scaling etc. However, for the signal data most of these methods cannot be applied. Instead, there are several other techniques. For a computational simplification only a few of them will be used in this paper.

One of the simplest methods is the **noise addition**. The Gaussian noise can be added to the signal or to the spectrogram when considering using time-frequency transformations. When dealing with multi-channel signals, **channel swapping** can be used to In the multi-channel audio the order of the channel may not be important. In that case the channels can be swapped for example with a probability of 0.5. In this paper only one channel networks will be examined. Therefore, a channel is always taken randomly.

A simple augmentation method is the **Gain**. Here the signal is amplified using a value taken for a logarithmical decibel scale. In this paper a value is taken from $\mathcal{U}_{[-15,15]}$.

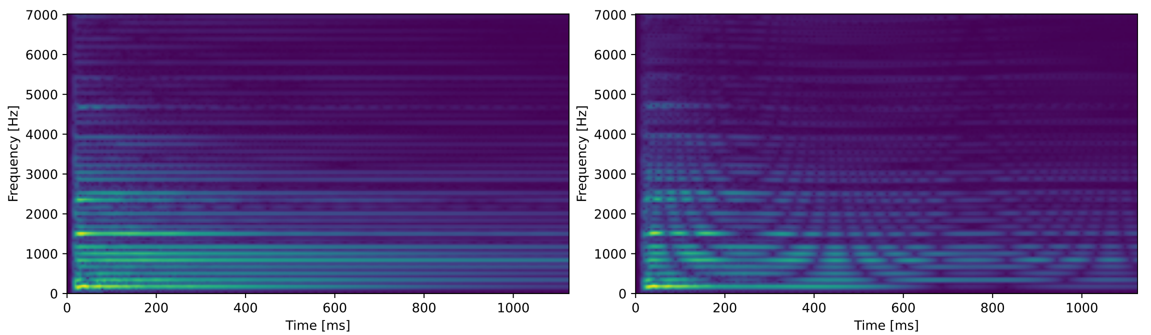


Figure 1.5: Spectrogram of the e2 piano tone before (left) and after (right) the application of a chorus effect.

Next audio effect is the **Chorus**. Naturally this effect happens when multiple sources produce the same (or very similar) signal. For data augmentation multiple copies of a signal are summed together with a small delay (usually between 15-35 ms). Example of this effect is depicted in Figure 1.5

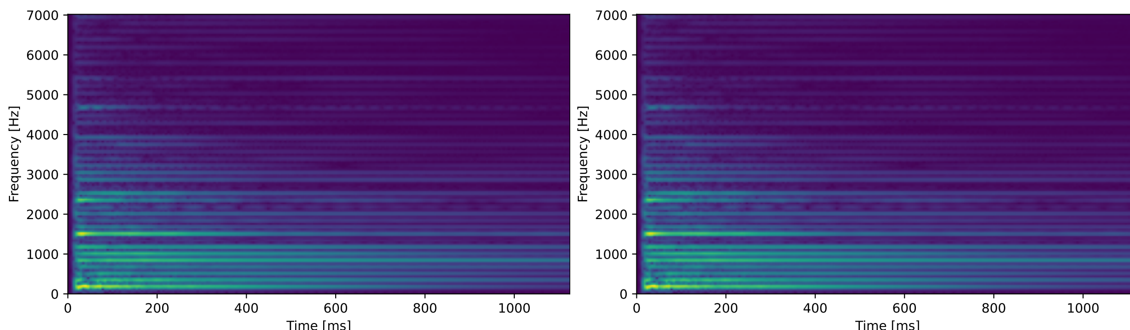


Figure 1.6: Spectrogram of the e2 piano tone before (left) and after (right) the application of a reverb.

Finally, the **Reverb** represents a persistence of sound created by reflections, e.g. in a room. This effect depends on many factors, the shape and size of a room, objects present in a room and many air conditions. The `Pedalboard` library developed by Spotify allows to model these conditions digitally and add this effect to the signal. In this paper data will be augmented using parameters room size, damping, wet level, dry level, and freeze mode. All these parameters are in interval $[0, 1]$, see Figure 1.6.

1.10 Time-frequency transformations

There are many tasks where the signal first needs to be transformed into some time-frequency representation. In this paper many neural networks are operating on these representations since it introduces many benefits. As an example, 2D convolutional layers can be applied on these tensors (2D plus a channel axis) which can enable many useful feature extractions.

STFT

One of the main transforms used for discrete signals is the **Short-time Fourier transform** (STFT) defined as

$$\mathrm{STFT}[x(n)](m, \omega) =: X(m, \omega) = \sum_{-\infty}^{\infty} x(n)w(n-m)\mathrm{e}^{-\mathrm{i}\omega n}, \tag{1.41}$$

for a signal $x(n)$ and a window $w(n)$. In practice, the FFT algorithm is used to increase the speed. The resulting shape depends on the used window. As an example, the `librosa` library in Python uses the symmetric *Hann window* defined as

$$w(n) := \frac{1}{2} - \frac{1}{2}\cos\left(\frac{2\pi n}{M-1}\right), \qquad 0 \leqslant n \leqslant M-1, \tag{1.42}$$

where M denotes the window length. The **spectrogram** $S(m, \omega)$ is then defined as

$$S(m, \omega) := |X(m, \omega)|^2. \tag{1.43}$$

MEL spectrogram

The spectrogram representation has one potential problem when working with musical data. Humans do not perceive frequency linearly, but logarithmically. For this reason, to make a representation closer to human perception, the frequency scale (in Hz) to Mel scale as

$$M(f) := 1127 \ln\left(1 + \frac{f}{700}\right), \qquad M^{-1}(m) = 700\left(\exp\left(\frac{m}{1127}\right)\right). \tag{1.44}$$

Another issue is that even the amplitude is being perceived logarithmically so it has to be scaled as well.

First, the STFT S of the signal is extracted and the amplitude is transformed to the decibel spectrum as

$$D_{ij} := 20 * log_{10}(S_{ij}^2). \tag{1.45}$$

Then the lowest and highest considered frequency of the original representation is taken and converted into mels. A list of all used mels is created by simply equidistantly fill the interval $(m_{\min}, m_{\max})$ using a chosen number of mel bands B (usually between 40-128),

$$m_{\min} < m_1 < m_2 < ... < m_B < m_{\max}, \qquad m_i = m_{\min} + \frac{i(m_{\max} - m_{\min})}{B+1}, \ \forall i \in \{1, ..., B\}. \tag{1.46}$$

These mel points are then converted to Hz as

$$h_i := M^{-1}(m_i), \quad \forall i \in \{1, ..., B\} \tag{1.47}$$

and rounded down to the nearest STFT bin using a formula

$$f_i := \left\lfloor \frac{(\text{n_fft} + 1)h_i}{\text{sample_rate}} \right\rfloor, \quad \forall i \in \{1, ..., B\}, \tag{1.48}$$

where n_fft stands for the length of the windowed signal after padding with zeros (if needed). The next phase is to create triangular filter matrix as

$$\mathbf{H}_{mk} := \begin{cases} 0, & k < f_{m-1}, \\ \frac{k - f_{m-1}}{f_m - f_{m-1}}, & f_{m-1} \leqslant k \leqslant f_m, \\ \frac{f_{m+1} - k}{f_{m+1} - f_m}, & f_m \leqslant k \leqslant f_{m+1}, \\ 0, & k > f_{m+1}, \end{cases} \qquad \forall m \in \{1, ..., B\},\ \forall k \in \{1, ..., s_f\},\ s_f := \left\lfloor \frac{\text{n_fft}}{2} + 1 \right\rfloor, \tag{1.49}$$

where $f_0 := f_{\min}$ and $f_{B+1} := f_{\max}$. These filter banks are then applied to the transformed STFT as

$$\text{MEL} := S \cdot H^T. \tag{1.50}$$

1.11 Autoencoders

According to the manifold hypothesis, the real world high-dimensional data are just a projection from a low-dimensional space. Lowering the dimensionality of data can be done by many approaches, e.g. by principal component analysis (PCA). In last years research showed that this can be also achieved by neural networks which can better specialize to a chosen dataset. This task belongs to the unsupervised learning, i.e. a task which only has unlabeled data.

Each autoencoder can be defined as a combination of two functions. The first one called the **encoder** $\mathcal{E}$ is used to compress input data $\mathbf{x} \in \mathbb{R}^n$ into a smaller representation $\mathbf{y} \in \mathbb{R}^m$,

$$\mathcal{E} : \mathbb{R}^n \to \mathbb{R}^m, \quad n > m, \quad n, m \in \mathbb{N}, \tag{1.51}$$

where $\mathbb{R}^m$ is called the **latent space**. Similar input tensors should also be similar in this latent space. The second function is called the **decoder** $\mathcal{D}$ and its purpose is to approximate an inverse function of $\mathcal{E}$,

$$\mathcal{D} : \mathbb{R}^m \to \mathbb{R}^n. \tag{1.52}$$

The goal of this chapter is to find neural networks $\mathcal{E}$ and $\mathcal{D}$ as

$$\underset{\mathcal{E},\mathcal{D}}{\operatorname{argmin}}\, \mathbb{E} L\big(\mathbf{x}, \mathcal{D}\big(\mathcal{E}(\mathbf{x})\big)\big), \quad \mathcal{E}(\mathbf{x}) \in \mathbb{R}^m, \tag{1.53}$$

where L is a reconstruction loss function, m is a number of dimensions in the latent space, and $\mathbb{E}$ is an expected value over the distribution of $\mathbf{x}$ [7].

Autoencoders can be used for many purposes such as a dimensionality reduction, data compression, noise reduction, feature extraction, etc. One of the simplest autoencoder neural network is a simple FFNN which includes a hidden bottleneck layer, i.e. a layer with a small number of neurons, see Figure 1.7. The output vector must have the same size as the input vector. The first part of the network can be considered as an encoder and the rest of the network as a decoder. This network is trained so that the input vector $\mathbf{x}$ represents also the output vector. With this approach the network learns how to make a representation of the input vectors which would have a dimensionality of the bottleneck layer.

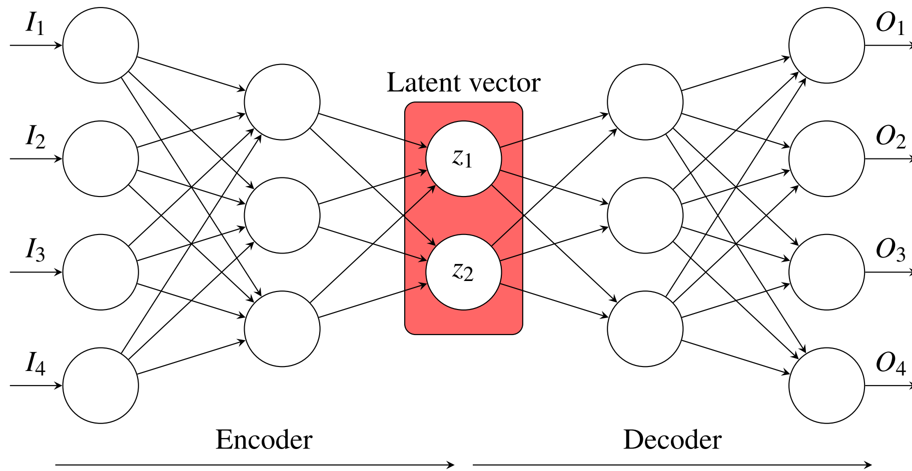


Figure 1.7: Simple autoencoder architecture.

Variational Autoencoders (VAE)

Autoencoders may be also used for generating new data. In that case a randomly generated vector from the latent space is evaluated by the decoder network. However, in practice the latent space usually is not continuous since the encoder tries to make the latent vector easy to decode. This may cause problems since the decoder is not able to decode combinations of the latent vector which were not part of the training. In this case the decoded results would not make any sense.

A way how to solve this problem is to use Variational Autoencoders. Instead of predicting the latent vector directly, a normal distribution of the latent vector is predicted, see Figure 1.8. The latent vector is then generated from this distribution. The decoding part then works the same as in the simple autoencoder.

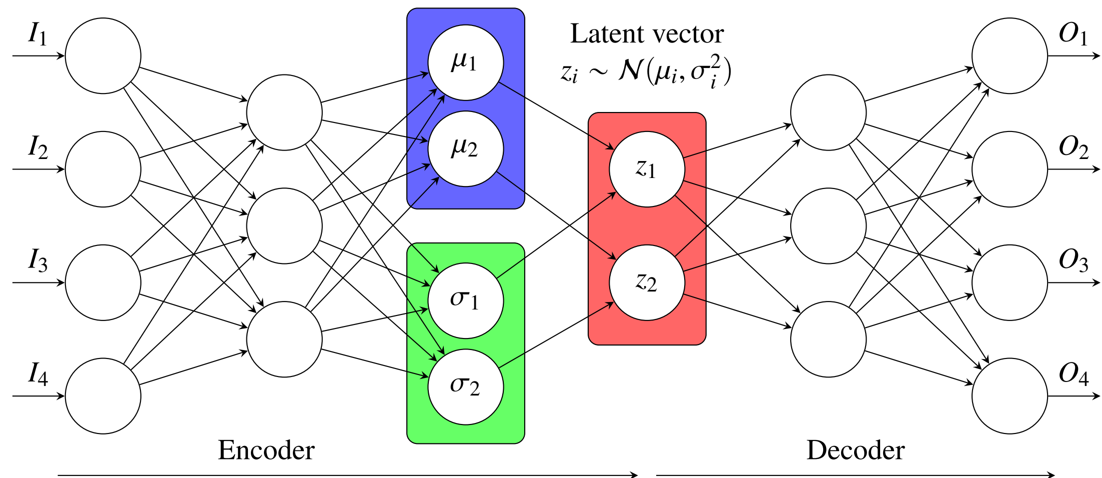


Figure 1.8: Simple VAE architecture.

In order to keep the data continuous, it is beneficial to add an auxiliary loss in order to keep the latent distribution closer to $\mathcal{N}(0,1)$ distribution. This additional loss is based on Kullback–Leibler divergence,

$$\mathbb{KL}\left(\mathcal{N}(\mu,\sigma) \parallel \mathcal{N}(0,1)\right) = \sigma^2 + \mu^2 - \log\sigma - \frac{1}{2}. \tag{1.54}$$

With this approach the latent space is more smooth and compact since the additional loss keeps latent vectors to be close to each other.

Chapter 2

Neural Network Architectures

Over the past years many convolutional neural networks (ConvNets) were designed and used for many tasks including image recognition, object detection and time series classification. There are many datasets available for the image recognition task. One example is the CIFAR-10 which is a dataset consisting of 60000 32x32 RGB images uniformly distributed in 10 classes. Another famous example is the ImageNet dataset [8] which consists of 1.2 million images in 1000 classes. There are many research teams focused on developing more sophisticated networks which can be used for these classification tasks.

Most of the state-of-the-art networks are well documented and therefore became popular among developers and scientists around the world. Comparison of chosen networks is shown in Figure 2.1. This graph represents the test accuracy on the ImageNet dataset. Some of the illustrated architectures will be described in detail in this chapter since they introduce many useful concepts needed for this paper. Furthermore, some of the architectures used in this paper are applied to the spectrogram data, which allows using some of the Image Classification modules directly.

2.1 Development summary

The approach of network development changed a lot in the last years. In the early 21st century the technological progress did not allow training deep ConvNets so the networks tended to be shallow with less neurons in each layer. This was caused by huge computational and memory demand of training these networks. The architecture structure also changed a lot. Until the Inception network appeared in 2014, all the state-of-the-art ConvNets consisted of stacked convolution and pooling layers followed by dense layers where the last layer used the softmax activation. A modification of this idea was to stack more convolutional layers together without any pooling layers between them. The motivation behind this was that adding more non-linearities to the model could possibly increase the generalization ability in complex data. This all approach led to the creation of VGG-16 and VGG-19 models which use more than 138 million parameters and became one of the biggest networks used for the ImageNet classification task.

With the Inception a whole new era of model development began. VGG models have too many parameters to be simply used on low-cost devices. The Inception network includes modules called the *Inception modules* which use 4 parallel threads of convolutional and pooling layers. With this approach the network not only have the *depth* but also a *width*, i.e. how many parallel threads there are in the used modules. The Inception-v1 (also called GoogLeNet) had a little lower accuracy than the VGG network, however, with just about 5 million parameters compared to 138 million of

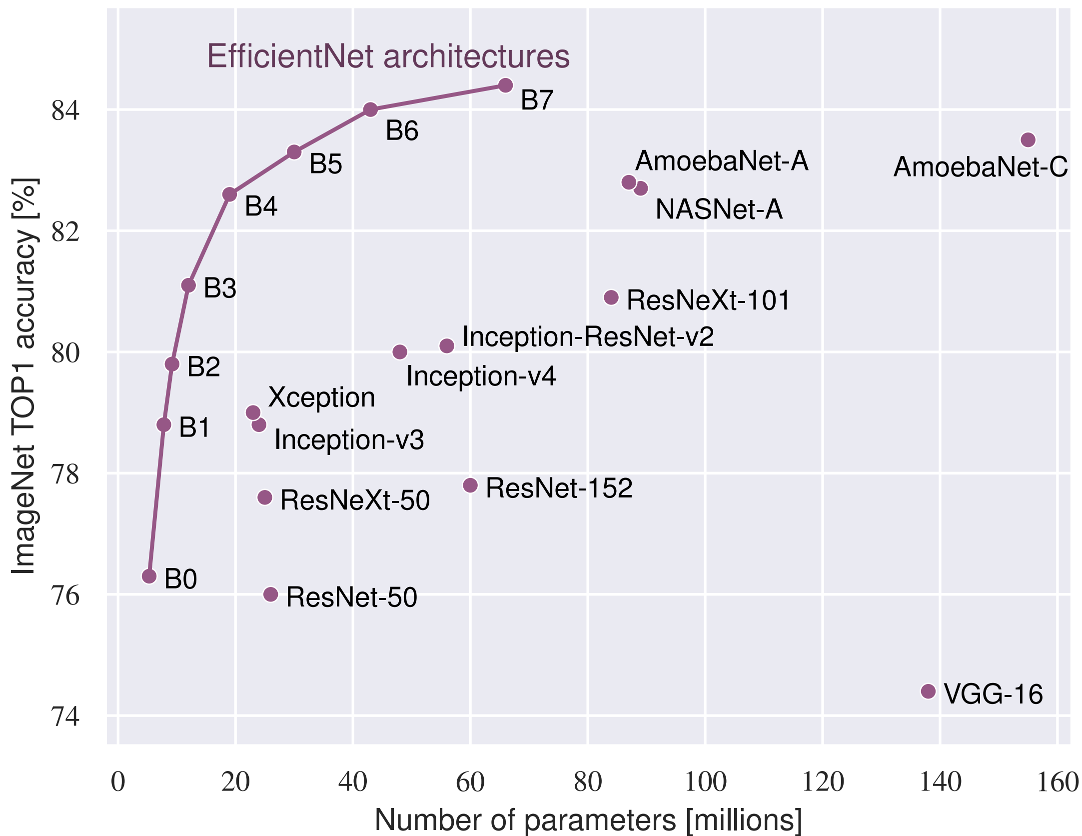


Figure 2.1: Comparison of chosen ConvNet architectures.

the VGG (about 4%). This opened the deep learning for mobile devices and low-cost hardware as it was now possible to train a small network which had almost the same accuracy as the VGG.

Another milestone of the ConvNet development was the usage of *residual learning* in the ResNet architecture which introduced using shortcuts in the model, i.e. threads with no transformation. The idea behind this method was to solve the *degradation problem* which happens in very deep ConvNets. The suitability of this method was proved in many experiments from many research teams around the world. One of them was Szegedy et al. who besides other things created and compared two models, Inception-v4 and Inception-ResNet-v2, where only the second one used residual learning while the first one did not.

In 2018 the NASNet architecture was created using the *NASNet Search Space* which was created to lower down the number of possible architectures. The idea behind this was to create a search space with pre-defined option for the used transformations and the structure of used modules so that the architecture can be created using a search algorithm. This space was also used to design a network using an evolutionary algorithm. With this method the AmoebaNet architecture was created and slightly overcame the NASNet accuracy while using just a fraction of computational time.

In 2020 the network family called EfficientNets were created in order to find a balance between the model depth, width and input image resolution. In order to make the networks usable in mobile devices 7 architectures were created so that every developer/scientist could decide which version of EfficientNet is appropriate for his task.

2.2 Architecture visualization

Since there are many various network architectures, we invented a method how to visualize them. The example of a module is shown in Figure 2.2. A prolonged hexagon stands either for a convolutional layer or a pooling layer with a core shape written in the bottom of each hexagon.

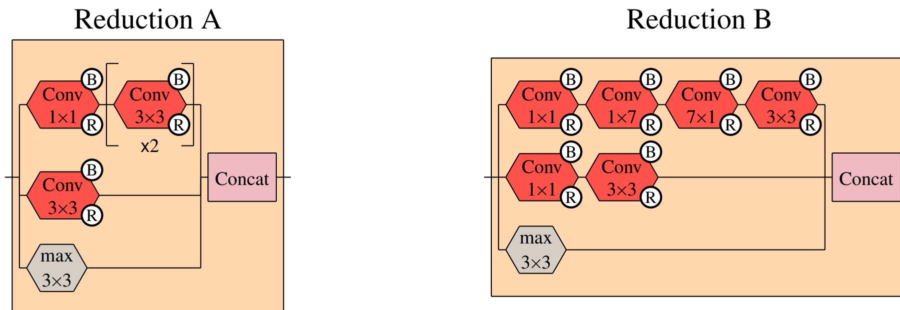


Figure 2.2: Examples of the module visualization.

The convolutional layer is colored red and always includes the information about the activation function and batch normalization in the adjacent circles. Here R stands for ReLU and T for Tanh activation functions whereas B indicates using the batch normalization. If two circles are shown, the top one is always applied before the bottom one. For example, in Figure 2.2 in all of the convolutional layers the batch normalization is applied before the activation function.

The pooling layers colored gray include information about the pooling type where *avg* stands for average pooling, *max* for maximum pooling and the "glob avg" for global average pooling [9]. The cell repetition is depicted by square brackets with the number of repetition written below the repeated cells. There are also two different kind of cells, the *Concat* cell depicted in a rectangle, which stands for the tensor concatenation (stacking tensors next to each other along the depth axis). The other cell is called the *Add* cell which stands for summing the tensors element-wise.

An example of the network structure is shown in Figure 2.3. In addition to the module illustration all the networks also include the input layer and at least one dense layer. Below the input layer there is the input image shape, while below the dense layers the number stands for how many neurons are presented in each layer. Every dense layer can also include information about the used activation function, where R stands for ReLU, T for Tanh and S for Softmax activation function. The middle rectangle cells represent modules which are usually illustrated in another figures.

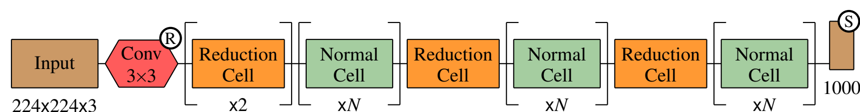


Figure 2.3: Example of the network visualization.

2.3 Early development

The first considered network is called the LeNet-5. This simple model developed in 1998 is composed of two convolutional layers stacked with pooling layers followed by three dense layers (see Figure 2.4). The number 5 in the name refers to the sum of its convolutional and dense layers. This is a common practice among many ConvNet developers. The *Tanh* function is used

as the activation function of both convolutional and dense layers [10]. All the layers combined have about 60000 parameters. Many following models are inspired by this network in stacking convolutional and pooling layers together followed by dense layers at the end of the model. The last layer uses the Softmax activation function with 10 neurons since it is used for the classification task with 10 classes.

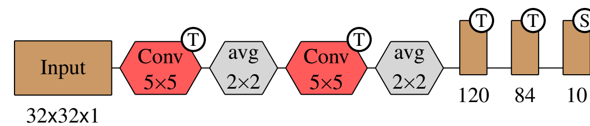


Figure 2.4: Structure of LeNet-5.

AlexNet

In 2012 the AlexNet was designed by Krizhevsky et al. This model is composed of 5 convolutional layers followed by 3 dense layers [11]. However, unlike the LeNet-5 architecture, not all the convolutional layers are stacked with pooling layers anymore (see Figure 2.5). AlexNet contains around 60 million parameters which is approximately 1000 times more than in LeNet-5. This made AlexNet one of the biggest ConvNet used for the image classification task. Training such a large network had a big computation and memory demand, therefore the ReLU activation function is used instead of Tanh [11].

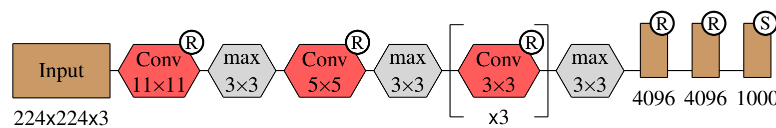


Figure 2.5: Structure of AlexNet.

Since there is around 60 million parameters, the problem with overfitting became serious despite the fact that the ImageNet dataset contains over a million images [11]. To deal with this problem, Krizhevsky et al. used many data ugmentation methods, such as random cropping or using dropout. The main idea behind dropout was to solve the problem of convergence to the global minimum instead of the local one. One approach would be to create several models of the same structure and to choose the one with the lowest loss value [11]. However, training a model of this size is a matter of days and therefore training more models would not be appropriate. A way how to deal with this problem is to apply dropout to some layers. When using dropout, each neuron has a 50% chance to set its value to zero and hence not contribute to the training process of a particular train data. This means that a different network is provided for each training sample but all these networks share weights together [11]. This causes that the network has to learn more general features from the training dataset.

Assessing the test dataset consisted of evaluating 10 crops, 4 corners plus the center crop together with their horizontal flipped versions, and making an average value of those values [11].

VGG-16

VGG-16 is a network developed for the ImageNet classification task by Karen Simonyan and Andrew Zisserman. Unlike the AlexNet which uses an 11x11 convolutional layer after the input layer, the VGG-16 begins with 3x3 convolutional layers together with max-pooling. Replacing a

huge convolutional layer by more 3x3 layers can lower the number of parameters while creating an output tensor with the same shape. Furthermore, this replacement also adds two additional non-linearities to the model which can lead to a better generalization of the network [12]. This approach is therefore used in many following state-of-the-art networks such as the Inception, ResNet etc.

Many variants of the VGG model were created in order to find the best performing network. The VGG-16 contains 13 convolutional, 5 pooling and 3 dense layers (see Figure 2.6). A bigger model called VGG-19 uses 3 additional convolutional layers [12]. One of the variants also contains a 1x1 convolutional layer. This can be understood as a transformation solely between particular channels in the input tensor. This approach is inspired by the Network-in-Network architecture created by Lin et al. in 2014 [9].

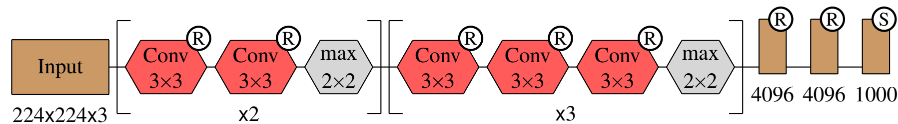


Figure 2.6: Structure of VGG-16.

The VGG training process followed the same principles as in AlexNet (data augmentation etc.) except for cropping the input images [12]. The first two convolutional layers also had a dropout value of 0.5 and the training started with a learning rate of 10^{-2}, however, when the validation accuracy stopped improving, the learning rate was decreased by a factor of 10 [12]. This situation happened 3 times during the training in 74 epochs.

Choosing the right weight initialization is also crucial for a proper training e.g. because of the vanishing or exploding gradient problem [13]. At first a shallow network was trained using a random weight initialization. After that more layers were added to the network using the $\mathcal{N}(0, 10^{-2})$ initialization while reusing weights from the already trained layers [12]. In 2022, the *Glorot* initialization is used to prevent overfitting and therefore it is possible to train a model without any need for pre-trained weights [14].

2.4 Inception

Theoretically, a well performing network can be created by simply adding more layers with more neurons to the model [15]. This works especially when a big number of training data is available. However, this construction is no efficient. With a bigger model there are many computational and memory problems together with increasing probability of overfitting [15]. In addition, it is often very hard and expensive to create a big dataset. This means that in practice there is a big need for an efficient utilization of computational resources, e.g. prevent too many weights to be close to zero [15]. Furthermore, in 2014 mobile technology was improving rapidly and therefore the need for a smaller efficient network increased even more.

To solve all these problems the *Inception* family of networks was developed by Szegedy et al. A famous configuration from this family with 22 layers is called the *GoogLeNet* [15]. Thanks to the more efficient structure only 5M parameters are needed to reach a comparable accuracy as the VGG-16 which uses staggering 138M parameters.

The idea behind the Inception architectures is to increase the network width in order to exploit the computational resources more efficiently. This led to creation of the Inception module depicted in Figure 2.7. Instead of using stacked convolutional and pooling layers, the input tensor is processed independently in four threads while keeping the same spatial dimensions [15]. The results of these

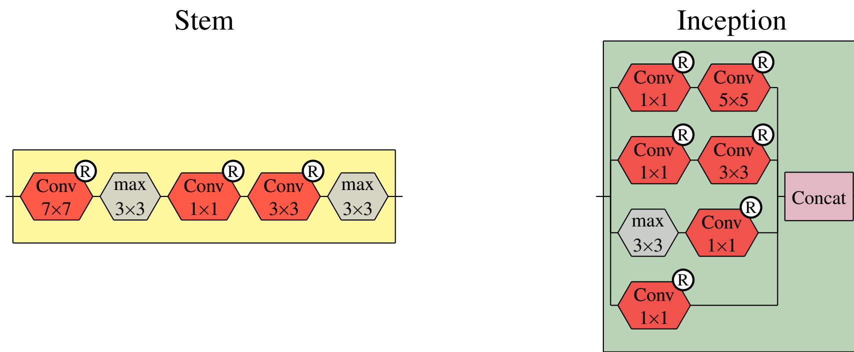


Figure 2.7: Structure of the Stem and Inception modules used in GoogLeNet network.

four transformations are then concatenated along the channel axes into one bigger output tensor. This approach also allowed constructing much deeper models. The next invention is the Stem module which works as a pre-processing unit of the input image [15]. This module, however, uses only one thread.

In order to prevent the vanishing gradient problem two auxiliary classifiers are added to the network [15]. They are only used in the process of training in order to deal with that problem and are removed after the network is fully trained [15]. The overall structure of GoogLeNet is depicted in Figure 2.8.

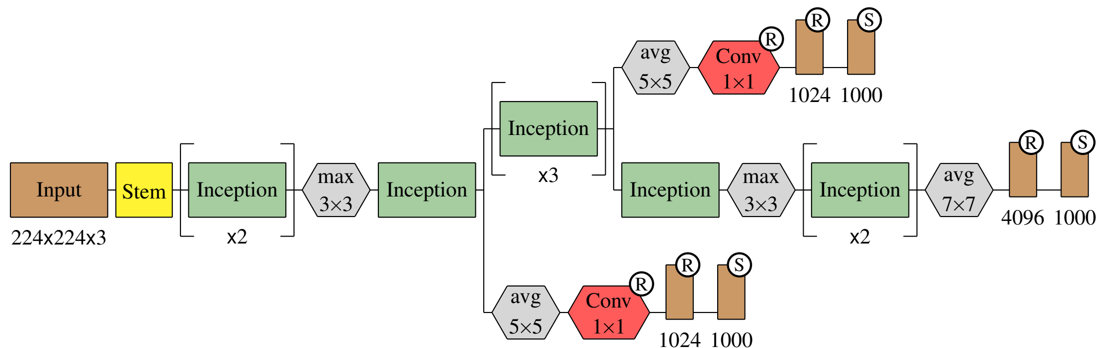


Figure 2.8: GoogLeNet architecture.

Inception-v3

In the next years, Szegedy et al. created several simple rules how to create a good-performing network [16]. First, none of the layers should be so small that it would lead to a loss of information needed for a proper classification. Second, the representation size should gradually decrease between the first and the last layer. The representation structure is also important since it can contain an additional information about correlation etc. Spatial aggregation such as a 3x3 convolution is performing better on low-dimensional data. This means that using a pooling layer before a convolutional one is usually a more efficient way how to process data [16]. There should also be a balance between the network width and depth. Increasing both at once can increase the network accuracy, however, it could also lead to unbearable computational demand [16].

Large convolution filters have a big computational demand since a lot of multiplication has to be done each time it is being used. However, some of these filters can be successfully replaced with several smaller ones. As an example, a 5x5 convolutional layer can be replaced with two

3x3 layers (see Figure 2.9). Analogically, 3x3 filters can be split into 1x3 and 3x1 filters saving about 33% of the memory resources (see Figure 2.9). It can reduce the computational and memory demands, however, a downside of this method is that not all the kernels can be separated in this way. One of the most famous examples of kernels which can be spatially separated is the *Sobel* kernel,

$$S := \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}. \tag{2.1}$$

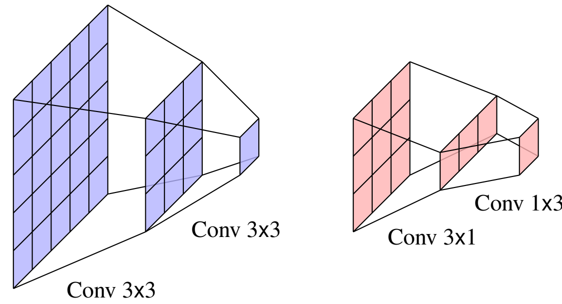


Figure 2.9: LEFT: replacing a 5x5 *convolutional filter with two* 3x3 *filters. RIGHT: replacing a* 3x3 *filter by a combination of* 3x1 *and* 1x3 *filters.*

Szegedy et al. experimentally found out that this approach works good only on medium sized kernels [16]. For example, a combination of 1x7 and 7x1 filter can contribute to a better network accuracy. On the other hand, replacing a 3x3 layer by two 2x2 layers would save just around 11% of memory while negatively affecting the network accuracy.

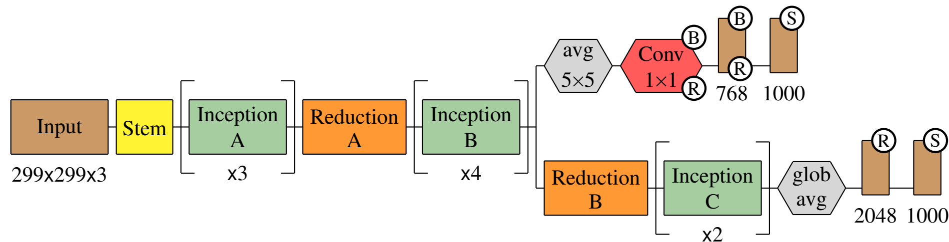


Figure 2.10: Structure of Inception-v3.

In the next generation of Inception all the 5x5 convolutional layers are replaced by two stacked 3x3 layers and 7x7 layers by three 3x3 layers. The overall structure created following the principles mentioned above is depicted in Figure 2.10. Furthermore, the Inception modules are shown in Figure 2.11 and reduction modules in Figure 2.12. Stem has the same structure as the one used in GoogLeNet, however, the 7x7 layer is replaced by three 3x3 layers and after each convolutional layer there is a batch normalization layer. In the Inception-v3 only one auxiliary classifier is used.

The final version of the inception-v3 model has a depth of 42 layers compared to 22 layers of the GoogLeNet version. However, the computational cost is only increased by a factor of 2.5 which makes Inception-v3 very efficient compared to e.g. VGG. It is also worth mentioning that the best network was trained using the *RMSProp* optimization method [17] with $\epsilon = 1.0$ and decay with a value of 0.9 [16] while the learning rate was set to 0.045 reduced every two epochs by a factor of 0.94. In addition, to stabilize the training, the gradient clipping [18] was used with a threshold of 2.0 [16].

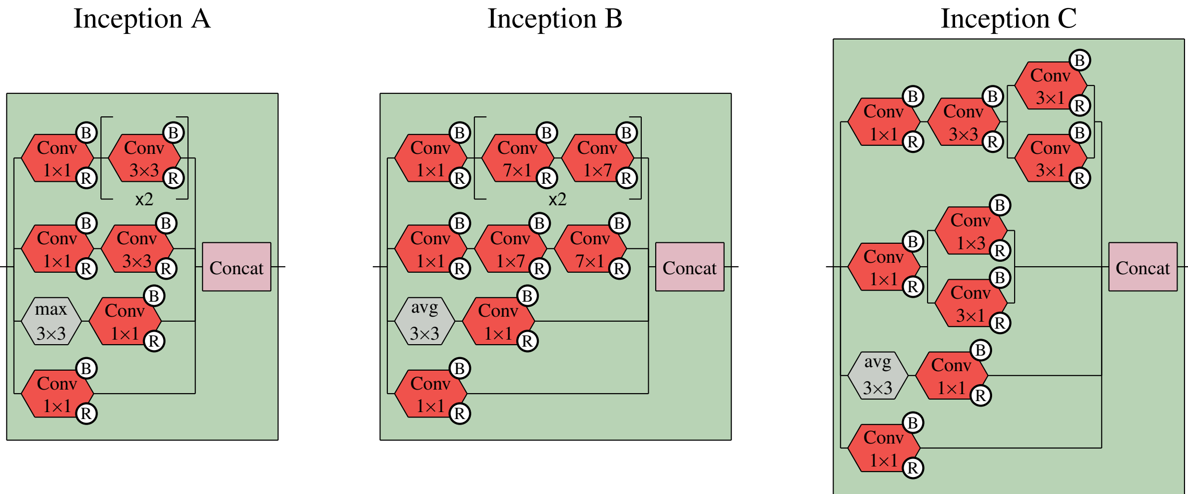


Figure 2.11: Structure of Inception modules used in the Inception-v3 network.

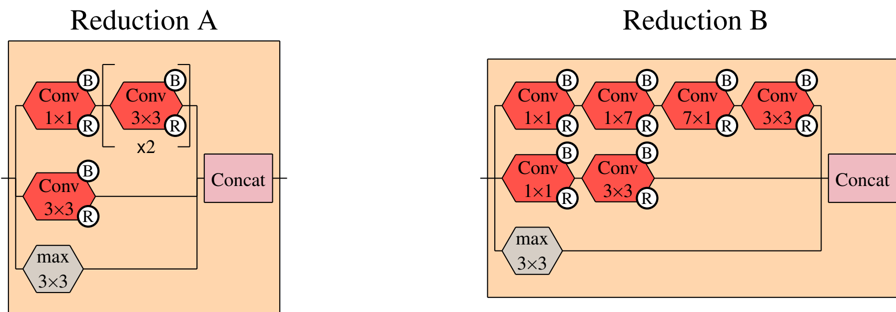


Figure 2.12: Structure of Reduction modules used in the Inception-v3 network.

2.5 ResNet

Authors of the previous architectures revealed that balancing a network length and depth is important in order to make the network good at the ImageNet classification task [15] [12]. Historically there was a problem with vanishing gradient which made deep neural networks learning unbalanced and led to worse predictions. This problem was, however, partially solved by better initialization strategies together with setting normalization layers inside the network [19].

When building deeper convolutional networks another problem soon appeared. While increasing the depth of these networks the accuracy constantly increases and then degrades significantly [19]. This problem is hence called the *degradation problem* (see Figure 2.13). Note that the error decreases around the 30000 iteration independently on the degradation problem. This can be cause e.g. by reaching another training epoch. The problem is that the 56-layer network has the training error higher than the 20-layer network. Unexpectedly, this problem is not caused by overfitting [19].

By adding more layers into the network, the training error should not be higher. This follows the fact that all those added layers can be identity layers and by training the error should not get higher in the process of learning.

He et al. solved this problem by introducing a *deep residual learning* framework [19]. In this approach *shortcut connections* (those skipping at least one layer) are used as a part of the ResNet

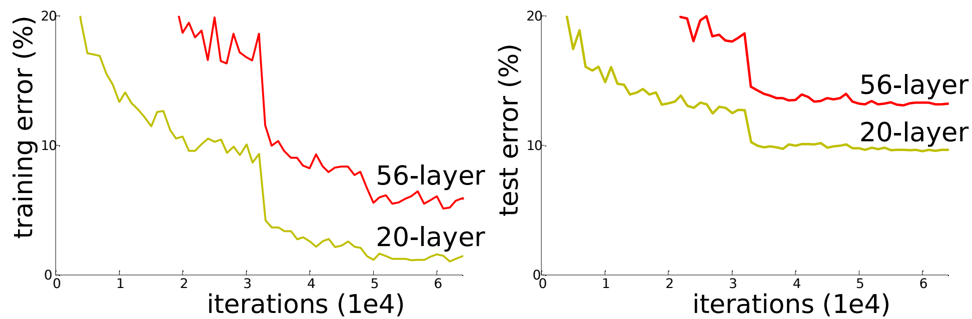


Figure 2.13: Training and testing error on CIFAR-10 dataset with plain (stacked ConvNet layers only) 20-layer network compared to a 56-layer network. Source: [19].

ID module shown in Figure 2.14. Here the original input tensor is added to the transformed tensor in order to simplify the weight optimization [19].

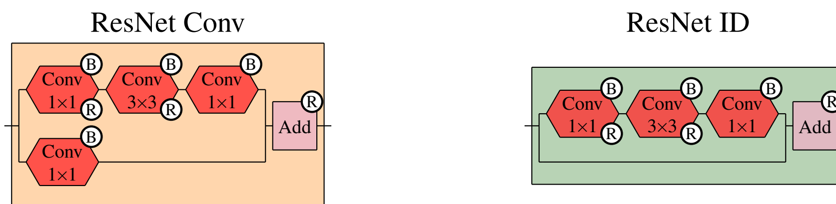


Figure 2.14: Structure of the ResNet Conv and ResNet ID modules.

This process does not increase the computational and the memory demand, since no new parameters are added to the network. A module which uses a residual learning can be written as $Y_i = \mathcal{F}(X_i) + X_i$, where X_i is the input tensor, Y_i is the output tensor and $\mathcal{F}$ stands for the operator. This also means that the X_i and $\mathcal{F}(X_i)$ dimensions must be the same. If not, one possible solution is to use a linear projection to match the dimensions [19]. The final architecture is shown in Figure 2.15.

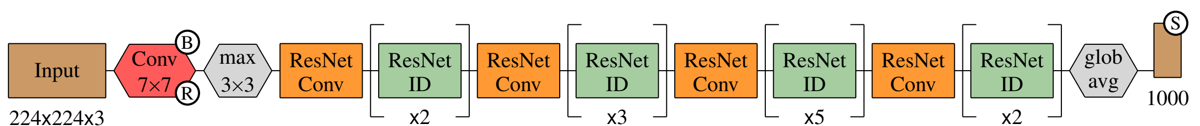


Figure 2.15: ResNet-50 architecture.

The experiments show that extremely deep networks using the residual learning are easier to optimize and therefore can profit from the large depth to decrease the training error [19]. A comparison of residual networks and plain networks (without the residual learning) with the same number of parameters is shown in Figure 2.16.

With this approach networks with 100 and even 1000 layers can be successfully trained with no problems connected to the degradation problem. This led He et al. to create extremely deep networks that won many computer vision competitions such as the ImageNet detection and localization tasks, the COCO detection task etc [19]. This can be the evidence that the residual learning principle can be used to many different tasks connected to deep learning methods [19].

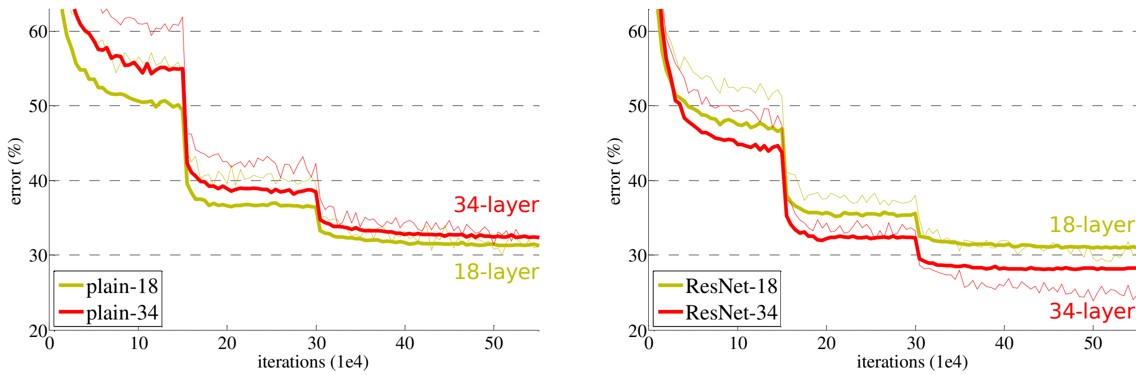


Figure 2.16: Training (thin) and validation (bold) error of plain and ResNet networks in the process of learning on the ImageNet dataset. Source: [19].

2.6 Inception-v4, Inception-ResNet-v2

Experiments made by Szegedy et al. do not fully support the necessity of using residual learning, however, they found out that using it can significantly improve the training time which is alone a great reason for its usage [20].

The next generation of Inception also profits from the hardware and software progress. In recent years the TensorFlow library came, besides other things, with memory optimization in backpropagation. This allowed Inception to have more complex structure because it was now possible to put a bigger model into the memory [20].

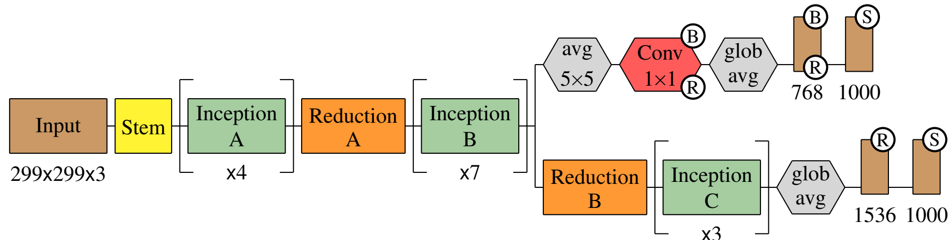


Figure 2.17: Inception-v4 architecture.

Two incarnations of Inception were created in order to compare a model which uses the residual learning against one which does not. The variation which does not use residual learning is called the Inception-v4 (see Figure 2.17). Inception A, B and C modules are shown in Figure 2.11, Reduction A and B are shown in Figure 2.12. The difference between the Inception-v3 and Inception-v4 networks is the number of used modules and complete remaking of the Stem module, which is shown in Figure 2.18.

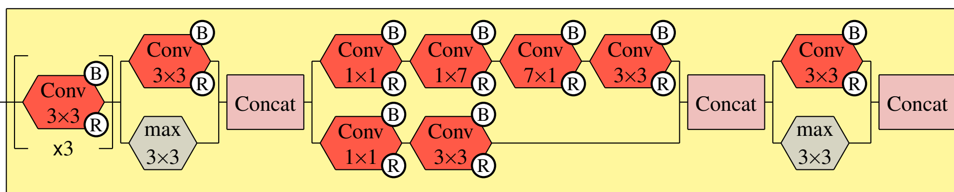


Figure 2.18: Stem module used in the Inception-v4 architecture.

The network which includes residual learning is called the Inception-ResNet-v2 (see Figure 2.19).

Inception ResNet A, B and C modules are shown in Figure 2.20. Reduction A and B and Stem modules have the same structure as of the Inception-v4 network and are shown in figures 2.12 and 2.18 respectively. Inception A module is almost the same as the one used in Inception-v4 with only one node changed [20].

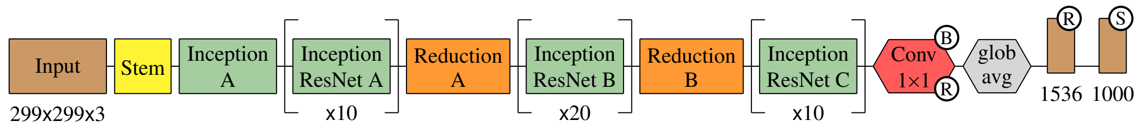


Figure 2.19: Inception-ResNet-v2 architecture.

Both networks are made with a similar computational cost. One of the differences is that in case of Inception-ResNet the batch normalization is not used in the Inception-ResNet modules (see Figure 2.20) in order to save the GPU memory, which then allows using more modules in one network. Szegedy et al. wanted to create these networks so that they can be both trained using a single GPU [20].

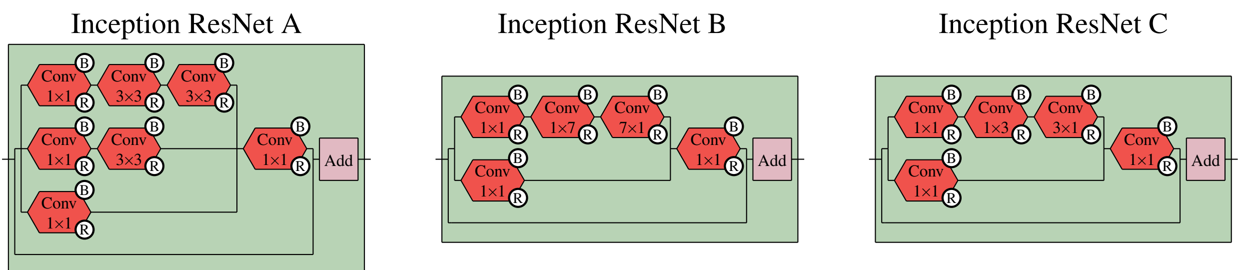


Figure 2.20: Structure of the Inception ResNet A, B and C modules.

When creating Inception networks, Szegedy et al. found out that if the number of ConvNet filters exceeds 1000, the network cannot be successfully train. This problem could not be solved using any used method [20]. The best result was reached using RMSProp optimization with decay of 0.9 and $\epsilon = 1.0$ [20]. The learning rate was set to 0.045 which was decayed with an exponential rate of 0.94.

2.7 ResNeXt-50

In recent models the difficulty of setting the right hyper-parameters increased because a new dimension was opened. The ResNet came with stacking modules of the same shape together [19]. This approach simplified the network construction by reducing the free choice of hyper-parameters. Xie et al. also argues that this step can prevent over-adaption of the network to a specific dataset [21].

On the other hand, Inception networks have demonstrated that a careful combination of all the used modules plays a big role in the model performance [15]. Szegedy et al. designed the model very precisely and customized the model in order to reach the best accuracy on the ImageNet dataset. However, with this approach it is not clear how to adapt the network parameters for a different dataset [21]. The motivation behind this work is therefore to merge the idea of simplicity by repeating layers with the idea of Inception modules with multiple threads [21].

In this work Xie et al. introduces the ResNeXt-50 network (see Figure 2.21) which consists of the ResNeXt Conv and ResNeXt ID modules (see Figure 2.22). These modules are comprised of a big number of identically constructed threads and one additional thread which is different [21]. In the ResNeXt Conv module the last thread consists of a single 1x1 convolutional layer while in the ResNeXt ID module this thread is a shortcut connection inspired by the ResNet architecture [19]. Furthermore, the number of threads is understood as a hyper-parameter which has to be found in order to make the network better in prediction [21].

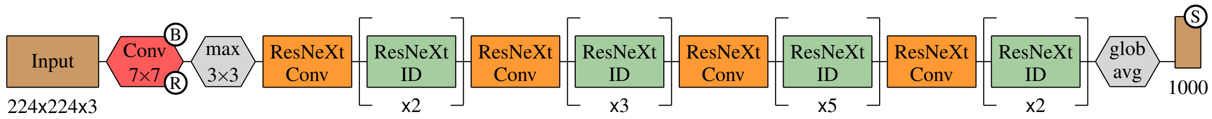


Figure 2.21: ResNeXt-50 architecture.

In the process of training the data augmentation was applied to the ImageNet dataset inspired by the AlexNet architecture [11]. In this process the input images were randomly scaled and cropped in order to artificially increase the dataset size. The ResNeXt-50 network was trained using the SGD optimization with a batch size of 256.

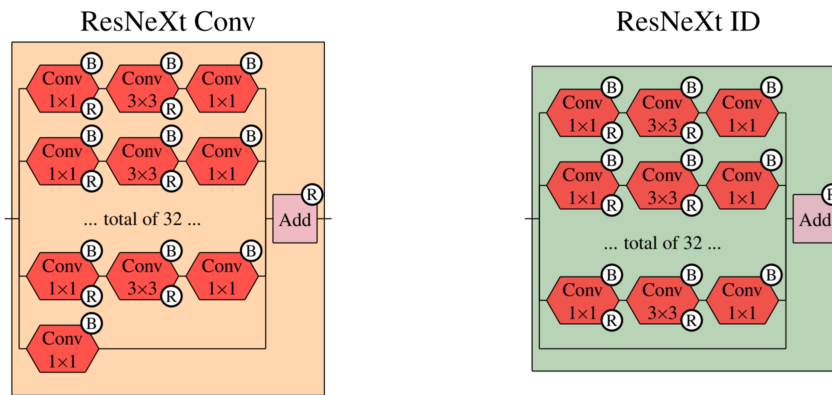


Figure 2.22: Structure of the Inception ResNeXt Conv and ResNeXt ID modules both with 32 identically constructed threads.

2.8 EfficientNets

Commonly used convolutional networks are being trained on one setting and the result network is then scaled up if more computational resources are available. As an example, the ResNet architecture can be scaled up from ResNet-18 to ResNet-200 simply by stacking more modules together [19]. To this moment there were many ways how to scale a network, however, most of them did not improve the accuracy efficiently. The vast majority of them focused solely on scaling up either the model depth or the input image resolution [22]. The problem is that trying to scale up more dimensions at the same time manually requires a lot of work without reaching the optimal accuracy [22].

This problem led Tan & Le to create a whole new method that can be used for this purpose. Surprisingly, a constant scaling seems to provide excellent results [22]. With using 2^N more computational resources, the network depth should be increased α^N times, the width β^N times and the image resolution γ^N times, where α, β, γ are constant coefficients which can be found on the

basement model using a grid search. The idea behind this *compound scaling* is that for a bigger image more layers are needed in order to extract all the necessary features, hence a larger network has to be created [22].

With this simple method a network can be efficiently scaled up in order to fully utilize the available computational resources. However, this process still heavily depends on the basement network. Tan & Le therefore used a network search [23] in order to create a baseline model for a whole family of models called EfficientNets. The best model called EfficientNet-B7 then surpasses the state-of-the-art model called GPipe while using 8.4 times less parameters [22]. The EfficientNet-B4 has similar number of parameters as the ResNet-50 while having the TOP-1 accuracy of 83% compared to 76.3% of the ResNet-50.

Scaling the network width, depth and resolution is a non-trivial task because they highly depend on each other. Many ConvNets have already been created using the depth scaling, however, scaling both width and resolution was common only in small networks used e.g. in mobile devices [22]. The intuition behind the depth scaling is that deeper models could learn more features and especially in high resolution input images they can look at the input from more perspectives extracting more complex features. Scaling a depth of a base network is shown in Figure 2.23. Deep networks have more problems with vanishing gradient, even though many techniques were invented to deal with this problem, e.g. batch normalization and shortcut connections. One example of this problem is that ResNet-1000 reaches a similar accuracy as ResNet-101, even though it uses 10 times more parameters [22].

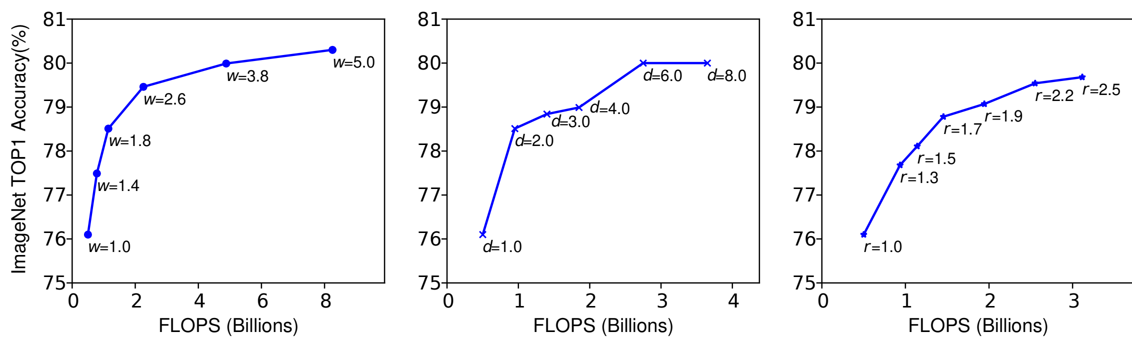


Figure 2.23: Scaling a network with different width (w), depth (d) and resolution (r) coefficients. Source: [22]

Width scaling many times leads to networks which have problems with capturing high-level features [22]. On the other hand, they are usually easier to train. The resolution scaling may add more information to the model and therefore can lead to better models. Resolution scaling is shown in Figure 2.23, where $r = 1$ stands for 224x224 resolution (e.g. $r = 2$ would refer to a resolution of 448x448 etc).

One hypothesis derived from the above analysis is that scaling a model in just one dimension increases the model accuracy, however, only until the scaling coefficient does not reach some value. After that the network saturates and increasing that dimension further does not improve the network accuracy anymore. This hypothesis corresponds with experiments made by Tan & Le depicted in Figure 2.24. In this experiment the width coefficient is scaled with different depth and resolution coefficients. Scaling all three parameters seems to produce a model with the highest accuracy.

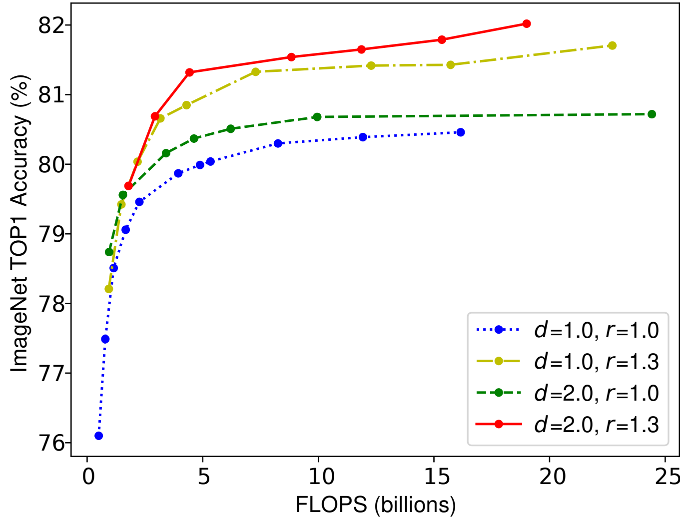


Figure 2.24: Scaling network width with different coefficient configurations. Source: [22]

Compound scaling method

Compound scaling method was invented to balance the coefficients of width w, depth d and resolution r by using a *compound coefficient* θ [22] as

$$d = \alpha^{\theta}, \qquad w = \beta^{\theta}, \qquad r = \gamma^{\theta}, \qquad \alpha\beta^2\gamma^2 \approx 2, \tag{2.2}$$

where $\alpha \geqslant 1$, $\beta \geqslant 1$ and $\gamma \geqslant 1$ are found by a grid search [22]. The equation $\alpha\beta^2\gamma^2 \approx 2$ is arbitrarily added since doubling the depth coefficient doubles the number of FLOPS, but the depth and resolution coefficients would increase it four times. With this approach the number of FLOPS is increased approximately 2^{θ} times after the number θ is chosen.

Since the scaling method does not change the baseline model architecture, building the model is also critical to create an appropriate network. The compound scaling method was applied on existing ConvNets [22] with successes, however, to fully demonstrate the power of this method, Tan & Le developed a whole new baseline architecture. The model structure was inspired by Architecture Search for Mobile (MnasNet) introduced by Tan et al. [24]. The optimization was made by searching for

$$\max \left[\text{Accuracy}(\mathcal{N})[\text{FLOPS}(\mathcal{N})/T]^{-0.07} \right], \tag{2.3}$$

where $\mathcal{N}$ is the basement model, T is the desired FLOPS value and -0.07 is a chosen hyperparameter balancing the accuracy with the FLOPS [22]. In the optimization of some models the latency is chosen instead of the FLOPS, however, the EfficientNet is not targeted to any specific device. After the basement model was created, the scaling coefficients were found as $\alpha = 1.2$, $\beta = 1.1$ and $\gamma = 1.15$. With them all the models were created (the full architecture documentation is specified in [22]).

2.9 InceptionTime

Despite many similarities between the Time Series Classification and Image Classification, the state-of-the-art architectures for these tasks did not have much in common [25]. This fact led Fawaz et al. to propose a convolutional network called the InceptionTime.

InceptionTime was created in 2020 inspired by the Inception-v4 model proposed by Szegedy et al. in 2016. Inception-v4 was chosen since it used to be a state-of-the-art architecture based on the residual learning. The main idea behind the InceptionTime is to apply multiple filters simultaneously to an input time series. Compared to the Image Classification task, one of the advantages in TSC is that the time series only have one dimension together with the channel one. This allows 1D convolutional filters to be much longer.

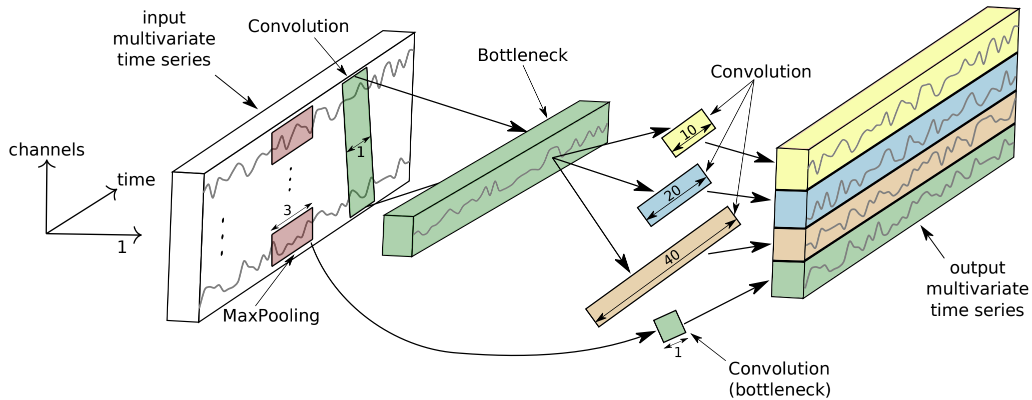


Figure 2.25: InceptionTime module diagram. Source: [25].

The InceptionTime module (see Figure 2.25) consists of two parallel threads. In the first one a simple 1-convolutional layer is applied to the input matrix along the channel axis in order to extract the cross-channel features. To the result matrix three convolutional layers composed of 32 cores with a size of $l \in \{10, 20, 40\}$ are applied along the time axis. In the second thread the max-pooling is applied followed by a 1-convolutional layer in order to reduce dimensionality [25]. All these result matrices are then concatenated into one output matrix. After that the batch normalization is applied followed by the ReLU activation. A simplified structure of the InceptionTime module is shown in Figure 2.26.

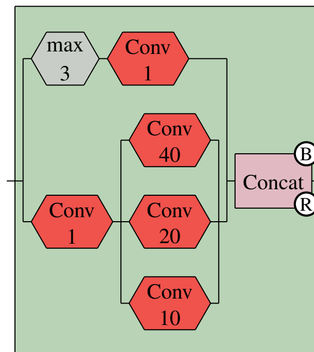


Figure 2.26: InceptionTime module.

The overall structure is highly inspired by the Network-in-Network, Inception-v4 [20] and ResNet [19] architectures. InceptionTime network consists of three stacked InceptionTime modules in parallel with one convolutional layer followed by a batch normalization. These result matrices are then summed element-wise into the output matrix. This whole process is repeated N times according to the time series complexity, see Figure 2.27.

Inspired by many other architectures [19], instead of using several stacked fully-connected layers after the last convolutional layer, the global average pooling is applied in order to significantly

reduce the dimension of the final output matrix. The last layer consists of Y neurons with the softmax activation, where Y denotes the number of classes in the classification task.

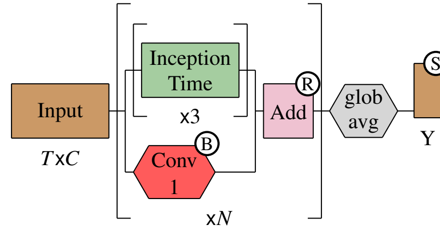


Figure 2.27: InceptionTime architecture.

Since the result network showed some instability in the predictions, Fawaz et al. decided to train five independent networks of the same architecture [25]. In the prediction process the input matrices are evaluated by all of them and these results are then averaged into the final prediction. This approach can help overcoming the weight initialization problem together with the loss function convergence to a local minimum [25].

Receptive field

Unlike the fully-connected networks, in ConvNets each output value always depend on some region in the input matrix [25]. This region is called the Receptive Field (RF) which is different for each neuron. In theory, the longer is the RF, the better should the neuron become in extracting longer patterns from the input series [25]. Consider sliding each convolutional core with a stride of 1. The RF can be then calculated as

$$\mathrm{RF}_d = 1 + \sum_{i=1}^{d} (k_i - 1), \tag{2.4}$$

where d denotes the network depth and k_i denotes the core length in the i-th layer. The equation (2.4) shows that in order to enlarge the RF value it is more efficient to increase the core length rather than to add more layers on top of the current ones.

The final InceptionTime network was trained using the Adam optimization introduced by Kingma & Ba [26] on multiple GPUs [25]. Compared to the HIVE-COTE model (state of the art model which does not use neural networks), the InceptionTime needs less time to be trained, especially on a big data task.

2.10 U-Net

In this paper there is also a need for being able to split a spectrogram into several parts in order to get individual signals. This will be done using image segmentation. Unlike image classification, the segmentation aims to create a soft mask over the input image which would pinpoint the location of examined data class. For this reason, the output of these neural networks is a tensor with values in range of $[0, 1]$. This mask may be smaller than the input image because of a potential scaling or cropping.

One of the most known architecture still used these days (2022) is called the **U-net** [27]. This network introduced by Ronneberger et al. in 2015 outperformed the previous state-of-the-art network which used a sliding-window approach. Another advantage is that this network is fully convolutional, i.e. does not need any additional transformations etc.

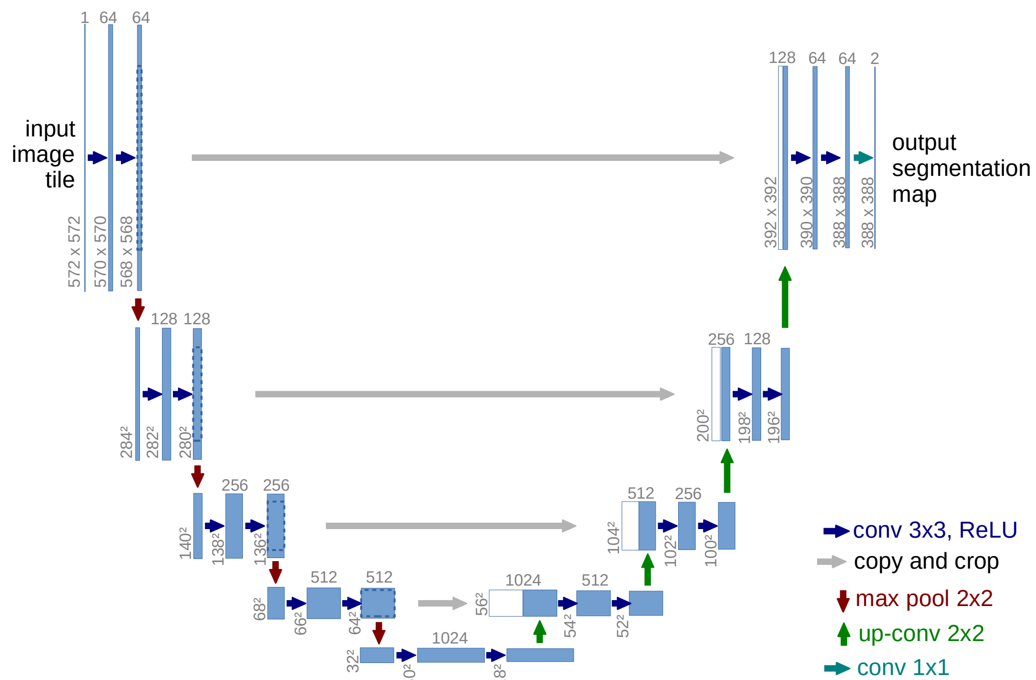


Figure 2.28: U-net architecture. The x and y size is shown on the left of each feature map while the number of channels is denoted on top of them. Source: [27].

The network consists of two main parts, see Figure 2.28. First, the input tensor is being processed by both convolutional and pooling layers in order to determine rough position of the class objects in the tensor. Each convolutional layer is followed by ReLU activation. This is possible since this approach highly increases the receptive field of the first part of the network and therefore can process bigger context of these objects.

In the second part, the produced tensor is being repeatedly upscaled, processed by convolutional layers and concatenated with properly cropped previous layers. This approach allows to better predict details in the object class location. Finally, a 1x1 convolution is used in order to produce a desired number of output channels. The result is therefore a segmentation mask with the same number of channels as the classes which are being predicted.

Overall, this architecture allows to efficiently predict the segmentation mask. Since the network is small compared to architecture like VGG-16 etc., the computational demand is not high and both training and inference is very fast (around 1 second per image as for 2015 [27]). The architecture is built so that only the middle area of the input image can be mapped to the output. This is caused by non-padding convolutions used inside the network. In order to fully process the input tensor, the context of the borders has to be added using e.g. a mirror padding, see Figure 2.29.

The usage on signal separation task is very similar. First, the STFT is applied to the input signal. Then either all the information or just the magnitude is taken and processed using the U-net. Experiments made by Jansson et al. show that this approach gives sufficiently good results [28]. The resulting soft mask is then multiplied piece-wise with the input tensor.

In Jansson's version of U-net the network uses batch normalization, 50% dropout in the first

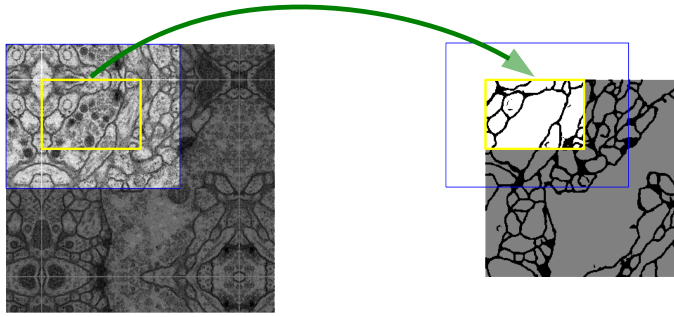


Figure 2.29: Mirror padding added in order to allow processing all the input image. Source: [27].

three layers, and Adam optimizer. The signal is also downsampled before processing in order to minimize the computational demand and the magnitude is normalized to the range of $[0, 1]$. The phase is not modified in the whole process.

With only 10 hours of training, the original U-Net managed to reach an Intersection over Union (IoU) value of 77.5% which was significantly better than the second best method with only 46% IoU [27]. Furthermore, this architecture can be also used for a signal decomposition which will be fully defined in Chapter 3.

Chapter 3

Signal Decomposition Task

Signal Decomposition Task (SDT) is a challenging task since it intends to approximate the inverse of signal addition which can never be done correctly. There are also many problems where this process cannot be done at all, e.g. when two summed signals have the same frequency or the same frequency range, since there would be an infinite number of possible input combinations. On the other hand, if the signal is separable in the frequency spectrum, the signals must be approximately separable as well.

Unfortunately, real signals are usually not frequency separable but rather contain some time-frequency patterns. A good example of this are tones created by musical instruments. Here the patterns are recognizable by a human brain and a goal of this chapter is to create a neural network capable of splitting these patterns into individual sources, see Figure 3.1.

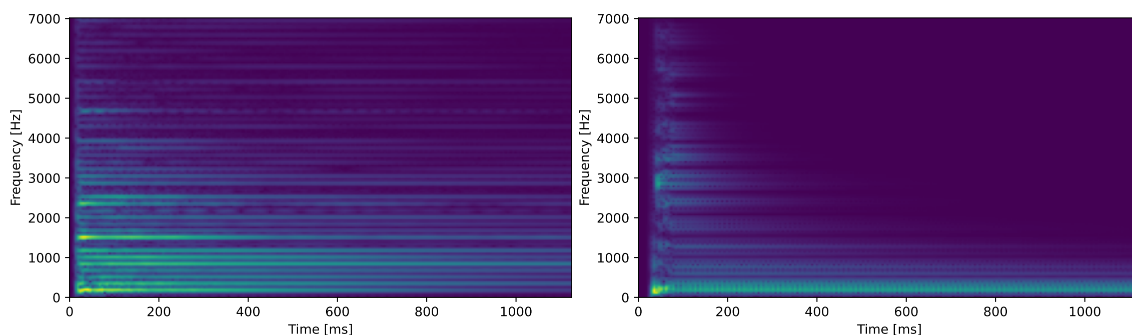


Figure 3.1: Crop of a spectrogram of a piano E2 tone (left) compared to the bass E2 tone (right).

Furthermore, different tones created by the same instrument also create diverse patterns, however, they are more similar to each other and therefore may be harder to separate, see Figure 3.2.

These patterns are usually hard to be defined and may be distorted by outside factors such as noise or air conditions. For this reason, it may be a good approach to use neural networks to find these patterns and only keep the relevant part of the signal while suppressing the noise or other signal sources.

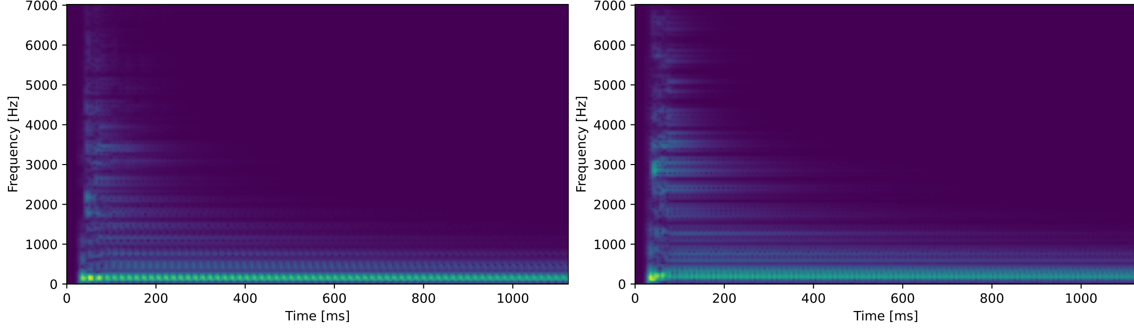


Figure 3.2: Crop of a spectrogram of a bass B1 tone (left) compared to the bass E2 tone (right).

3.1 Problem definition

Assume having S signal sources containing c channels and a signal

$$\overline{\mathbf{x}} = \sum_{i=1}^{S} \mathbf{x}^{(i)} \in \mathbb{R}^{c,N}, \tag{3.1}$$

where $x^{(i)} \in \mathbb{R}^{c,N}$ stands for a signal generated by the i-th source. The goal of SDT is then to make estimations $\widehat{\mathbf{x}}^{(i)}, \ \forall i \in \{1, ..., S\}$.

There are several approaches how to deal with SDT. First option is to take a signal $\overline{\mathbf{x}} \in \mathbb{R}^{c,N}$ and directly predict all the $\mathbf{x}^{(i)}$. In this approach there would be a neural network

$$\sigma : \mathbb{R}^{c,N} \to \mathbb{R}^{cS,N}. \tag{3.2}$$

However, there is a problem of choosing the right loss function. Comparing two signals might be difficult because of a potential small phase shift or a noise present in the signal etc. There are many options which loss function to choose. In this paper we try L1 and L2 functions.

A way how to deal with this problem is to compare products of STFT of these signals. There are many advantages of this approach. First, the phase shift of some frequencies does not cause a crucial problem for the overall loss. Another advantage is that the signal $\overline{\mathbf{x}} \in \mathbb{R}^{c,N}$ is now transformed to $\overline{\mathbf{x}}^* \in \mathbb{C}^{c,s_f,s_t} \sim \mathbb{R}^{2c,s_f,s_t}$ which allows to use 2D convolutional layers. This enables searching for time-frequency patterns while reducing the computational complexity since there is no need for long 1D convolutional layers anymore. In this case, there would be a neural network

$$\sigma : \mathbb{R}^{2c,s_f,s_t} \to \mathbb{R}^{2cS,s_f,s_t}. \tag{3.3}$$

Another option is to only use the magnitude component of STFT product while the angle is kept only for the final inverse transform, i.e.

$$\left(\overline{\mathbf{m}}^*\right)_{klm} := |\overline{x}^*_{klm}|, \quad \left(\overline{\boldsymbol{\theta}}^*\right)_{klm} := \arctan\left(\frac{\mathrm{Im}(\overline{x}^*_{klm})}{\mathrm{Re}(\overline{x}^*_{klm})}\right), \qquad \overline{\mathbf{m}}^*, \overline{\boldsymbol{\theta}}^* \in \mathbb{R}^{c,s_f,s_t}. \tag{3.4}$$

The idea behind this step is to reduce model complexity by only focusing on more relevant magnitude information. The magnitude $\overline{\mathbf{m}}^*$ is then evaluated by a neural network

$$\sigma : \mathbb{R}^{c,s_f,s_t} \to \mathbb{R}^{cS,s_f,s_t}. \tag{3.5}$$

The resulting tensor is then split into S individual segments $\widehat{\mathbf{m}}^{(i)*} \in \mathbb{R}^{c,s_f,s_t}$. All these segments are then inverted as

$$\left(\widehat{\mathbf{x}}^{(i)*}\right)_{klm} := \widehat{m}^{(i)*}_{klm} \exp(\mathrm{i}\overline{\theta}^*_{klm}), \qquad \widehat{\mathbf{x}}^{(i)*} \in \mathbb{C}^{c,s_f,s_t}. \tag{3.6}$$

In order not to create any artificial signal a soft masking may be used. Instead of re-creating a whole new tensor (STFT product), this method would only remove some components of the spectrogram by properly deleting corresponding areas in the input tensor.

3.2 Dataset

A lot of research groups (e.g. Deezer [29]) use private datasets, e.g. the Bean dataset used by Prétet et al. [30] composed of 24097 songs. One of the biggest free dataset is the MUSDB18 music dataset containing 150 full-length songs split into 4 components [31]. Most songs have a length of 200-300s, see Figure 3.4. There are 3 instrumental tracks (drum, bass, and others) plus a vocal track. Music dataset is chosen since it is easy to demonstrate whether the music decomposition methods work while maintaining the ability to use this method for other signal tasks, i.e. acoustic emission signals etc.

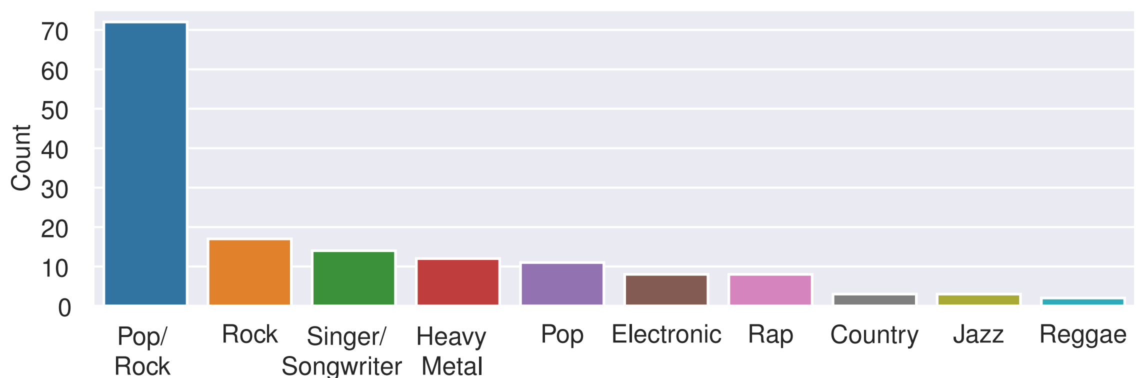


Figure 3.3: Distribution of genres in the MUSDB18 dataset.

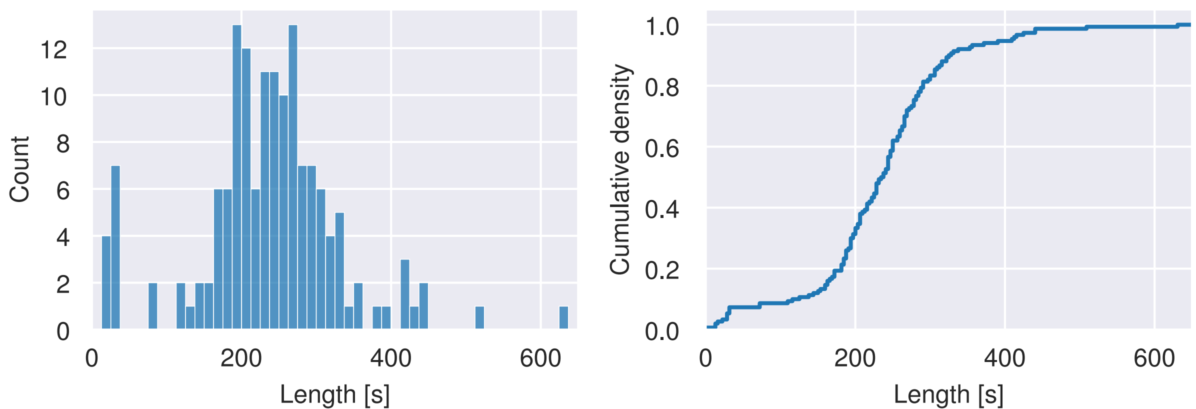


Figure 3.4: Distribution of song length in the MUSDB18 dataset.

The MUSDB18 dataset is composed mainly from pop and rock songs. This must be taken into account when training neural networks since the results may be poor on low covered genres. Another problem is that this dataset is uncomparably smaller than the private datasets and the resulting network cannot fully compete with networks created by Deezer etc.

3.3 Metrics

The SDT is very hard for evaluation since the $\hat{\mathbf{x}}^{(i)}$ can have many frequencies in common and the overall success highly depends on the found patterns rather than the frequencies themselves. Furthermore, comparing the signals together with L1/L2 may not give appropriate results since they cannot operate in the frequency spectrum.

For this reason, the spectrograms of $\hat{\mathbf{x}}^{(i)}$ and $\mathbf{x}^{(i)}$, $\forall i \in S$, will be also compared using the L1 and L2 metrics. In addition, MEL spectrograms described in section 1.10 will be also compared since they should better correspond to how human ears perceive sound in general.

The final metrics for signal comparison will be L1 and L2 applied to the signal (**L1**, **L2**), to the spectrogram (**S-L1**, **S-L2**), and to the MEL spectrogram (**MEL-L1**, **MEL-L2**).

At the same time, the audio results will be also subjectively rated by a human in order to verify which metrics are appropriate. The **rating** will use a scale of 0-10, where 0 means a complete failure and 10 stand for a perfectly predicted result. The predicted and original audio records will be available on *https://github.com/AIKovanda/signalai-audio*.

3.4 Autoencoders

A secondary goal of this paper is to assess the possibility of using autoencoders for comparing signals. In this approach, instead of comparing the signals directly, first their latent vectors are created and the final comparison is done in the latent space. The advantage of this comparison is that the latent space should contain only the most important information about the original signal which may enhance the meaning of L1 or L2 metrics.

In this section all the networks take a signal with a length of 131072 samples. The encoder part of the networks is composed of 1D convolutional layers with a stride greater than 1 in order to reduce the size of the input signal. The length of the convolutional cores is chosen so that

$$\frac{input_length - core_length}{stride} \in \mathbb{N} \tag{3.7}$$

in order to process the signal without any need for zero padding. The decoder part consists of 1D transposed convolutional layers with the same setting as the convolutional layers in the encoder part in the opposite order. This ensures that the resulting signal has the same length as the input one. The setting of 7 chosen networks is shown in Table 3.1.

	layer 1		**layer 2**		**layer 3**		**layer 4**	
id	**core length**	**stride**	**core length**	**stride**	**core length**	**stride**	**core length**	**stride**
1	50	2	50	2	-	-	-	-
2	50	2	50	2	50	2	-	-
3	50	2	50	2	50	2	50	2
4	256	4	-	-	-	-	-	-
5	256	4	253	4	-	-	-	-
6	156	4	154	4	-	-	-	-
7	156	4	154	4	153	4	-	-

Table 3.1: Setting of the encoder part of all the simple autoencoders.

Spectrogram-based autoencoders were tried as well but the results were poor compared to the signal-based architectures. Finding a proper setting of the STFT transform together with the network hyperparameters is left for a next research.

In the next phase the chosen networks are trained on individual sources from the MUSDB18 dataset since the best model will be used as a metric for comparing solely separate sources. In the training process the networks try to compress the input signal into a latent vector and then reverse this process to predict the original signal. The predicted signal is then compared to the input signal to determine the autoencoder performance. The results of 7 trained autoencoders are shown in Table 3.2.

id	compression [%]	L1	L2	S-L1	S-L2	MEL-L1	MEL-L2	rating
1	50.05	0.0047	0.0002	0.0828	0.1633	0.0576	39.01	9.0/10
2	75.06	0.0100	0.0008	0.1462	0.5809	0.1118	97.36	7.0/10
3	87.57	0.0116	0.0010	0.1570	0.7402	0.1437	37.32	6.5/10
4	50.10	0.0018	0.0001	0.0342	0.0368	0.0133	1.22	9.5/10
5	87.62	0.0084	0.0006	0.1264	0.4205	0.0848	28.30	7.0/10
6	75.14	0.0072	0.0005	0.1123	0.3327	0.0853	106.66	6.5/10
7	93.90	0.0141	0.0013	0.1720	0.9785	0.2076	42.64	6.0/10

Table 3.2: Results of the simple autoencoder on augmented test data.

The first and the fifth network seem to sufficiently predict the original signal with a compression rate of around 50%. The resulting audio seems to be a little distorted but still close enough to the original audio. All the other networks have low performance resulting in very distorted signals. This corresponds with the values of all metrics. For this reason, network 5 is chosen for the following signal comparison.

In the next sections the signals will be additionally compared using the L1 metric in the latent space generated by the 5th trained autoencoder architecture (**AE**).

3.5 Training strategy

The MUSDB18 dataset consists of 150 songs split into 4 components. In order to model real life situations, data augmentation is used in all the training. Reverb, Gain and Chorus are used all with a probability of 20%. In Reverb, all the parameters described in section 1.9 are chosen randomly from uniform distribution $\mathcal{U}_{[0,1]}$ except for the freeze mode which comes from $\mathcal{U}_{[0,0.2]}$. The reason for it is that with high freeze mode value the sound becomes destroyed too much to bring any benefit for the training.

The Gain augmentation adjusts the sound volume by a value from $\mathcal{U}_{[-15,15]}$ Db. Finally, the Chorus is also applied with a center delay from $\mathcal{U}_{[6.5,8.5]}$. In case of working with time-frequency representations the STFT or spectrogram transformation is applied after all the augmentations.

In the beginning, the $\mathbf{x}^{(i)} \in \mathbb{R}^{c,N}$ are randomly taken from the training set. Then all the transformations are applied separately for each $\mathbf{x}^{(i)}$, resulting into $\widetilde{\mathbf{x}}^{(i)}$. The network is then trained

on

$$\text{input}: \ \overline{\widetilde{\mathbf{x}}} := \sum_{i=1}^{S} \widetilde{\mathbf{x}}^{(i)}, \qquad \text{output}: \begin{pmatrix} \widetilde{\mathbf{x}}^{(1)} \\ \widetilde{\mathbf{x}}^{(2)} \\ \vdots \\ \widetilde{\mathbf{x}}^{(S)} \end{pmatrix}. \tag{3.8}$$

3.6 Signal to signal networks

One of the simplest approaches to the decomposition task is to use networks $\sigma : \mathbb{R}^{c,N} \rightarrow \mathbb{R}^{cS,N}$ which take signal $\overline{\mathbf{x}}$ as an input and produce $\widehat{\mathbf{x}}^{(i)}$, $\forall i \in \{1, ..., S\}$ as an output. First, the InceptionTime architecture described in section 2.9 will be used as it is a simply scalable architecture. There are many possible ways how to design the network. The number of convolutional cores as well as their length can be set in each Inception module. The number of these modules is also arbitrary.

Generally, a big network with a lot of parameters has a big computational and memory demand. There can also be problems with overfitting. This is the key problem here since the processed signal should be as long as possible due to the potential long-term audio features. For this reason the convolutional layers in the first module should be bigger than those introduced by Fawaz et al. Finally, it was decided that those layers would have a size of 51, 121 and 255. A bigger version then consists of 3 modules while the small version contains only the first module.

For both versions the training is slow and the results are not good enough, see Table 3.3. Instead of separating the sources all the trained networks only slightly suppress signals coming from the other sources. This means that in all the predictions other components are clearly hearable. Furthermore, a big network trained with a small learning rate of 0.0002 seems to only produce a whistling noise. This sound is clearly not desired and therefore leads to very poor subjective rating. Networks trained with a high learning rate of 0.001 give less clear sound compared to the first network. In addition, all the networks struggle completely with predicting the *other accompaniment* source, which in all cases sounds identical to the input sound.

id	version	lr	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
1	small	0.0002	0.0316	0.0029	0.0118	0.2106	1.7062	0.6164	93.59	5.0/10
2	big	0.0002	0.0305	0.0026	0.0116	0.2355	1.8581	0.6189	96.05	1.6/10
3	small	0.0010	0.0299	0.0023	0.0113	0.2194	1.4598	0.5913	70.94	3.9/10
4	big	0.0010	0.0311	0.0026	0.0117	0.2222	1.7413	0.6080	92.21	3.4/10

Table 3.3: Results of the Inceptiontime architecture on augmented test data.

The second approach is based on the idea of time-frequency transformation, which is all learned from the input signal by the network. The proposed architecture SepNet consists of 4 parts. In the first part, two 1D convolutional layers are applied to the signal with a stride of 128 inspired by the STFT. The resulting matrix can be considered as a time-frequency interpretation. In the next part 2D convolutional layers are used in order to further process the output tensor. In some networks the attention module inspired by YOLOv4 [32] object detection architecture is used to reduce irrelevant information. Finally, the tensor is transformed back into a signal resulting into the final prediction. The overall architecture is depicted in Figure 3.5.

The results are shown in Table 3.4. Most of the metrics are clearly better than those of the InceptionTime network. However, the results sound worse. All the predictions are highly distorted

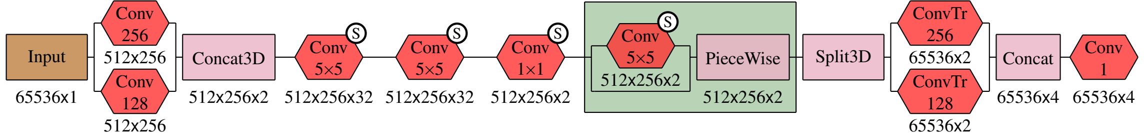


Figure 3.5: Structure of SepNet. Output shape of each layer is depicted below the corresponding cell. S stands for the SELU activation.

which makes SepNet the worst tried network for music source separation. This effect is even higher for networks trained with a high learning rate.

id	attention	lr	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
1	no	0.0002	0.0272	0.0021	0.0104	0.1989	1.2972	0.5253	57.64	4.0/10
2	yes	0.0002	0.0269	0.0020	0.0103	0.1976	1.2737	0.5198	54.01	4.0/10
3	no	0.0010	0.0293	0.0024	0.0111	0.2179	1.6311	0.5973	77.19	3.5/10
4	yes	0.0010	0.0294	0.0024	0.0111	0.2296	1.6028	0.5892	76.04	3.1/10

Table 3.4: Results of the SepNet architecture on augmented test data.

3.7 STFT to STFT networks

In the next section the $\sigma : \mathbb{R}^{2c,s_f,s_t} \rightarrow \mathbb{R}^{2cS,s_f,s_t}$ networks are evaluated. First, a ResNeXt-based architecture called SpecResNeXt is tried. The bigger version consists of 5 ResNeXt modules (ID, Conv, Id, Conv, Id) depicted in Figure 2.22. A smaller version has only 3 modules (ID, Conv, Id).

The results are shown in Table 3.5. These results are made after cropping first and last 1000 samples from the predicted signal since it always contains an intense crack. In a potential application a zero padding should be added to the input signal in order to compensate for the lost 2000 samples in the output. The metrics values are in average higher than those of SepNet, however, a subjective rating is better. The output is not distorted but contains all the other components similarly to the InceptionTime architecture. The big version trained with a high learning rate only produces crackings and therefore has a zero subjective rating.

id	version	lr	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
1	small	0.0002	0.0374	0.0051	0.0140	0.2655	3.6491	1.2133	2274.85	4.1/10
2	big	0.0002	0.0365	0.0048	0.0137	0.2678	3.3733	1.1527	1756.71	4.1/10
3	small	0.0010	0.0368	0.0048	0.0139	0.2761	3.3891	1.1206	1458.83	4.0/10
4	big	0.0010	0.0370	0.0066	0.0139	0.2750	5.1414	1.1854	2755.50	0.0/10

Table 3.5: Results of the SpecResNeXt architecture on augmented test data.

Finally, the U-Net inspired by Jansson et al. [28] was tried. This model is well designed on separating exactly 1 source. With a small modification it would be possible to predict all the sources, however, it would raise computational demand and therefore it was decided to train a model for each source separately.

The results made for all signal sources separately are shown in Table 3.6. The overall average is calculated as well for comparison with the previous models. The U-Net seems to have a good ability to separate music sources, which is visible especially at the *bass* component with a very low S-L1 metric value. The metrics values are low compared to the other models with an exception of

the SepNet. SepNet has comparable metrics values and even beats the U-Net in MEL-L2 metric, however, in the subjective rating the U-Net is much better.

id	source	lr	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
1	drums	0.0002	0.0236	0.0023	0.0093	0.1761	1.3410	0.5776	426.69	8.5/10
1	bass	0.0002	0.0254	0.0024	0.0084	0.0814	1.3522	0.6004	555.03	9.5/10
1	others	0.0002	0.0312	0.0031	0.0118	0.2132	1.8942	0.7212	242.31	7.5/10
1	vocals	0.0002	0.0199	0.0017	0.0078	0.1510	1.1777	0.4041	114.21	6.5/10
1	AVG	0.0002	0.0250	0.0024	0.0093	0.1554	1.4413	0.5758	334.56	8.0/10
2	drums	0.0010	0.0234	0.0023	0.0092	0.1783	1.4070	0.5825	469.22	8.5/10
2	bass	0.0010	0.0246	0.0022	0.0082	0.0792	1.2361	0.5637	384.56	9.5/10
2	others	0.0010	0.0349	0.0042	0.0130	0.2208	2.5624	0.9249	1493.87	7.0/10
2	vocals	0.0010	0.0226	0.0027	0.0089	0.1813	1.9712	0.5776	386.10	6.0/10
2	AVG	0.0010	0.0264	0.0028	0.0098	0.1649	1.7942	0.6622	683.44	7.8/10

Table 3.6: Results of the U-Net architecture on augmented test data.

The *bass* component produced by U-Net is almost unrecognizable from the original one. The prediction of *drums* partly contains the *bass* component. A similar problem happens with the *other accompaniment* part which contains *vocals*. The *vocal* prediction also contains cymbals from the *drums* component. However, all these problems are minor considering the small size of the MUSDB18 dataset.

3.8 Separate instrument comparison

Finally, the Table 3.7 shows the overall result of the best trained model from each architecture for every signal component separately. The U-Net architecture is by far the best tool for signal decomposition task by the subjective rating. Unfortunately, this rating is not fully confirmed by any metric used for the evaluation.

The Pearson correlation table of all the used metrics based on all the examined architectures is shown in Figure 3.6. Based on 56 observations, the L2, S-L2 and MEL-L1 metrics seem to be highly correlated. Another high correlation is among L1 and AE metrics. The subjective rating is correlated the most with the AE metric and L1 metric, which was also used as a loss function for training the networks. The absolute value of correlation with AE is only 0.507, however, that is still the highest value among all the tested metrics.

This result shows the potential of using autoencoders for signal comparison. On the other hand, the subjective metric is not deterministic and different people may evaluate the audio results differently. More data from various people would be needed in order to fully evaluate the tested metrics.

architecture	id	source	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
InceptionTime	1	drums	0.0364	0.0052	0.0136	0.2531	2.7035	0.7997	259.04	5.5/10
InceptionTime	1	bass	0.0282	0.0020	0.0094	0.0997	1.1442	0.5578	60.06	6.0/10
InceptionTime	1	others	0.0353	0.0027	0.0135	0.2687	1.6172	0.6376	27.53	4.0/10
InceptionTime	1	vocals	0.0265	0.0019	0.0106	0.2208	1.3599	0.4705	27.73	4.5/10
SepNet	2	drums	0.0261	0.0024	0.0105	0.2614	1.6462	0.6226	124.88	4.5/10
SepNet	2	bass	0.0253	0.0017	0.0085	0.0868	0.8692	0.4628	43.08	5.0/10
SepNet	2	others	0.0320	0.0023	0.0125	0.2424	1.3373	0.5683	28.06	2.0/10
SepNet	2	vocals	0.0241	0.0017	0.0097	0.1998	1.2421	0.4254	20.03	4.5/10
SpecResNeXt	1	drums	0.0356	0.0061	0.0138	0.2930	4.4668	1.4176	4601.90	4.0/10
SpecResNeXt	1	bass	0.0361	0.0043	0.0122	0.1453	2.9934	1.1031	1515.35	4.5/10
SpecResNeXt	1	others	0.0416	0.0053	0.0160	0.3387	3.6196	1.2203	729.89	4.0/10
SpecResNeXt	1	vocals	0.0364	0.0048	0.0141	0.2851	3.5164	1.1121	2252.26	4.0/10
U-Net	1	drums	0.0236	0.0023	0.0093	0.1761	1.3410	0.5776	426.69	8.5/10
U-Net	1	bass	0.0254	0.0024	0.0084	0.0814	1.3522	0.6004	555.03	9.5/10
U-Net	1	others	0.0312	0.0031	0.0118	0.2132	1.8942	0.7212	242.31	7.5/10
U-Net	1	vocals	0.0199	0.0017	0.0078	0.1510	1.1777	0.4041	114.21	6.5/10

Table 3.7: Results of the best architectures on augmented test data for each component.

	L1	L2	AE	S-L1	S-L2	MEL-L1	MEL-L2	rating
L1	1.000	0.769	0.942	0.641	0.711	0.784	0.381	-0.463
L2	0.769	1.000	0.749	0.555	0.983	0.950	0.749	-0.346
AE	0.942	0.749	1.000	0.853	0.717	0.739	0.340	-0.507
S-L1	0.641	0.555	0.853	1.000	0.571	0.514	0.202	-0.489
S-L2	0.711	0.983	0.717	0.571	1.000	0.927	0.761	-0.413
MEL-L1	0.784	0.950	0.739	0.514	0.927	1.000	0.787	-0.257
MEL-L2	0.381	0.749	0.340	0.202	0.761	0.787	1.000	-0.191
rating	-0.463	-0.346	-0.507	-0.489	-0.413	-0.257	-0.191	1.000

Figure 3.6: Correlation matrix of used metrics based on all the results.

Chapter 4

Event List Generation

Event List Generation (ELG) is a task in which individual types of events are being detected. This can be used for various problems, such as event analysis of acoustic emission or MIDI files generation from the source audio records. In practice, a system based on ELG could detect potentially harmful material flaws using acoustic emission signal analysis. This could help preventing serious malfunctions of pipes, bridges, turbines etc.

4.1 Problem definition

Let us have E event types and a signal with a length of N samples composed of c channels. The input to the ELG is a signal $\overline{\mathbf{x}} = \sum_{i=1}^{E} \mathbf{x}^{(i)} + \boldsymbol{\epsilon} \in \mathbb{R}^{c,N}$, where $\mathbf{x}^{(i)} \in \mathbb{R}^{c,N}$ stands for a signal containing solely the i-th event type and $\boldsymbol{\epsilon} \in \mathbb{R}^{c,N}$ represents a background noise and other sounds which are not relevant. Let $\mathbf{Y} \in \{0, 1\}^{E,N}$ be a **characteristic matrix** of the events defined as

$$y_{ij} := \begin{cases} 1, & \text{if } j\text{-th sample corresponds to the } i\text{-th event,} \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Next, let $\mathbf{Z} \in \{0, 1\}^{E,M}$ be a **binary map** of the events defined as

$$z_{ij} := y_{i,\lfloor \frac{jN}{M} + 0.5 \rfloor}, \tag{4.2}$$

where M can be chosen arbitrarily depending on the specific problem. The goal of the ELG is then to make the estimation $\widehat{\mathbf{Z}}$ for a chosen M, see Figure 4.1.

Figure 4.1: Example of a single-channel signal $\overline{\mathbf{x}}$ and a binary map $\mathbf{Z}$ including $E = 3$ event types.

4.2 Approach

There are several approaches how to deal with this task. First option is to take the signal $\overline{\mathbf{x}} \in \mathbb{R}^{c,N}$ and directly predict the binary map $\mathbf{Z}$. In this approach there would be a neural network

$$\sigma : \mathbb{R}^{c,N} \rightarrow [0,1]^{E,M}. \tag{4.3}$$

This can be understood as a multi-label classification made of EM labels. In this chapter we use the cross-entropy loss function. An alternative L2 loss was tested but all resulting architectures were performing very poorly.

Another way to solve this task is to create a spectrogram with a shape of $[s_f, s_t]$ from the input signal. The idea behind this step is to provide the network information about the frequencies in a 3D tensor (including the channel axis). One advantage is the ability to use 2D convolutional layers capable of dealing with a full time-frequency information. In the next phase this tensor could be reduced to a matrix representation by creating a 3D tensor with only one channel. 1D convolutional layers can be further used to process this matrix and to create a final binary map estimation. In this approach there would be a neural network

$$\sigma : \mathbb{R}^{c,s_f,s_t} \rightarrow [0,1]^{E,M}. \tag{4.4}$$

4.3 Dataset - Tones

The dataset consists of 85 piano tones generated digitally by external software using sound bank containing real recordings captured by 8 various microphones and simulating 2 key velocities with a sampling frequency of 48kHz. A data generator was created to provide a randomly mixed signal $\overline{\mathbf{x}}$ together with a corresponding matrix $\mathbf{Z}$. For the signal networks, the M parameter is set to $\frac{N}{512}$ which corresponds to a time interval of 10.6ms. For the networks estimating the matrix $\mathbf{Z}$ directly from the spectrogram, $M = s_t$ is used so that the output binary map had the same time scale as the spectrogram. In this chapter this means a time interval of 5.3ms.

This generator randomly chooses 2-30 tones with corresponding lengths and put them into an interval of $2.74s$ (2^{17} samples), see Figure 4.2.

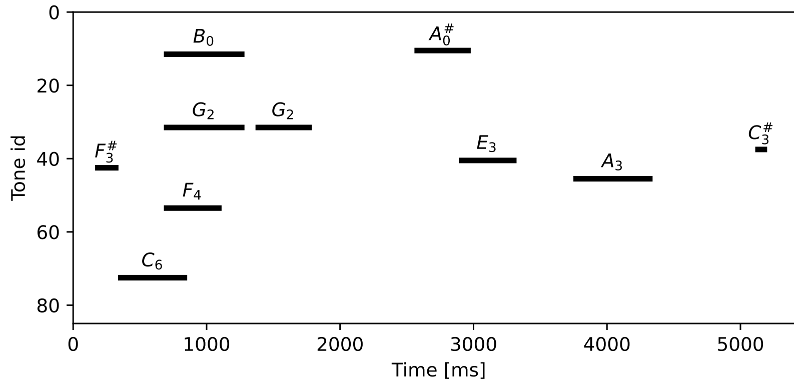


Figure 4.2: Example of a generated tone binary map $\mathbf{Z}$.

4.4 Evaluation

In this task there must be a compromise between precision and recall which are used to get more intuition about what is happening in the prediction. This balance will be done using the F1 metric defined as

$$F1 := 2\,\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}, \tag{4.5}$$

where precision and recall are defined as

$$\text{precision} := \frac{\text{true positives}}{\text{predicted positives}}, \qquad \text{recall} := \frac{\text{true positives}}{\text{real positives}}. \tag{4.6}$$

Since neural networks predict a number in a range of $[0, 1]$, a threshold is needed to determine a binary prediction. For simplicity, only a threshold of 0.5 will be considered in the following classifier evaluation.

Another metric which will be used is the ROC AUC metric. This metric is defined as an area below the **Receiver Operating Characteristic** curve (ROC curve). This curve is used for evaluating a binary classifier by moving a threshold t over the interval $[0, 1]$ and calculating the true positive rate and false positive rate. A completely random classifier would give a ROC curve of $f(x) = x$. A good classifier should approximate $f(x) = 1$. An example of a ROC curve is depicted in Figure 4.3. Here the curve lies above $f(x) = x$ which means that for every threshold the classifier is better than a random one. The ROC AUC is equal to 0.96.

4.5 Raw signal networks

First, the architectures $\sigma : \mathbb{R}^{c,N} \to [0, 1]^{E,M}$ are examined. All of them are based on InceptionTime network since it proves to be a good choice for Time Series Classification. However, tone prediction has some characteristics that must be taken into account. InceptionTime uses convolutional cores with a length of 10, 20 and 40 by default. These sizes however seem to be too small to capture crucial features needed for a proper event list generation. For comparison, the STFT transformation uses a window with a length of 1024. Having a convolutional core with this length has a big computational demand and takes a lot of time. In this section the first Inception module is taken always with cores with a length of 63, 127, and 255.

Since the output of each module has the same size of the time axis, a pooling is applied after each Inception module to fit the output binary map. This approach also lowers the computational demand while predicting event list of sufficient resolution. A bigger version then takes additional 2 original modules.

Since the training takes many hours, only a few parameters is fully examined. First examined parameter is the size of the network. Second, the learning rate of 0.001 or 0.0001 is used since other experiments showed the importance of having it in a balance. Dynamically changing learning rate while training is kept for a next research. All these architectures are trained on augmented signals in order to get more realistic data. They all take an interval of 131584 samples as an input (2.74s) with a sample rate of 48kHz.

The results shown in Table 4.1 indicate that the bigger network performs better that the smaller one. Networks with lower learning rate tend to have a better ROC AUC metric (see Figure 4.3), while the ones trained with higher learning rate have a better F1 metric with a threshold of 0.5. Balancing a threshold can increase the F1 metric, however, this leads to another problem. Here

id	size	lr	accuracy	precision	recall	F1	ROC AUC
1	small	0.0002	0.99418	0.94562	0.24547	0.38976	0.91846
2	big	0.0002	0.99518	0.99781	0.36274	0.53206	0.95667
3	small	0.0010	0.99594	0.97756	0.47818	0.64221	0.91773
4	big	0.0010	0.99702	0.98260	0.62497	0.76400	0.93236

Table 4.1: Results of the InceptionTime architecture on augmented test data.

the low F1 value is caused by small recall and pushing the threshold down would increase F1 metric at the cost of lowering the precision value. In practice, this would lead to the prediction of non-existing tones which is not wanted.

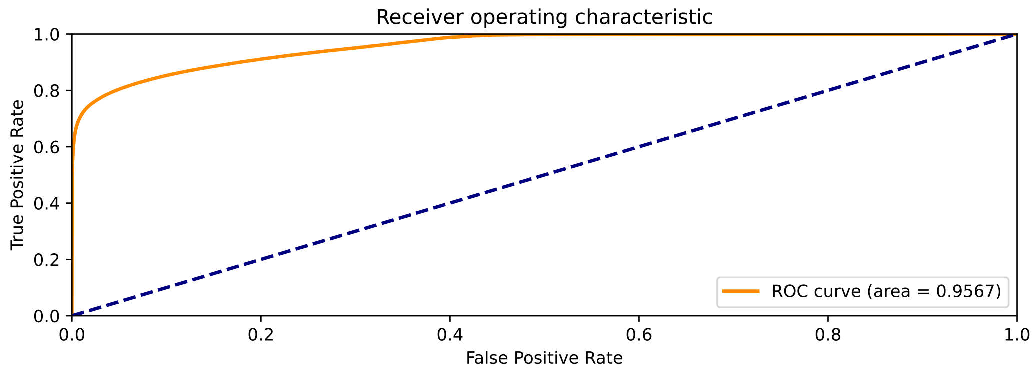


Figure 4.3: ROC AUC of the winning InceptionTime architecture.

The problem with a small recall is caused by a poor prediction of multiple simultaneous tones at once, see Figure 4.4. This effect applies to all the trained InceptionTime architectures. The overlapping tones are usually cropped or interrupted in time. Together with a high computational time, this makes this architecture a poor choice for ELG task.

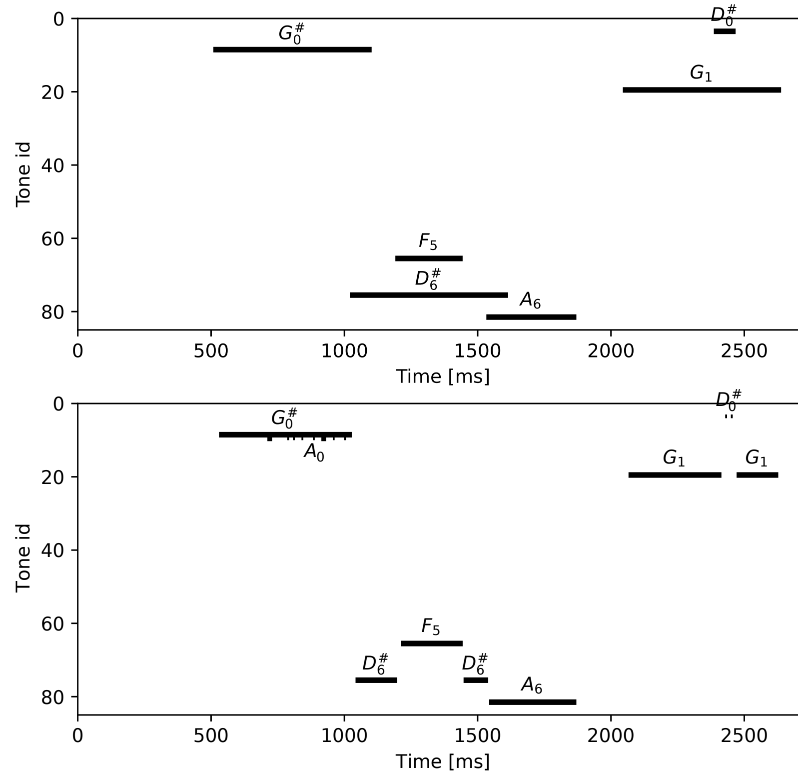


Figure 4.4: Typical prediction of InceptionTime. Top: ground truth, bottom: prediction.

4.6 STFT networks

Using long 1D convolutional cores is very inefficient and leads to a big memory and computational demand. Alternative option is to first transform the input signal into an STFT representation which allows to compress the data in the time axis and create a new frequency axis. This step enables to use 2D convolutional layers which can have dramatically less parameters (more than 10 times less). In this section only the STFT magnitude is taken into account since our experiments show that this approach is satisfactory.

Two versions of a simple ConvNet were created. The big version consists of 4 2D convolutional layers with a core of 5x5 which do not change the first two dimensions of the input tensor, see Figure 4.5. This strategy is chosen because of the desire to have an overall output with the same time scale. In the next step the 3D tensor with only one channel is reshaped into a matrix. The frequency axis is now interpreted as a channel axis used for the following small 1D convolutional layers.

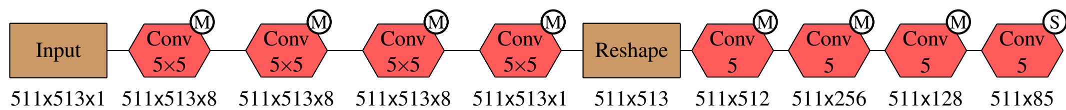


Figure 4.5: Structure of Spec2Map-big. Output shape of each layer is depicted below the corresponding cell.

The idea behind this is to use 2D convolutional layers to reduce noise and to extract local 2D features, whereas the second part is used to determine tones based on a modified magnitude matrix. The smaller version uses 2 less 2D convolutional layers and one less 1D layer, see Figure 4.6.

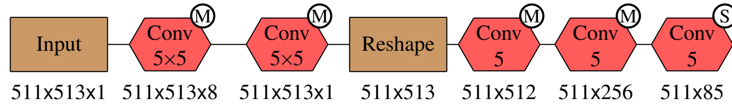


Figure 4.6: Structure of Spec2Map-small. Output shape of each layer is depicted below the corresponding cell.

In the training process many variables were tested in order to determine their importance. It was found that too big learning rate (lr=0.001) leads to a complete failure of the network in every tested version. For this reason, all the following experiments were made with lr=0.0001. Changing this value in the training process is again left for a next research. Each network was trained on 40000 batches made of 16 samples. In all of them the loss value was declining. After around 30000 batches the loss stabilized and declined very slowly. Early stopping seemed not to have a significant effect.

The overall results are shown in Table 4.2. Linear regression shows a significant effect of augmentation on the resulting F1 metric. This was expected since the evaluation process includes the reverb augmentation in order to generate more real-life data since it should simulate various ambient conditions. On the other hand, the size of the network or used sample rate (48kHz or 24kHz) do not appear to be significant. For the next research we therefore suggest to use smaller sample rate in order to lower the computational demand. The ROC AUC characteristic is shown in Figure 4.7 and an example of its performance is depicted in Figure 4.8.

id	version	augmentation	sr [kHz]	accuracy	precision	recall	F1 (t=0.5)	ROC AUC
1	small	no	24	0.99855	0.76937	0.94451	0.84799	0.99932
2	big	no	24	0.99828	0.72988	0.95656	0.82798	0.99940
3	small	yes	24	0.99946	0.94984	0.92290	0.93618	0.99952
4	big	yes	24	0.99952	0.95359	0.93245	0.94290	0.99974
5	small	no	48	0.99859	0.77263	0.93335	0.84541	0.99835
6	big	no	48	0.99841	0.74698	0.94465	0.83426	0.99886
7	small	yes	48	0.99942	0.95550	0.90165	0.92779	0.99864
8	big	yes	48	0.99949	0.96322	0.91196	0.93689	0.99911

Table 4.2: Results of the Spec2Map architecture on augmented test data.

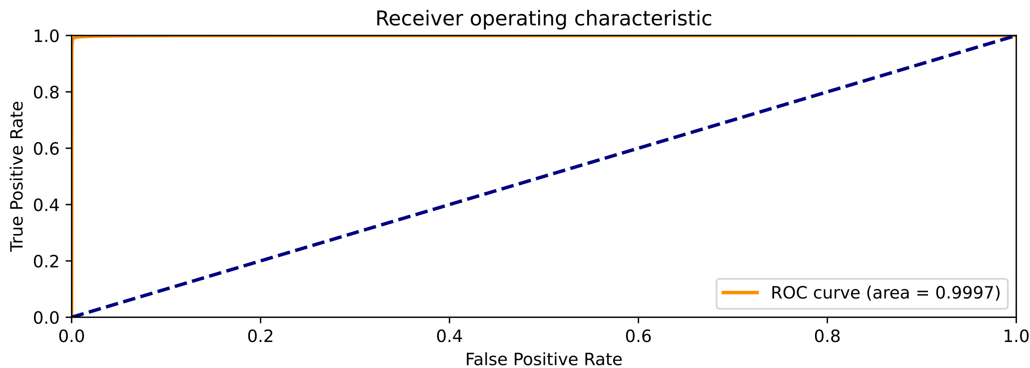


Figure 4.7: ROC AUC of the winning Spec2Map architecture.

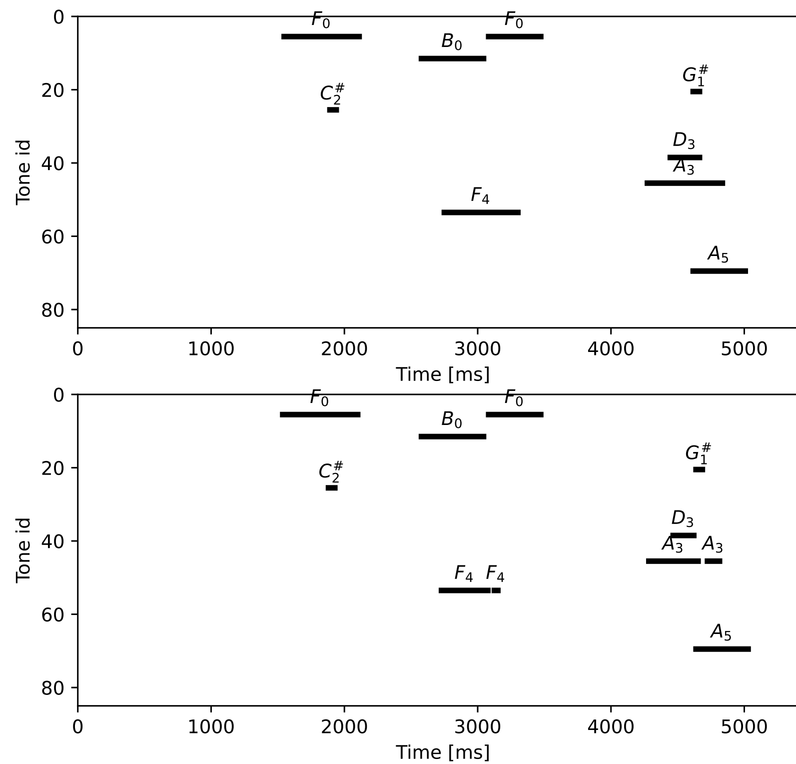


Figure 4.8: Ground truth (top) and predicted (down) tones by the winning Spec2Map architecture.

4.7 Identification of acoustic emission bursts

The basic principle of the acoustic emission method is to monitor ultrasonic signals generated at various points in the structure as the effect of mechanical loading. The primary objective of the experiment illustrated in Figure 4.10 was to capture and identify characteristic cracking signals accompanying microscopic processes of crack nucleation at the root of the artificial notch and consecutive fatigue crack propagation.

The captured signals often contain high levels of noise which originates from the hydraulic loading unit and the vibrations induced by friction of the test body in the bearing slots. The goal of deploying deep learning methods was to identify crack growth acoustic manifestations in a very high noise background and to verify the applicability on real continuous acoustic emission recordings. An example of a spectrogram containing 2 bursts is depicted in Figure 4.9.

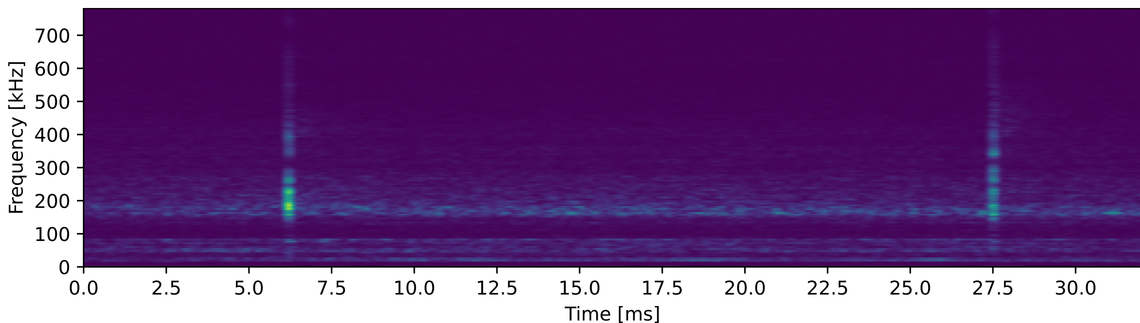


Figure 4.9: Spectrogram of the measured signal containing 2 bursts.

The dataset consists of 139s long background noise measured with a sample frequency of 1562.5kHz

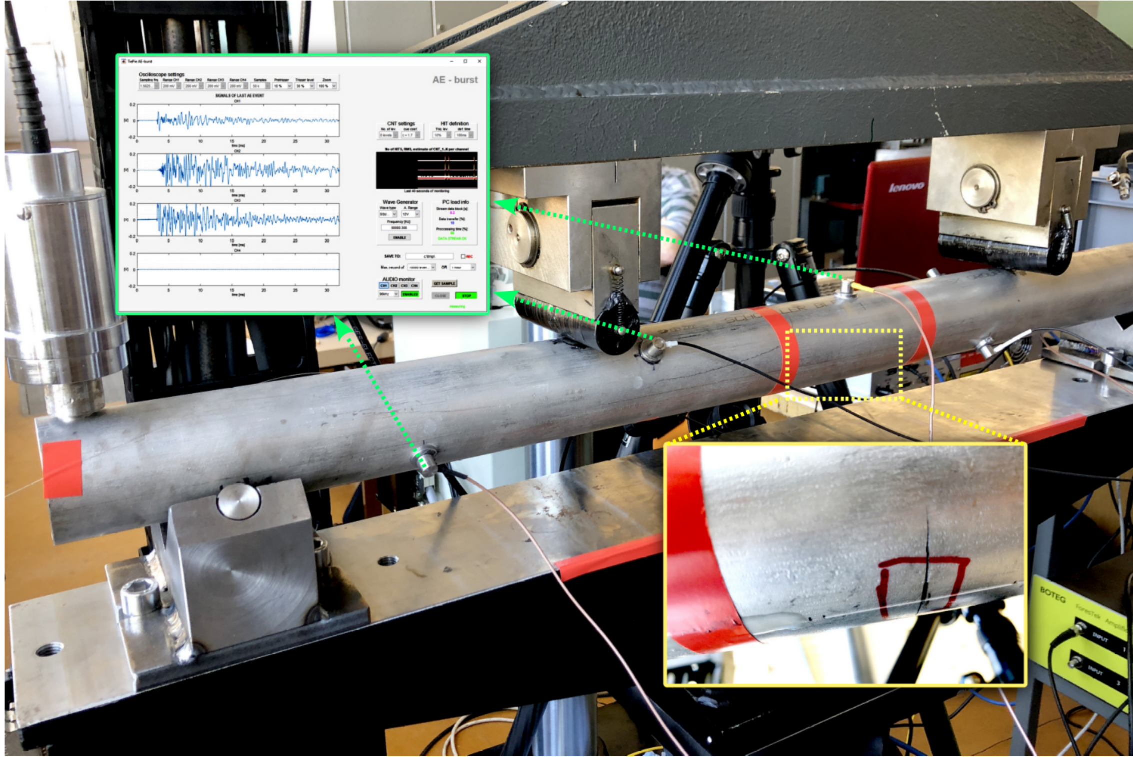


Figure 4.10: Diagram of tensile testing machine experiment.

and 6000 bursts of two types manually isolated from a tensile test signal. These bursts are split into the train set (4800) and the test set (1200). For this reason the final binary map only contains 2 channels. The networks are trained on randomly generated mixture of the background noise and bursts from the train set. The evaluation is made in the same way with the test set.

There are many differences between this task and piano tones detection. In this task most of the signal comes from background sources while the bursts are only about 252 samples long. This significantly lowers the chance of having 2 bursts in the same time. Furthermore, the spectrogram has to be made with lower stride to capture the information about the beginning and the end of each burst. This makes the spectrogram bigger and harder to be processed by the networks. On the other hand, smaller interval of the signal may be processed because of the lack of long-term dependencies. This makes the InceptionTime-based networks fast compared to the previous task.

id	model	size	accuracy	precision	recall	F1 (t=0.5)	ROC AUC
1	InceptionTime	small	0.99943	0.98438	0.96064	0.97237	0.99806
2	InceptionTime	middle	0.99949	0.98148	0.97055	0.97598	0.99872
3	InceptionTime	big	0.99938	0.97952	0.96039	0.96986	0.99737
4	Spec2Map	small	0.98980	0.32635	0.00516	0.01016	0.97918
5	Spec2Map	big	0.98981	0.71328	0.00333	0.00662	0.98109

Table 4.3: Results of the InceptionTime and Spec2Map architectures on augmented test data.

Three networks were tested. The smallest one consists of one big InceptionTime module with convolutional cores with lengths of 63, 127, and 255. The result is then processed by a max pooling with a size of 16 in order to lower the output binary map dimensionality. The middle size network has one additional small module with cores with lengths of 11, 21, and 41. After each module there is a pooling layer with a size of 4. The biggest model contains 3 modules, one

big and two small InceptionTime modules. The pooling after first two modules has a length of 2, whereas the final one has a size of 4. The spectrogram-based networks have the same setting as those from the previous section. All of the models are trained with a learning rate of 0.0002. The results are shown in Table 4.3.

The winning Spec2Map architecture from the previous task seems to struggle on predicting bursts. With a threshold of 0.5 the network almost only predicts zeros. This results in very low recall value. The ROC AUC metric indicates that lowering the threshold would increase the recall significantly, however, with further decreasing the precision which is not high even for $t = 0.5$, see Figure 4.11.

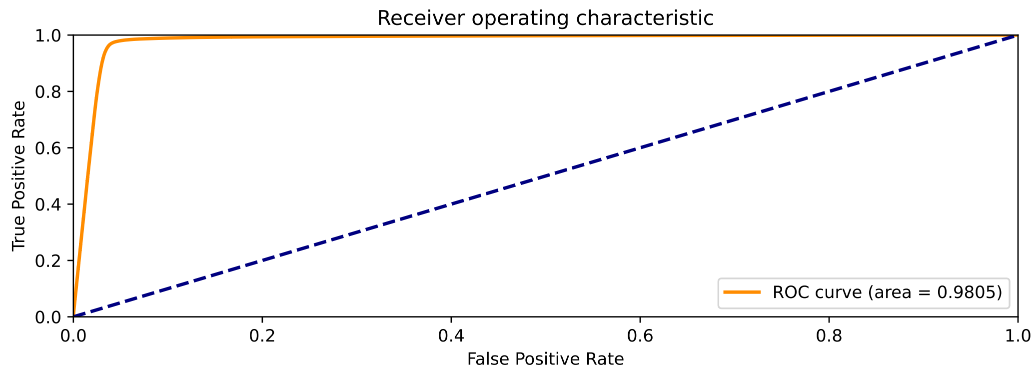


Figure 4.11: ROC AUC of the big Spec2Map architecture.

On the other hand, InceptionTime-based architectures seem to perform very well. All the metrics are high and the ROC curve approximates $f(x) = 1$, see Figure 4.12. This generally indicates a high performing classifier. An example of a typical prediction is depicted in Figure 4.13.

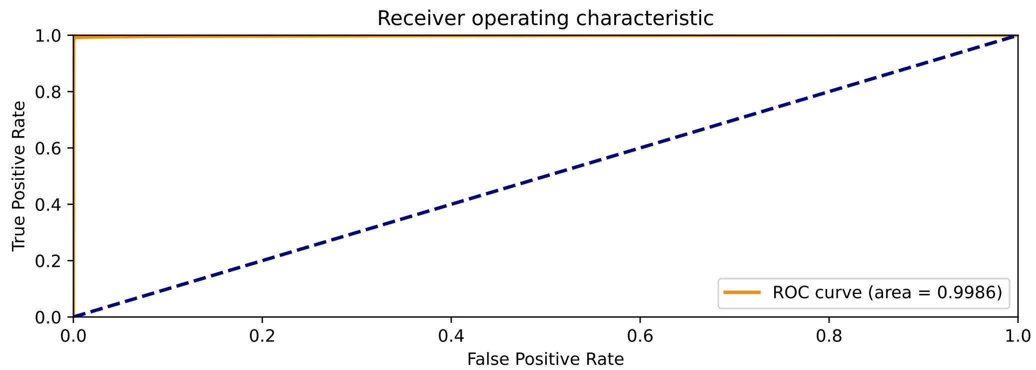


Figure 4.12: ROC AUC of the winning InceptionTime architecture.

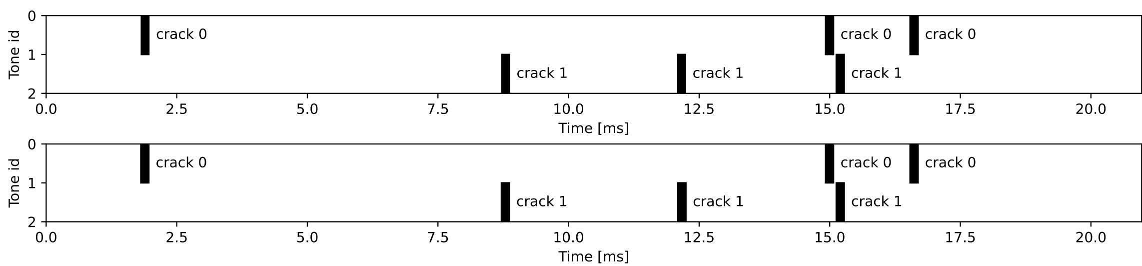


Figure 4.13: Ground truth (top) and predicted (down) bursts by the winning InceptionTime architecture.

Conclusion

One goal of this paper was to create proper neural networks which would be capable of separating a signal into individual components. Another goal was to identify particular events present in the input signal.

In the first chapter the theory behind neural networks is described. This theory is needed in order to fully understand the neural network functionality. In the second chapter various state-of-the-art convolutional neural networks (ConvNets) are described. Most of these ConvNets were designed for the ImageNet classification task which became a respected benchmark for ConvNets comparison. There are also architectures from the Series Classification and Segmentation tasks as their modified versions are used in this paper.

Signal Decomposition task is defined in the third chapter. Here many architectures are tested on the MUSDB18 dataset which consists of 150 songs decomposed into 4 sources. Networks based on processing signals directly did not perform well and took a lot of time in the process of training. Finally, the U-Net architecture seems to be able to catch the important time-frequency patterns and produces satisfactory results. Many metrics are also tested together with one chosen autoencoder in order to explore their correlation with a subjective rating. Results show the potential of autoencoders as they may better focus on relevant information. However, bigger data would be needed to make more relevant conclusion.

In the last chapter individual event are being detected in the input signal. This task is about classifying small time intervals for each data class. First, the strategy of creating binary maps is defined. All the networks are binary classifiers and binary metrics such as F1 and ROC AUC may be used for the model evaluation. There are two datasets available for this task. The first one consists of piano tones generated digitally from a sound bank. The second one is composed of 6000 bursts and a background hum from acoustic emission captured from a tensile test. In both cases the training data were randomly generated using data augmentation methods. The bursts are very small and there is only a small chance that two of them would happen in the same time. On the other hand, multiple piano tones usually play at the same time.

All the networks based the InceptionTime seem to struggle predicting multiple events happening in the same time. This makes them useless on the piano dataset, whereas on the emission dataset they perform very well. The spectrogram network seems to make good predictions on the piano tones but struggle catching small bursts from the emission dataset.

Bibliography

[1] P. Ramachandran, B. Zoph, and Q. V. Le, *Searching for Activation Functions*, 2017. arXiv: 1710.05941 [cs.NE].

[2] D. Misra, *Mish: A Self Regularized Non-Monotonic Activation Function*, 2020. arXiv: 1908.08681 [cs.LG].

[3] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *CoRR*, vol. abs/1502.01852, 2015. arXiv: 1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852.

[5] Jefkine, *Backpropagation In Convolutional Neural Networks*, en-us, Sep. 2016. [Online]. Available: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/ (visited on 04/09/2022).

[6] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv:1609.04747 [cs]*, Jun. 2017, arXiv: 1609.04747. [Online]. Available: http://arxiv.org/abs/1609.04747 (visited on 11/22/2021).

[7] D. Bank, N. Koenigstein, and R. Giryes, "Autoencoders," *arXiv:2003.05991 [cs, stat]*, Apr. 2021, arXiv: 2003.05991. [Online]. Available: http://arxiv.org/abs/2003.05991 (visited on 11/07/2021).

[8] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *arXiv: 1409.0575 [cs]*, Jan. 2015, arXiv: 1409.0575. [Online]. Available: http://arxiv.org/abs/1409.0575 (visited on 04/05/2021).

[9] M. Lin, Q. Chen, and S. Yan, "Network In Network," *arXiv: 1312.4400 [cs]*, Mar. 2014, arXiv: 1312.4400. [Online]. Available: http://arxiv.org/abs/1312.4400 (visited on 11/28/2020).

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," in *Intelligent Signal Processing*, IEEE Press, 2001, pp. 306–351.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[12] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arXiv: 1409.1556 [cs]*, Apr. 2015, arXiv: 1409.1556. [Online]. Available: `http://arxiv.org/abs/1409.1556` (visited on 11/28/2020).

[13] S. Kaul, *Gradient Descent Problems and Solutions in Neural Networks*, en, Mar. 2020. [Online]. Available: `https://medium.com/analytics-vidhya/gradient-descent-problems-and-solutions-in-deep-learning-8002bbac09d5` (visited on 11/28/2020).

[14] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: JMLR Workshop and Conference Proceedings, 2010, pp. 249–256. [Online]. Available: `http://proceedings.mlr.press/v9/glorot10a.html`.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *arXiv: 1409.4842 [cs]*, Sep. 2014, arXiv: 1409.4842. [Online]. Available: `http://arxiv.org/abs/1409.4842` (visited on 11/29/2020).

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," *arXiv: 1512.00567 [cs]*, Dec. 2015, arXiv: 1512.00567. [Online]. Available: `http://arxiv.org/abs/1512.00567` (visited on 11/29/2020).

[17] *Keras: the Python deep learning API.* [Online]. Available: `https://keras.io/` (visited on 11/17/2020).

[18] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks," *arXiv: 1211.5063 [cs]*, Feb. 2013, arXiv: 1211.5063. [Online]. Available: `http://arxiv.org/abs/1211.5063` (visited on 02/09/2021).

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *arXiv: 1512.03385 [cs]*, Dec. 2015, arXiv: 1512.03385. [Online]. Available: `http://arxiv.org/abs/1512.03385` (visited on 02/09/2021).

[20] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," *arXiv: 1602.07261 [cs]*, Aug. 2016, arXiv: 1602.07261. [Online]. Available: `http://arxiv.org/abs/1602.07261` (visited on 02/09/2021).

[21] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated Residual Transformations for Deep Neural Networks," *arXiv: 1611.05431 [cs]*, Apr. 2017, arXiv: 1611.05431. [Online]. Available: `http://arxiv.org/abs/1611.05431` (visited on 02/09/2021).

[22] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," *arXiv: 1905.11946 [cs, stat]*, Sep. 2020, arXiv: 1905.11946. [Online]. Available: `http://arxiv.org/abs/1905.11946` (visited on 02/09/2021).

[23] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv: 1611.01578 [cs]*, Feb. 2017, arXiv: 1611.01578. [Online]. Available: `http://arxiv.org/abs/1611.01578` (visited on 03/25/2021).

[24] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," *arXiv: 1807.11626 [cs]*, May 2019, arXiv: 1807.11626. [Online]. Available: `http://arxiv.org/abs/1807.11626` (visited on 04/05/2021).

[25] H. I. Fawaz, B. Lucas, G. Forestier, C. Pelletier, D. F. Schmidt, J. Weber, G. I. Webb, L. Idoumghar, P.-A. Muller, and F. Petitjean, "InceptionTime: Finding AlexNet for Time Series Classification," *Data Mining and Knowledge Discovery*, vol. 34, no. 6, pp. 1936–1962, Nov. 2020, arXiv: 1909.04939, ISSN: 1384-5810, 1573-756X. DOI: 10.1007/s10618-020-00710-y. [Online]. Available: http://arxiv.org/abs/1909.04939 (visited on 04/13/2021).

[26] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *arXiv:1412.6980 [cs]*, Jan. 2017, arXiv: 1412.6980. [Online]. Available: http://arxiv.org/abs/1412.6980 (visited on 04/13/2021).

[27] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *arXiv:1505.04597 [cs]*, May 2015, arXiv: 1505.04597. [Online]. Available: http://arxiv.org/abs/1505.04597 (visited on 04/03/2022).

[28] A. Jansson, E. J. Humphrey, N. Montecchio, R. M. Bittner, A. Kumar, and T. Weyde, "Singing Voice Separation with Deep U-Net Convolutional Networks," in *ISMIR*, 2017.

[29] R. Hennequin, A. Khlif, F. Voituret, and M. Moussallam, "Spleeter: a fast and efficient music source separation tool with pre-trained models," *Journal of Open Source Software*, vol. 5, no. 50, p. 2154, 2020, Deezer Research. DOI: 10.21105/joss.02154. [Online]. Available: https://doi.org/10.21105/joss.02154.

[30] L. Prétet, R. Hennequin, J. Royo-Letelier, and A. Vaglio, "Singing Voice Separation: A Study on Training Data," in *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2019, pp. 506–510. DOI: 10.1109/ICASSP.2019.8683555.

[31] Z. Rafii, A. Liutkus, F.-R. Stöter, S. I. Mimilakis, and R. Bittner, *The MUSDB18 corpus for music separation*, Dec. 2017. DOI: 10.5281/zenodo.1117372. [Online]. Available: https://doi.org/10.5281/zenodo.1117372.

[32] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," *arXiv:2004.10934 [cs, eess]*, Apr. 2020, arXiv: 2004.10934. [Online]. Available: http://arxiv.org/abs/2004.10934 (visited on 07/01/2021).