DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning for Determining Audio Similarity

Jonathan Rösner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Deep Learning for Determining Audio Similarity

Deep Learning zur Bestimmung von Audio Ähnlichkeit

Author:	Jonathan Rösner
Supervisor:	Prof. Dr. Simon Hegelich
Advisor:	Prof. Dr. Simon Hegelich
Submission Date:	11 / 15 / 2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 11 / 15 / 2020 Jonathan Rösner

Acknowledgments

I would like to thank my supervisor, Professor Dr. Hegelich. By giving me the opportunity to write this thesis he showed me that universities truly are magnificent places that let young scientists find their way in this world by letting them follow their passion. I also want to thank my partner Elisabeth, who always supports me in whatever I do, I love you. Lastly I want to thank my parents and brother who lay the foundation to everything I am and ever will be.

Abstract

We propose a deep learning framework for similarity learning of audio representations. It is inspired by the most recent successes in self-supervised representation learning in the domain of image data. Our framework transfers those successes to the domain of audio data. With our framework we show that (1) self-supervised contrastive learning can successfully learn robust audio representations without the need for labels, (2) the learned representations can be applied to other audio-based downstream tasks using transfer learning and (3) we show that our approach outperforms recently published results in the area of few-shot classification for audio data. We further describe the preliminary knowledge in signal processing and deep learning required to understand the inner workings of our proposed framework and investigate the most recent and most important related work in this field of research.

Kurzfassung

In dieser Arbeit stellen wir ein neues Deep-Learning Framework zum Lernen von Ähnlichkeiten zwischen Audiorepräsentationen vor. Diese Arbeit ist inspiriert von den kürzlich erschienenen Erfolgen von “self-supervised contrastive learning” im Bereich der Bilderkennung. Unser Framework übernimmt diese neuesten Erkenntnisse und wendet sie auf Audiodaten an. Mit unserem Framework zeigen wir, dass (1) “self-supervised contrastive learning” ohne gelabelte Daten robuste Audiorepräsentationen lernen kann, (2) dass die gelernten Repräsentationen mittels “transfer learning” auf andere Bereiche übertragen werden können und (3) dass unser Ansatz kürzlich erschienene Arbeiten im Bereich “few-shot classification” übertrifft. Zudem erklären wir die Grundlagen in den Bereichen Signalverarbeitung und Deep-Learning, die notwendig sind, um unser Framework zu verstehen und untersuchen die aktuellsten und relevantesten Arbeiten in diesem Bereich.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Problem Formulation . . . 3
2.2 Learning Paradigms . . . 3
2.2.1 Supervised Learning . . . 4
2.2.2 Self-Supervised Learning . . . 4
2.2.3 Contrastive Learning . . . 6
2.2.4 Transfer Learning . . . 6
2.3 Self-Supervised Loss Functions . . . 8
2.3.1 Triplet Margin Loss . . . 9
2.3.2 Normalized Temperature-Scaled Cross Entropy . . . 10
2.4 Neural Networks for Audio Data . . . 10
2.4.1 Convolutional Neural Network . . . 11
2.4.2 Long Short-Term Memory . . . 12
2.4.3 Self-Attention . . . 14
2.5 Fast Fourier Transformation . . . 15
2.6 Filters . . . 18
2.7 Phase Vocoder . . . 20

3 Related Work **21**
3.1 Similarity Learning of Representations . . . 21
3.2 Few-Shot Audio Classification . . . 22
3.3 Self-Supervised Few-Shot Transfer Learning . . . 23

4 Contrastive Learning for Audio **24**
4.1 Learning Framework . . . 24
4.1.1 Pre-Training . . . 24
4.1.2 Transfer . . . 27
4.1.3 Fine-Tuning . . . 28
4.2 Data Preprocessing . . . 28
4.3 Augmentation . . . 28
4.3.1 Crop . . . 29
4.3.2 Gain . . . 30
4.3.3 White Noise . . . 31
4.3.4 Low-pass and High-pass Filter . . . 31

4.3.5 Time-Stretch . . . 33
4.3.6 Pitch-Shift . . . 34
4.4 STFT Enhancement . . . 35

5 Experiments **37**
5.1 Datasets . . . 37
5.1.1 Self-Supervised Pre-Training Dataset . . . 37
5.1.2 VoxCeleb . . . 38
5.1.3 Music Classification . . . 39
5.1.4 British Birdsong Dataset . . . 39
5.2 Encoder Model Architectures . . . 39
5.2.1 VGG-Vox . . . 39
5.2.2 LSTM . . . 41
5.2.3 Speech Transformer . . . 42
5.3 Evaluation Metrics . . . 45
5.4 Implementation Details . . . 45
5.5 Results . . . 46

6 Conclusion and Future Work **51**

List of Figures **54**

List of Tables **55**

Acronyms **56**

Bibliography **58**

1 Introduction

In this thesis, we explore the use of deep learning and neural networks for finding similarities in audio data. For this, we use recent discoveries in the area of deep learning and combine them with established methods of signal processing to create a novel learning framework suited for the task of finding similarity patterns in the highly complex domain of audio data.

Contrastive learning, the task of learning to distinguish similar pairs of data from dissimilar ones, has become very successful in recent years to train machine learning classifiers. This was made possible mainly due to new methods leveraging on the ability to learn without the need for human-annotated examples. Instead, those methods made use of what is now known as self-supervised learning, a learning paradigm that does not require any human supervision. Most prominent in the domain of image classification was the work of Chen et al. [1] released earlier this year called "A Simple Framework for Contrastive Learning of Visual Representations (SimCLR)". The authors showed that their proposed framework could outperform classical methods by a large margin without the need for labeled data. Since then several papers have been published that back the strong results of SimCLR and self-supervised learning in the domain of image data ([2, 3, 4]). In this work, we show that similar results can be obtained in the domain of audio data with slight changes to the framework itself.

Deep neural networks have a long history in their application of compressing information from high-dimensional data to dense representations. The first major work to apply this property to problems in the domain of audio data was Hinton et al. in 2012 [5]. After their work the use of such modern learning architectures found increasing popularity by researchers of several fields, such as speech recognition [5], audio classification [6] or speaker verification [7]. In this work, we also make use of neural networks to create dense representations of audio data (also known as embeddings). Such embeddings can then be more easily compared for similarities by a machine. We also show that our framework is detached from the precise architecture of the neural network that is used to create the embeddings. Therefore it is possible to plug in any model architecture into the framework. We demonstrate this property by evaluating the framework using three very different modern architectures.

To make the framework applicable to several downstream tasks, such as audio classification or speech recognition, we introduce the paradigm of transfer learning into the framework. Transfer learning is a concept that was first introduced by L. Y. Pratt in 1993 [8]. At its core transfer learning gives a neural network the ability to take the knowledge obtained from one task and transfer it to another one. Hence we split our learning framework into two stages: *i*) A pre-training stage that uses self-supervision and a large, unlabeled dataset to learn similarities in audio data and *ii*) a transfer stage where the network trained in *i* is transferred to three different domains to solve the downstream task of audio classification. We show that

this way we beat the most recent results for few-shot classification. Few-shot classification means that the domain that is being transferred to has little or no training data, which occurs when the creation of such datasets is too difficult. The reasons for this can be manifold. For music, intellectual property rights make it impossible to create an open dataset. For the annotation of complex audio events, trained humans have to manually annotate the data in a slow, cumbersome process, making the creation of a large-scale dataset time consuming and expensive. By pre-training our network on unlabeled data we overcome this problem and make it possible to obtain good results in the transfer domain, even when labeled training data is scarce.

This thesis is structured as follows: First (Chapter 2) we briefly explain the most important preliminary knowledge in the fields of deep learning and signal processing that is required to understand this work. Afterward, we look at the most relevant related work in Chapter 3, especially at the contributions of SimCLR and other self-supervised, contrastive methods. In Chapter 4 we go into the details of our proposed framework. We explain how it is structured and do a step by step analysis of each of the necessary components. Chapter 5 contains all the experimental setup and evaluation done to demonstrate the results of the framework. Lastly (Chapter 6) we conclude the findings of this thesis and propose potential future work.

2 Preliminaries

This chapter explains the most important preliminary knowledge required for this work. An in-depth analysis of all of the concepts explained in the remainder of this chapter would be out of scope for this thesis, therefore we limit it to a brief but concise overview. For the readers that are interested in gaining deeper insights we recommend the books "Scientist and Engineer's Guide to Digital Signal Processing" by Steven W Smith [9] and "Deep Learning (Adaptive Computation and Machine Learning series)" by Ian J Goodfellow [10].

2.1 Problem Formulation

Much of the success of deep learning comes from the availability of large, open datasets of high quality. There is a strong correlation between the release of such datasets and the increase in deep learning progress in its corresponding domain. The best example being *ImageNet* [11] for image data. The major reason for this is the reproducibility and comparability of works on such datasets. Unfortunately, datasets of such quality and size do not yet exist for audio data. The datasets that do exist are usually of lower quantity and quality, so therefore training good neural networks is a difficult task.

The problem that we are trying to solve can be broken down into three subproblems: *i*) find a way to make use of unlabeled data to train a neural network in finding similarities in pairs of audio data, *ii*) explore ways to transfer the network trained in *i* to another domain *iii*) show that by combining *i* and *ii* we can solve complex downstream classification tasks where only a few quantities of labeled training data exist.

2.2 Learning Paradigms

There exist many different learning paradigms in the realm of gradient-based machine learning today. In this section, we define and differentiate four of the most important paradigms for our work: *Supervised Learning*, *Self-Supervised Learning*, *Contrastive Learning* and *Transfer Learning*. We later describe how these paradigms can be combined to create new, sophisticated learning methods.

Note that the modern literature of deep learning does not fully agree on the definition and differentiation of some of these terms, so for example some would call an approach unsupervised while others would call it self-supervised. We therefore try to use the most recent definitions of the terms and stick to them throughout this thesis but beware that other works might have slightly different definitions.

2.2.1 Supervised Learning

Supervised learning is the most used and most successful learning paradigm for neural networks today. Its goal is to learn a mapping from unknown inputs to specific outputs based on a set of given input-output pairs called training data. In the case of classification, the outputs are called labels and can be seen as a certain input belonging to one of many predefined categories, e.g in the case of hand-written number-classification labels would be in the range of 0 to 9 where each label corresponds to the same number-category. Learning such a mapping is achieved by minimizing the discrepancy between the predicted label and the actual label. Equation 2.1 shows such a target, usually called the loss function, using the de-facto standard function for classification called Categorical Cross-Entropy (CCE).

$$CCE(p,t) = -\sum_{c=0}^{N-1} t_c log(p_c) \tag{2.1}$$

t_c denotes the actual probability that one input is of category c. p_c is the probability a classifier assigns to an input being of category c. Often times the words category and class are used synonymously. In our case t_c is a sparse vector where all entries are 0 except for a 1 at the position of the correct class. This is also known as a one-hot vector. Note that p_c must be a probability, meaning $\sum_c p_c = 1$ and $p_c \in (0,1) \forall c$. So outputs of classifiers that are not probabilities, as is the case in most neural networks, must first be normalized. Usually, this is achieved using the softmax function, defined in Equation 2.2, on the outputs. Here z denotes a vector of size K, for example p_c with $K = |c|$.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.2}$$

In fact this procedure is so common that today most deep learning frameworks implicitly apply softmax to the inputs of CCE.

Simply by learning this mapping from predefined input-output pairs, a trained model can learn to correctly classify unknown inputs. A neural network can learn this mapping so well that in some cases it can outperform even humans. Because of its rather simplistic paradigm and its superb results, supervised learning has become the most used way to train neural networks today. Unfortunately, it requires a lot of data to generalize well and training a network on one specific task does not provide a good mapping for another task, even if the two tasks might be closely related. Since modern architectures usually contain billions of parameters that have to be tuned on incredibly large datasets, retraining a network for each task requires enormous resources, time and money. Other paradigms are required to mitigate this problem.

2.2.2 Self-Supervised Learning

Self-supervised learning is a special case of unsupervised learning. In unsupervised learning, a model tries to find inherent patterns in a dataset without labels. An example of this is clustering or principal component analysis [12]. Self-Supervised learning has the same target

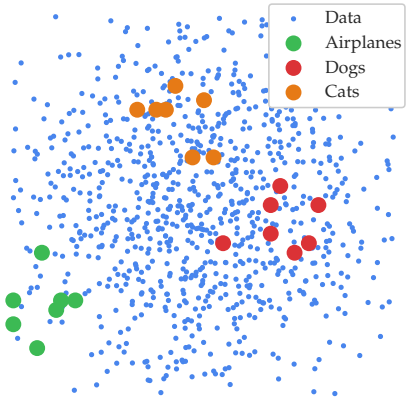


Figure 2.1: An example of a latent space where similar data points would be mapped closer together while not similar ones would be further apart.

as unsupervised learning but obtains labels from the data itself. This means that from one or many data points, a supervised task is generated, which is then evaluated in the same way a supervised system would be. Choosing the exact nature of this task is a critical part of self-supervised learning. There have been many proposed tasks for such systems but all fall into either one of two categories: generative or discriminative. An example of a generative task is proposed in Hossein et al. (2020) [13] where the authors try to predict the next frames in a video by minimizing the Mean Squared Error (MSE) between an image generated by a neural network and the actual next frame. On the other hand, an example of a discriminative task is Gidaris et al. (2018) [14]. Here the authors try to find the original image out of four rotated copies of that image. Discriminative methods, such as contrastive learning explained in 2.2.3, try to distinguish different kinds of inputs.

In general the target in self-supervised learning is "a proxy task that forces the network to learn what we really care about" [15]. *Word2Vec*, proposed by Mikolov et al. [16], though not called self-supervised by the authors, can be seen as such a system that, by predicting words in a sentence, is forced to learn a semantic representation of those words. In our case what we care about is a representation of audio input into a latent space where close distance is equivalent to a close semantic distance in the real world, e.g. perceived similarity in music or speech. We call this semantic distance the similarity of two data points.

Figure 2.1 shows how we think of similarity in a made-up scenario of differentiating input images of certain categories. Close points in this two-dimensional plane are considered similar, while further away points are considered not similar. Therefore data of the same class should be mapped close together since they are of high similarity. Note that this similarity measure can vary from being strictly objective in the case of same speakers to highly subjective in the case of song genres. One can clearly see how this paradigm can be used to solve problem *i* defined in 2.1.

2.2.3 Contrastive Learning

The problem of learning similarity between a pair of inputs is a common problem in machine learning. This problem consists of three pieces: *i*) A way to represent the data that is comparable by a machine, *ii*) a notion of similarity in the target domain and *iii*) labels that describe the membership of each input to a certain class. Neural networks can be used to solve *i* by compressing the input signal into a dimensionality-reduced representation, without loosing too much information. For vectors, *ii* can be solved easily since there are several functions that compute distance or similarity for a pair of vectors. See section 2.3 for more details. As explained in section 2.2.1 and 2.2.2 labels required for *iii* can be obtained either by human annotation or by creating a task that the machine can produce itself. Contrastive learning is a discriminative method that combines those three solutions to form a learning paradigm that learns to distinguish pairs of inputs by producing vectors that are either close or distant to each other. Figure 2.2 shows the general task of contrastive learning in the domain of image data.

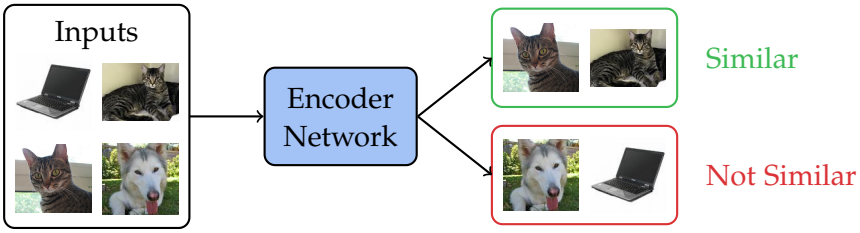


Figure 2.2: The general contrastive learning task in the domain of image data. Different images are fed through an encoder network that predicts whether or not two images are similar or not. Images are from *ImageNet* [11].

As explained earlier labels can be created automatically using a self-supervised learning task. By combining self-supervised and contrastive learning it is possible to create a fully autonomous learning framework for similarities without the need for human annotation. This makes it possible to scale the training dataset to a much larger amount than in a fully supervised scenario. The way that we make use of this property is to create two augmented views from the same input and train the network to become invariant to those augmentations, meaning it will produce similar vectors for augmented views, but not for two completely unrelated views. This concept will be explained in more detail in Chapter 4.

2.2.4 Transfer Learning

Learning basic concepts first and applying those concepts to other tasks later on is an important part of deep learning research today. Transfer Learning can be seen as an extension to other learning paradigms that enables the transference of knowledge from one domain to another [17]. A method for transfer learning of neural networks was proposed as early as 1993 by L. Y. Pratt [8] and is described in more detail in Algorithm 1.

The basic idea behind transfer learning is that a neural network $f \circ g$, where f is an

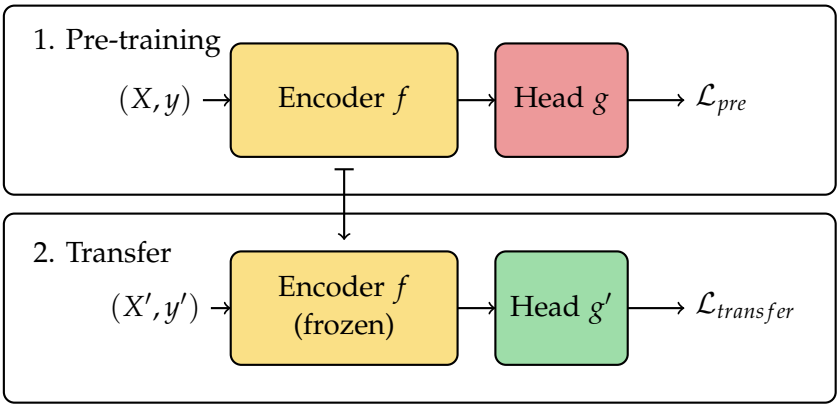


Figure 2.3: The two stages of transfer learning. The encoder network is transferred after pre-training from the first domain to the second domain. Head is usually a small fully-connected network.

arbitrary network architecture called encoder network that outputs vectors and g is a small fully connected network called classification head that takes in the outputs of f and produces predictions in the target domain. This network can be trained on a dataset (X, y), containing training inputs X and training outputs y, for example, images and the class of the depicted object in each image. During training the network f learns general low-level features of image recognition while g learns to map those features to the desired output y.

Now consider a second domain of image classification (X', y') where X' is of the same kind as X. Instead of training an entirely new network of the aforementioned structure, one can instead take the encoder network f trained on the other dataset and transfer it into a new architecture $f \circ g'$ where g' is a new classification head that takes in the outputs of f and produces predictions in the new target domain y'. This is possible because the underlying data of both datasets share similar features, like edges and objects. The encoder network f learns those features from the first dataset and transfers the knowledge to the second dataset. Shaha et al. [18] showed that using transfer learning on neural networks robust features can be transferred between multiple vision tasks. Figure 2.3 shows this idea in a single diagram.

The need for transfer learning arises when the domain to be transferred to has little or no training data or when making mistakes in unknown situations is expensive, as is the case in self-driving cars [19]. In our case transfer learning will be used to take the latent representations learned from self-supervision and apply them to downstream supervised tasks with low amounts of available training data. This approach is very common and has shown good results in recent years ([20, 21, 22]). A thorough survey of modern transfer learning techniques can be found in Pan et al. (2010) [23].

Algorithm 1 A supervised transfer-learning framework

Input: encoder network f, classification heads g and g', trainable parameters $\boldsymbol{\theta}_f$, $\boldsymbol{\theta}_g$, $\boldsymbol{\theta}_{g'}$, pre-training dataset (X, Y), transfer dataset (X', Y'), loss functions $\mathcal{L}_{pre}$, $\mathcal{L}_{transfer}$, learning rate α
Randomly initialize θ_f and θ_g ▷ Stage 1: Pre-training
while not done **do**
    sample minibatch $(\boldsymbol{x}, \boldsymbol{y}) \sim (X, Y)$
    compute $\mathcal{L}_{pre}$ from $g(f(\boldsymbol{x}))$ and $\boldsymbol{y}$
    $\boldsymbol{\theta}_{f\circ g} \leftarrow \boldsymbol{\theta}_{f\circ g} - \alpha \nabla_{\theta_{f\circ g}} \mathcal{L}_{pre}$ ▷ $\theta_{f\circ g}$ is the concatenation of θ_f and θ_g
end while
Randomly initialize $\boldsymbol{\theta}_{g'}$ ▷ Stage 2: Transfer
freeze $\boldsymbol{\theta}_f$
while not done **do**
    sample minibatch $(\boldsymbol{x'}, \boldsymbol{y'}) \sim (X', Y')$
    compute $\mathcal{L}_{transfer}$ from $g'(f(\boldsymbol{x'}))$ and $\boldsymbol{y'}$
    $\boldsymbol{\theta}_{f\circ g'} \leftarrow \boldsymbol{\theta}_{f\circ g'} - \alpha \nabla_{\theta_{f\circ g'}} \mathcal{L}_{transfer}$
end while
return $f \circ g'$

2.3 Self-Supervised Loss Functions

Expressing the similarity of entities in the real world is effortless for humans. Be it sound events, images, words, all can easily be compared to other entities of that same domain. For machines this task is different. Since understanding similarity involves a lot of contextual knowledge that machines do not possess, the question if two entities are similar becomes much harder to answer for an algorithm. Self-supervised learning tries to bridge that knowledge gap by transforming complex real-world entities into a more simple representation that a machine can process. This representation is usually in the form of vectors. There are many ways of calculating the similarity of two vectors. Equations 2.3, 2.4, 2.5 describe three of the most common distance measures for vectors: *euclidian distance*, *manhattan distance* and *cosine similarity*. Note that $s_{cosine}(p, q)$ is actually a similarity metric where 1 is equal to the most similarity and -1 to the least similarity. One can simply convert it to a distance: $d_{cosine}(p, q) = 1 - s_{cosine}(p, q)$, but this is commonly not used. It is called cosine similarity because the result matches the cosine of the angle between the two vectors.

$$d_{euclidian}(\mathbf{p}, \mathbf{q}) = \|\mathbf{p}, \mathbf{q}\|_2 = \sum_{i=1}^{n} \sqrt{(p_i - q_i)^2}, \quad \in \mathbb{R}^+ \tag{2.3}$$

$$d_{manhattan}(\mathbf{p}, \mathbf{q}) = \|\mathbf{p}, \mathbf{q}\|_1 = \sum_{i=1}^{n} |p_i - q_i|, \quad \in \mathbb{R}^+ \tag{2.4}$$

$$s_{cosine}(\mathbf{p}, \mathbf{q}) = \frac{\mathbf{p} \cdot \mathbf{q}}{\|\mathbf{p}\| \|\mathbf{q}\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \sqrt{\sum_{i=1}^{n} q_i^2}}, \quad \in [-1, 1] \tag{2.5}$$

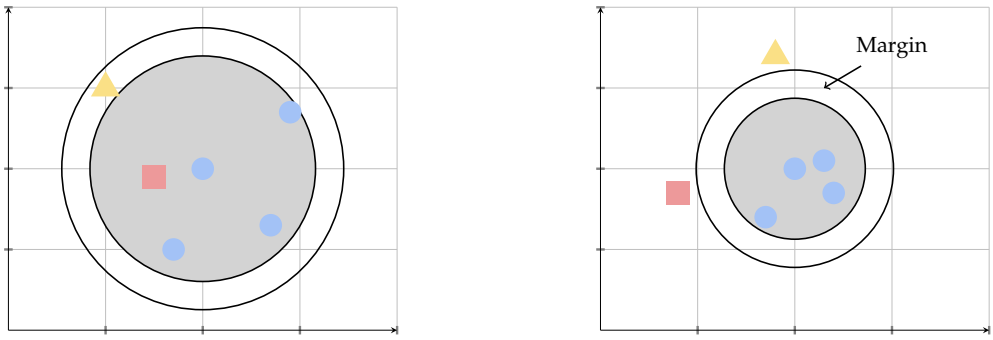
In the following subsections we look at two loss functions that implement these distance metrics to give self-supervised algorithms a way to learn good vector representations. A loss function is a mathematical way of expressing a certain target for a neural network to train towards to. Stochastic gradient descent works by minimizing this loss function over time using stochastically sampled batches of data.

2.3.1 Triplet Margin Loss

As the name suggests the *Triplet Margin Loss* requires triplets of input data points. These data points are called anchor A, positive P and negative N. The positive is supposed to be of high similarity to the anchor, whereas the negative is supposed to be of no or low similarity to the anchor. Equation 2.6 describes this notion in mathematical terms.

$$\mathcal{L}(A, P, N) = max\Big(\|f(A), f(P)\|_2 - \|f(A), f(N)\|_2 + \alpha, 0\Big) \tag{2.6}$$

This loss becomes larger the higher the distance between A and P and the lower the distance between A and N. Here the euclidean distance is used but any distance function can be applied. By minimizing this loss the network is forced to learn representations that increase the distance to N and decrease the distance to P, relative to A. Figure 2.4 shows how such representations could look like before and after learning. Note that usually a margin hyperparameter α is used to better distinguish positive from negatives, it's effect can be seen in 2.4b.



(a) Before training

(b) After training

Figure 2.4: **Left**: Potential latent representations of input data points before training. They are mapped randomly into the latent space with no clear boundary. **Right**: After training with the triplet loss representations of the same class are pulled together while other classes are pushed apart. A margin zone, controlled by the margin hyperparameter, separates uncorrelated samples from correlated ones.

Since its first proposal in 2009 by Weinberger et al. [24] the triplet loss is one of the most used loss functions for self-supervised learning. One major drawback of triplet loss is the fact that for randomly chosen inputs the loss approaches 0 very quickly since for most

randomly chosen inputs $d(A, N) > d(A, P)$. These examples become irrelevant for training once this threshold is reached. Therefore a novel technique called online hard negative triplet mining was introduced by Schroff et al. [25]. Online hard negative triplet mining ensures consistently increasing difficulty of triplets as the network trains but it requires one to track all comparisons and build up a memory bank that stores previous results. This is very inefficient and therefore more modern approaches try to replace it with simpler and more effective methods.

2.3.2 Normalized Temperature-Scaled Cross Entropy

Normalized Temperature-Scaled Cross Entropy (NT-Xent) loss was first proposed by Sohn in 2016 [26] but was coined *NT-Xent* only recently by Chen et al. [1]. It was found that triplet loss functions "often suffer from slow convergence and poor local optima, partially due to that the loss function employs only one negative example while not interacting with the other negative classes per each update" [26]. To deal with this issue, NT-Xent loss takes one positive example and multiple negative examples per training step. The loss is defined as follows:

$$\ell(i, j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)} \tag{2.7}$$

Here $\mathbb{1}$ is the indicator function which evaluates to 1 only if $k \neq i$ (the similarity of an input to itself is always 1 and therefore discarded). Usually for this loss $s_{i,j}$ denotes the cosine similarity of two vector representations but any similarity measure can be used. τ is a temperature hyperparameter used to expand the range of the exponential. This helps in stabilizing training. One can clearly see that this loss closely resembles a softmax distribution trying to maximize agreement of similar pairs.

The NT-Xent loss has gained a lot of popularity since the release of SimCLR ([27], [28], [29]), mainly due to it scaling well to large batch sizes. This can be leveraged by ever-increasing memory sizes in deep learning specific processing units and better distribution strategies. Because of this and the fact that it does not require explicit example mining, we believe that NT-Xent will replace triplet loss in most applications for self-supervised learning.

2.4 Neural Networks for Audio Data

Vibration is a repetitive motion relative to an equilibrium point [30]. Sound is therefore the vibration of molecules, usually air molecules, that is picked up by our ears and transformed into electrical signals that are then perceived by our brains. A similar thing happens when we try to digitize sound. First, a transducer converts sound into an electrical signal. This continuous signal is then transformed into a discrete signal by an *Analog-to-digital converter* (ADC). The signal is periodically quantized to create a stream of samples. This way of digitally representing an analog signal is called Pulse-Code Modulation (PCM).

Quantization maps a certain amplitude of an analog signal to a digital integer value, represented as a sequence of bits. The higher the number of bits per integer, also known as the bit-depth, the lower the error introduced by quantization. The frequency of this quantization

is called the sampling frequency, usually denoted in kHz. The sampling frequency and the quantization error are the major factors of the quality of the digital audio signal. The higher the sampling rate and the bit-depth, the better the quality, but the more space is required to store the audio. In today's recordings, a sampling rate of 44.1kHz and a bit-depth of 16bit is most commonly used.

The resulting sequence of integer values can then be fed into a neural network for further processing. Unfortunately, classical network architectures are practically limited by the number of input values. A fully connected network with a single hidden layer consisting of 2048 hidden neurons would require more than 270 million weights to process a 3-second audio clip sampled at 44.1kHz. Nowadays it is common to make use of the time-series quality of audio data and use architectures that are specialized in this domain as shown in Lezhenin et al. [31] or Zhao et al. [32].

Another common approach is to exploit the local dependency of samples in an audio signal by first applying convolutional layers to reduce the input dimensionality while preserving as much information as possible. It was also found that fully convolutional networks can produce remarkable results without any need for time-dependent computation [33]. Most of the architectures used today are combinations of the three. All of these network types can be used with one-dimensional data, as most recently shown in [34] but are more often used with two-dimensional inputs. We therefore always transform our input signal using the Short-time Fourier transform described in 2.5 to create a two-dimensional matrix where $x_{i,j}$ represents the ith Fourier coefficient at timestep j. This process reduces the number of timesteps significantly and thus reduces the size of the networks. The final goal is to produce a single vector that best describes the data it is trying to represent. In the remaining subsections, we dive deeper into the theory of all three architectures.

2.4.1 Convolutional Neural Network

The kernel convolution for two-dimensional inputs used in a Convolutional Neural Network (CNN) is defined in Equation 2.8 for discrete inputs.

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy) \tag{2.8}$$

Here ω is the kernel matrix ranging from $[-a,-b]$ to $[a,b]$. $f(x,y)$ is a pixel of an input image f. The kernel ω is shifted over the two dimensions x and y of f and repeatedly multiplied with a part of the input image centered around $f(x,y)$. In the end, the resulting output pixels $g(x,y)$ are put back together to produce a filtered output g. Note that the kernel does not have to move one pixel at a time but can move any desired distance per step. This distance is called stride. If the stride is larger than 1, the size of the output image is reduced.

The kernel matrix is composed of learnable weights. Training such a network using backpropagation was first proposed by LeCun et al. in 1989 [35] and has since seen magnificent results in image-related tasks but also in other domains like audio.

The advantage of this compared to fully connected layers is that the size of the kernel

matrix is not determined by the size of the input, so larger input matrices do not necessarily require more trainable weights. This kernel convolution is usually followed by a non-linear activation function. The most prominent activation function for CNNs is the Rectified Linear Unit (ReLU), proposed by Hinton et al. [36] and denoted in Equation 2.9.

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.9}$$

Due to its non-saturating gradient, it accelerates stochastic gradient-based learning, as opposed to alternatives like sigmoid or tanh [37]. Note that this comes with a drawback that neurons can potentially die out, meaning they can enter states that produce negative outputs for all inputs and therefore never produce any gradient other than 0. There exist many alternatives that mitigate this issue, like the *leaky ReLU* that assigns small values for inputs lower than zero, so therefore always producing some gradient. In practice, however, this problem is often ignored since large networks can cope with a few dead neurons.

To further reduce dimensionality to subsequent layers, CNNs commonly employ pooling after each convolution step. As evaluated in Scherer et al. [38], max-pooling, which returns the maximum value of a given receptive field, is vastly superior to other pooling techniques. Figure 2.5 shows a single stack of convolution, ReLU and max-pooling layers. In practice, there are many such stacks combined to produce a condensed set of features that best describes the given input and that can be used for further downstream tasks such as classification.

CNNs for time series classification were first introduced by Wang et al. in 2017 [39]. They form a strong baseline, that is difficult to beat on arbitrary data without heavy fine-tuning to a specific task. Even though CNNs are extremely powerful in finding local dependencies in data, they fail to learn dependencies that are far apart. This is due to the kernel convolution only looking at specific areas of the input one at a time.

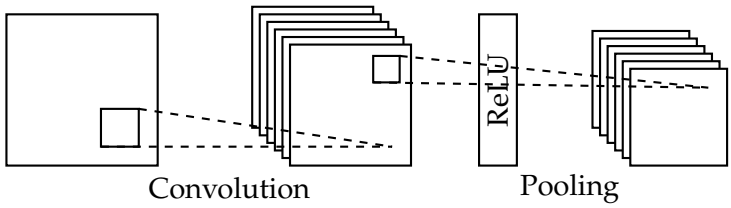


Figure 2.5: A single stack of a convolutional, ReLU and max-pooling layer. First n filters are convoluted with the input, resulting in n output channels. All output channels are then fed through a ReLU and finally max-pooled to further decrease the output size.

2.4.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) is a type of Recurrent Neural Network (RNN) used for processing time-series data. It was first proposed by Hochreiter and Schmidhuber in 1997 [40]. A common problem for pre-LSTM RNNs was to learn long-term temporal dependencies

due to exponentially decaying gradients over time. To mitigate this issue LSTM employs a series of gates, described in Equation 2.10 to 2.14, that regulate the memory state of a cell.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{c}_{t-1} + \mathbf{b}_f) \tag{2.10}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{c}_{t-1} + \mathbf{b}_i) \tag{2.11}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{c}_{t-1} + \mathbf{b}_o) \tag{2.12}$$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{b}_c) \tag{2.13}$$

$$\mathbf{h}_t = \mathbf{o}_t \circ tanh(\mathbf{c}_t) \tag{2.14}$$

Here σ denotes the sigmoid function and $\circ$ denotes the element-wise vector multiplication. $\mathbf{x}_t \in \mathbb{R}^d$ is the input at time-step t, $\mathbf{h}_t \in \mathbb{R}^h$ is the outputted hidden state at timestep t. $\mathbf{W}$, $\mathbf{U} \in \mathbb{R}^{h \times d}$ are traininable weights and $\mathbf{b} \in \mathbb{R}^h$ are trainable biases. $\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t \in \mathbb{R}^h$ are intermediate vectors called gates. Figure 2.6 shows how the different gates interact. At its core the cell state is described by Equation 2.13: a new state $\mathbf{c}_t$ is the sum of two terms. The first term $\mathbf{f}_t \circ \mathbf{c}_{t-1}$ includes the forget gate $\mathbf{f}_t$, which regulates when old state should be forgotten, while the second term $\mathbf{i}_t \circ tanh(\mathbf{W}_c \mathbf{x}_t + \mathbf{b}_c)$ includes the input gate $\mathbf{i}_t$, which regulates when new information should be remembered.

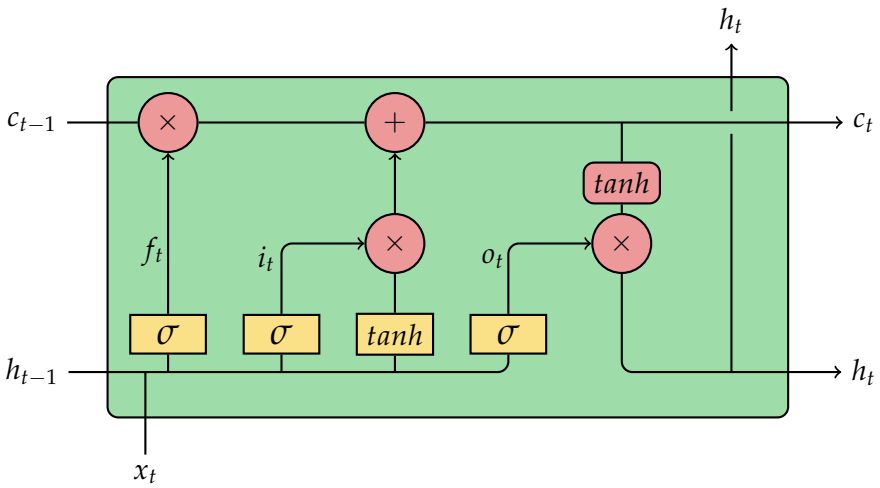


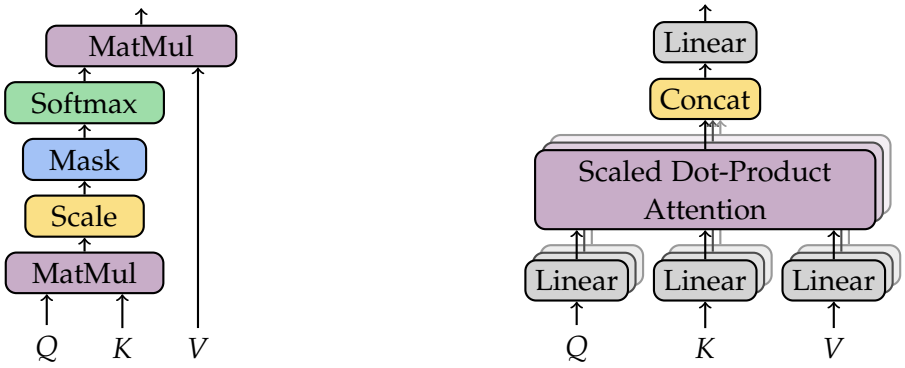
Figure 2.6: The flow through a LSTM cell. The last hidden state h_t is used as the latent representation of the input.

In Hinton et al. [41] the authors showed that LSTMs can be successfully used on audio data. But even though its performance on long time temporal dependencies is really strong, its computational speed is not as efficient as the other methods described in this section. This

comes from the fact that a LSTM cell state $\mathbf{c}_t$ depends on the previous cell state $\mathbf{c}_{t-1}$. This makes the parallelization of RNNs impossible and so the benefits of modern deep learning hardware cannot be leveraged.

2.4.3 Self-Attention

Self-Attention networks have gained a lot of popularity in the last two years mostly resulting from the success of Transformers in Natural Language Processing (NLP). This architecture was first proposed in the paper called "Attention is all you need" [42]. The *Transformer* is an encoder-decoder based architecture for sequence data that is not reliant on any recurrence whatsoever. This makes the model extremely parallelizable which in turn resulted in huge models being trained on big GPU-clusters in parallel [43]. The results of these models raised the bar for state of the art models in almost all NLP tasks by a significant amount. Figure 2.7 shows the Scaled Dot-Product Attention (often just called self-attention) and the Multi-Head Attention (MHA), both proposed by Vaswani et al. in the original *Transformer* paper [42].



(a) Scaled Dot-Product Attention

(b) Multi-Head-Attention

Figure 2.7: The basic components of a self-attention block. In practice multiple Attention-blocks are stacked onto each other to produce better representations. Figures are from the original Transformer paper [42].

Self-attention is a mechanism in which each input of a sequence can compute a certain relationship to each other input. The input of the scaled dot product attention is a linear transformation of the inputs with three different, learnable matrices, called query, key and value. All three matrices have the same output dimension. MHA extends this concept by applying multiple such self-attention computations in parallel and concatenating the results. MHA is usually preceded and followed by linear transformations. Just like multiple kernel matrices in the same convolutional layer, MHA allows the network to learn multiple representations of the input at the same time, thus decreasing the probability that information is lost.

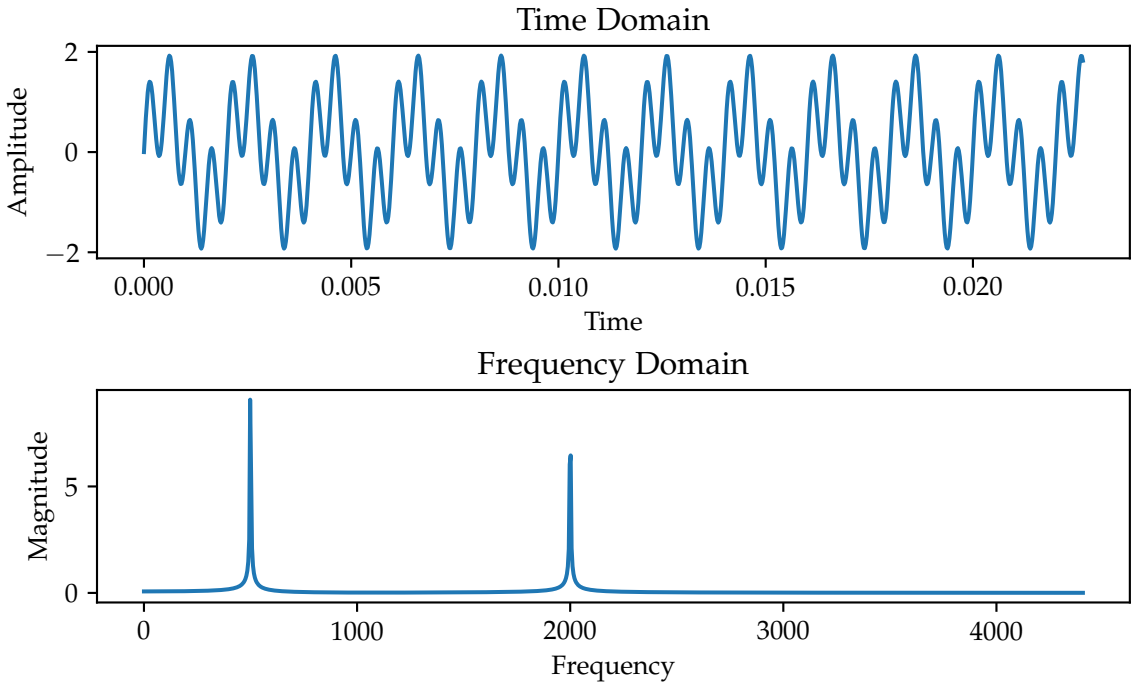


Figure 2.8: **Top:** A signal consisting of two sinusoids of $500Hz$ and $2,000Hz$. **Bottom:** The same signal transformed into the frequency domain using DFT.

2.5 Fast Fourier Transformation

The inputs to our encoder models are not raw samples of audio data but rather spectrograms of it. To understand what spectrograms are one first has to know about the Fourier transformation. The Fourier transformation is the most widely used method to break down a non-periodic signal into its individual components. In fact, all mammal ears do something quite similar to it [9]. The basic idea behind this transformation is that any signal can be represented by the sum of different sine and cosine waves. The equation of the complex Fourier transformation for discrete signals, called Discrete Fourier Transformation (DFT) is shown in Equation 2.15.

$$F[j] = \sum_{k=0}^{N-1} f[k] e^{-i2\pi kj/N}, \quad 0 \leq j \leq N-1 \tag{2.15}$$

Where f is an input signal of length N. The results $F[j]$ are complex numbers that can be seen as the amount that a specific sinusoid with frequency j contributes to the signal. It is often called the Fourier coefficient. The real part of this coefficient, the magnitude of the corresponding sinusoid, will be the input to our models, while the imaginary part, the phase, will be discarded. We chose this input representation because we believe that it most similarly

resembles the way that our brains perceive sound. In fact, the ear's basilar membrane inside the cochlea contains around 12 thousand sensory cells that function as frequency detectors, just like the coefficients of a Fourier transformation would. A visualization of this can be seen in Figure 2.8 in the case of a signal consisting of two sinusoids of $500Hz$ and $2,000Hz$ respectively. One can see that once converted into the frequency domain, both individual frequencies become clearly visible.

To convert back from the frequency domain to the time domain one can use the inverse Fourier Transformation shown in Equation 2.16. Note that both representations contain the same information, therefore one can freely convert between the two.

$$f[j] = \frac{1}{N} \sum_{k=0}^{N-1} F[k] e^{i2\pi kj/N}, \quad 0 \leq j \leq N-1 \tag{2.16}$$

Here F is a sequence of complex numbers of length N like the result of a DFT and f is the original signal back in the time domain.

Since the default DFT has a complexity $O(n^2)$, most commonly the Fast Fourier Transform (FFT) is used to calculate the DFT. It achieves the same result in $O(n \log n)$. Algorithm 2 shows the procedure for a basic FFT.

Algorithm 2 Fast Fourier Transform

Input: Complex input samples X of length N where N is a power of 2
$bit\_reverse(X)$
for all $s \in \{1, ..., log(N)\}$ **do**
    $m \leftarrow 2^s$
    $w_m \leftarrow e^{-2\pi i/m}$
    for all $k \in \{0, m, 2m, ..., N-1\}$ **do**
        $w \leftarrow 1$
        for all $j \in \{0, 1, ..., \frac{m}{2} - 1\}$ **do**
            $t \leftarrow X[k + j + \frac{m}{2}]$
            $u \leftarrow X[k + j]$
            $X[k + j] \leftarrow u + t$
            $X[k + j + m/2] \leftarrow u - t$
            $w \leftarrow w \cdot w_m$
        end for
    end for
end for
return X

To model the change of magnitudes of the frequencies over time, we employ a method called Short-Time Fourer Transform (STFT), which splits the full signal into overlapping frames and then applies the FFT to each frame respectively. This results in a matrix of shape $[t, f]$, also called spectrograms, where each entry is the magnitude of the frequency f in frame t. A plot of such a spectrogram can be seen in Figure 2.9.

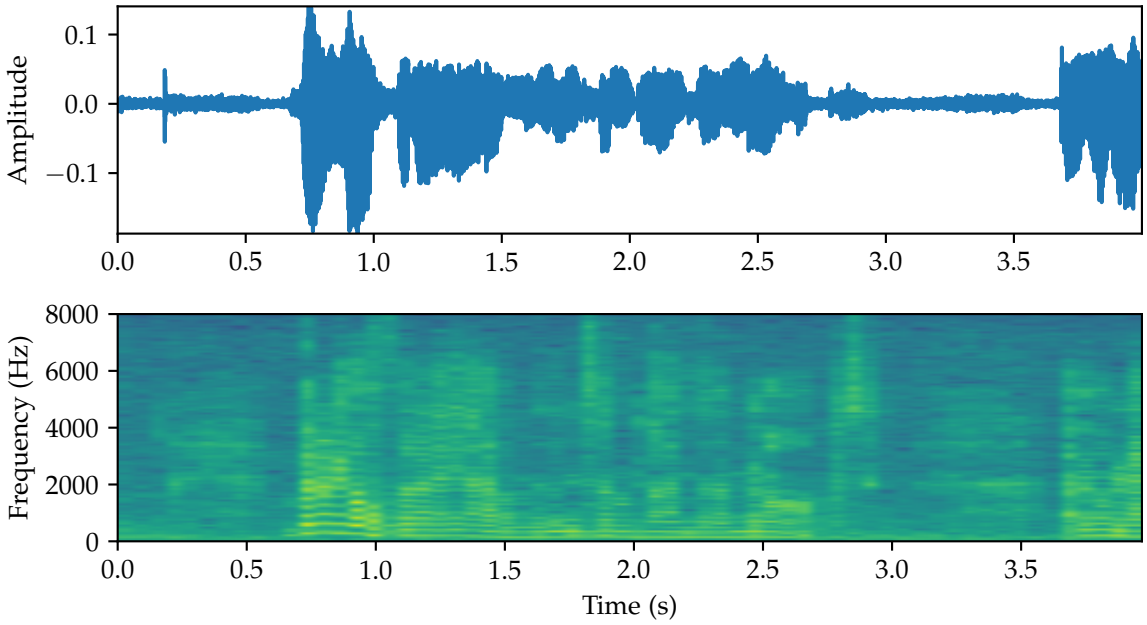


Figure 2.9: **Top:** A 4 second audio signal sampled at 16kHz. **Bottom:** The spectrogram of the same signal created using STFT with no overlap and a window size of 1024.

In a discrete signal the number of Fourier coefficients, also called FFT-bins, is determined by the number of samples collected, while the frequency resolution of each bin is determined by the sampling rate of the signal and the number of bins. Equation 2.17 and 2.18 show the relationship between the different parameters.

$$N = \frac{|x|}{2} \tag{2.17}$$

$$\Delta_{bin} = \frac{fs}{N} \tag{2.18}$$

Here N is the number of FFT-bins, $|x|$ is the number of samples. Δ_{bin} is the size per FFT-bin in Hz and fs is the sampling rate of the signal in Hz. The smaller N the better the time resolution. The smaller the bin size the better the frequency resolution. One can see that there always exists a trade-off between frequency resolution and time resolution. If we decrease the number of samples per frame of the STFT we get a better temporal resolution but therefore a worse frequency resolution. It is important to choose good parameters that fit the need of the task. The more exact frequency differentiation, the weaker the temporal differentiation becomes. For our task, a well-balanced resolution in both domains is required.

2.6 Filters

We make use of filters to create augmented views of the input data before feeding it into the neural network. Filters are one of the most important parts of Digital Signal Processing (DSP). Their applications range from telecommunication to computer graphics [44]. An abstract view of two of the most important types of filters, called *low-pass* and *high-pass* filters can be seen in Figure 2.10. The frequencies in the passband are the ones that should not be altered by the filter while frequencies in the transition band should be linearly reduced and frequencies in the stopband should be removed entirely. As can be seen in Figure 2.10 a low-pass filter stops all frequencies above the transition band, while a high-pass filter stops all below it.

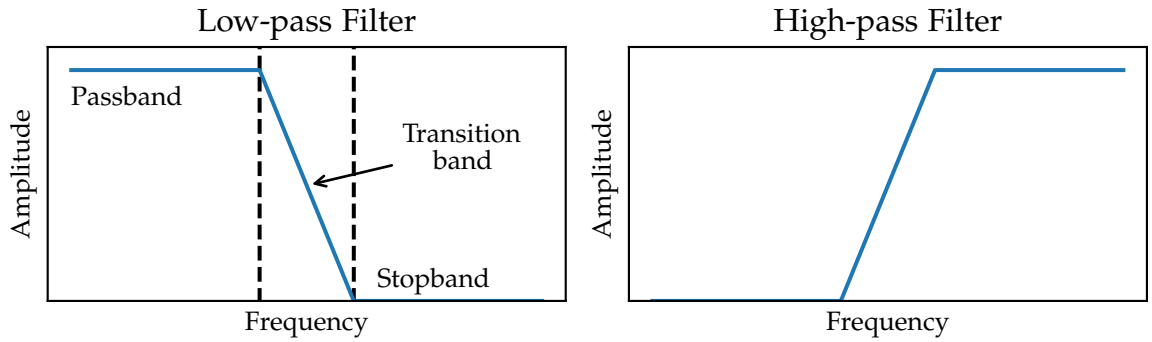


Figure 2.10: **Left:** Abstract illustration of a *low-pass* filter. **Right:** Abstract illustration of a *high-pass* filter

Figure 2.11 shows how the same filter can be represented in time and in the frequency domain. The frequency response describes how the filter will affect the individual frequencies of an incoming signal. In this example, frequencies above 100Hz will be cut off linearly. The impulse response is the actual signal of the filter in the time domain. Applying a filter in the frequency domain can be achieved by multiplying an input's frequency spectrum and the filter's frequency response. It can be easily seen why multiplication will have the desired effect of removing all frequencies in the stopband while preserving all frequencies in the passband.

It can be proven that multiplication in the frequency domain is equivalent to convolution in the time domain [45]. The convolution operation for two finite sequences f and g is defined in Equation 2.19

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m] \tag{2.19}$$

So therefore in order to apply a filter in the time domain, one has to convolve an incoming signal with the impulse response of the filter. Figure 2.12 shows how the time domain and the frequency domain are impacted when a signal is fed through a low-pass filter.

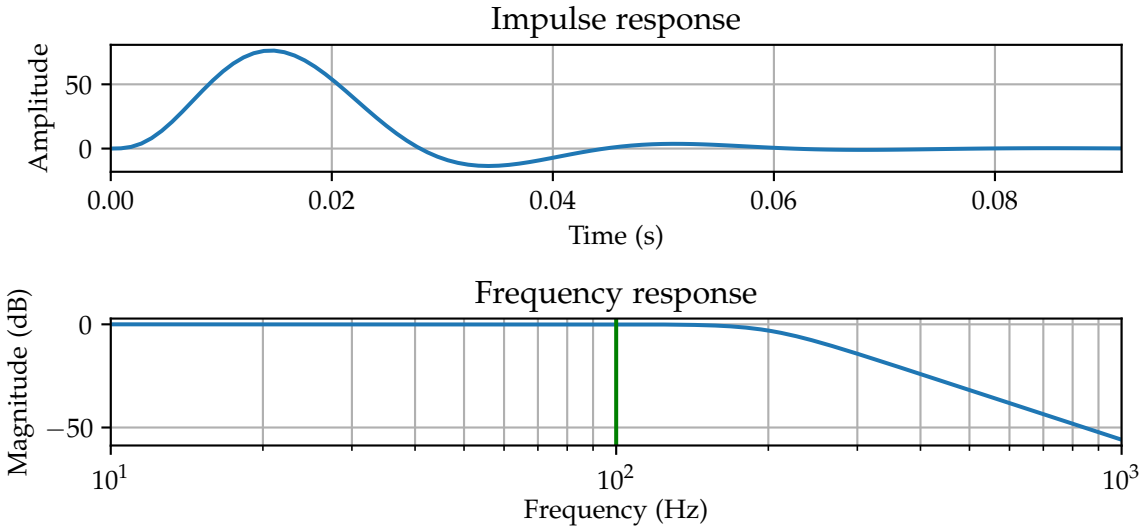


Figure 2.11: **Top:** Time-domain signature of a low-pass filter called the "Impulse response". **Bottom:** Frequency-domain signature of a low-pass filter called the "Frequency response".

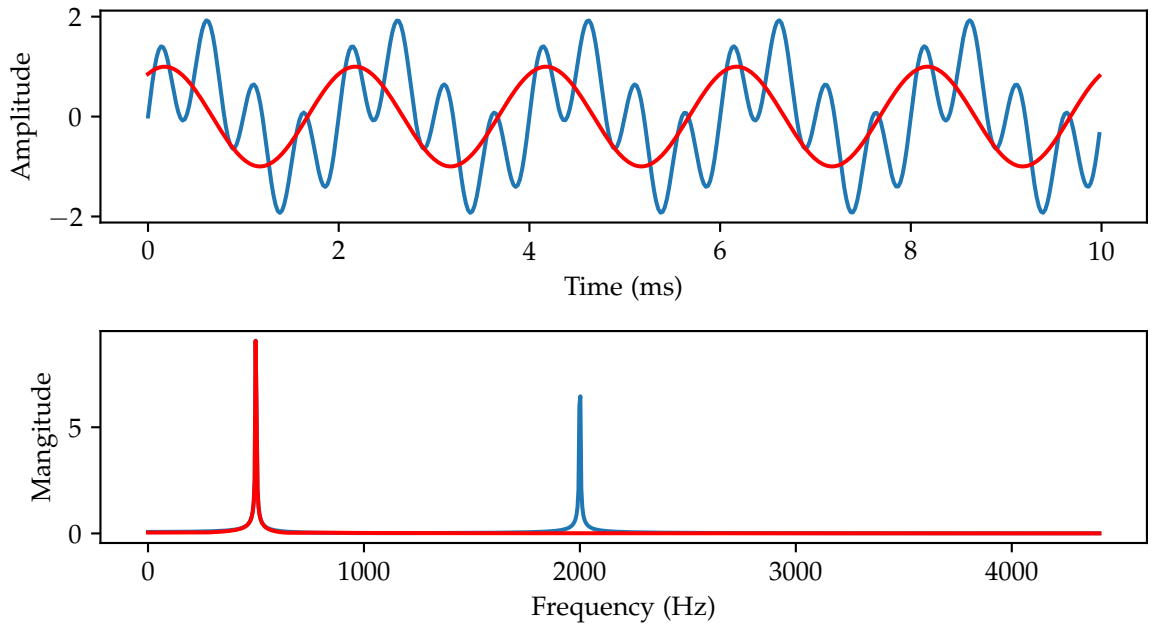


Figure 2.12: A 500 Hz + 2000 Hz sinusoid in blue and the same signal low-pass-filtered at 1000 Hz in red. **Top:** Time domain. **Bottom:** Frequency domain.

2.7 Phase Vocoder

Another augmentation that we want to apply is to speed up or slow down an audio clip. A trivial approach to speeding up an audio signal would be to skip every other sample while preserving the sampling rate of the signal. In fact this approach does have the desired effect of speeding up the signal but also changes the frequency of the signal. The Phase Vocoder proposed by Flanagan et al. in 1966 [46] is an algorithm that achieves scaling in the time domain without changing the frequency domain [47]. At its core, the phase vocoder is made of three stages. *Analysis*, *Processing* and *Synthesis*. Figure 2.13 shows a basic overview of the different stages of the phase vocoder.

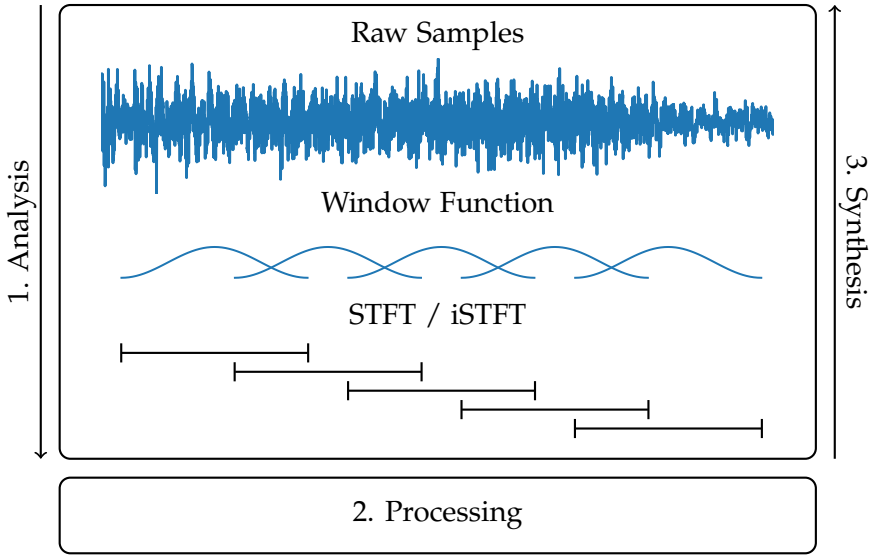


Figure 2.13: Basic overview of the phase vocoder algorithm. In the "Analysis Stage" raw samples are being transformed into the frequency domain using a window function and STFT. Then in the "Processing Stage" the audio is perturbed as required and in the "Synthesis Stage" raw samples are reconstructed from the perturbed STFT-frames using the inverse STFT.

The analysis stage leverages the STFT [48], described in Section 2.5, to split the signal into multiple frames. In the processing stage the signal can be perturbed in any desired way, e.g. the overlap between frames can be increased or decreased to achieve time-stretching. After that, in the synthesis stage, the signal is restored by applying the inverse STFT to the altered frames. Unfortunately, this naive method of time-stretching introduces unwanted artifacts in the output signal. This results from the fact that the relationship between phase and time is linear [48] and stretching changes the time domain of the signal. Therefore phase at each frame for each frequency must be adjusted accordingly by subtracting the propagating phase from one time-step to another. This adjustment of phase over time yields good results but is known to be imperfect [49]. To this day finding better solutions to this problem is an ongoing field of research.

3 Related Work

In this chapter, we look at the previous work most relevant to ours. We first look at the concept of similarity learning for dense representations of high dimensional data. Then we focus on the progress made in the field of few-shot learning with a focus on audio classification and lastly we investigate the combination of self-supervision and transfer learning to solve the problem of few-shot classification and how it is already being successfully used in the domain of image data. We focus solely on related work on deep neural network architectures trained with stochastic gradient descent.

3.1 Similarity Learning of Representations

Similarity learning, also known as metric learning or distance learning plays an important role in many machine learning or pattern recognition tasks. A wide variety of algorithms, like *K-nearest neighbor* or *K-Means* can be classified under this learning paradigm since they utilize distance metrics to find patterns in datasets. In this chapter though we focus only on one particular kind of similarity learning, namely distance evaluation on dense representations of the input data created by neural networks. Dense representations, also known as embeddings, are simply low dimensional vectors that can be compared for similarity quite easily by making use of one of the distance or similarity functions defined in Section 2.3. As explained in 2.2.3 contrastive learning is one way of training a neural network to learn the similarity of embeddings. The modern form of the contrastive loss function used in contrastive learning was first proposed in Hadsell et al. 2006 [50]. There have been many extensions to these initial proposals, such as using a memory bank to store the representations [51]. Even though these papers have shown good results they have also been criticized for becoming too complex for the marginal gains they provide [1]. Hinton et al. (2020) [1] SimCLR. Their claim is that without the need for complex, specialized architectures like memory banks, good embeddings can be learned by combining self-supervised contrastive learning and data augmentation. Figure 3.1 shows the learning framework of SimCLR. The input at each pass through the framework is a batch of unlabeled training images x. Each image is being transformed two times by an augmentation t and another augmentation t', both sampled from a distribution of augmentations T. Then all augmented images are passed through an encoder network $f(\cdot)$ (a Residual Neural Network (ResNet) [52]) and a projection head $g(\cdot)$, which is simply a two-layer fully connected network. A loss function then maximizes agreement between the outputs z_i and z_j of the two augmented views. The authors found that the representations obtained from the encoder network $f(\cdot)$ rather than the projection head are better suited for downstream tasks and therefore after training $g(\cdot)$ is discarded.

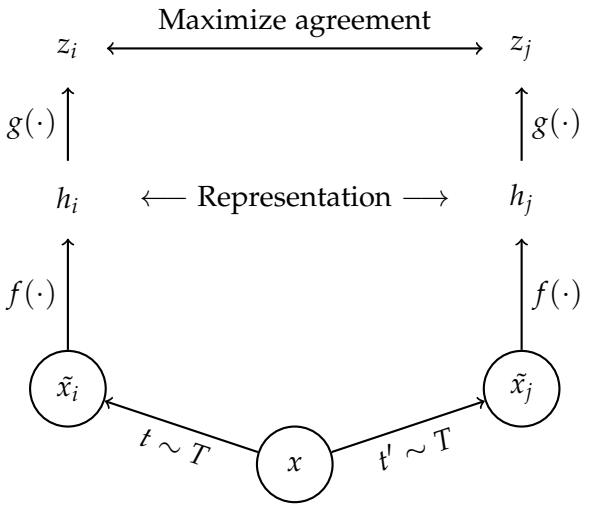


Figure 3.1: The contrastive learning framework of *SimCLR*: The input x is augmentated two times by randomly sampled augmentations ($t \sim T$ and $t' \sim T$). A base encoder network $f(\cdot)$ and a projection head $g(\cdot)$ are trained using a contrastive loss to maximize agreement. Figure is from the original paper of SimCLR [1].

SimCLR uses the NT-Xent loss function explained in Section 2.3.2. Each image in a batch of size N is augmented two times, resulting in $2N$ examples. Instead of sampling negative pairs explicitly, given a positive pair, all other $2(N-1)$ examples are treated as negative pairs. Using this loss they claim to achieve 85.8% top-5 accuracy on only 1% of the labels, outperforming *AlexNet*, a supervised network proposed by Krizhevsky et al. [37], with 100 times fewer labels. Note that *AlexNet* was the state-of-the-art model for image classification just 8 years before this paper was released.

Chung et al. [53] extensively evaluated similarity learning techniques in the realm of audio data and showed that they can outperform classification-based techniques in the task of speaker recognition. The authors also showed that the triplet loss often has slow convergence or falls into local minima.

In this work, we combine the findings of Hinton and Chung to create a self-supervised, contrastive learning framework for determining audio similarity. Compared to SimCLR we propose a new set of augmentation techniques to create a suited self-supervised learning task in the domain of audio data. We also show that the triplet loss is inferior compared to the novel NT-Xent loss used in SimCLR, supporting the findings of Chung et al. that the triplet loss takes a long time to converge to an optimal solution [53].

3.2 Few-Shot Audio Classification

Humans are very effective in learning new tasks with only little training data available. This is the case because we learn knowledge of the world and reapply this knowledge to other domains later on. Like a person being able to drive a car is later able to learn how to drive a

truck much faster since specific concepts apply to both domains. Inspired by this and the fact that labeled data is scarce and difficult to obtain for certain fields a lot of research is now focused on training neural networks with as little data as possible. This concept is called few-shot learning and was firstly investigated in the domain of image data by Rezende et al. in 2016 [54] and was later transferred to the domain of audio data by Arik et al. in 2018 [55]. An even more drastic approach is called one-shot or even zero-shot learning, where at test time only a single instance of the new classes is given to the network. Chou et al. (2019) [56] showed that using attentional similarity, few-shot sound recognition can be performed with great success. Anand et al. [57] tried to identify speakers using a prototypical loss function. They proposed to replace CNNs with Capsule Networks, first introduced by Hinton et al. [58]. Though it has not yet been shown that it can actually outperform more classical architectures. In this thesis, we show that Capsule Networks can be outperformed on few-shot classification tasks using classical established architectures like convolutional neural networks and combining them with our proposed learning framework.

3.3 Self-Supervised Few-Shot Transfer Learning

In this section, we take a closer look at the work of Medina et al. (2020) [59] which has many similarities to our proposed framework. The authors combine the two learning paradigms self-supervised learning and transfer learning to create a new solution for the few-shot classification problem. They state that several works ([60], [61]) have shown the superiority of transfer learning compared to other methods on cross-domain settings. The authors claim that before their work, unsupervised non-episodical techniques for few-shot transfer learning have not been explored and so they propose *ProtoTransfer* which "performs self-supervised pre-training on an unlabeled training domain and can transfer to few-shot target domain task" [59]. In contrast to our proposed system, they use a prototypical loss function as their self-supervised training target to minimize distances between noisy transformations of the same input. The difference between our method and SimCLR is that they try to minimize the distance between one augmented image to the original image while we minimize the distance between two augmented images and therefore creating an even more difficult task for the network to learn. We chose this since we found that increasing the difficulty of the pre-training stage prevents overfitting and enables better generalization. Another difference is their fine-tuning stage, where they first calculate class-cluster centers based on the support set of the few-shot task. These centers c are used as weights to a final linear layer, which is then fine-tuned using softmax cross-entropy on the few-shot dataset. We do not make use of these clusters and train a randomly initialized one layer fully connected network on the few-shot data before finally fine-tuning the entire network with a very small learning rate. We argue that this approach is simpler but at the same time yields just as good results. All in all their proposed framework is very similar to ours and draws its ideas from similar sources as we do. They showed that using this approach state-of-the-art results can be achieved in the domain of image data. Our task will be to show similar results for audio data.

4 Contrastive Learning for Audio

We combine the learnings of the previous chapters to propose a new framework for the problem of similarity learning of audio representations based on recent advances in self-supervised, contrastive and transfer learning. We call it "Contrastive Learning for Audio (CL-Audio)". In the following sections we take a closer look at the individual components of the framework and explain the reasoning behind different design choices.

4.1 Learning Framework

In this section, we talk about the different stages and specifics of our learning framework. In each subsection, we go into detail about the specific tasks. Algorithms 3 and 4 show the execution process of our framework in the pre-training and transfer stage. Analogously Figure 4.1 describes those stages in a more visual way. All stages are explained in more detail in the following subsections. Circled numbers in the text correspond to the same marker in Figure 4.1.

But Before going into the details of CL-Audio we first want to state the intuition behind it. The final goal of CL-Audio is to be able to represent audio signals in such a way that a machine can tell whether or not two signals are similar or not. To achieve this we employ contrastive learning, which enables us to train a neural network to transform input signals in such a way that the outputs have our required properties. Unfortunately, contrastive learning requires labels to learn the similarity of input pairs, but as explained in Section 2.2.3 we can make use of self-supervised learning to create our own labels. In simple words, this self-supervised task is to find two randomly augmented signals, constructed from the same source signal, out of a batch of other unrelated signals. To achieve this we make use of the NT-Xent loss that tries to maximize similarity between those two augmented signals and minimizes similarity towards the others. In doing so we train an encoder network that solves our problem of determining audio similarity. This encoder network can later be transferred into a low-data domain to solve various downstream tasks.

4.1.1 Pre-Training

CL-Audio is a deep-learning framework for few-shot transfer learning based on initial self-supervised similarity learning using a contrastive loss. The left side of Figure 4.1 shows the self-supervised pre-training stage of the framework. Input on each pass through the framework is a sampled minibatch of audio data $\{x_k\}_{k=1}^{N}$ of size N. Each sample is first augmented by two stochastic transformations t and t' sampled from a set of transformations $\mathcal{T}$ (1). The specifics of these augmentations are explained in Section 4.3. The two resulting

Algorithm 3 CL-Audio Pretraining

Input: encoder network f, projection network g, trainable parameters $\boldsymbol{\theta}$, random transformations $\mathcal{T}$, batch size N, learning rate α, temperature τ, audio enhancements E
Randomly initialize $\boldsymbol{\theta}$
while not done **do**
  for sampled minibatch $\{\boldsymbol{x}_k\}_{k=1}^{N}$ **do**
    for all $k \in \{1, ..., N\}$ **do**
      convert $\boldsymbol{x}_k$ to 16-bit PCM
      $t \sim \mathcal{T}$ ▷ 1. augmentation
      $\tilde{\boldsymbol{x}}_{2k-1} \leftarrow t(\boldsymbol{x}_k)$
      $\tilde{\boldsymbol{x}}_{2k-1} \leftarrow E(\tilde{\boldsymbol{x}}_{2k-1})$ ▷ preemphasis, DC-removal, dither
      $\hat{\boldsymbol{x}}_{2k-1} \leftarrow STFT(\tilde{\boldsymbol{x}}_{2k-1})$
      $\boldsymbol{z}_{2k-1} \leftarrow g(f(\hat{\boldsymbol{x}}_{2k-1}))$
      $t' \sim \mathcal{T}$ ▷ 2. augmentation
      $\tilde{\boldsymbol{x}}_{2k} \leftarrow t'(\boldsymbol{x}_k)$
      $\tilde{\boldsymbol{x}}_{2k} \leftarrow E(\tilde{\boldsymbol{x}}_{2k})$ ▷ preemphasis, DC-removal, dither
      $\hat{\boldsymbol{x}}_{2k} \leftarrow STFT(\tilde{\boldsymbol{x}}_{2k})$
      $\boldsymbol{z}_{2k} \leftarrow g(f(\hat{\boldsymbol{x}}_{2k}))$
    end for
    for all $i \in \{1, ..., 2N\}$ and $j \in \{1, ..., 2N\}$ **do**
      $s_{i,j} \leftarrow \boldsymbol{z}_i^{\intercal} \boldsymbol{z}_j / (\|\boldsymbol{z}_i\| \, \|\boldsymbol{z}_j\|)$ ▷ Cosine Similarity
    end for
  end for
  let $\ell(i,j) = -\log \frac{\exp(s_{i,j}/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(s_{i,k}/\tau)}$ ▷ NT-Xent loss
  $\mathcal{L} \leftarrow \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$
  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\theta} \mathcal{L}$
end while
return f ▷ Throw away g

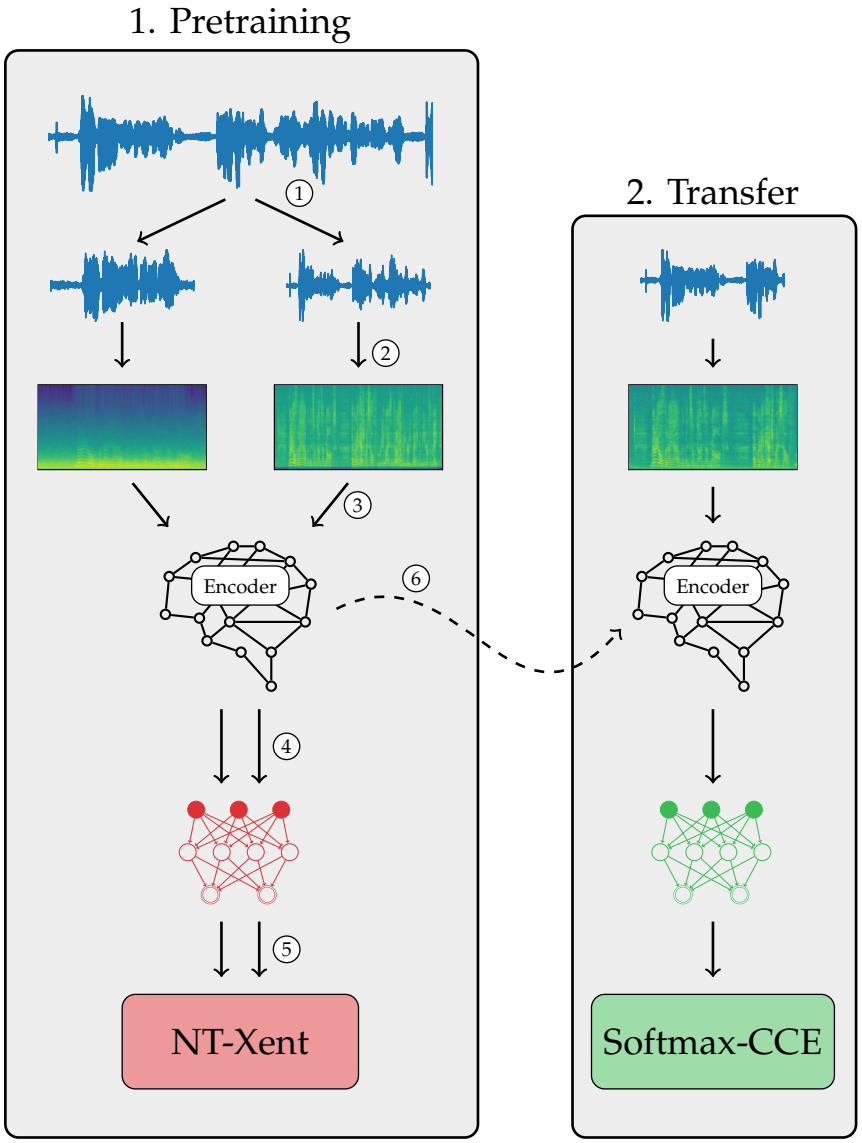


Figure 4.1: Graphical overview of the CL-Audio framework. Marked arrows are explained in more detail in the text.

audio signals are both transformed using the STFT into the frequency domain ②. In this step, we also apply a few audio enhancement techniques. Details regarding these enhancements can be found in Section 4.4. The resulting spectrograms $\hat{x}_k$ act as the input to the encoder network $f(\cdot)$. The encoder network can be any deep neural network that accepts matrix-like time-series data and outputs vectors. Both spectrograms are fed into the encoder network ③. The output of this are two vectors. These vectors, also called embeddings, are the latent representations of the two augmentations of the input data. Those embeddings are again transformed using a fully-connected network $g(\cdot)$ ④, also called the projection head. Chen

Algorithm 4 CL-Audio Transfer

Input: frozen encoder network f from pretraining, classification head g' with trainable parameters $\boldsymbol{\theta'}$, batch size N, learning rate α, audio enhancements E
Randomly initialize $\boldsymbol{\theta'}$
while not done **do**
  for sampled minibatch $(\boldsymbol{X}, \boldsymbol{y})$ of size N **do**
    for all $k \in \{1, ..., N\}$ **do**
      convert $\boldsymbol{x}_k$ to 16-bit PCM
      $\boldsymbol{x}_k \leftarrow E(\boldsymbol{x}_k)$ ▷ preemphasis, DC-removal, dither
      $\boldsymbol{x}_k \leftarrow STFT(\boldsymbol{x}_k)$
    end for
    $\hat{\boldsymbol{y}} \leftarrow softmax(g'(f(\boldsymbol{X})))$
    $\mathcal{L} \leftarrow \sum_{k=1}^{N} \boldsymbol{y}_k \log \hat{\boldsymbol{y}}_k$ ▷ Categorical Cross-Entropy
    $\boldsymbol{\theta'} \leftarrow \boldsymbol{\theta'} - \alpha \nabla_{\theta'} \mathcal{L}$
  end for
end while
return $f \circ g'$

et al. 2020 [1] found that using such a projection head drastically improves the performance of a contrastive learning algorithm. The two outputs of the projection head z_k and z_{2k} will act as the positive input pairs for the contrastive NT-Xent loss term $\mathcal{L}$ (5) while all others act as negative input pairs. Note that all this is performed using a batch of N input samples simultaneously. This results in N positive input pairs and $2(N-1)$ negative input pairs for each pass through the network. Both networks are trained using backpropagation and stochastic gradient descent to minimize $\mathcal{L}$. This initial stage is trained with as much unlabeled data as possible. For this purpose we created a 310GB large dataset which is explained in more detail in Section 5.1.1. The number of epochs to train in this stage is hard to determine since there is no verification loss that we can compare to. To prevent overfitting and to lower training time we stop as soon as the training loss does not decrease from one epoch to another. After training is complete $g(\cdot)$ is discarded.

4.1.2 Transfer

The next stage is the transfer stage shown in the right side of Figure 4.1, which transfers the embedding network $f(\cdot)$ trained in the pre-training phase and puts it into a new, low-data classification domain (6). Here the input is again a single audio file represented as samples x_k but this time we also use corresponding labels y_k. This time the audio is transformed into a spectrogram without any previous augmentation. The spectrogram is then fed into the pre-trained embedding network $f(\cdot)$ and the resulting embedding vector is fed into a new fully-connected network $g'(\cdot)$, also called the classification head, which transforms the dimensionality of the latent vector into the correct number of classes of the classification task. The output of the classification head is often called logits. Contrary to the dataset in the

pre-training phase, this dataset does contain labels y_k, which are used together with the logits in a softmax categorical cross-entropy loss $\mathcal{L}$ to train the new classification head $g'(\cdot)$ while the weights of $f(\cdot)$ are frozen, which means they are not trainable. In this stage, we use a verification loss to determine when to stop training. We fix the number of training epochs and in the end return the network that performed the best on the verification dataset.

4.1.3 Fine-Tuning

In a third, optional stage, both the embedding network and the classification head can be made trainable and the entire network is trained on the same data again with a very low learning rate, to further increase accuracy. This stage has to be tested for performance. It is not guaranteed that fine-tuning actually increases performance but in a few experiments we even found that it decreases performance. If the smaller dataset of the transfer stage is too small compared to the size of the classification head, we found that finetuning had no effect on accuracy but as the size of the dataset increases, we found that finetuning got more and more important.

4.2 Data Preprocessing

To correctly input the data into the network, there are several preprocessing steps that have to be performed first to assure optimal performance of the network. We start off with audio files on disc which can be stored in many different formats and codecs like *.wav*, *.mp3* or *.flac*. It is important that our network receives only data that is of the same kind, therefore we first transform the content of the file to raw signed 16-bit PCM samples with sample rate 16kHz on a single channel (mono). We chose this format because most of the datasets available also use these parameters. Converting from a standard *.mp3*-file to this configuration is straight forward. We first decode the codec to obtain raw samples, make it mono by averaging over both stereo channels, then resample from the original sample rate (usually 44.1kHz) to our target rate 16kHz and then convert the resulting samples to 16-bit PCM by multiplying each sample with 2^8. The conversion for a *.flac*-file is achieved analogously.

4.3 Augmentation

We now take a closer look at all the employed augmentations. The reason we use augmentations to create two distinct views of the same source is twofold. First, we need to create a self-supervised task for the algorithm and secondly, we need a task that is sufficiently difficult for the machine to learn strong embeddings. As explained in Section 2.2.2 there exist many different kinds of self-supervised tasks to choose from but most of them yield suboptimal results. The reason we chose data augmentation as our contrastive prediction task is because it decouples the target from the encoder architecture. This way we can very precisely compare different encoder models and are able to adapt our framework when new, stronger models are discovered.

In each remaining subsection, we describe the augmentation process and our design decision for each augmentation respectively. All augmentations are applied with a fixed probability of 60% in the order that they are listed here. Figure 4.2 shows the same audio clip with every augmentation applied individually. Note that the original audio clip was chosen because of its distinctive activity in the higher frequencies so that the individual augmentations become more visible.

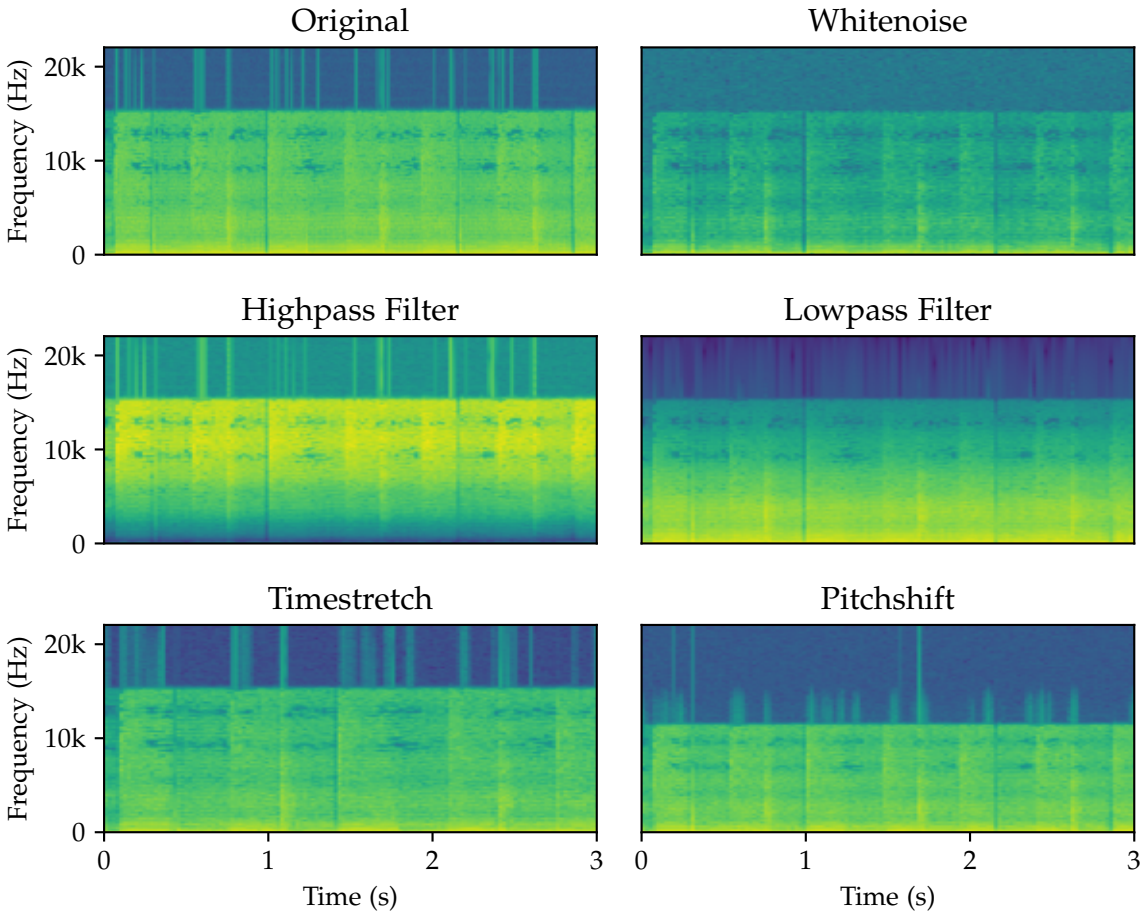


Figure 4.2: Each augmentation applied individually to one sound file with distinctive high ends to better distinguish the effects of the respective augmentation.

4.3.1 Crop

At first, any audio clip is cropped at random. We input 10 seconds of samples and crop a 4.5-second clip from it even though the input to our network will only be 3 seconds long. We do this because we anticipate the later time stretching stage. As we will explain later we shrink up to a factor of 0.7, which means that a 4.5-second clip will result in at least 3 seconds of input data after time stretching. Cropping can be seen as a way to artificially increase the

size of the dataset since a single input clip can potentially be split into several independent input clips. Figure 4.3 shows a 10-second audio clip in blue and a 3-second crop of it in red.

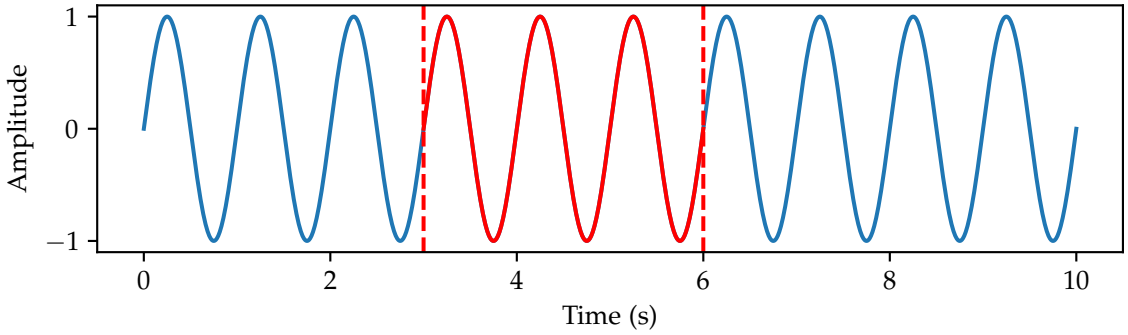


Figure 4.3: 1Hz sine wave in blue and cropped clip in red.

4.3.2 Gain

Gain is added to the signal in a straightforward fashion by multiplying the signal with a factor and clipping it afterward as depicted in Equation 4.1. Random gain changes can be seen as simulating different distances of the same source to a microphone.

$$y_{gain}(n) = min(max((y(n) \cdot \gamma), -1), 1) \tag{4.1}$$

Here γ is a hyperparameter that controls the amount of gain. It will be the hyperparameter sampled stochastically in our augmentation pipeline. $y(n)$ denotes the input signal. The output is then clipped in the range of $[-1, 1]$. Figure 4.4 shows a sine wave in blue and the same sine wave with 2dB of gain applied to it in red.

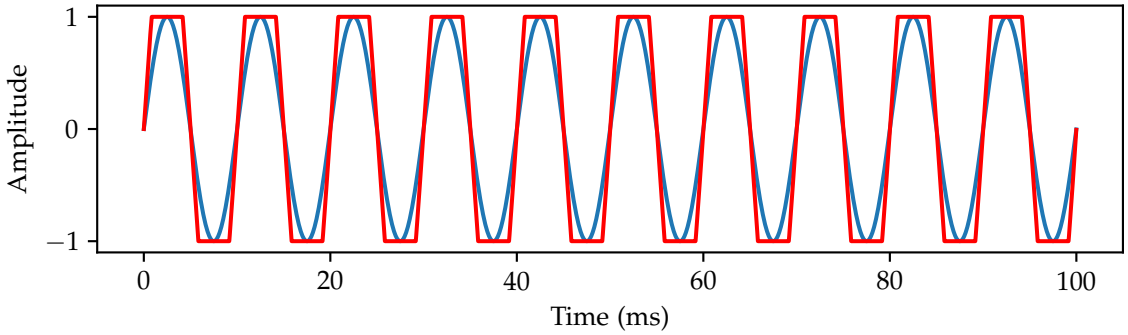


Figure 4.4: 100Hz sinus in blue and with gain of 2dB applied in red.

4.3.3 White Noise

Even though there are many different kinds of noise, like Brownian noise, pink noise, etc. we limit ourselves to adding white noise to a signal. White noise is simply uniformly distributed noise over all frequencies. After adding this random noise we clip the resulting signal. We scale the white noise uniformly between -40 dB and +20 dB. Adding white noise can be seen as a regularizer for quiet frequencies, since overall amplitude is increased uniformly, the percentage of low-level frequencies is decreased, therefore it is harder to hear them.

$$y_{wn}(n) = min(max(y(n) + \gamma \cdot \mathcal{U}(-1,1), -1), 1) \tag{4.2}$$

where γ is again a stochastically sampled parameter that controls the amount of noise added to the signal. $\mathcal{U}(-1,1)$ is a random distribution between -1 and 1. Again the output must be clipped to stay between the minimal and maximal amplitude. Figure 4.5 shows a sine wave and the same wave with white noise and $\gamma = of 1.03dB$ added to it.

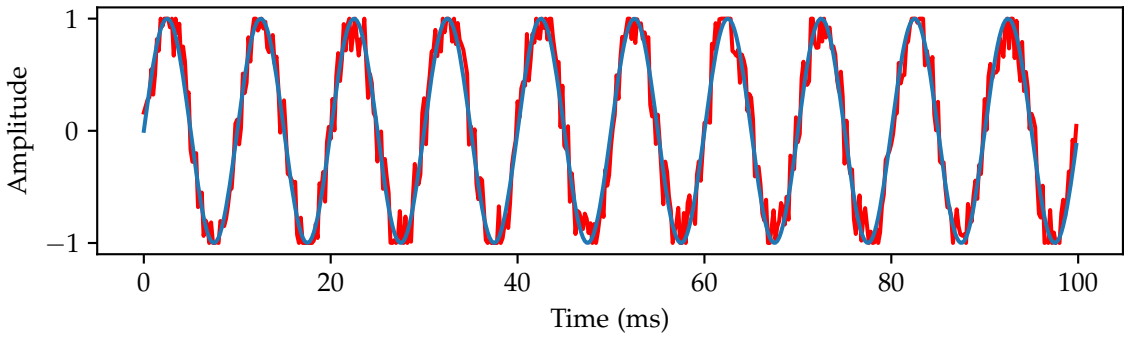


Figure 4.5: 100Hz sinus in blue and with white noise added in red.

4.3.4 Low-pass and High-pass Filter

We apply low pass filtering as an environmental effect. Just like gain can be seen as a change in distance to the observer, a low pass filtering effect occurs in nature when objects between the producer and the observer alter the sound, like for example a wall can dampen a sound. To reproduce this effect we employ a Butterworth low-pass filter at a frequency between 100 and 2000 Hz and a random order from 1 to 5. Figure 4.6 shows the frequency response of a low-pass Butterworth filter at 200 Hz with different orders.

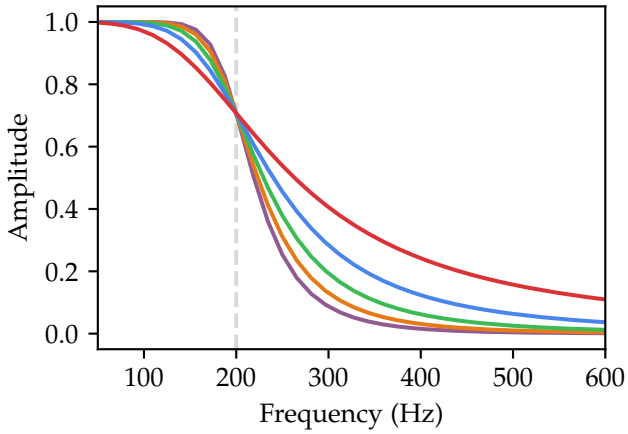


Figure 4.6: Frequency response of a low-pass Butterworth filter with a cutoff frequency of 200Hz and orders ranging from 1 to 5.

We chose this filter because it has a very flat frequency response, meaning it does not produce any audible artifacts on the cutoff frequency and above. This kind of filter was first proposed by S. Butterworth in 1930 [62].

Figure 4.7 shows a comparison to others filters. This clearly shows that the Butterworth filter rolls off more smoothly than the others without any ripple effects. Analogously to the low-pass filter, we use the Butterworth filter as a high-pass filter.

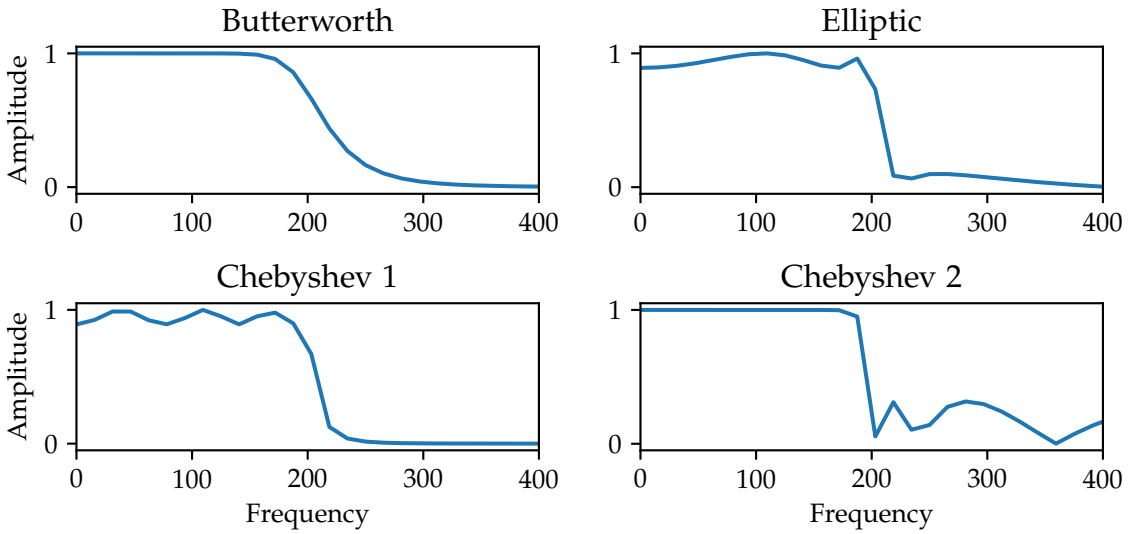


Figure 4.7: Frequency response of four different low-pass filters with a cutoff frequency at 200Hz.

4.3.5 Time-Stretch

Time-stretching is an effect that tries to increase the duration of an audio clip without introducing any other audible sound effects, like changing the pitch of the audio. Time-stretching is achieved using the phase vocoder algorithm described in Section 2.7. In the analysis phase of the vocoder algorithm, the separate STFT buckets are pushed together to achieve speedup and pulled apart to achieve a slowdown. Afterwards, they are synthesized back together to obtain the original audio but in a different length. Since time-stretching, besides cropping, is the only augmentation that changes the shape of the data we have to make sure that after time-stretching, all examples are cropped again into the correct shape, therefore we cannot allow time stretching to decrease the number of samples below the input-size of our network. We start with a 4.5 second audio clip and randomly shrink or stretch it by a factor of 0.7 to 1.3 and then crop all excess samples to yield a 3-second clip.

We experimented with many different implementations and found that there does not yet exist a perfect solution for our use case. Whereas all other computations fall into the range of 2 to 20 milliseconds, the minimum time for time stretching was 150ms with the pyrubberband [63] python library. It is a wrapper to the more common command-line interface rubberband [64]. Rubberband produces the best results of all the approaches that we have tried. The wrapper simply calls the library's Command Line Interface (CLI) and stores and loads intermediate results on disk. This is of course a significant overhead. We made the tradeoff of merely decreasing the probability that time stretching is applied to 5%, which results in similar computational time as the other augmentations. Figure 4.8 shows a simple sine-wave of 200Hz in blue that has a short period of silence in it, indicated by the zero values around 50 and 60 ms. The red line is the time-stretched version by a factor of 2. Both lines overlap almost perfectly while the period of silence is cut in half. This means the frequency of this signal is maintained while time is shortened. We can see artifacts at the end of the audio clip which are introduced by the last STFT window having to zero-pad the signal, also the amplitude changes slightly. One should note that though visible here, these differences are in fact not audible by humans.

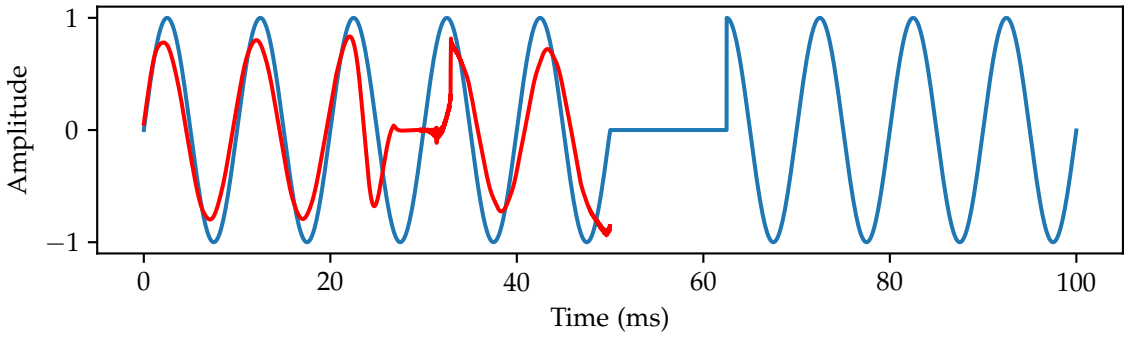


Figure 4.8: 100Hz sinus with a short period of silence in blue and the same signal speeded up by a factor of 2 in red.

4.3.6 Pitch-Shift

Pitch is the perceived frequency of sound. Pitch-shifting is an effect that changes the pitch of a signal without changing the duration of it. To understand how pitch-shifting works, we first look at the effect of resampling. The simplest way to change the time and pitch of a digital audio signal is through resampling. Resampling means changing the sample rate of a digital signal and then playing it back at the original sample rate. The effect of this can be seen in Figure 4.9. The blue line is again a sine wave with 100Hz sampled at 16kHz and the red line is the same signal resampled to 32kHz and plotted on the same timescale as the 16kHz signal It is obvious that this operation halves the duration of the signal, since only every other sample is used in the new signal, but also the entire frequency spectrum is scaled with the same factor, thus increasing or decreasing the perceived pitch accordingly.

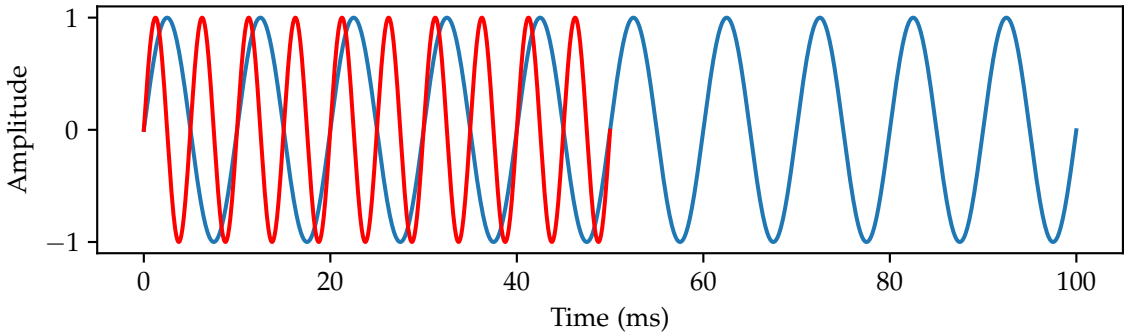


Figure 4.9: 100Hz sinus sampled at 16kHz in blue and the same signal resampled to 8kHz in red.

Even though this is not the desired effect pitch-shifting makes use of it by introducing it into a two-stage process. First the signal is time-stretched by a factor of $1/\alpha$ and then resampled to a target frequency $fs_t = \alpha \cdot fs_o$ and finally played back at the original sample rate fs_o. The time-stretch of the first stage is canceled out by the opposing time-stretch of the second stage, leaving behind only the change in pitch by a factor of α. Figure 4.10 shows again the same 100Hz signal in blue and the same signal down-pitched by 3 semitones in red. Pitching by 12 semitones is equivalent to a doubling in frequency.

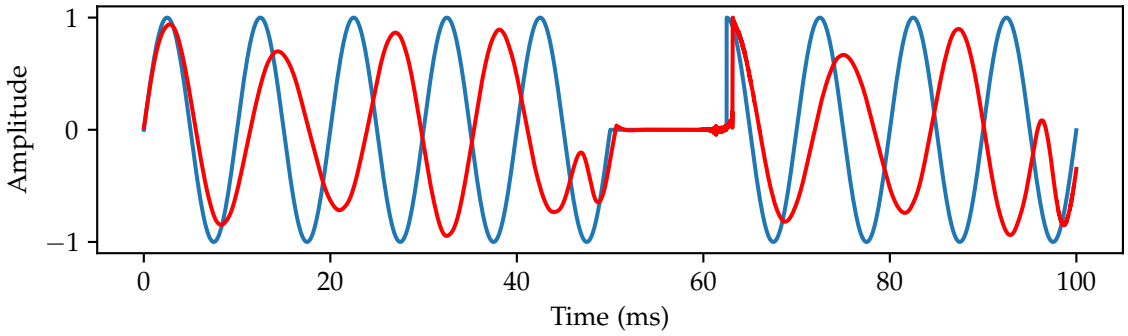


Figure 4.10: 100Hz sinus with a short period of silence in blue and the same signal pitch-shifted by -3 semitones.

4.4 STFT Enhancement

After augmentation, we now want to transform the resulting audio from the time domain to the frequency domain to have the correct input shape for our network. But instead of simply transforming it using plain STFT we first apply some audio enhancement techniques to mitigate the negative effects of possible bad quality audio recordings and to increase the quality of the resulting STFT. Three of such enhancement techniques were used here. (1) Pre-emphasis filter, (2) Direct Current (DC) removal, (3) dither. The pre-emphasis effect decreases the volume of lower frequencies and boosts higher ones. This is a common technique in speech processing to increase the volume of speech-specific frequencies while lowering others. Equation 4.3 shows the most common way of pre-emphasis filtering of a signal. We use a default value for α of 0.97.

$$pre(x_n) = x_n - \alpha x_{n-1} \tag{4.3}$$

DC removal counters the fact that bad audio recordings usually have an equipment-introduced DC offset, which means the mean of the waveform is not at zero. This can cause low-frequency distortion. We achieve DC removal by applying a digital high-pass filter at a frequency close to zero. The equation for this filter is shown in Equation 4.4.

$$y_n = x_n - x_{n-1} + y_{n-1} - 0.99 y_{n-2} \tag{4.4}$$

The last enhancement step is called dithering. It is the process of adding noise to a signal to increase its quality, which at first sounds counter-intuitive. To understand why dithering can actually improve perceived sound quality one has to know about the negative effects of quantization as explained in Section 2.4. Since digitized signals with a low bit depth introduce a lot of quantization error, these distortion effects can become audible especially in higher frequencies. To cover these effects we introduce a special kind of noise called dither to the signal that masks higher order distortions. Audible higher-order harmonics are swapped

with inaudible noise. Equation 4.5 shows a simple way of adding a small amount of dither to a signal.

$$dither(x_n) = x_n + 10^{-6} * \sigma_x \cdot \mathcal{U}(-1, 1) \tag{4.5}$$

$\mathcal{U}(-1, 1)$ denotes a uniform distribution between -1 and 1.

5 Experiments

In this chapter, we describe all the experiments that we conducted to test the quality of the CL-Audio learning framework. The experiments go as follows: First, we train each network using CL-Audio to obtain a pre-trained encoder network. Then we transfer this network to a novel domain as depicted in Figure 4.1 and train a small, randomly initialized classification head network on the outputs of the encoder network. All the experiments have the same final goal of classifying unknown audio samples into predefined categories. We will look at multiple domains, represented by three different datasets, two of which are publically available for the purpose of reproducibility. The first domain is speaker recognition which we will explore using the VoxCeleb [65] dataset. We will also purposefully lower the amount of training data to simulate a low-data environment in this domain. We call this dataset VoxCeleb50-20 because it contains 50 classes and 20 samples per class. For the second domain, artist classification, we created a dataset only for this thesis to see if CL-Audio can successfully classify the rich and complex information contained in music. The last domain will be birdsong classification, which will be explored using the British Birdsong Dataset [66]. In all of the three domains, we will compare the performance of CL-Audio towards the same networks trained using only supervision and, if available, results of related work on the same domain. This chapter is structured as follows: (1) We first look at the individual datasets used for training and for our individual domains, (2) we go into more detail about the three models we use for our experiments. (3) We briefly report all the necessary evaluation metrics and hyperparameters used in the experiments. (4) We show all results and discuss the findings.

5.1 Datasets

First, we take a closer look at all datasets used for training and testing. The first dataset we look at is the large, unannotated training dataset for the CL-Audio self-supervision pre-training task. After that, we go into more detail on the VoxCeleb dataset, then discuss our music-classification dataset and lastly look at the British Birdsong dataset. Key metrics of the individual datasets can be found in Table 5.1.

5.1.1 Self-Supervised Pre-Training Dataset

This is the training data that we use for the pre-training phase of our framework. It is a combination of different datasets. The main goal of this dataset is to maximize the amount of information and variation that the network will be exposed to. This data does not have to be labeled and can therefore be extracted from any source available. We combined three sources: *i*) *LibriSpeech* [67], *ii*) a collection of music and *iii*) *AudioSet* [68].

Dataset	Classes	Instances per class	Classification target
VoxCeleb	1,251	123	Speaker
Music	58	40	Artist
Birdsong	54	20	Species
VoxCeleb50-20	50	20	Speaker

Table 5.1: Key metrics of the datasets used in the experiments.

The LibriSpeech corpus is a dataset that contains 1000 hours of read English speech. Its data is extracted from audiobooks from the LibriVox project. Since the clips are stored as .flac files, we first have to extract the samples and convert them to 16-bit PCM as explained in Section 4.2. We then split the result into 10 second long clips for easier processing with our input-pipeline.

The next part of the set consists of a large collection of music. In total 169.5 GB, resulting in 3,533 hours of music, parsed into 10 second long clips. Since most of the files are stored in the .mp3 format, they also have to be parsed. The last part of the dataset is a subset of the AudioSet dataset. AudioSet consists of over 5,800 hours of audio from over 2.1 million videos. From this entire set, we downloaded 1,000 hours of audio from random videos in the dataset to increase the diversity of our dataset. These clips include anything from speech, music, vehicles, lawnmowers or weather. All these combined results in a dataset of over one million 10 second sample clips and one can easily see that since no labeling effort goes into the creation of this dataset, the upper bound is practically only determined by the amount of hard drive available which in our case was roughly 310GB.

5.1.2 VoxCeleb

VoxCeleb is a speaker-identification and verification dataset. The authors of VoxCeleb claimed that previous datasets in this field struggled to create large corpora due to the intense manual labor involved in the annotation. Therefore what they proposed was “a fully automated pipeline based on computer vision techniques to create the dataset from open-source media" [65]. This pipeline starts by collecting videos from YouTube, then extracting the sound and speakers from these videos and performing active speaker recognition, which is the task of determining whether the person in the picture produced the sound or not. This was done using *SyncNet* [69]. After that a face verification algorithm determines if the face in the frame is actually the person of interest. If this is the case, the audio clip is added to the dataset and correctly labeled. With this method, they curated a dataset of over 100,000 utterances of over 1,000 celebrities. For few-shot learning, we purposefully decreased the number of speakers to the first 50 of those. In their paper, the authors also proposed a strong baseline for the tasks of audio-based speaker identification and verification. In this thesis, we used their VGG-based model as the main model for all experiments. It is described in more detail in Section 5.2.1. Unfortunately, the paper lacks some crucial implementation details which we, in turn, had to fill in. As described later we could not fully reproduce their reported results, so we used our

results as a point of reference since they align more with similar experiments done by others [70]. The VoxCeleb is by far the most used dataset of the three presented in this chapter and therefore, to increase reproducibility and comparability of this work, it will be the main focus in our experiments. Other work on this dataset include Chung et al. [71], Okabe et al. [72] and Yang et al. [73]. Fortunately, since all the files are stored as 16-bit PCM *.wav*-files sampled at 16kHz, there is no processing required to use them with our input-pipeline.

5.1.3 Music Classification

For the task of music artist classification we parsed and annotated 1,450 songs by 58 artists. We made sure that each artist had at least 3 albums to increase musical and environmental diversity per artist and that all songs contained vocals, to increase distinct audio features per artist. We then split the songs into 20 train and 5 test songs and from those sampled 3 utterances each, therefore no song must have a length less than 30 seconds. Again these clips first have to be parsed from *.mp3* to 16-bit PCM format at 16kHz.

5.1.4 British Birdsong Dataset

To explore CL-Audio's performance on an entirely unknown category we selected the relatively small British Birdsong dataset. It is a specific subset gathered from the *Xeno Canto collection* [66]. For our case we try to classify each sample by the annotated species. For us to take a species into the dataset it must have at least 3 recordings all of which combined must have no less than 250 seconds of sound. To create a balanced dataset we filter out all the excess recordings. 20 10-second clips from the first two recordings are then used as training data. Note that due to input-size restrictions of the networks only 3 seconds of the 10-second clips are randomly sampled on each run. 5 10-second clips of the third recording are then used for testing. This results in a total of 180 minutes of training data. The files come as *.flac* files so they have to be parsed and split accordingly. This dataset is a good showcase for an algorithm's versatility on low-data, highly specialized domains.

5.2 Encoder Model Architectures

Now we take a closer look at the three different encoder model architectures implemented for the experiments. The reason for implementing and testing on several models is to showcase the model-agnostic properties of our framework. All models have proven track records in the field of audio classification. In the following subsections, we take a closer look at each model, including a graphical overview and its major key metrics. Key figures of the models are listed in Table 5.2.

5.2.1 VGG-Vox

The first model that we look at is called VGG-Vox, proposed by the creators of the VoxCeleb dataset. It is a convolutional neural network based on the VGG-M architecture first proposed

Model	Trainable Parameters	Evaluation Method
VGG-Vox	17,911,267	Full utterances
LSTM	19,425,871	Avg. of 10s cuts
SpeechTransformer	15,571,773	Avg. of 10s cuts

Table 5.2: Encoder models used in the experiments and their respective number of trainable parameters and evaluation method.

by Chatfield et al. in 2014 [74]. The VGG model architecture consists of several stacked blocks of convolution, followed by batch normalization, ReLU activation and max-pooling. Most of the architecture stayed the same but for slight modifications to adapt to spectrogram inputs. The single most important contribution is the replacement of the VGG-M's fully connected layer that reshapes the two-dimensional convolutional input into one-dimensional vectors fit for dense input.

VGG-Vox replaced this layer with a fully connected layer of size 9×1 followed by an average pooling layer of size n, where n is the size of the temporal dimension of the network's last convolutional layer's output. This change is of significant importance for audio data since it removes the need for a fixed input size in the temporal dimension but not in the frequency dimension, which means we can input clips of arbitrary length as long as the number of FFT-bins stays the same. This is a significant advantage compared to all other models in this thesis. Note that in none of the major modern deep learning libraries, one can stack data of different shapes into a single batch. Therefore to feed clips of different durations to the network, we have to pass one after another, which greatly decreases performance. Due to this, we limit this approach to test time and train on fixed-size batches of 3-second clips.

Figure 5.1 shows the entire model. In the case of self-supervised pre-training, the final fully connected layer and softmax will be replaced by a projection head consisting of two fully connected layers of size 1024. Only in the transfer stage will a classification head of the depicted form replace the projection head. In Figure 5.1, c is the number of classes of the supervised transfer-domain. The model was trained using a batch size of 80.

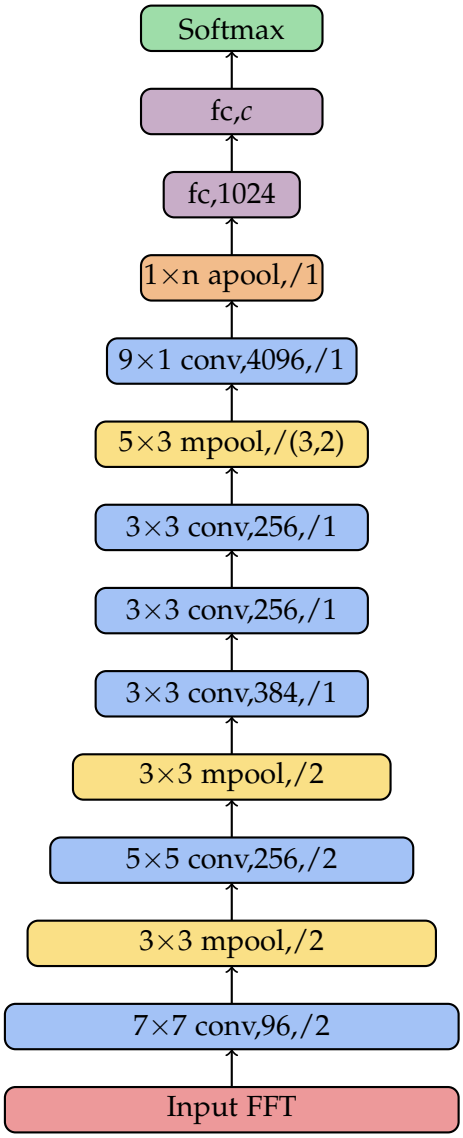


Figure 5.1: The VGG-Vox model. Each operation has the form $n \times m$ (kernel size), operation, N (number of filters), $/(a, b)$ (stride: $/a$ when $a = b$)

5.2.2 LSTM

Recurrent Neural Networks and especially LSTMs have long been a staple in the realm of time series prediction. Its ability to prevent gradients from vanishing over long series proved highly useful in the domain of audio data as well. As explained in more detail in Section 2.4.2 a LSTM layer consists of a cell that successively updates its own state based on a portion of the input data and its previous state. LSTM layers can be stacked just like convolutional layers to produce even more specific features based on hidden states of the previous layer. In our architecture we stack 2 LSTM layers. To produce an output that can be fed to a fully

connected layer, we simply reject all hidden states returned by the LSTM except the last one. Another extension we used to LSTM is called bidirectional LSTM. Instead of only passing the data to the cell from beginning to end, we also pass it vice versa on a second run and then concatenate the hidden states of the two passes. Therefore we can capture temporal dependencies in both directions. Figure 5.2 shows an overview of our architecture.

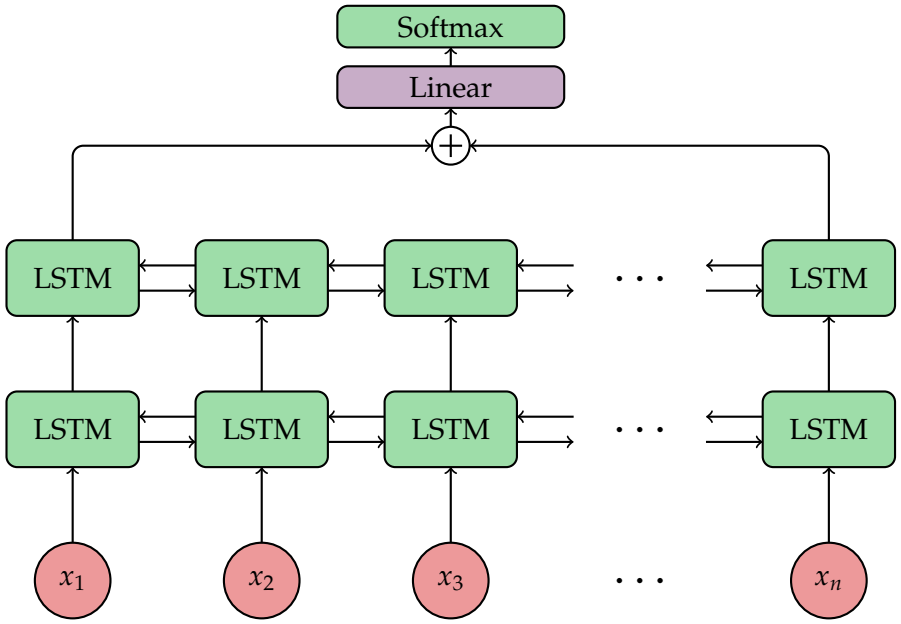


Figure 5.2: Two-layer Bi-LSTM model with a classification head.

Unfortunately, this way of computation is inherently not parallelizable and therefore LSTMs generally have slower training times compared to CNNs or attention-based sequence classifiers. Although counter-intuitive, most deep learning libraries do not offer the ability to pass inputs of arbitrary lengths to the LSTM. This is due to the fact that those libraries usually build a static execution graph during compile time, which is basically an unrolled LSTM to increase performance. Due to this we cannot test on full-length audio clips but rather have to cut them first into the shape of the training data and average the results of all of these cuts. This way, though computationally more efficient, is not as accurate. Again the softmax layer at the end is replaced during the pre-training stage for a projection head that consists of two fully connected layers.

5.2.3 Speech Transformer

The last model we will look at is based on the attention mechanism proposed by Vaswani et al. in 2017 [42] called Transformers. Whereas the major application area of Transformers today is natural language processing, we wanted to try out this architecture in the realm of audio data because of its highly parallelizable time-series transformation quality. The model that we will experiment with in this thesis is based on the Speech Transformer proposed by [32]. The full

transformer model consists of an encoder and a decoder part to solve sequence-to-sequence tasks. Since we limit ourselves to the case of classification we use only the encoder part of the model. To reduce the input size of the Multi-Head-Attention blocks we prepend two convolutional layers to the model with a stride of two. Since self-attention does not have any notion of positional knowledge we have to add a positional encoding before feeding it to the multi-head attention block. We use the same positional encoding proposed by the original authors of the transformers defined in Equation 5.1 for even positions and Equation 5.2 for uneven positions. In our case *pos* denotes the time-step and i denotes the frequency bin of the STFT. A plot of the encoding signal is shown in Figure 5.3.

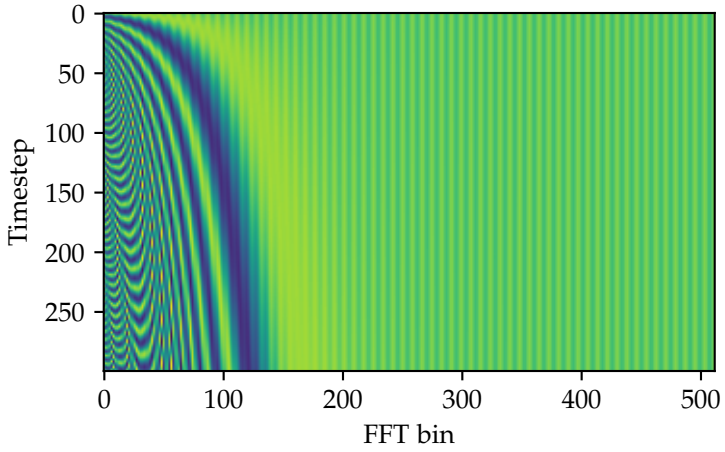


Figure 5.3: A visualization of the positional encoding that is added to the STFT input of the network. Darker colors indicate higher values

$$PE(pos, 2i) = sin\Big(\frac{pos}{10000^{2i/d_{model}}}\Big) \tag{5.1}$$

$$PE(pos, 2i+1) = cos\Big(\frac{pos}{10000^{2i/d_{model}}}\Big) \tag{5.2}$$

Each transformer-block consists of an initial multi-head-attention followed by a skip connection and layer normalization. The output is forwarded into a feed-forward layer that is again followed by a skip connection and layer normalization to stabilize training. In total we stack 8 of these transformer-blocks. Note that these blocks do not reshape the data, therefore after 8 blocks we still have data shaped the same way as the input. After the transformer blocks, the output is averaged across the time dimension to create one-dimensional vectors. This is one of the major bottlenecks of the transformer. We decided against simply flattening the output, which would be analogous to a CNN because the shape of the resulting vector would simply be too large to be feasible. An output of $128x74$ would result in a linear layer

with 9,699,328 weights if the output size would be 1024 just like in the other models. We also tried appending a learnable vector at the start of every sequence. The idea was to use this vector's last hidden state to be used as a summarization vector of the entire sequence since the transformer blocks make it attend to all other time-steps. This is a similar approach that the Bidirectional Encoder Representations from Transformers (BERT) architecture [75] uses by introducing a $[CLS]$ token that is prepended to every sentence to be used as a sentence-level representation. Unfortunately, this approach did not yield good results so it was also discarded. These single vector representations are then piped into a linear layer and finally a softmax classifier on top of it. Again during pre-training, this classifier is replaced by a projection head consisting of two fully connected layers.

Just like the LSTM based network, the Speech Transformer is relying on equally shaped input data, meaning all clips need to have the same duration. Again we employ the evaluation method of splitting the test-clips into 3-second cuts, feeding them through the network, and averaging the resulting probabilities to obtain one single prediction for each clip. Figure 5.4 shows the full model.

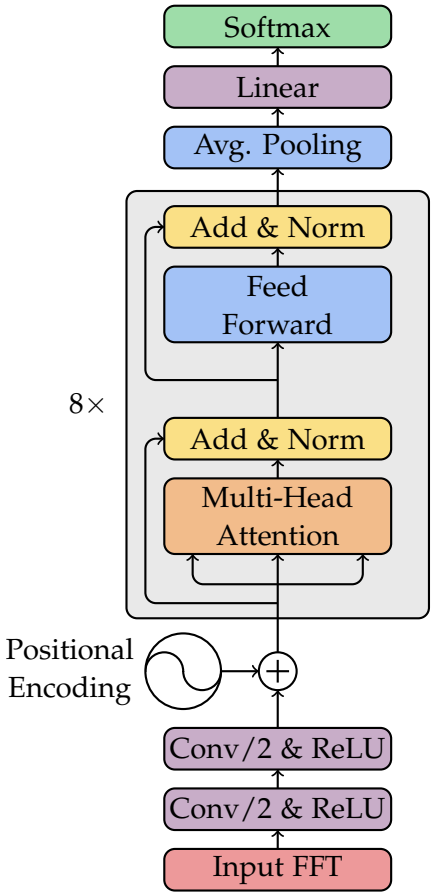


Figure 5.4: The encoder part of the SpeechTransformer model. The figure is from [32] but with our changes applied.

5.3 Evaluation Metrics

Since all of our experiments are classification tasks we will always be tracking the accuracy of our model. Accuracy is simply defined as the number of correct predictions divided by the number of total predictions. We will also oftentimes be tracking the top-5 accuracy of our model which is calculated just like the accuracy but a correct prediction is counted when the correct class was in the top 5 highest scoring classes. This metric shows, especially in difficult, low-data domains, if the model is on the right track even when it is not confident about its predictions.

5.4 Implementation Details

To reduce variability in the same experiment but with different models, we tried to keep all hyperparameters the same. We pre-trained all CL-Audio networks using the Adam optimizer [76] with an initial learning rate of 10^{-4} that reduces in a logarithmic fashion until it reaches 10^{-5} after 10 epochs. Since we found that Stochastic Gradient Descent (SGD) with a momentum of 0.9 performs better in the transfer stage, we used this for all transfer and fully supervised runs. Again the learning rate is decreased accordingly from an initial one of 10^{-2} to 10^{-4}. All convolutional and fully connected layers have an L2 regularizer with a factor of 10^{-4} as well as a batch normalization layer following it. The temperature parameter of the NT-Xent loss was fixed at 0.1. No dropout was used in any of the models. Table 5.3 shows the hyperparameters used for the augmentation pipeline of the pre-training stage.

Augmentation	Probability	Parameter	Min	Max
Crop	1.0	Duration	3	3
Gain	0.6	γ	-10dB	10dB
Whitenoise	0.6	γ	-40dB	-10dB
Lowpass	0.6	Cutoff	100Hz	2,000Hz
		Order	1	4
Highpass	0.6	Cutoff	400Hz	10,000Hz
		Order	1	4
Timestretch	0.1	Factor	0.7	1.3
Pitchshift	0.1	Factor	-600cent	600cent

Table 5.3: Hyperparameters of all augmentations used during contrastive pretraining.

We found that one bottleneck of our augmentations pipeline was the fact that we sequentially applied time-stretching and pitch-shifting. As mentioned earlier pitch-shifting is just an extension to time stretching and so the two can be combined. When both, time-stretching by a factor of α_1 and pitch-shifting by a factor of α_2 should be applied we can instead time-stretch by a factor of α_1/α_2 and then resample to $fs_o \cdot \alpha_2$ where fs_o is the original sampling frequency. Thus we get a final signal that is correctly stretched and shifted at only half the computational

cost.

The input shape of all models is fixed at 300 time-steps and 512 FFT-bins. To obtain this a basic STFT with a window size of 25ms and a step size of 10ms is used on 3-second crops. We use the hamming window function for all experiments.

All code is written in python using various open-source libraries, most notably: NumPy [77], Pandas [78], SciPy [79], Librosa [80] and TensorFlow [81]. Training was done on a single *NVIDIA RTX 2080 Ti*. Batch sizes varied between different models so that the 11GB of VRAM was always fully utilized.

5.5 Results

We present the quantitative results of all models on the three different datasets. We look primarily at the accuracy as well as the top-5 accuracy evaluation metric but we will also take a closer look at other metrics like training time, model sizes, memory consumption and convergence speed. We also compare the VGG-Vox model trained on a small portion of the VoxCeleb dataset towards results of comparable works as well as towards a fully supervised version of the same network and show a comparison of the triplet loss towards the NT-Xent loss on the same dataset. All the results stated here are the maximally reached results for a particular experiment, denoted as percentages rounded to two decimal places.

In the end, we present qualitative results of the obtained latent representations of our framework by plotting 2,000 embeddings, dimensionality-reduced by t-distributed Stochastic Neighbor Embedding (t-SNE).

Table 5.4 shows the overall top-1 and top-5 accuracy of the three networks on the three datasets as well as on a portion of the VoxCeleb dataset, named VoxCeb50-20. This dataset consists of 20 examples per class of the first 50 classes of the VoxCeleb dataset. To restrict the dataset even further and make it comparable to the work of Anand et al. [57], we expose only the first three seconds of each training sample to the network. In total, this results in 50 minutes of training data. The best result for a dataset is highlighted in bold numbers.

		VoxCeleb50-20		Music		Birdsong		VoxCeleb	
		Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
VGG-Vox	Superv.	26.07	53.70	13.81	**36.15**	14.85	32.96	**74.42**	**92.78**
	CL-Audio	**50.18**	**72.89**	**18.91**	35.48	**23.36**	**39.58**	70.56	91.55
LSTM	Superv.	15.38	41.02	3.55	10.01	13.67	30.09	65.41	87.85
	CL-Audio	35.18	65.91	11.25	30.85	16.69	39.18	55.27	81.59
Transformer	Superv.	13.81	36.85	2.52	7.41	7.21	17.66	45.18	66.71
	CL-Audio	13.54	36.87	5.22	15.17	7.81	21.58	25.18	52.54

Table 5.4: Top-1 and Top-5 accuracy of all models on each dataset trained using a supervised method and our method.

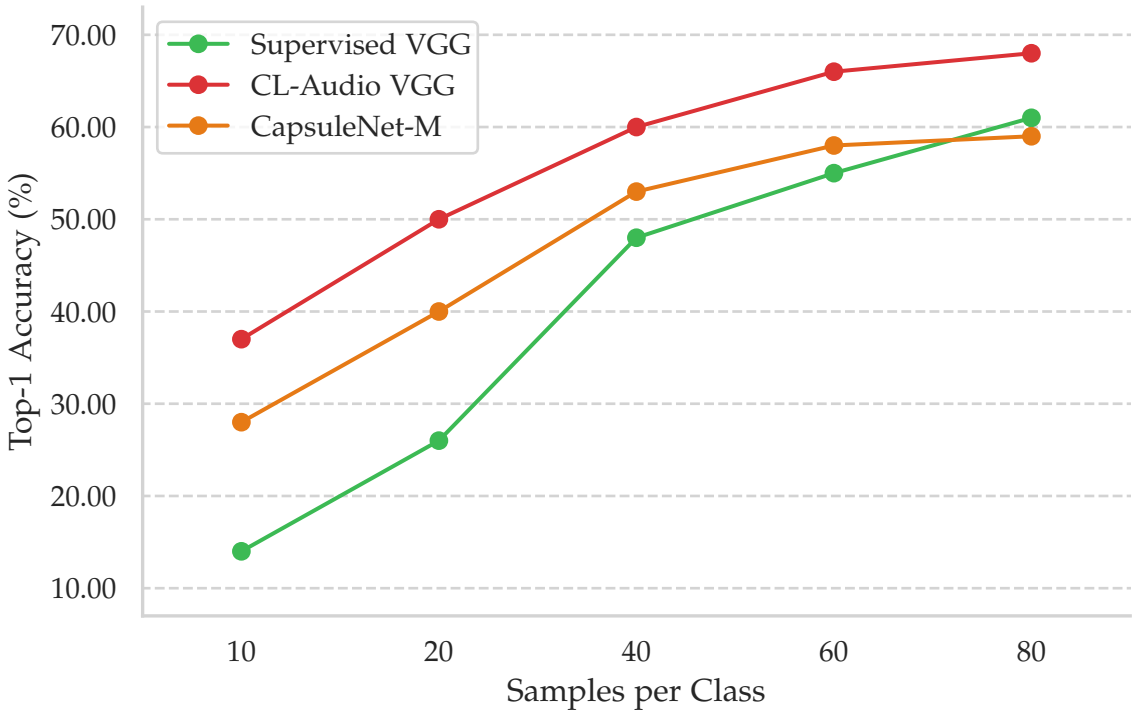
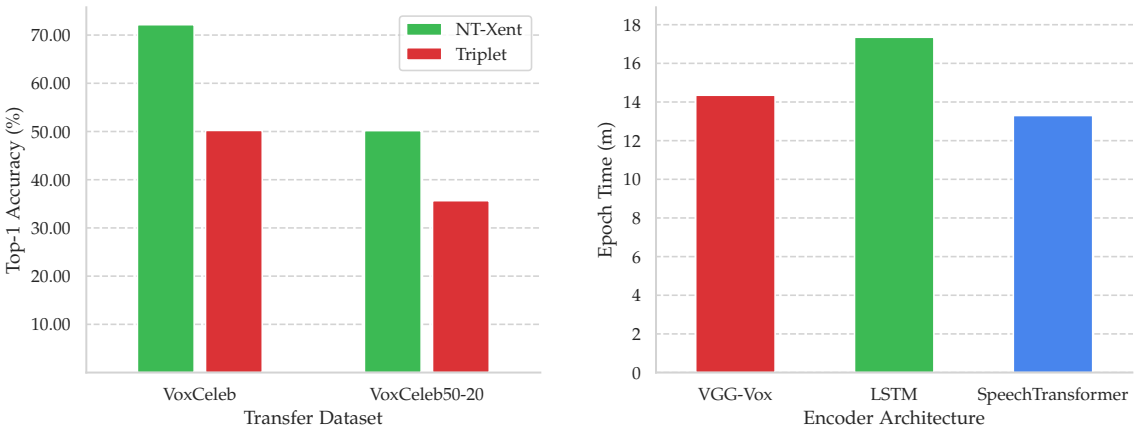


Figure 5.5: Test accuracy on 50 classes of VoxCeleb for different networks trained with different amounts of training data. Our method is in red. Orange line is reprinted from [57]. Green line is the supervised baseline.

It can be observed that the CNN architecture outperforms all other network architectures by a large margin. We want to note that the big discrepancy in the VoxCeleb dataset is due to the fact this model is the one used by the creators of the dataset, meaning it already underwent a long series of optimization rounds to find optimal hyperparameters and network architecture fine-tuned to the VoxCeleb dataset, whereas the other two models are not fine-tuned towards this dataset. Nonetheless, it also outperforms the other architectures in the other datasets but by a smaller margin. Overall all models performed best on the full VoxCeleb dataset, which is to be expected as it has the most training data. Interestingly all models perform the worst on the music classification dataset. The LSTM and the SpeechTransformer model trained purely supervised completely failed to learn good representations, scoring only slightly higher than random, whereas the networks trained using CL-Audio scored at least 10% and 11% respectively. The observed top-5 accuracy on all experiments is very consistently around 3 times as high as the top-1 accuracy.

Figure 5.5 shows the comparison of our method towards the results of Anand et al. [57] by plotting the maximum top-1 accuracies of all models as training samples per class increase on the VoxCeleb dataset limited to the first 50 classes.

We clearly outperform all other methods up until 80 training examples per class. Using 20



(a) Comparison of loss functions

(b) Comparison of training times

Figure 5.6: **Left**: Comparison of top-1 accuracies of a VGG-Vox model pre-trained using a NT-Xent loss and one using a triplet loss. Higher is better. **Right**: Comparison of the training time per epoch in minutes for all three models implemented for this thesis. Lower is better.

training examples we outperform the Capsule Network by as much as 10.22%. Note that we outperform the other networks with only as little as 10% of the number of trainable parameters. Whereas our classification head only has 140,000 trainable parameters the capsule network has 16.7 million. This reduction in parameters obviously reduces the required memory for gradient computation by 90% and it enables training on devices with very small Graphical Processing Units (GPUs). The results of the capsule network were not reproduced for this work but rather adopted from the original paper. To ensure consistency of the results we reproduced the results of the fully supervised VGG-based network and found almost a perfect match in performance compared to the results stated by [57].

Next we compare two different loss functions for our learning framework. The results of the triplet and NT-Xent loss function applied to the VGG-Vox network trained using the CL-Audio framework and transferred to the VoxCeleb dataset can be found in Figure 5.6a. We see that, consistent with the findings by Chen et al. [1], NT-Xent outperforms the standard triplet-loss by a margin of 20%. Note that no hard-triplet mining was conducted during training with the triplet loss.

Figure 5.6b shows the training time per epoch in minutes for the VGG-Vox, the LSTM-based network and the SpeechTransformer. The dataset used was the full VoxCeleb dataset, so one epoch corresponds to 145,265 utterances. The LSTM is the slowest of the three networks, as was to be expected. Interestingly the SpeechTransformer is significantly faster than the VGG-Vox CNN. One reason for this is the lower amount of trainable parameters, as shown in Table 5.4. Since the maximum batch size for the SpeechTransformer was only 60, compared to 80 for the VGG-Vox and LSTM, there might even be more performance gains for this model with more VRAM. Even though the SpeechTransformer trained fast we still do not believe that it is a feasible option due to the significantly worse accuracy it provided. We argue that

(a) Colors indicate Artists

(b) Colors indicate Genres

Figure 5.7: Randomly chosen clips from six artists of the music classification dataset embedded using a VGG-Vox pre-trained using CL-Audio. The embeddings are reduced to two dimensions using t-SNE.

with its good training time and strong accuracy the VGG-Vox model clearly outperforms the other two models.

Now we look at the relevance of classification-head architecture. Table 5.5 shows various architectures, their respective number of trainable parameters and their top-1 accuracy on the VoxCeleb50-20 dataset. The number of neurons per layer is annotated as a tuple where the first entry represents the first layer's neurons and the last entry represents the last layer's neurons. All layers are initializers with a normal distribution. They are purely dense layers and use L2 kernel regularization, batch normalization and ReLU activations. As can be clearly seen an increase in parameters does not correlate with an increase in accuracy. Only at very low parameterized models can we observe a decrease in performance. Additional layers actually decrease the performance of the model as a whole.

Now we take a closer look at qualitative results in the form of latent representations of songs, created using a VGG-Vox network trained using CL-Audio. Figure 5.7 shows two scatterplots of these embeddings, reduced to 2-dimensions using t-SNE. In Figure 5.7a, the colors indicate different artists while in Figure 5.7b the colors indicate genres. We can clearly see that the network distinguishes genre better than artists. One problem with our dataset might be the high percentage of rock artists and the considerably low percentage of classical artists. This explains the mediocre performance of the models on the artist classification dataset.

Shape	Top-1 Accuracy	Trainable weights
[1024]	41.25	1,104,946
[512]	43.78	553,522
[256]	**50.18**	277,811
[128]	46.91	139,954
[64]	45.51	71,026
[32]	33.14	36,562
[16]	26.28	19,331
[1024,1024]	44.36	2,156,594
[1024,512]	47.39	1,605,170
[256,512]	47.19	423,218
[256,256]	45.76	344,114
[128,128]	46.28	156,722
[512,64]	40.11	564,082
[1024,1024,1024]	42.30	3,208,242
[1024,512,256]	42.49	1,724,210
[512,256,128]	43.22	699,314
[256,512,128]	43.52	469,938

Table 5.5: Different classification head architectures. All heads are fully-connected networks. The shape describes the number of neurons per layer, where the leftmost number represents the first layer. Respective top-1 accuracies on VoxCeleb50-20 and number of trainable parameters of the head (rest of the model excluded).

6 Conclusion and Future Work

In this thesis, we investigated various subfields of machine learning. While all subfields have their own interesting peculiarities, they all have one common goal: to create strong models that generalize well to unseen data. Looking at our proposed framework we achieved exactly that. In much detail we described our learning framework called "CL-Audio" in Chapter 4. It is a general-purpose framework that can be applied to any network architecture to learn similarities in audio data.

We then used the representations learned by CL-Audio in a series of few-shot classification tasks. This problem of few-shot learning is very common in ML and becomes relevant every time data cannot be easily obtained or annotated. In various experiments in Chapter 5 we showed that using this approach we outperform previous attempts of few-shot learning on audio data by as much as 10.22%. Not only did we collect impressive results, we also had several very interesting findings along the way and many of our conducted experiments also failed to provide the expected outcome. We were especially surprised by the weak performance of the Transformer architecture in the context of audio data. While the Transformer still dominates almost all NLP tasks, for audio it does not seem to be easily transferable. The first major problem that we encountered was with the non-existence of positional information. While traditionally this is solved using an added positional encoding signal, we found that this is not enough for the network to learn reasonable temporal dependencies in audio. Leaving the embedding out provided similar results. The convolutions required at the beginning of the network also provided troubles later on. While reducing the amount of data in two dimensions, time and frequency, convolutions add a third dimension for the different filters, called channels. Since we need two-dimensional input for the multi-head-attention block, we had to either average over the channels or flatten frequency and channels to one dimension. Since averaging failed to learn completely, flattening provided us with an even larger input-matrix. To mitigate this we introduced a dense layer that reshaped every time slot to a smaller dimension of the wanted size. Even though this worked it required a lot of fine-tuning to get this layer to learn a reasonable mapping. Lastly, we encountered the problem of obtaining a one-dimensional vector representation out of the two-dimensional output of the last MHA-block. Again we had two choices. The first option was to add a learnable vector to the input-matrix in the temporal dimension and use its last hidden state as a summary of the entire sequence. This is a similar approach as the BERT architecture [75] uses with its $[CLS]$ token appended to each sentence. This approach did not learn a good representation though, so we were left with averaging over all time-steps. This worked but is an imperfect solution since averaging over the 300 provided time slots loses a lot of information contained in the outputs. Though all these problems were non-existent in the LSTM model, since the model was very basic, the results were also weak. The best performing

model by far was the CNN. This was to be expected since it is also the most used network for audio classification in the literature, meaning its architecture was already fine-tuned and polished by many other works while the transformer is comparably new. Looking at the problems we encountered with the transformer almost all of them are irrelevant for the CNN based model. Even though no global temporal dependencies can be found, local ones seem to work just as well. The problem of obtaining a single vector representation is solved by averaging over the frequency axis. This works only because at this point the information is so compressed that we average over only 9 dimensions instead of 300 for the Transformer.

As for the performance of the models on the various datasets we were surprised by the discrepancy of a model's top-1 accuracy between different datasets. While all models scored the best on *VoxCeleb*, every model's performance was weakest on music classification. Even though a substantial amount of the training data in the pre-training stage was music, it was still a very hard task for the network to classify the artists. The major problem was the fact that the network seemed to distinguish genre better than the artists as seen in Figure 5.7. The gain of CL-Audio was highest in the VGG-Vox model and the *VoxCeleb50-20* dataset though it can be clearly seen that all domains benefited from not initializing weights at random. We found that the amount that a model benefits do vary depending on the task. We argue that in no task should the weights be initialized at random but rather using a pre-trained encoder model and only retraining a small classification head network on the specific domain. The gains achieved towards either fully supervised methods or towards more complex architectures like the capsule network become clear when looking at the results stated in this work.

Another problem we found was that the time-stretching and pitch-shifting operations performed during augmentation required a lot of computational time, especially as batch sizes increased. This meant that the Central Processing Unit (CPU) could not be fully utilized due to the fact that augmentation was done on the CPU and therefore training time increased drastically. Propagating those costly operations to the GPU is not easily achievable though since it requires other operations, like the STFT, to be offloaded as well.

Looking at the good results of this work, we are even more confident that the future of machine learning lies in self-supervised learning and transfer learning. The sheer amount of data that can be processed in a self-supervised way will always outperform the human-centric way of data annotation and supervised learning. But the even more important part is about the transferal of learned knowledge to a new task. For humans this is natural, but current-day ML algorithms learn knowledge from scratch on every task over and over. One can clearly see the limitations of this approach as networks become larger and integration with the real world becomes ever more important. We argue that real-world robots trained using ML will only be able to operate well if they have an excellent internal model of the real physical world that can only be obtained by reusing information from one problem to another. This combination of self-supervised learning and few-shot-transfer-learning will only become more important in the future as limits of supervised methods will be reached.

This thesis is a foundation for future works on the task of self-supervised few-shot transfer-learning in the domain of audio data. In the future this framework could be tested with

even bigger, state of the art, models like the ResNet architecture proposed by He et al. [52] which was already successfully applied to audio data. We believe that bigger models like the ResNet can profit even more from the benefits of pre-training using CL-Audio. Also, the augmentations were not yet fully tested on their individual contribution towards the final goal. Future work might look into this and see which augmentations are most important and how to set the hyperparameters for an optimal result. Maybe an increase in augmentation difficulty as the network converges can be used in a similar fashion as learning rate decay. Another interesting task that was not tackled in this thesis is verification. Verification is the task of verifying if two speakers have the same identity. Our framework would be an optimal fit for this problem.

List of Figures

2.1 Latent space . . . 5
2.2 Contrastive Learning . . . 6
2.3 Transfer Learning Overview . . . 7
2.4 Triplet loss example . . . 9
2.5 Convolution, ReLU and max-pool overview . . . 12
2.6 LSTM cell Overview . . . 13
2.7 Scaled Dot-Product Attention and Multi-Head-Attention . . . 14
2.8 Discrete Fourier Transform . . . 15
2.9 Spectrogram example . . . 17
2.11 Low-pass filter responses . . . 19
2.12 Filter demonstration . . . 19
2.13 Phase Vocoder Overview . . . 20

3.1 SimCLR overview . . . 22

4.1 Contrastive learning for Audio . . . 26
4.2 Augmentations . . . 29
4.3 Crop . . . 30
4.4 Gain . . . 30
4.5 Whitenoise . . . 31
4.6 Butterworth-Filter with different orders . . . 32
4.7 Low-pass filter comparison . . . 32
4.8 Time-stretch . . . 33
4.9 Resampling . . . 34
4.10 Pitchshift . . . 35

5.1 VGG-Vox Model . . . 41
5.2 LSTM Model . . . 42
5.3 Positional Encoding . . . 43
5.4 SpeechTransformer Model . . . 44
5.5 Comparison of models on different dataset sizes . . . 47
5.6 Comparison of loss functions and models . . . 48
5.7 t-SNE plots of music embeddings . . . 49

List of Tables

5.1 Datasets 38
5.2 Encoder Models 40
5.3 Augmentation hyperparameters 45
5.4 Model comparison results 46
5.5 Comparison of classification head shapes 50

Acronyms

BERT Bidirectional Encoder Representations from Transformers. 44, 51

CCE Categorical Cross-Entropy. 4

CL-Audio Contrastive Learning for Audio. 24, 26, 37, 39, 45, 47–49, 51–53

CLI Command Line Interface. 33

CNN Convolutional Neural Network. 11, 12, 23, 42, 47, 48, 52

CPU Central Processing Unit. 52

DC Direct Current. 35

DFT Discrete Fourier Transformation. 15, 16

DSP Digital Signal Processing. 18

FFT Fast Fourier Transform. 16

GPU Graphical Processing Unit. 48, 52

LSTM Long Short-Term Memory. 12–14, 41, 42, 44, 47, 48, 51

MHA Multi-Head Attention. 14, 51

MSE Mean Squared Error. 5

NLP Natural Language Processing. 14, 51

NT-Xent Normalized Temperature-Scaled Cross Entropy. 10, 22, 24, 27, 45, 46, 48

PCM Pulse-Code Modulation. 10, 28, 38, 39

ReLU Rectified Linear Unit. 12, 40, 49

ResNet Residual Neural Network. 21, 53

RNN Recurrent Neural Network. 12, 14

SGD Stochastic Gradient Descent. 45

SimCLR A Simple Framework for Contrastive Learning of Visual Representations. 1, 2, 10, 21–23

STFT Short-Time Fourer Transform. 16, 17, 20, 26, 33, 35, 43, 46, 52

t-SNE t-distributed Stochastic Neighbor Embedding. 46, 49

Bibliography

[1] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. *A Simple Framework for Contrastive Learning of Visual Representations*. 2020. arXiv: 2002.05709 [cs.LG].

[2] J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, M. G. Azar, B. Piot, K. Kavukcuoglu, R. Munos, and M. Valko. *Bootstrap your own latent: A new approach to self-supervised Learning*. 2020. arXiv: 2006.07733 [cs.LG].

[3] P. H. Richemond, J.-B. Grill, F. Altché, C. Tallec, F. Strub, A. Brock, S. Smith, S. De, R. Pascanu, B. Piot, and M. Valko. *BYOL works even without batch statistics*. 2020. arXiv: 2010.10241 [stat.ML].

[4] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, and G. Hinton. *Big Self-Supervised Models are Strong Semi-Supervised Learners*. 2020. arXiv: 2006.10029 [cs.LG].

[5] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97. DOI: 10.1109/MSP.2012.2205597.

[6] A. Khamparia, D. Gupta, N. G. Nguyen, A. Khanna, B. Pandey, and P. Tiwari. "Sound Classification Using Convolutional Neural Network and Tensor Deep Stacking Network". In: *IEEE Access* 7 (2019), pp. 7717–7727. DOI: 10.1109/ACCESS.2018.2888882.

[7] H. Chen, W. Xie, A. Vedaldi, and A. Zisserman. *VGGSound: A Large-scale Audio-Visual Dataset*. 2020. arXiv: 2004.14368 [cs.CV].

[8] L. Y. Pratt. "Discriminability-Based Transfer between Neural Networks". In: *Advances in Neural Information Processing Systems 5*. Ed. by S. J. Hanson, J. D. Cowan, and C. L. Giles. Morgan-Kaufmann, 1993, pp. 204–211. URL: http://papers.nips.cc/paper/641-discriminability-based-transfer-between-neural-networks.pdf.

[9] S. W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. USA: California Technical Publishing, 1997. ISBN: 0966017633.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. The MIT Press, 2016. ISBN: 0262035618.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[12] K. P. F.R.S. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572. DOI: 10.1080/14786440109462720.

[13] M. Hosseini, A. S. Maida, M. Hosseini, and G. Raju. *Inception-inspired LSTM for Next-frame Video Prediction*. 2020. arXiv: 1909.05622 [cs.CV].

[14] S. Gidaris, P. Singh, and N. Komodakis. *Unsupervised Representation Learning by Predicting Image Rotations*. 2018. arXiv: 1803.07728 [cs.CV].

[15] A. Zisserman. *Self-Supervised Learning*. 2018. URL: https://project.inria.fr/paiss/files/2018/07/zisserman-self-supervised.pdf (visited on 10/25/2020).

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[17] Q. Yang, Y. Zhang, W. Dai, and S. J. Pan. *Transfer Learning*. Cambridge University Press, 2020. DOI: 10.1017/9781139061773.

[18] M. Shaha and M. Pawar. "Transfer Learning for Image Classification". In: *2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA)*. 2018, pp. 656–660. DOI: 10.1109/ICECA.2018.8474802.

[19] C. Fellicious. *Transfer Learning and Organic Computing for Autonomous Vehicles*. 2018. arXiv: 1808.05443 [cs.LG].

[20] A. Dosovitskiy, P. Fischer, J. T. Springenberg, M. Riedmiller, and T. Brox. *Discriminative Unsupervised Feature Learning with Exemplar Convolutional Neural Networks*. 2015. arXiv: 1406.6909 [cs.LG].

[21] A. van den Oord, Y. Li, and O. Vinyals. *Representation Learning with Contrastive Predictive Coding*. 2019. arXiv: 1807.03748 [cs.LG].

[22] P. Bachman, R. D. Hjelm, and W. Buchwalter. *Learning Representations by Maximizing Mutual Information Across Views*. 2019. arXiv: 1906.00910 [cs.LG].

[23] S. J. Pan and Q. Yang. "A Survey on Transfer Learning". In: *IEEE Transactions on Knowledge and Data Engineering* 22.10 (2010), pp. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[24] K. Q. Weinberger, J. Blitzer, and L. K. Saul. "Distance metric learning for large margin nearest neighbor classification". In: *In NIPS*. MIT Press, 2009.

[25] F. Schroff, D. Kalenichenko, and J. Philbin. "FaceNet: A unified embedding for face recognition and clustering". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2015). DOI: 10.1109/cvpr.2015.7298682. URL: http://dx.doi.org/10.1109/CVPR.2015.7298682.

[26] K. Sohn. "Improved Deep Metric Learning with Multi-class N-pair Loss Objective". In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 29 (NIPS 2016)* (2016), pp. 1849–1857.

[27] Y. Tu, J. Feng, and Y. Yang. *AAG: Self-Supervised Representation Learning by Auxiliary Augmentation with GNT-Xent Loss*. 2020. arXiv: 2009.07994 [cs.CV].

[28] J. M. Giorgi, O. Nitski, G. D. Bader, and B. Wang. *DeCLUTR: Deep Contrastive Learning for Unsupervised Textual Representations*. 2020. arXiv: 2006.03659 [cs.CL].

[29] A. Nandan and J. Vepa. *Language Agnostic Speech Embeddings for Emotion Classification*. 2020.

[30] N. Ajavakom. *Introduction to Mechanical Vibration*. 2017. URL: http://pioneer.netserv.chula.ac.th/~anopdana/433/ch1.pdf (visited on 10/25/2020).

[31] I. Lezhenin, N. Bogach, and E. Pyshkin. "Urban Sound Classification using Long Short-Term Memory Neural Network". In: Sept. 2019, pp. 57–60. DOI: 10.15439/2019F185.

[32] Y. zhao, J. Li, X. Wang, and Y. Li. "The Speechtransformer for Large-scale Mandarin Chinese Speech Recognition". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 7095–7099. DOI: 10.1109/ICASSP.2019.8682586.

[33] S.-W. Fu, Y. Tsao, X. Lu, and H. Kawai. *Raw Waveform-based Speech Enhancement by Fully Convolutional Networks*. 2017. arXiv: 1703.02205 [stat.ML].

[34] P. Dhariwal, H. Jun, C. Payne, J. W. Kim, A. Radford, and I. Sutskever. *Jukebox: A Generative Model for Music*. 2020. arXiv: 2005.00341 [eess.AS].

[35] Y. Lecun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. "Backpropagation applied to handwritten zip code recognition". English (US). In: *Neural Computation* 1.4 (1989), pp. 541–551. ISSN: 0899-7667.

[36] V. Nair and G. E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines." In: *ICML*. Ed. by J. Fürnkranz and T. Joachims. Omnipress, 2010, pp. 807–814. URL: http://dblp.uni-trier.de/db/conf/icml/icml2010.html#NairH10.

[37] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[38] D. Scherer, A. Müller, and S. Behnke. "Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition". In: Jan. 2010, pp. 92–101. ISBN: 978-3-642-15824-7. DOI: 10.1007/978-3-642-15825-4_10.

[39] Z. Wang, W. Yan, and T. Oates. *Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline*. 2016. arXiv: 1611.06455 [cs.LG].

[40] S. Hochreiter and J. Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[41] A. Graves, A.-r. Mohamed, and G. Hinton. *Speech Recognition with Deep Recurrent Neural Networks*. 2013. arXiv: 1303.5778 [cs.NE].

[42] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008. URL: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

[43] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[44] J. O. Smith. *Introduction to Digital Filters with Audio Applications*. online book. http://-ccrma.stanford.edu/~jos/filters/, 2007.

[45] C. McGillem and G. Cooper. *Continuous and Discrete Signal and System Analysis*. Holt, Rinehart, and Winston series in electrical engineering. Saunders College Pub., 1991. ISBN: 9780030510199. URL: https://books.google.de/books?id=EI4oAQAAMAAJ.

[46] J. L. Flanagan and R. M. Golden. "Phase Vocoder". In: *Bell System Technical Journal* 45.9 (1966), pp. 1493–1509. DOI: 10.1002/j.1538-7305.1966.tb01706.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1966.tb01706.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1966.tb01706.x.

[47] M. Portnoff. "Time-scale modification of speech based on short-time Fourier analysis." In: (Aug. 2005).

[48] E. Moulines and J. Laroche. "Non-parametric techniques for pitch-scale and time-scale modification of speech". In: *Speech Communication* 16.2 (1995). Voice Conversion: State of the Art and Perspectives, pp. 175–205. ISSN: 0167-6393. DOI: https://doi.org/10.1016/0167-6393(94)00054-E. URL: http://www.sciencedirect.com/science/article/pii/016763939400054E.

[49] Z. Průša and N. Holighaus. "Phase vocoder done right". In: *2017 25th European Signal Processing Conference (EUSIPCO)*. 2017, pp. 976–980. DOI: 10.23919/EUSIPCO.2017.8081353.

[50] R. Hadsell, S. Chopra, and Y. LeCun. "Dimensionality Reduction by Learning an Invariant Mapping". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100.

[51] Z. Wu, Y. Xiong, S. Yu, and D. Lin. *Unsupervised Feature Learning via Non-Parametric Instance-level Discrimination*. 2018. arXiv: 1805.01978 [cs.CV].

[52] K. He, X. Zhang, S. Ren, and J. Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[53] J. S. Chung, J. Huh, S. Mun, M. Lee, H. S. Heo, S. Choe, C. Ham, S. Jung, B.-J. Lee, and I. Han. *In defence of metric learning for speaker recognition*. 2020. arXiv: 2003.11982 [eess.AS].

[54] D. J. Rezende, S. Mohamed, I. Danihelka, K. Gregor, and D. Wierstra. *One-Shot Generalization in Deep Generative Models*. 2016. arXiv: 1603.05106 [stat.ML].

[55] S. O. Arik, J. Chen, K. Peng, W. Ping, and Y. Zhou. *Neural Voice Cloning with a Few Samples*. 2018. arXiv: 1802.06006 [cs.CL].

[56] S.-Y. Chou, K.-H. Cheng, J.-S. R. Jang, and Y.-H. Yang. *Learning to match transient sound events using attentional similarity for few-shot sound recognition*. 2019. arXiv: 1812.01269 [cs.SD].

[57] P. Anand, A. K. Singh, S. Srivastava, and B. Lall. *Few Shot Speaker Recognition using Deep Neural Networks*. 2019. arXiv: 1904.08775 [eess.AS].

[58] S. Sabour, N. Frosst, and G. E. Hinton. *Dynamic Routing Between Capsules*. 2017. arXiv: 1710.09829 [cs.CV].

[59] C. Medina, A. Devos, and M. Grossglauser. *Self-Supervised Prototypical Transfer Learning for Few-Shot Classification*. 2020. arXiv: 2006.11325 [cs.LG].

[60] D. Chen, Y. Chen, Y. Li, F. Mao, Y. He, and H. Xue. *Self-Supervised Learning For Few-Shot Image Classification*. 2020. arXiv: 1911.06045 [cs.CV].

[61] Y. Guo, N. C. Codella, L. Karlinsky, J. V. Codella, J. R. Smith, K. Saenko, T. Rosing, and R. Feris. *A Broader Study of Cross-Domain Few-Shot Learning*. 2020. arXiv: 1912.07200 [cs.CV].

[62] S. Butterworth. *On the Theory of Filter Amplifiers*. Vol. 7. 1930, pp. 536–541.

[63] B. McFee. *pyrubberband*. Version latest. URL: https://github.com/bmcfee/pyrubberband.

[64] P. P. Ltd. *Rubberband*. Version latest. URL: https://breakfastquay.com/rubberband/.

[65] J. S. Chung, A. Nagrani, and A. Zisserman. "VoxCeleb2: Deep Speaker Recognition". In: *INTERSPEECH*. 2018.

[66] D. Stowell and X. C. contributors. *xccoverbl*. 2014. URL: https://archive.org/details/xccoverbl_2014 (visited on 10/25/2020).

[67] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur. "Librispeech: An ASR corpus based on public domain audio books". In: Apr. 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964.

[68] J. F. Gemmeke, D. P. W. Ellis, D. Freedman, A. Jansen, W. Lawrence, R. C. Moore, M. Plakal, and M. Ritter. "Audio Set: An ontology and human-labeled dataset for audio events". In: *Proc. IEEE ICASSP 2017*. New Orleans, LA, 2017.

[69] J. Chung and A. Zisserman. "Out of time: automated lip sync in the wild". In: *Workshop on Multi-view Lip-reading, ACCV*. 2016.

[70] V. Iashin. *v-iashin/VoxCeleb*. 2019. URL: https://github.com/v-iashin/VoxCeleb (visited on 10/25/2020).

[71] J. S. Chung, J. Huh, and S. Mun. *Delving into VoxCeleb: environment invariant speaker recognition*. 2020. arXiv: 1910.11238 [cs.SD].

[72] K. Okabe, T. Koshinaka, and K. Shinoda. "Attentive Statistics Pooling for Deep Speaker Embedding". In: *Interspeech 2018* (Sept. 2018). DOI: 10.21437/interspeech.2018-993. URL: http://dx.doi.org/10.21437/Interspeech.2018-993.

[73] Y. Yufeng. "Automatic speaker verification and diarization on VoxCeleb data collection". In: (June 2020). URL: https://smartech.gatech.edu/handle/1853/62830.

[74] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. *Return of the Devil in the Details: Delving Deep into Convolutional Nets*. 2014. arXiv: 1405.3531 [cs.CV].

[75] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[76] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[77] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'io, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[78] T. pandas development team. *pandas-dev/pandas: Pandas*. Version latest. Feb. 2020. DOI: 10.5281/zenodo.3509134. URL: https://doi.org/10.5281/zenodo.3509134.

[79] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[80] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. "librosa: Audio and music signal analysis in python". In: *Proceedings of the 14th python in science conference*. Vol. 8. 2015.

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