People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Mohamed Khider, Biskra
Faculty of Exact Sciences, Natural Sciences and Life
Computer Science Department

Order Number: IA1/M2/2018

Report

Presented to obtain the Academic Master Diploma in

Computer Science

Option: **Artificial Intelligence**

Environment Sensing using Acoustic Data

By:

Redjimi Adel

Defended the 24/06/2018, in front of the jury composed of:

Mr. Bourekkache Samir	MCB	President
Mr. Bachir Abdelmalik	Professor	Supervisor
Mr. Kerdoudi Mohammed Lamine	MCB	Examiner

Academic year: 2017/2018

Abstract

With the recent rise of deep learning approaches for artificial intelligence, we came to realize that developing computer systems that are reliably capable of understanding visual scenes and recognizing speech is finally possible. Unfortunately, little attention has been given to the idea of developing computer systems that can emulate the way we use our sense of hearing to make sense of what's around us. Making intelligent systems that can reliably recognize environmental sounds will cause a paradigm shift in technological areas such as audio surveillance, noise pollution analysis, audio-based search engines, and hearing aids. This work digs into the use of convolutional neural networks (CNN) for robust environmental sounds recognition, while also proposing a method for dealing with small datasets called stochastic continuous data augmentation. Obtained results have been compared with some previous related works.

Acknowledgements

I would first like to thank my project supervisor Prof Abdelmalik Bachir for guiding me to the right direction throughout the project and for giving me the sort of feedback that helped shaping this work.

I would also like to thank Justin Salamon of New York University for delivering further details on his previous work.

And finally, tremendous thanks the friends and family for their continuous support and belief throughout my entire academic career.

Contents

List of Figures

Chapter 1

Introduction

Everyday, the average human receives a considerable amount of cues about their environment through sounds. While the information carried through speech sounds serves the purpose of explicit human-to-human communication, the information carried through environmental sounds serves the purpose of implicit environment-to-human communication. Environmental sounds can carry various kinds of information: sounds like a police siren approaching or a dog bark getting louder may carry information about a potentially dangerous situation, a child's cry in the next room may carry valuable time-critical information about the child's health or safety, rain and thunder sounds carry weather-related information, ... etc.

The problem of Environmental Sounds Recognition (ESR) is defined as the automatic identification of environmental sounds. Developing the future's intelligent machines involves emulating human perceptions such as vision and hearing, such capabilities are necessary for machines with direct contact with the real world. Compared to other areas such as object recognition and speech recognition, environmental sounds recognition is relatively under-developed and under-investigated [1].

Developing computer systems that can make sense of environmental sounds has the potential to improve existing technologies as well as making way to newer ones. Time will reveal potential applications that are yet to be imagined. Recent Android-powered smartphones can perform speech recognition without using remote computational resources, this shows how powerful mobile processing units are getting. Having the possibility to embed such computational power on a small device convinces us to think about the potential of developing intelligent hearing aids, that are able to make sense of the environmental sounds surrounding the hearing-impaired user, and informing them in the cleanest possible way. Audio surveillance systems may also make use of environmental sounds recognition. Video surveillance systems fail to detect potentially dangerous events that aren't visible through the lenses of a camera, for e.g., a gunshot in the neighbor's house. Another potential application of ESR systems is urban noise analytics. Noise pollution is negatively affecting our physical and mental health [2]. While current studies rely solely on noise levels measured in decibels, ESR will cause a huge leap in the field of urban noise analytics. With ESR-capable urban noise analytics systems, we will be able to answer questions like: what kinds of noise have the most negative impact on students? and the keyword here is "kinds".

Part of the reason why it is not currently straightforward to tackle such problem is the relative lack of adequate data, again, compared to other kinds of data such as speech and images. Compiling a dataset of environmental sounds is tougher than compiling a dataset of speech or visual data. For speech data, we'd exploit the fact

that there are millions of professionally transcribed audio records and videos, that makes it relatively easy to get label speech data. Now consider this scenario to have an idea on why collecting environmental sounds data is tougher than image data: do a web search for pictures of dogs, then do a web search for audio clips of dog barks, of course you will find more images than audio clips, given images are more relevant on the web (Google, despite being the most widely used search engine, doesn't even offer a "sounds" category as it does with images), also, it will take a simple scrolling through a folder of hundreds of images to check for pictures without dogs and to clean up the data you collected, while it may require hours of listening to the dog barks you collected in order to be able to spot unwanted data. Along with the relative lack of adequate data, the dimensionality and variety of raw audio data make it even harder.

In the last decade, the field of machine learning has witnessed a paradigm shift with problems involving large amounts of high-dimensional data thanks to the idea of deep learning. Deep learning came to revolutionize the way we go about problems such as speech recognition [3], image classification [4], and object recognition [5].

While this work is not the first deep learning approach to the problem, it aims to tackle the problem using a deep learning model called the convolutional neural network along with two other techniques in order to improve the performance. The first technique, that was proposed to compensate for the lack of data, is named stochastic continuous data augmentation. Also, we employed an existing machine learning technique called transfer learning that allows us to keep the knowledge learned from one dataset, and use it with another one.

Chapter 2

Audio Signal Processing Basics

This chapter is heavily inspired by the first chapter "Sounds and signals" from the book "Think DSP: Digital Signal Processing in Python" by Allen B. Downey [6].

A **signal** represents a quantity that varies in time or space. It's a function that carries information about the behavior or the attributes of a particular phenomenon [7]. E.g, an image signal represents variations of brightness in space. An **audio** signal (or sound signal) represents variations of air pressure over time [6].

A **digital signal** is a sequence of discrete values that represents the variations of a quantity over time. In contrast to **analog signal**, which is a continuous stream of values. Let's have a concrete look at these two in the case of audio signals:

- In the physical world, air pressure changes continuously, thus, carried by an analog signal.
- A microphone converts the analog audio signal to an analog electrical signal, representing a change in voltage instead of air pressure.
- A digital recording device measures and stores the values of the analog electrical signal at a constant time rate. The result of this operation is a sequence of discrete values that we call a digital audio signal.

2.1 Digital audio signals

A digital audio signal is defined by three elements:

1. A **sequence of discrete values** that represent the variations in voltage that themselves represent the variations in air pressure carrying sounds. These values are picked from the analog electrical signal at a constant amount of time. Also note that these values can be stored as decimal or floating-point numbers. A single value among them is called a **sample**.
2. A **sample rate** which is the number of measurement in a ***second***. Sample rates are expressed in **Hertz** where ($1Hz = 1s^{-1}$). Having a sample rate of X Hertz means:
   - every second, the recording device measured and stored X samples from the analog signal,

– and every X samples from the sequence correspond to 1 second of sound.

3. A **bit-depth** which is the number of bits used to store each sample. More bits means a higher number of possible values, thus a higher fidelity to the digitized signal. Less bits means a lower number of possible values causing higher noise levels.

This representation of audio signals is called the **waveform** representation.

2.2 Periodic signals

A periodic signal is a signal that repeat itself after a constant amount of time. See Figure 2.1.

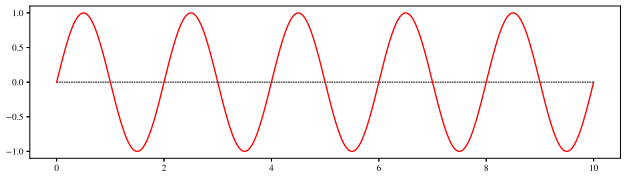


Figure 2.1: Example of a periodic signal

Notice that this periodic signal has the same shape as a trigonometric sine function. We call this type of curves **sinusoids**. As you can see, the signal illustrated in Figure 2.1 is repeated 5 times, also known as cycles. The duration of each cycle, called the period, and is equal to 2 time units in this example.

The **frequency** of a signal is the number of cycles per ***second***. This number is expressed in Hertz. If we consider the time unit in the example to be milliseconds ($ms = 10^{-3}s$), the signal would have a frequency of 500 Hertz ($= \frac{1}{2 \times 10^{-3}s}$).

The amplitude of a sinusoid periodic signal is a measure of its strength. The stronger the signal, the higher the amplitude.

2.3 Spectral decomposition

Spectral decomposition is the idea that any signal can be expressed as the sum of sinusoids periodic signals with different frequencies and different amplitudes.

2.3.1 Spectrum

The **spectrum** of a signal is the set of sinusoids (defined by their respective frequencies and amplitudes) that add up to produce the signal. To compute the spectrum of a particular signal, we use the **Fast Fourier Transform (FFT)**. The way FFT works is beyond the scope of this work. See Figure 2.2.

2.3.2 Time domain vs frequency domain

A **time domain** representation describes how a signal behaves over time. Whereas a **frequency domain** representation describes the signal with respect to frequency

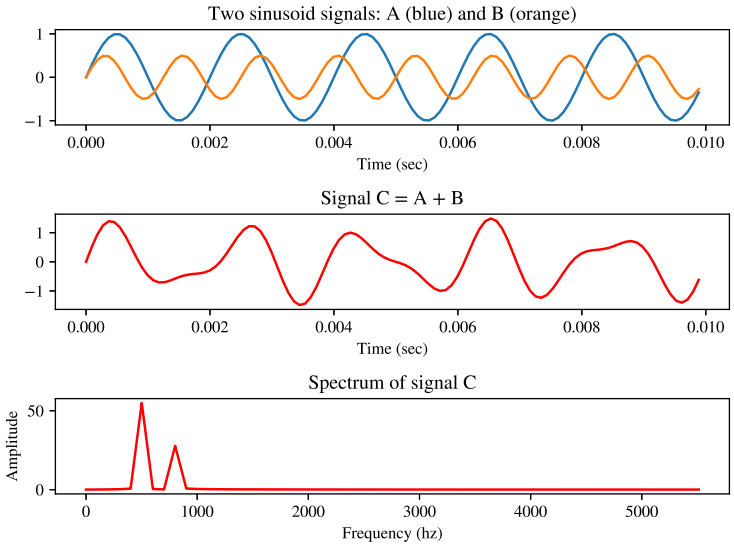


Figure 2.2: We consider two sinusoid signals for a segment of 0.01 seconds: A (blue) with a frequency of $500Hz$ and an amplitude of 1, and B (orange) with a frequency of $800Hz$ and an amplitude of 0.5. In the second subfigure, you can see the signal C as the result of adding up A and B. Using the Fast Fourier Transform, we computed the spectrum of the signal C. You can clearly see in the third subfigure that, according to the computed spectrum, C is the result of adding up two sinusoids signal of frequencies $500Hz$ and $800Hz$ with the first one having twice the amplitude of the second one.

rather than time, it shows how much of the signal lies within each frequency. The spectrum is a frequency domain representation.

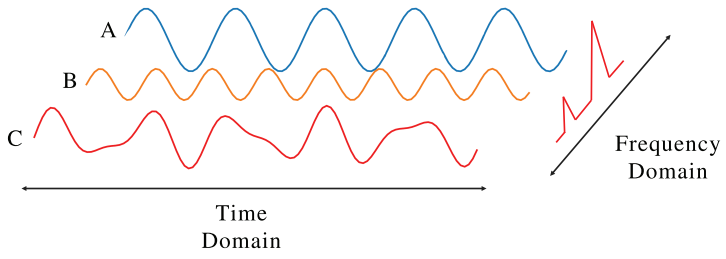


Figure 2.3: Time domain vs Frequency domain. Signals from Figure 2.2 were used.

The obvious disadvantage of spectrums is that they obscure out the relationship between frequency and time. For example, we cannot tell by looking at a spectrum whether some frequencies were changing strength over time. We lose track of time! See the example shown in Figure 2.4.

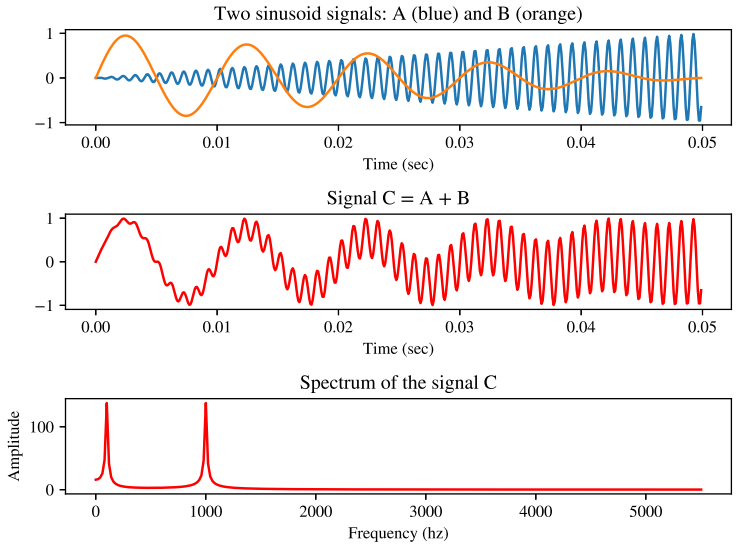


Figure 2.4: The spectrum fails to show the relationship between frequencies and time. It fails to show that one frequency (blue) kept getting stronger over time and the other (orange) kept getting weaker over time.

2.3.3 Spectrogram

A **spectrogram** is a representation that shows the relationship between frequency and time, in other words, it shows how frequencies behave over time. In order to make a spectrogram for a given signal, we break the signal into a number of short equally-sized segments, then we compute the spectrum for each segment. Mapping every time segment to its corresponding spectrum gives a 2 dimensional representation of the original signal, the 2 dimensions being: time and frequency. In Figure 2.5 we show how the spectrogram representation solves the problem faced in the example illustrated in Figure 2.4.

Windowing and overlapping

Discontinuities at the ends of the each segment can cause noise in the resulting spectrogram. In order to eliminate such noise, we can use a window function that gradually tapers the signal at both ends of each segment, see Figure 2.6. When building spectrograms out of windowed segments, it is preferred to use overlap in order to compensate for the tapered parts [8].

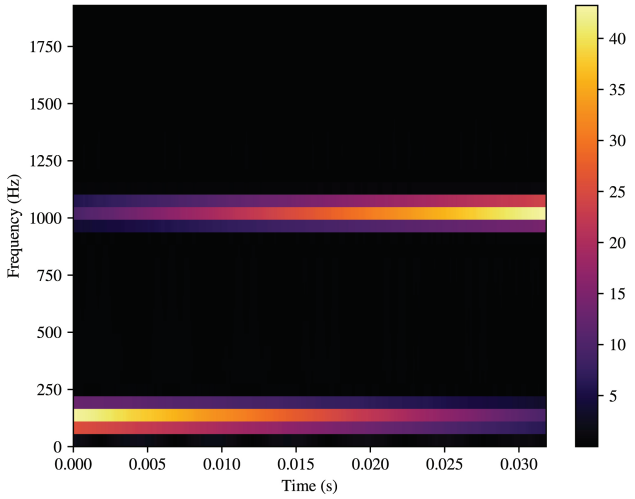


Figure 2.5: Spectrogram of the signal C from Figure 2.4. This representation can show us how frequencies, that makes our signal, behave over time. Notice the higher one ($\sim$ 1000Hz) getting stronger over time, whilst the lower one ($\sim$ 200Hz) getting weaker.

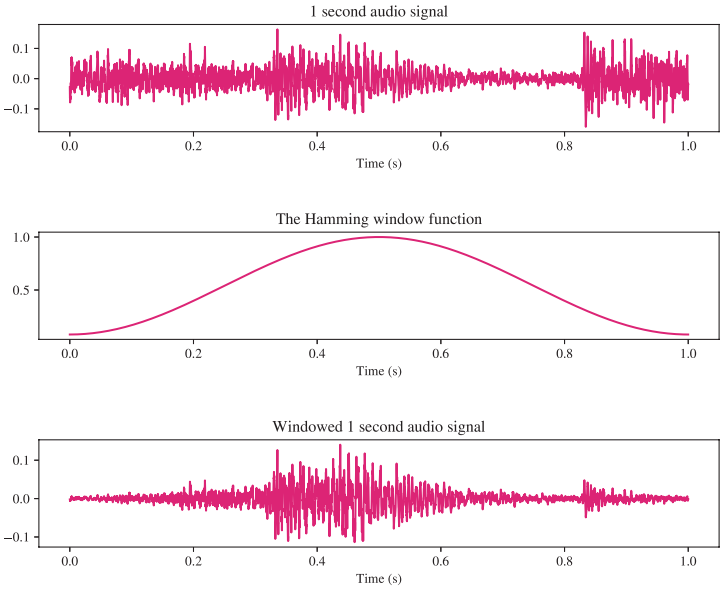


Figure 2.6: Using the Hamming window function to gradually taper the signal at both ends.

2.4 Sound perception

For most tasks (such as editing, generating, mixing, ...), computers use the waveform representation of audio signals. On the other side, human brains get access to information about the frequencies over time only.

2.4.1 Pitch

Pitch is a frequency-related perceptual property of sounds. The higher the frequency, the higher the pitch. Since it's not possible to include sounds on a paper format, few familiar sounds can you help understand the concept of high and low pitch:

Remarkably low pitch	**Remarkably high pitch**
Thunder	Siren
Lion	Baby cat
Boat horn	Whistle

If you're unfamiliar with this concept, we strongly recommend checking videos[1] regarding this concept given it's not possible to illustrate it on paper.

2.4.2 Mel scale

Experiments led on human subjects [9] revealed that: if a human hears 3 frequencies (on the Hertz scale) with equal distances from one another in order one after the other (say: 300hz, 500hz, then 800hz), he won't perceive the distances in pitch as equal. This means that evenly spaced frequencies along the Hertz scale aren't evenly spaced perceptually. The Mel scale is a perceptual scale of pitches where evenly spaced frequencies within it are also evenly spaced perceptually. The following equation shows the conversion between the frequency f in Hertz to m in mels:

$$m = 2595 \cdot \log_{10}(1 + \frac{f}{700})$$

2.4.3 Brain-inspired audio features

If we are willing to make computers that understand sounds the way humans do, a first step may be representing sounds loosely based on how the brains perceives them: *changes of mel-scaled frequencies over time*. As we've seen earlier, spectrograms express changes of frequencies (in Hertz) over time. The natural next step would be scaling the frequencies in these spectrograms to the mel scale obtaining mel-spectrograms (or mel-sclaed spectrograms) that will make the inputs to our learning model in this work.

[1] Video "Sound Waves: High Pitch and Low Pitch" : youtu.be/yMLTF_0PAQw

Chapter 3

Neural Networks and Deep Learning

Before diving into neural networks and deep learning, we will pass quickly through the definitions of some relevant concepts.

Artificial Intelligence

Artificial intelligence (AI) is a rapidly growing field of engineering and science that targets the development of computer systems that are capable of automating routine labor, understanding sounds, images or natural language, discovering valuable information in raw data, ... etc. The biggest challenge of modern AI is solving problems that are easy for humans to perform but hard to describe formally - problems that we solve intuitively, like recognizing sounds and images, and understanding natural language [10].

Machine Learning

Machine Learning (ML) is a field of AI that studies the design of computer programs that can solve problems by learning from data rather than being explicitly programmed how to solve them step-by-step.

Classification

A type of problems where machine learning has been absolutely useful are classification problems. Classification is identifying the class (or the category) of an observation (or an object). Examples:

- Classifying an email as spam or not spam
- Classifying an image into one of several classes: car, house, human, dog, ...
- Classifying an audio clip into one of the words: "okay", "google", "set", "alarm",
- etc ...

The reason why we use machine learning for classification is that we can't describe formally the way we classify things, thus, we can't come up with a programmable solution for such problems. The type of machine learning used for classification is supervised learning.

Supervised learning consists of automatically building a decision model (it can be a tree, a function, a set of rules, ...) that can classify observations (such emails) into one of many classes (spam or not spam) using a labeled dataset. A labeled dataset is a set of correctly classified examples. This decision model is called a classifier.

Mathematically, given a labeled dataset $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ where: x_i is an example, and y_i is its correct class (label). Supervised learning is finding a function h that approximates the function expressed by the data pairs in the dataset, and that generalizes well to examples from outside the dataset.

In supervised learning and classification, generalization is the ability of a classifier to perform at a good accuracy on examples outside the dataset.

Features

In classification, features are values that usefully characterize the things we wish to classify. These values are usually aligned in a vector called the feature vector. Examples:

Things to classify	Possible features
Faces	Distance between the eyes, eyes width, eyebrows width, eyebrows thickness, ...
Birds	Mass, wingspan, color, beak length, ...
Images	Histograms, edges, shapes, ...

To extract features from raw data we use programs called feature extractor. A feature extractor reduces raw data into a less complex form that may be easier for a machine learning algorithm to learn from.

An RGB 32x32 image has 3072 values that represent it ($3072 = 32 \times 32 \times 3$, 3 for RGB values). These 3072 values are also called low-level features. Dealing with such high dimensional data requires powerful hardware. For decades, machine learning practitioners relied on hard-coded feature extractors to extract higher-level features. However, the consequent dimensionality reduction of feature extraction process may lead to the loss of some information that can eventually help our learning algorithms perform better. The field of deep learning is based on the idea of learning feature extraction instead of hard-coding it.

Machine learning model evaluation

To evaluate the performance of a machine learning model on a given dataset we rely on some metrics such as accuracy, precision, recall, and others. Usually, the dataset is split into two subsets:

- the training set: used to train our model
- the test set: used to evaluate the performance of our model on examples that are not part of the training set

A validation set can be used to track the performance of the model during the learning process. Test and validation sets aren't meant to be as big as the training set.

However, there are two problems with splitting the dataset into two subsets:

- Sometimes we don't even have enough data, thus, selecting only a random subset of the available data may be counter-productive.

– What if the test data is biased? Therefore, the estimated performance can't be as reliable.

To deal with such kinds of problem, there is an evaluation strategy called **k-fold cross validation**. It works as follows:

– Pick an integer k, usually between 5 and 10.

– Split the dataset D into k equally-sized subsets. These subsets should be representative of the whole dataset, it means they contain samples from all classes.

– For each subset D_i, train the model (from scratch) on the other $k-1$ subsets, and evaluate it on D_i.

The average performance estimated over the k training is more reliable given we used all the data we have for both training and testing in the k setups.

Overfitting and generalization

Generalization describes the quality of a learning model that's capable of performing well enough on unseen data after training.

Overfitting is a term used to describe a machine learning model that successfully perform on the training set that was used for learning, but fails to perform on new examples (validation or test set). In such case, the learning model learned to memorize the training set and failed to generalize on unseen data. Usually this is caused by one of the following factors:

– There isn't much training data to generalize from.

– The model has too many parameters to learn, this leads to the model memorizing the training set rather than learning a generalized solution to the problem.

3.1 Artificial Neural Networks

Artificial neural networks (ANNs, neural networks or neural nets) are computational models that were initially inspired by the way our brains process information. The average human has about 100 billion neurons, each of these neurons is connected to a bunch of other neurons. A single neuron may receive an electrical signal from other neurons as an input, the strength of the signal coming from a particular neuron depends on how strong is the connection with that neuron, and depending on that signal, the neuron may fire or not. In this section we will explore the basics of artificial neural networks models and how they apply to machine learning.

3.1.1 Artificial Neuron

Let's jump now into the computational model of a neuron. We can see the artificial neuron as a function f that:

1. takes as inputs a set of values $x_1, x_2, \ldots x_n$,

2. each input value x_i is multiplied by a weight w_i that reflects how strong that connection is, giving us a weighted input $w_i \times x_i$

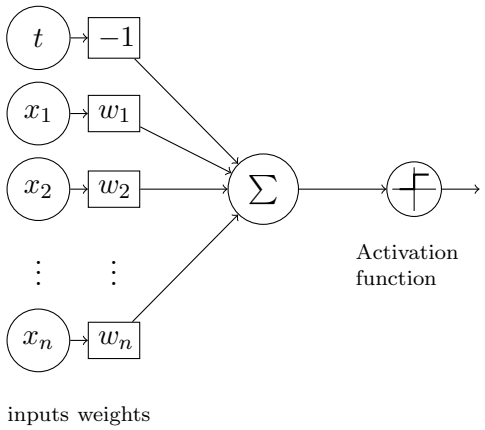


Figure 3.1: Computational model of a neuron

3. the weighted inputs now run through a sum $\sum_i w_i \times x_i$
4. then we get to decide whether the cumulative influence of inputs is sufficient to make the neuron fire (output = 1) or not (output = 0). We use a threshold value t for this, if the weighted sum of inputs is greater than t, the neuron should fire, otherwise it doesn't. This last step is called the activation.

We end up having this function that's parameterized by a vector of weights $W = [w_1, w_2, \ldots, w_n] \in \mathbb{R}^n$, a threshold $t \in \mathbb{R}$ and takes as inputs a vector $x = [x_1, x_2, \ldots, x_n]$:

$$f_{W,t}(x) = a(\sum_{i=1}^{n} w_i \times x_i - t)$$

$$a(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

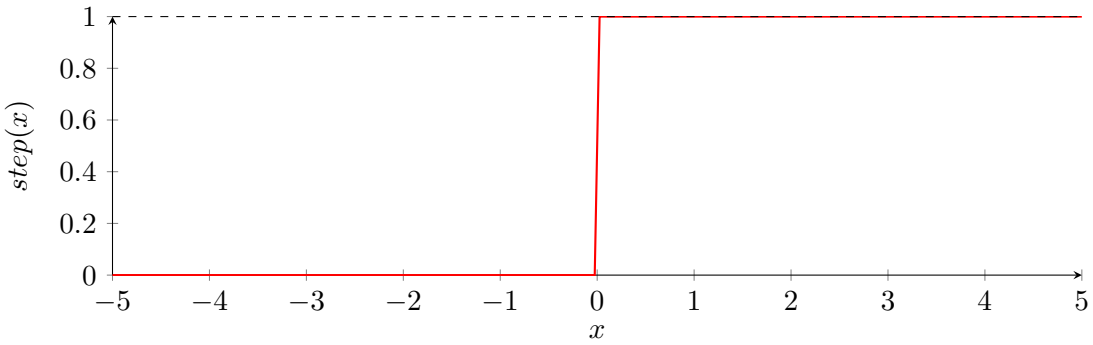


Figure 3.2: Step activation function

The function a is called the activation function, and in this basic case the activation function is a step function (see figure 3.2). Other models use other functions as activation functions.

The sum of the weighted inputs can be computed as a dot product of the vector x and W.

$$\sum_{i=1}^{n} w_i \times x_i - t = W \cdot x - t$$

To follow the conventions, we replace the term $-t$ with $+b$ ($b \in \mathbb{R}$) and we will call it the neuron's bias (please note that b can be negative and act the same as a threshold $-t$). We end up having the following equation that represents the pre-activation of a neuron:

$$\sum_{i=1}^{n} w_i \times x_i + b = W \cdot x + b$$

Thus, a neuron with an activation function a can be seen as a function f parameterized by a vector of weights $W = [w_1, w_2, \ldots, w_n] \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$ that takes as inputs a vector $x = [x_1, x_2, \ldots, x_n] \in \mathbb{R}^n$:

$$f_{W,b}(x) = a(W \cdot x + b)$$

Example: an AND gate neuron

Let's walk through a quick example of an artificial neuron implementing a logical AND gate. An AND gate takes 2 inputs in $\{0, 1\}$ and outputs 1 if, and only if, both inputs are 1.

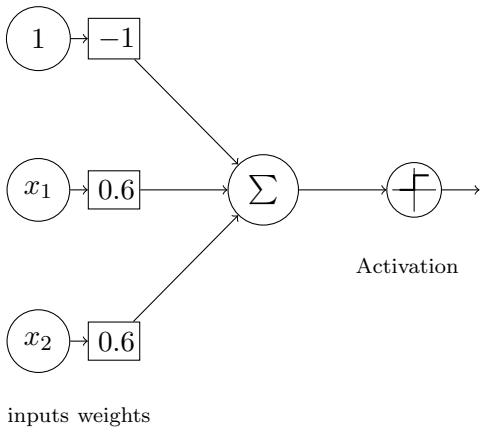


Figure 3.3: A neuron implementing a logical AND gate

So we have an artificial neuron with a threshold $t = 1$, and weights $w_1 = w_2 = 0.6$. Let's see how it computes the output for the inputs $(1, 0), (0, 0), (0, 1)$, and $(1, 1)$:

- $f(1, 0) = a((0.6 \times 1 + 0.6 \times 0) + (-1)) = a(-0.4) = 0$
- $f(0, 0) = a((0.6 \times 0 + 0.6 \times 0) + (-1)) = a(-1.2) = 0$
- $f(0, 1) = a((0.6 \times 0 + 0.6 \times 1) + (-1)) = a(-0.4) = 0$
- $f(1, 1) = a((0.6 \times 1 + 0.6 \times 1) + (-1)) = a(0.2) = 1$

Note: $a(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$

3.1.2 Activation Functions

An activation function a is a function that takes as inputs the pre-activation of a neurone and decides how the neuron should fire.

Several activations functions were proposed as alternatives to the step function already seen in the basic artificial neuron. As you will notice, that unlike the step function, proposed alternatives are differentiable functions. We will see in a while we tend to use differentiable functions as activation functions.

Notation: z is the pre-activation.

Sigmoid activation function

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.1}$$

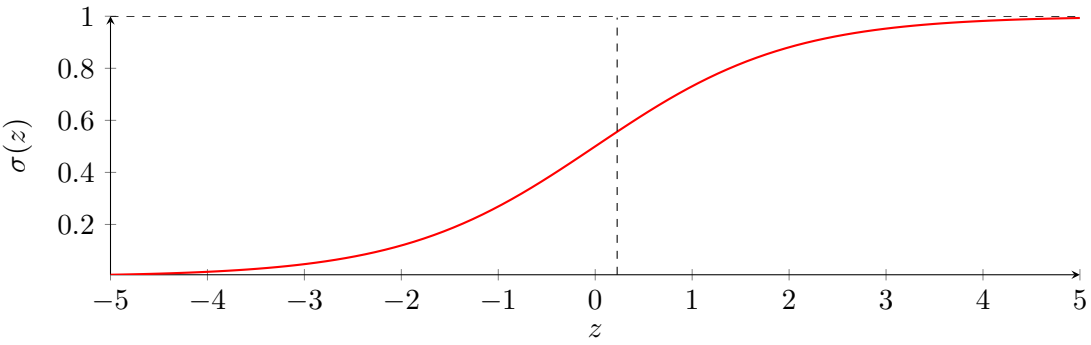


Figure 3.4: Sigmoid activation function

The first successful alternative to the step function and it was around for decades. The sigmoid function σ squashes the neuron's pre-activation between 0 and 1. It outputs a number closer to 1 when its inputs is so large, and a number closer to 0 when the input is so small (negative). See figure 3.4.

The sigmoid function has an interesting derivative, given as:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{3.2}$$

Hyperbolic tangent activation function

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{3.3}$$

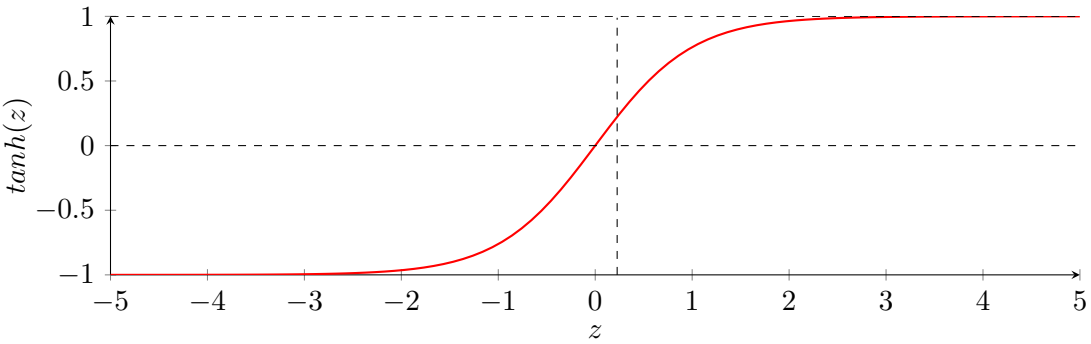


Figure 3.5: Hyperbolic tangent activation function

Another successful alternative is the hyperbolic tangent (tanh) function. It is very similar to sigmoid as it also performs a squashing to the pre-activation between -1 and

1. Notice that the inactivity of a neuron is represented by a -1 rather than 0 when using this function.

Rectified linear activation function

$$ReLU(z) = \max(z, 0) \tag{3.4}$$

$$ReLU'(x) = \begin{cases} 1 & x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

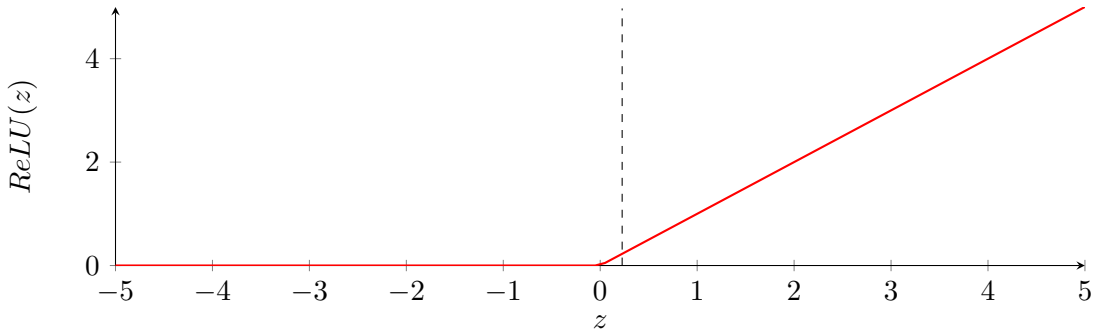


Figure 3.6: Hyperbolic tangent activation function

Also called ReLU (Rectified Linear Unit), the ReLU activation function is regarded by experts as the most popular activation function for deep learning [11]. It has strong biological motivations and mathematical justifications [12]. As well as some interesting properties:

- Computational cost: computing $max(z, 0)$ is certainly less power-intensive than its counterparts. The derivative is also easy to compute (see equation 3.5).
- Sparsity: suppose we have a large matrix of neurons, wouldn't be good if the inactive neuron have exactly 0 valued outputs so we have a sparse matrix? Of course it would be good, since operations on large sparse matrices are more efficient. Compared to the previous differentiable activations functions, ReLU is the only one that offers sparsity.

3.1.3 Capacity of a Single Neuron

An artificial neuron with the sigmoid σ activation function can be trained (by finding the right weights W and bias b) to perform binary classification. The output of the sigmoid function in $[0, 1]$ can be interpreted as estimating the probability of class $y = 1$ given a feature vector x, noted $P(y = 1|x)$. This classifier is also known under the name "logistic regression classifier" [10]. Note that the same can be applied to other activation functions with a slight change on how we interpret the output of the activation function.

Unfortunately, this simple classifier can only perform linear classification since it only uses a linear combination of its inputs. Linear classification is possible when the data can be separated by a hyperplan, or a line in the case of 2-dimensional data. See figure 3.7.

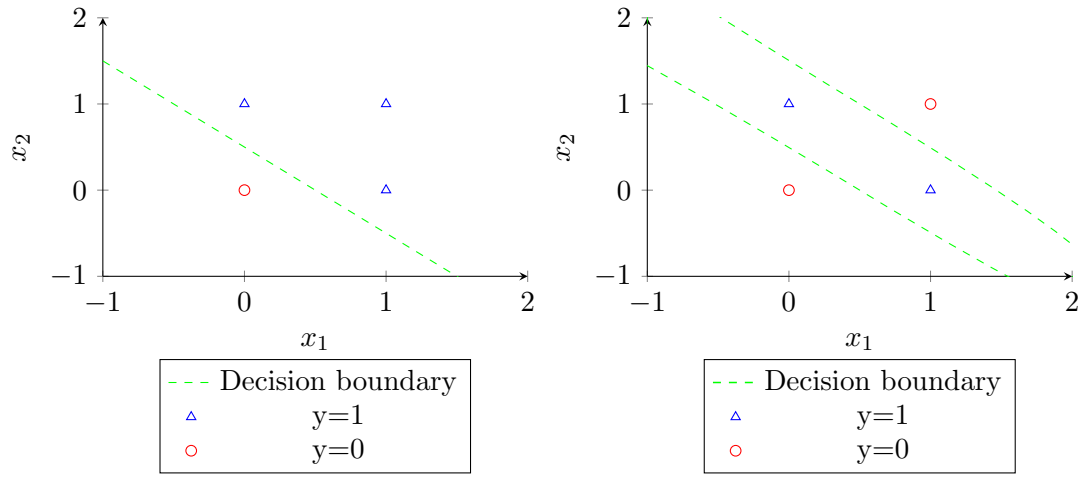


Figure 3.7: Logical OR data: linearly separable (left), Logical XOR data: non linearly separable (right)

Using multiple neurons to implement the logical XOR gate

As you can see in figure 3.7, we can't implement a logical XOR gate, which can be seen as a non linear classification problem, using a single neuron. But what if we change the representation of the the inputs using other neurons? Consider this:

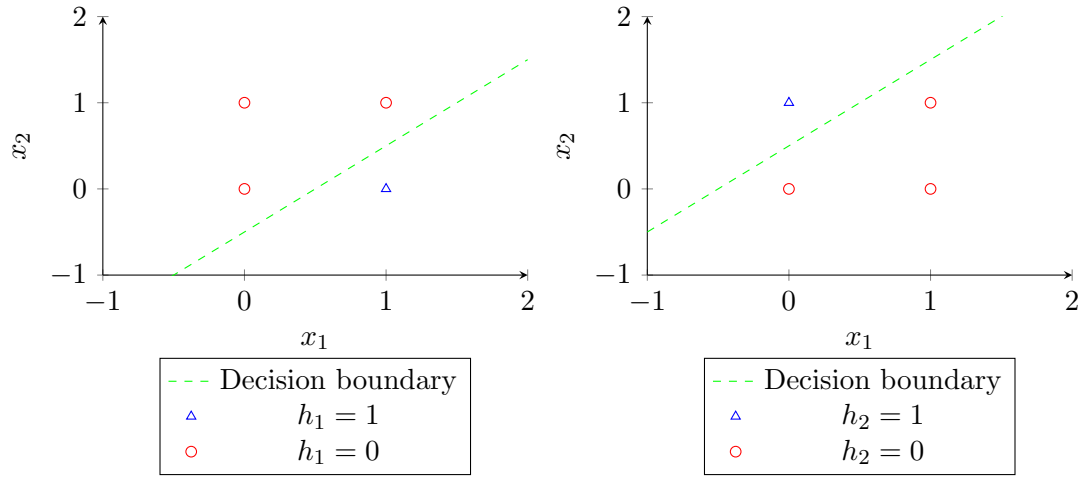


Figure 3.8: Linear separability of h_1 and h_2

- Instead of dealing with inputs (x_1, x_2) as they are we transform them into another representation (h_1, h_2) such that:
  - $h_1 = 1$ if $(x_1, x_2) = (1, 0)$, and equals 0 otherwise (linearly separable, see figure 3.8).
  - $h_2 = 1$ if $(x_1, x_2) = (0, 1)$, and equals 0 otherwise (linearly separable, see figure 3.8).

  both h_1 and h_2 are modeled using a single neuron each.

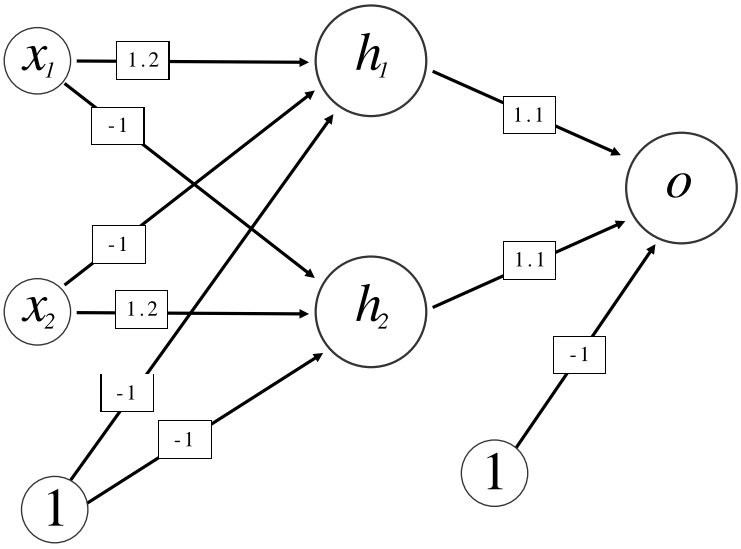


Figure 3.9: Neural network implementing logical XOR gate using an intermediate representation (h_1, h_2) of its inputs (x_1, x_2) to deal with non linear separability

– $XOR(x_1, x_2) = 1$ if one of h_1 or h_2 equals to 1 (of course they can't be both equal to 1 by definition), in other words we end up having

$$XOR(x_1, x_2) = OR(h_1, h_2)$$

and we've already seen in figure 3.7 that a logical OR gate satisfies linear separability, thus, it can be modeled using a neuron.

We end up having what we call a multi-layer neural network (See figure 3.9):

– The first layer consists of two neurons h_1 and h_2, and it performs representation transformation of the inputs.

– The second and final layer consists of just one neuron o (for output) that takes as inputs the outputs of h_1 and h_2, and it performs a linear classification on the new representation of the input

And this setup (when given the right parameters W and b for each neuron) is able to perform non linear classification.

3.1.4 Feed-forward Neural Network

Other names: *Multi-layer Perceptron (MLP), Vanilla Neural Network*

A feed-forward neural network (FFNN) is a multi-layer neural network that takes a vector of real values x and computes a forward computation through its layers [10]. It consists of:

– hidden layers (at least 1), usually used to perform representation transformation,

– a single output layer, that represents the output of the network, usually used to perform classification on the representation given by the last hidden layer.

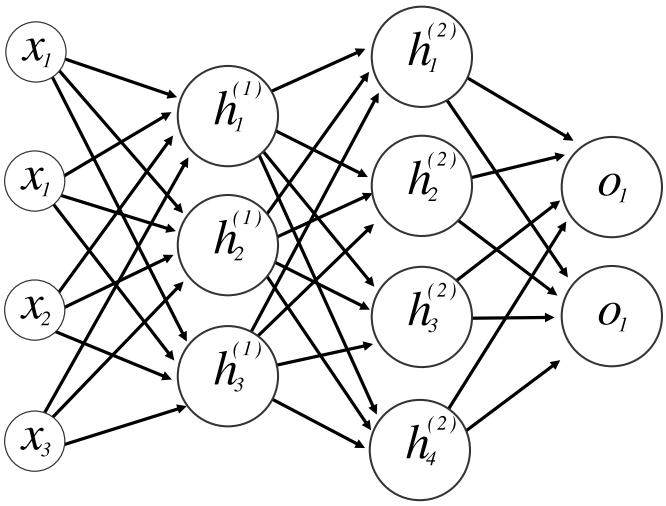


Figure 3.10: A simplified graphical visualization of the feed-forward neural network model. This FFNN takes 4 inputs, has 2 hidden layers $h^{(1)}$ and $h^{(2)}$ of sizes 3 and 4 respectively, and has an output layer of size 2. Biases were omitted only from this visualization. We can see how each two adjacent layers are fully connected to each other

A layer is a set of neurons that have the same activation function, the same number of inputs (thus, the same number of weights), but may have different values for their parameters (weights and bias).

- The first hidden layer, noted $h^{(1)}$,is the input layer, it takes as inputs the inputs of the network x: each neuron, noted $h_k^{(1)}$, of this layer takes as input all the inputs of the network x.
- Each hidden layer $h^{(i)}$ is fully connected to the next layer $h^{(i+1)}$:
  - the output (activation) of a neuron in the hidden layer $h^{(i)}$ is fed to all the neurons of the next layer $h^{(i+1)}$
  - the inputs of a neuron in a hidden layer $h^{(i>1)}$(except the first) are the outputs of the neurons in the previous layer $h^{(i-1)}$.
  - the inputs of a neuron in the output layer are the outputs of the final hidden layer, noted $h^{(L)}$.

Finally, the output of the network is the output of the final layer of neurons. It can be seen as a vector $y = [o_1, o_2, \ldots, o_m]$, with m being the size of the output layer. Note that each layer can have a number of neurons (or a size) that doesn't necessarily match the number of neurons in other layers.

Matrix-based approach to compute the output of a FFNN

The output of a single neuron u given an input vector x, a weight vector W, a bias b, and an activation function a can be computed this way:

$$u(x) = a(W \cdot x + b)$$

Now let's consider the output vector of a layer h of m neurons where:

- x is the input vector of this layer h
- h_i is the i-th neuron of this layer h
- W_i is the weight vector of the neuron h_i
- b_i is the bias of the neuron h_i
- a is the activation function of this layer h and thus for each neuron in this layer

$$\begin{aligned} h(x) &= [h_1(x), h_2(x), \ldots, h_m(x)] \\ &= [a(W_1 \cdot x + b_1), a(W_2 \cdot x + b_2), \ldots, a(W_m \cdot x + b_m)] \\ &= a([W_1 \cdot x + b_1, W_2 \cdot x + b_2, \ldots, W_m \cdot x + b_m]) \end{aligned}$$

We consider a to apply an element-wise activation to the element of its input vector:

$$a([x, y]) = [a(x), a(y)]$$

It turned out that this vector can be computed using a matrix-vector multiplication:

$$h(x) = a(W_h \cdot x + b_h)$$

Where:

- x is the input vector of the layer h
- W_h is an $m \times \dim(x)$ matrix that represents the weights of this layer such that:
  - the i-th row represents the weights of the i-th neuron,
  - thus, $W_{h_{ij}}$ represents the j-th weight of the i-th neuron.
- b_h is a vector of size m that represents the biases of this layer's neurons
- a applies an element-wise activation to the elements of the resulting m-sized vector.

Now we can consider a layer h of size m in the neural network a function, parameterized by a matrix of weights W_h and a vector of b_h, that takes in a vector of inputs and outputs a vector of size m which is the result of the activation of its neurons.

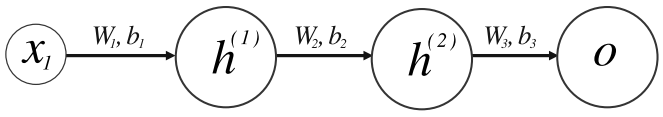


Figure 3.11: An even simpler way of looking at the neural network from Figure 3.10 after considering that each layer is a function that takes as input the output vector of the previous layer, applies a matrix-vector multiplication with weights, adds up the bias, and finally applies the activation function to the result yielding its output vector.

FFNN as a universal function approximator

A feed-forward neural network of L hidden layers is a composed function f that takes in a vector of inputs x, and returns a vector of outputs y:

$$y = f_\theta(x) = o(h^{(L)}(h^{(L-1)}(\dots(h^{(1)}(x))\dots)))$$

Where:

- $\theta = \{W_{h_1}, W_{h_2}, \dots, W_o\} \cup \{b_{h_1}, b_{h_2}, \dots, b_o\}$ the set of network parameters.
- o the output layer function, uses an activation function a_o
- h_i the i-th layer function, uses an activation function a_i.

It is proven that multi-layer feed-forward neural networks can approximate any function when given the right parameters θ.

3.2 Deep Learning

In the previous section, we discovered the power of the feed-forward neural network model: its multi-layer architecture allows it to perform several representation transformation of its input data, and the fact that, given the right architecture and parameters, it is theoretically capable of approximating any function. The question is: since we have large amounts of data, can we do machine learning and train these computational models to approximate complex functions such as classification? And that's the question deep learning answer: training deep neural network architectures to perform complex tasks such as recognizing faces, colorizing black and white images, classifying sounds, ...

Deep learning is a branche of machine learning that studies neural network architectures and learning algorithms. It is called deep because multi-layer neural networks (deep, more layers $\Rightarrow$ deeper) are able to learn more adequate representations of their input data in order to improve their performance, as opposed to other machine learning models that either operate on the input data directly or use hand-crafted feature extractor.

The word "deep" comes from the use of multi-layer neural networks. Each layer learns to transform its inputs to a representation that improves the performance of the next layer. As an emerging result, the first layers of a neural network tend to learn feature extraction, as opposed to traditional machine learning models that rely on hand-crafted feature extractors. There are empirical evidences that deep learning models outperform traditional machine learning ones in tasks involving complex or high-dimensional data such as images and sounds [13][14][15].

3.2.1 Training a Neural Network as an Optimization Problem

There exist several neural network models used in deep learning. Understanding the training process for these models comes down to understanding the training process for the most basic one of them, the feed-forward model. Also note that this applies to supervised learning, but later applications of neural networks to unsupervised learning problems use the same approach [16][17].

As we've seen before, a feed-forward neural network is a function f parameterized by the θ (the set of all weights and biases). The goal of training a neural network

is to find the right parameters θ for the function f_θ to approximate a function g. The function g is defined by a dataset $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ that represents the mapping this function applies to some inputs $y = g(x)$. In classification for example, g maps an input x to its category y.

Finding the right parameters θ for f_θ to approximate g can be done by minimizing the difference between f_θ and g. We can use a function that somehow reflects the difference between f_θ and g, and it will be our **cost function**, noted C. We end up having this minimization problem of C over θ:

$$\underset{\theta}{\operatorname{argmin}}\, C(\theta) = \frac{1}{n}\sum_{i=1}^{n} C^{(i)}(\theta)$$

Where:

- $\theta = \{W_1, W_2, \ldots, W_n\} \cup \{b_1, b_2, \ldots, b_n\}$: parameters of the neural network.
- n : size of our dataset.
- $C^{(i)}(\theta)$: the cost value for a given data point $(x^{(i)}, y^{(i)})$. This value should be equal to 0 when $f(x^{(i)}) = y^{(i)}$, and should get bigger as $f(x^{(i)})$ gets further from $y^{(i)} = g(x^{(i)})$.

3.2.2 Gradient Descent

To minimize the cost function C we will use an algorithm called gradient descent. Gradient descent is an iterative optimization algorithm that finds a local minimum of a differentiable function. This algorithms uses the information that the gradient gives us about the function at a given input: the direction of the steepest increase, but since we are looking to decrease the function's output, we follow the negative gradient at each step. See figure 3.12 for an intuition behind how the gradient descent work.

Here's how gradient descent works in our case:

- Randomly initializing the set of parameters

  $$\theta = \{W_1, W_2, \ldots, W_n\} \cup \{b_1, b_2, \ldots, b_n\}$$

- Computing the gradient of the cost function ∇C with respect to every element in θ (remember θ consists of the weight matrices and bias vectors of the neural network) by:

  averaging over the gradient of the cost function $\nabla C^{(i)}$ for every datapoint (x_i, y_i) in our dataset D.

  Note: to compute the gradient of the cost function $\nabla C^{(i)}$ with respect to every element in θ we use an algorithm called back-propagation.

- Updating parameters by taking a step proportional to the negative gradient of the cost function $-\eta\nabla C$

  η is called the learning rate, by increasing it we take bigger steps towards to local minimum, but we may miss the local minimum by taking huge steps.

- Repeating until a chosen stopping criteria is satisfied.

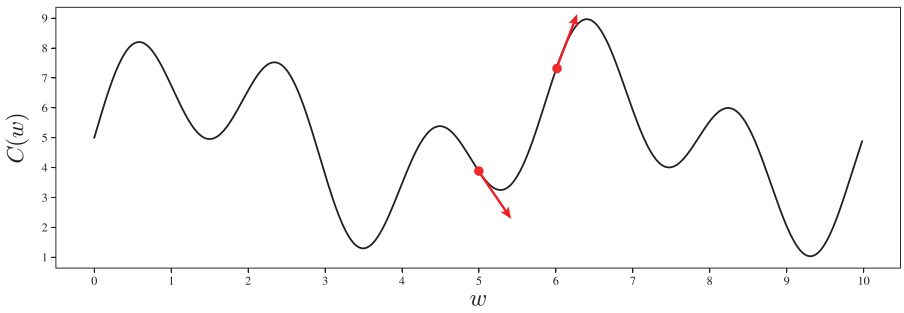


Figure 3.12: An intuitive example of gradient descent for minimizing a differentiable function with just one real input w. Gradient descent starts from a random input, computes the gradient at that input (in this case, we compute the derivative), we take a step in the input space proportional to the negative gradient: as you you can see for $w = 5$ the gradient is negative (the function is decreasing), thus taking a step proportional to the negative gradient will take us rightward on the input line and downward on the function landscape, for $w = 6$ we have a positive derivative, thus we take a negative step leftward on the input line and downward on the function landscape. This idea is generalized for multi-variable functions.

In batch gradient descent, we pass over the whole dataset to compute the gradient in order to take a step towards the local minimum of our cost function. Passing over the entire dataset can be time-consuming, and for large datasets, we can't even load the entire dataset all at once on the memory. Other variants of gradient descent were proposed to deal with this problem:

- Stochastic gradient descent: instead of passing over the whole dataset to compute the gradient, we pick a random example from the dataset to compute a stochastic approximation of the true gradient.
- Mini-batch gradient descent: we pass over a random mini-batch of data to compute a stochastic approximation of the true gradient. Mini-batch gradient descent can make use of highly optimized matrix operations libraries that can compute the gradient for mini-batches of data efficiently. *Mini-batch gradient descent is sometimes called stochastic gradient descent.*

3.2.3 Feed-forward neural networks for classification

Classification is probably the most popular problem that has been successfully tackled using machine learning, and later on deep learning. Of course, we can train feed-forward neural networks for classification.

A classification problem consists in finding a mapping from an input data x to its class y, where y is a distinct category. There are 2 usual scenarios to this problem:

Binary classification

Where y can only be 1 of 2 possibilities. For example, if x is an image and y can either be 1 = `CAR` or 0 = `NOT-A-CAR`. Since feed-forward neural networks take real vectors as

inputs, we can reshape the image matrix x to a 1-dimensional vector in order to be used as inputs to the neural network. y in this case is a single value 1 or 0, our network then should have a single neuron in its output layer that uses the sigmoid function σ which outputs a value between 0 and 1. We interpret the output of the network as the probability of the input x being a `CAR`.

Multi-class classification

Where y can a value among a finite set of $k > 2$ possible values. For example, if x is a audio spectrogram and y can be one among {`RAIN`, `WIND`, `RIVER`}. Since feed-forward neural networks take real vectors as inputs, we can reshape the spectrogram matrix x to a 1-dimensional vector in order to be used as inputs to the neural network. y in this case should be represented by a vector that encodes the category. The most popular, and useful, encoding is the One-Hot encoding: if we have k possible categories, we associate to each category c_i a vector of size k that contains 1 in its i-th component and 0 everywhere else. For example:

$$\texttt{RAIN} = 1 \longmapsto \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$\texttt{WIND} = 2 \longmapsto \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$\texttt{RIVER} = 3 \longmapsto \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, the output layer will contain three sigmoid neurons. We interpret the output of the network as the respective probability for each class, we then take the one with the highest probability. For example:

$$\begin{bmatrix} 0.8 \\ 0.6 \\ 0.1 \end{bmatrix} \longmapsto 1 = \texttt{RAIN}$$

Although this worked for years, a popular alternative came to replace it. First, notice that the sum of the output values can exceed 1, but for mutually exclusive classes that doesn't make sense as the sum of their probabilities should be equal to 1. For this matter, another activation function should be used instead to take into account such information. It is called the **softmax** activation function [18], and it is only used for the output layer. For k output neurons $o_1, o_2, \ldots, o_k$:

$$o_i = \frac{e^{p_i}}{\sum_{j=1}^{k} e^{p_j}}$$

Where p_i is the pre-activation (defined in 3.1.1) of the i-th output neuron. Notice the use of exponentials instead of summing over pre-activations as they are, that's because pre-activations can be negative. Therefore, softmax activation function guarantees that the sum is equal to one, and it delivers extra information to our neural network.

3.2.4 Convolutional Neural Network

Also called: *ConvNet, CNN*

The convolutional neural network [19] is a deep learning model that revolutionized the field of computer vision [3]. After getting much attention from the research community, CNN was applied to other problems such as natural language processing [20, 21]. The neural connectivity patterns in CNNs is biologically inspired from the animal visual cortex [22].

Multi-Dimensional layers

As opposed to the feed forward neural network, which operates only on vectors as input data, the convolutional neural network may operate on 2-dimensional input data (matrices) such as grayscale images or audio spectrograms. For other kinds of data, like RGB images, that have multiple channels, we can consider either having 3-dimensional input or multiple channels of 2-dimensional input.

Local connectivity

In the traditional feed forward neural network, every neuron is fully connected to the previous layer. In other words, every neuron in the layer L receives inputs from all the neurons in the previous layer $L-1$. In CNNs, we meet the concept of local connectivity, where a neuron receives inputs from a sub-region of neurons from the previous layers. This may also applies to neurons in the first hidden layer, thus, each neuron takes as input a sub-region of the input data. See Figure 3.13.

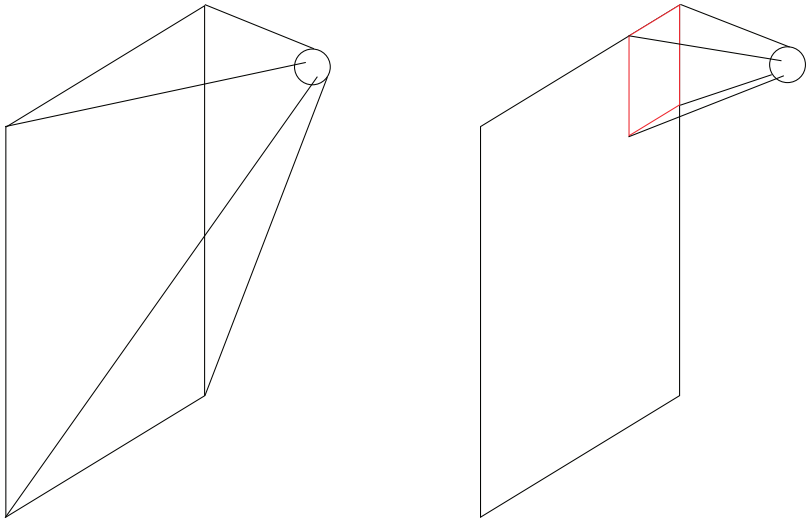
If a locally connected neuron receives multi-channeled inputs (such as an RGB image, with 3 channels), it will be connected to all the channels of the subregion. So if we take the subregion size as $r \times r$ and the input data has c channels, the neuron will therefore take inputs of size $r \times r \times c$. Taking inputs of size $r \times r \times c$ implies having $r \times r \times c$ weights. The output of such neuron is computed the same way as the traditional neuron: dot product between the weights and the inputs, adding the results to a bias parameter b, passing the result through an activation function.

Convolutional layer

Now let's talk about the convolution operation. A locally connected neuron gets input from a sub-region of the input tensor[1]. For the sake of clarity, we will use the term input matrix, even though it may be multi-channeled. The convolution operation consists in computing the activation of a single locally connected neuron for all the sub-regions that makes out the input matrix (sub-regions are usually overlapped). We, therefore, obtain a matrix of activation with each one corresponding to a sub-region in the input matrix. **Remember:** we slide the same neuron over all sub-regions to get this matrix. See Figure 3.14.

During a successful training process, a single locally connected neuron learns to detect a certain feature (intuitive example: horizontal dark edge, diagonal bright edge, ...). Convolving such kind of neuron over the input matrix gives as output a matrix

[1] In mathematics, a vector is an ordered 1-dimensional set of numbers, a matrix is 2-dimensional, a tensor is the generalization of these structured set of numbers to higher dimensions.



(a) fully connected neuron (b) locally connected neuron

Figure 3.13: Full connectivity vs. local connectivity of a neuron over 2-dimensional inputs

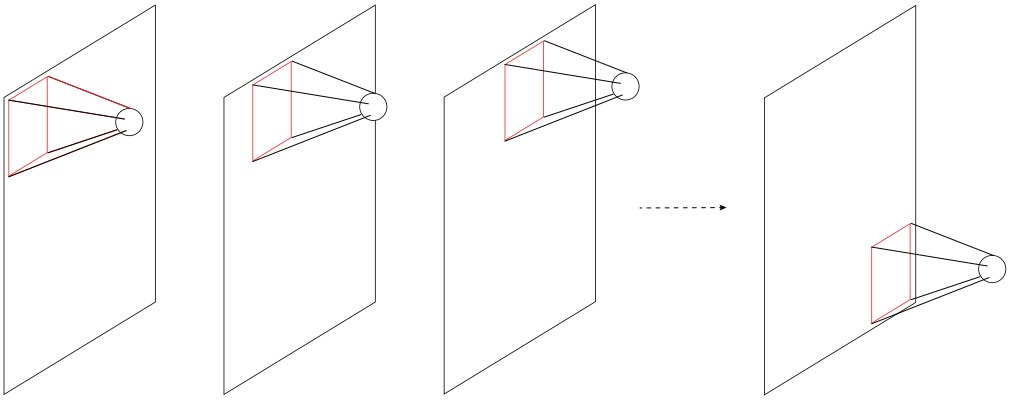


Figure 3.14: 2-dimensional convolution

that indicates the presence and the absence of the feature in the different sub-regions on the input matrix, this matrix is called the feature map.

A convolutional layer, which is one of the main building blocks of a CNN, consists of multiple neurons, where each one of these neurons convolves over the inputs yielding a feature map. The final output of a convolutional layer is multi-channeled matrix where each channel represents a feature map.

More formally, a convolutional layer is a function $\mathbf{conv}_{W,b}(x)$ parameterized by weights W and biases b:

- x is the input tensor, of size $N \times M$ if the the inputs aren't multi-channeled, of size $N \times M \times c$ with c being the number of channels.
- $r \times r'$ is the sub-regions size ($r \times r' \times c$ in the case multiple channels)
- The number of neurons n, which is also the number of output channels (feature maps).

– Weight parameters tensor W is of size $n \times r \times r' \times c$.

– Bias parameters vector b of size n.

– Usually, we keep sliding the regions one step at a time, resulting in highly overlapped sub-regions. In such case, the output tensor of this function (or the the multi-channeled matrix) is of size $(N - r + 1) \times (M - r' + 1) \times n$

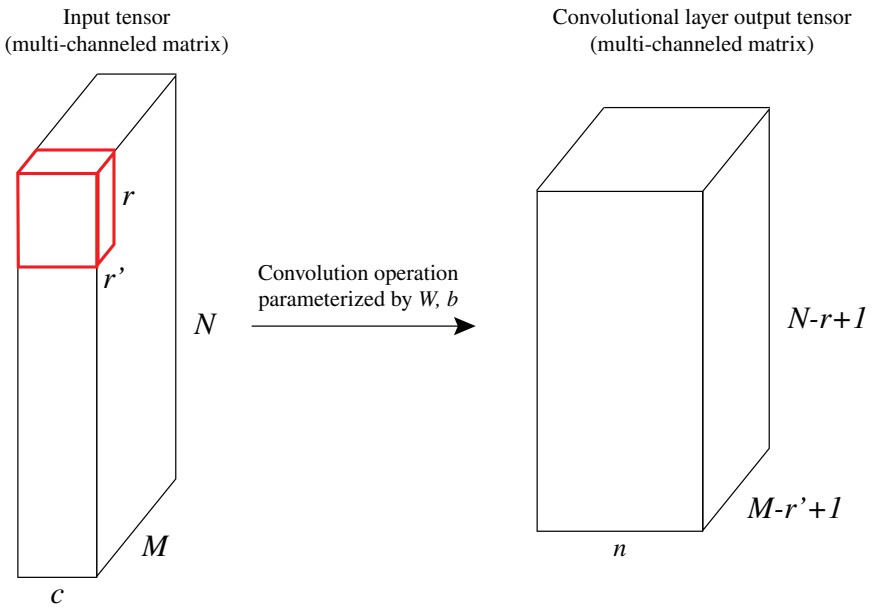


Figure 3.15: Convolutional layer

Convolutional layers are capable of learning feature detection, and it was empirically proven to be more successful than classical hand-crafted feature extraction [4][10][11]. The features detected at a given convolutional layer are of higher abstraction than these present at its input layer. For e.g, if we have pixel as inputs, a convolutional layer may learn how to detect edges, or if we have edges as inputs, a convolutional layer may learn how to detect objects or shapes.

Pooling layer

Pooling consists in aggregating values of non-overlapping subregions resulting in reducing the size of the representation (and consequently the number of parameters in the following layers). Pooling is applied to a multi-channeled matrix, like the ones convolutional layers take as inputs and give as outputs.

The most used aggregation function for pooling is the maximum function, and we call this max-pooling.

Given a multi-channeled matrix of size $N \times M \times c$ as input, $r \times r$ max-pooling is computed as the following:

– For each channel:

  · Compute the max value for each $r \times r$ sub-region (no overlap)

  · The result would be a matrix of size $\frac{N}{r} \times \frac{M}{r}$

  · See Figure 3.16

– The final result is a multi-channeled matrix of size $\frac{N}{r} \times \frac{M}{r} \times c$. See figure 3.17.

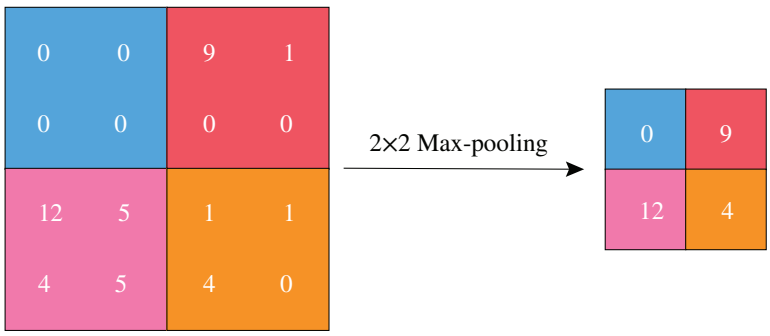


Figure 3.16: 2×2 Max-pooling on a single channel

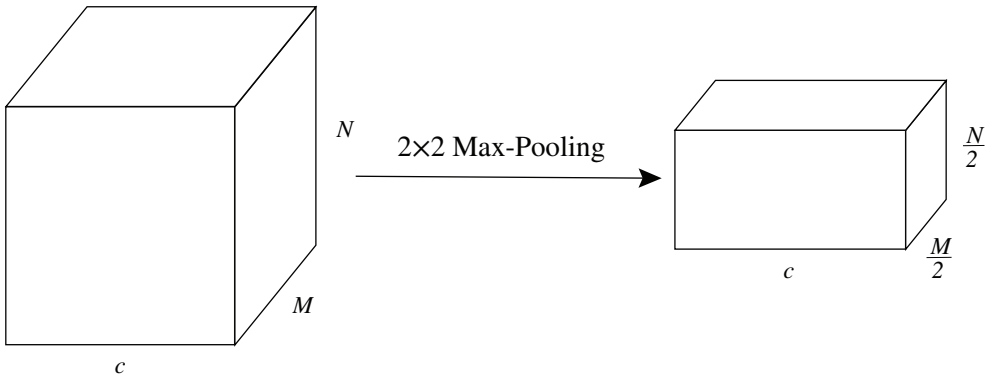


Figure 3.17: 2×2 Max-pooling on a multi-channeled input

Architecture of a CNN

Let's recap what we know so far about the new kinds of layer we saw:

– Convolutional layer:

· Capable of learning features detection.

· Uses the rectified linear function (ReLU) as activation function.

· Outputs a multi-channeled output, where each channel represents a feature map that indicates the presence and the absence of a certain feature spatially.

– Pooling layer:

· Reduces the size of its multi-channeled input.

· Introduces invariance to local translation. In other words, if a feature is slightly translated to a neighboring position, this slight change won't matter in the next layers.

A typical CNN architecture is better explained using the following regular expression:

$$\textbf{Input} \Rightarrow (\textbf{Conv} \Rightarrow \textbf{MaxPooling})+ \Rightarrow \textbf{FC}+ \Rightarrow \textbf{Output}$$

Where:

- **Input** represents the input to our network, usually a matrix (or a multi-channeled matrix).
- **Conv** represents a convolutional layer.
- **MaxPooling** represents a pooling operation.
- (**Conv** $\Rightarrow$ **MaxPooling**)+ means a sequence of convolutional layers with each one being followed by a pooling operation.
- **FC** stands for "fully connected", represents a fully connected layer of neurons. The + sign again means a sequence of fully connected layers. Generally, it's just one fully connected layer that learns classification over the highest level features learned by the previous convolutional layers.

Why CNN?

For the problem we are tackling, environmental sounds recognition, we decided to use a convolutional neural network model because of its robust feature learning and its recent success in classification tasks [3, 20, 21, 22]. Please note that convolutional neural network was picked instead of a temporal model (like RNN, LSTM) due to the fact that we are mainly focusing on correctly classifying audio segments, while temporal models would be potentially a better option if we were to tackle a sound event detection problem (temporally localizing, predicting, and recognizing sound events).

Chapter 4

Convolutional Neural Network for Environmental Sounds Recognition

4.1 Related work

Early environmental sounds recognition research focused primarily on engineering features [1]. Most pattern recognition fields went through the same paradigm before deep learning took over. Researchers and engineers had to come up with signal processing algorithms to extract features from audio signals . Popular hand-crafted features, some of them being borrowed from the more mature field of speech recognition, include:

- Zero-Crossing Rate (ZCR)
- Mel-Frequency Cepstral Coefficients (MFCC)
- Discrete Wavelet Transform (DWT)
- Continuous Wavelet Transform (CWT)

These hand-crafted features serve therefore as inputs to machine learning models such as:

- k-Nearest Neihbors (kNN) + Dynamic Time Warping (DWT)
- Support Vector Machines (SVM)
- Hidden Markov Models (HMM)
- Gaussian Mixture Models (CWT)
- Artificial Neural Networks (ANN)

Sigtia et al. [23] suggest that there is a trade-off between performance and cost in solving environmental sounds recognition. In other words, classification models that requires more computational resources tend to achieve higher accuracies. This raises concerns over the embeddability of accurate classifications models in small devices for time/energy-critical applications.

As the interest is growing in this relatively new field, datasets started to show up for benchmarking and experimentation purposes [24][25]. Also, it is worth noting

that the field is gaining giant corporates interest after Google Inc. released a dataset, called Audio Set, of over 600 category of indoor and outdoor sounds containing over 2 millions labeled example [26].

In recent years, people started taking the deep learning approach to tackle this problem. While some of the early deep learning approaches to the problem didn't go as far as getting rid of the hand-crafted feature extraction mechanisms [23][27][28] (which is what deep learning is all about), there has been other pieces of work that explored the possibility of feeding audio spectrograms as inputs to the neural network [29][30], and some others went even further by proposing neural network architectures that operate directly on the raw wave-form of audio signals [31][32].

4.2 Our proposals

In this work we propose two techniques to improve the performance of the convolutional neural network for environmental sounds classification. The first is a new method we propose to compensate for the lack of training data called stochastic continuous data augmentation, and the second is the use of transfer learning [33] to transfer the knowledge learned from a dataset to use it with another one.

4.2.1 Convolutional Neural Network Architecture

To tackle this classification problem, we propose a deep neural network model, precisely, a convolutional neural network. The architecture of this model is better explained through Figure 4.1:

- (a) represents the inputs to the network, for each 2 seconds audio signal we have, we take the mel-spectrogram of this audio signal (window size = 128ms = 2048 sample, overlap of 3 quarters). This is a mono-channeled matrix of size 59×128.
- (b) is a convolutional layer of region size = 4×4 and it learns to extract 32 low level features. It outputs a multi-channeled matrix of size $56 \times 125 \times 32$.
- (c) is a 2D-MaxPooling layer that down-samples the output of the previous convolutional layer by 3×2. It outputs a multi-channeled matrix of size $18 \times 62 \times 32$.
- (d) and (e) are another convolutional layer followed by a 2D-MaxPooling layer. The former learns to extract 48 higher level features. They share the same regions size with (b) and (c). The output of (e) is a multi-channeled matrix of size $5 \times 28 \times 48$.
- (f) is a convolutional layer that operates over considerably high level features to learn the extraction of even higher level features to be used for classification. It uses a region size = 2×2. It outputs a multi-channeled matrix of size $4 \times 28 \times 64$. This matrix is later flattened to a vector of size 7168 to be used with standard fully connected layers.
- (g) and (h) are fully connected layers. (g) learns a compressed representation of the 7168 features delivered by the last convolutional layer, while (h) learns classification and it uses a softmax activation function. For the ESC-50 dataset, (h) outputs the 50 respective probability for each class, and for the UrbanSounds8K data, the 10 respective probability for each class.

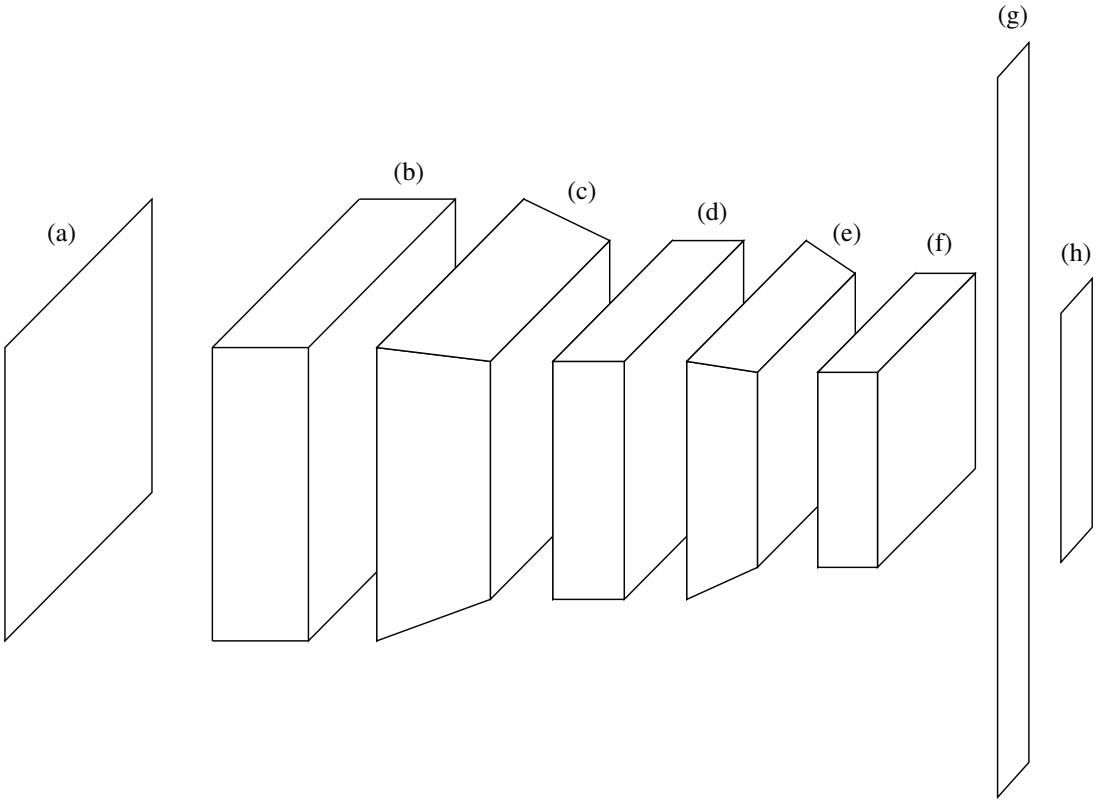


Figure 4.1: Convolutional neural network (CNN) architecture for environmental sounds classification

The model has about 700,000 learnable parameters.

As we mentioned earlier in this report, training a neural network is an optimization problem: we find the parameters (weights and biases) that minimize a certain cost function C.

Cost function

The cost function used in this work is the cross-entropy function and is defined as follows:

$$\begin{aligned} \underset{\theta}{\operatorname{argmin}}\, C(\theta) &= \frac{1}{n}\sum_{i=1}^{n} C^{(i)}(\theta) \\ C^{(i)}(\theta) &= -\sum_{j}[y_j^{(i)}\log(N(x^{(i)})_j)) + (1-y_j^{(i)})\log(1-N(x^{(i)})_j)] \end{aligned}$$

Where:

– n is the number of examples to compute the cost function over.

– θ is the set of all parameters.

– $C^{(i)}(\theta)$: the cost value for a given example $(x^{(i)}, y^{(i)})$.

– $N(x^{(i)})$ is the output of the neural network for the input example $x^{(i)}$.

– j subscript expresses the j-th element of a vector.

This cost function is used when training neural network for to perform classification.

Gradient descent algorithm

The optimization algorithm used to minimize the cross-entropy cost function is a variant of the stochastic gradient descent algorithm called **Adam**[34]. While the details of how Adam algorithms work are beyond the scope of this work, but it's worth noting that when compared to traditional stochastic gradient descent, Adam uses an adaptive learning rate (noted η in 3.2.2) for each trainable parameter. Adam is empirically proven to perform better than other variants of stochastic gradient descent[34].

Drop-out regularization

Drop-out is a regularization technique used with neural networks to fight overfitting [35]. During training, drop-out consists in shutting off a random proportion of the neurons. For example, in this work, we applied a drop-out with a probability of 0.5 (in the last 2 layers of the neural network), in other words, at each training iteration, we shut off half of the neurons (by setting their activation to 0).

Randomly shutting off half of the neurons at each training iteration helps preventing the neural network from memorizing the training data (and thus, overfit to the data), and forces the network to learn more general ways to deal with the data. It can be also seen as training simultaneously a set of smaller neural networks.

4.2.2 Data augmentation

To improve the performance of the convolutional neural network, we propose a method called stochastic continuous data augmentation, but before diving into that, we first must take a look at what data augmentation is.

Data augmentation is the task of increasing the size of a dataset without actually collecting new data but instead by introducing slight modification to the data we already have. Data augmentation has been shown to improve the performance of learning models. The concept of data augmentation is better explained visually, see Figure 4.3.

The two kinds of modification to be used on audio signals in this work are time-stretch and pitch shift.

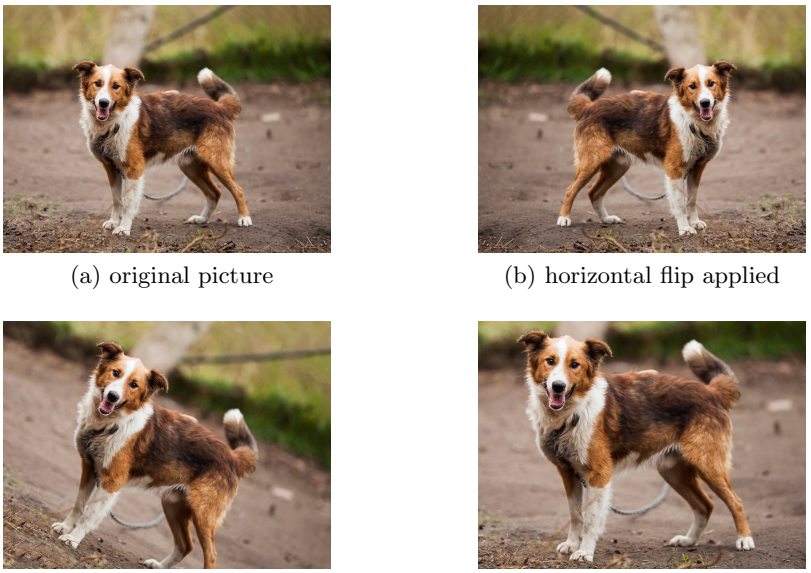
Time-stretch

Time-stretching an audio signal is slowing down or speeding up the audio signal while keeping pitch unchanged. Time-stretching an audio signal of t seconds by a factor f results in:

– the same audio signal if $t = 1$,

– a slower version of the audio signal if $t > 1$ of length $f \times t$ seconds,

– a faster version of the audio signal if $t < 1$ of length $f \times t$ seconds.



(a) original picture (b) horizontal flip applied

(c) clock-wise rotation applied (d) zoom-in applied

Figure 4.2: slight modifications (respectively: horizontal flip, rotation, and zoom) helped us synthesize new pictures based on the original one. Applying the modifications shown in this example for every image in a dataset would result in quadrupling the size of the dataset. Given they don't change the nature of the picture, in this case a dog, they can only help the learning algorithm generalize better.

Pitch-shift

Pitch-shifting an audio signal is increasing or decreasing the pitch while keeping the time unchanged. An intuitive example would be: if we have a 1 second audio signal that contains only the musical note A_4 (fr: La_4, frequency: 440Hz) played on a piano, pitch-shifting it by $+2$ semitones would result in a 1 second audio signal that contains the musical note B_4 (fr: Si_4, frequency: 493Hz), pitch-shifting it by -2 semitones would result in a 1 second audio signal that contains the musical note G_4 (fr: Sol_4, frequency: 392Hz), without affecting the duration of the note.

Note: although time and frequency (thus, pitch too) are directly related to each other, both time-stretching and pitch-shifting effects use specialized algorithms so they only affect time or pitch respectively.

Stochastic continuous data augmentation

In this work, we propose a data augmentation strategy where we continuously modify, using random parameters, examples from the dataset. Given that training a neural network is an iterative process, instead of reusing the same dataset for all iterations, we introduce random modifications to the dataset for every iteration. This way our neural networks gets the chance to explore continuously slight variations to every example we have on our dataset.

To get a better intuition for how this strategy work, we will take a look at the following 2 training algorithms:

– D is the dataset

– $t_f(x)$ is x time-stretched by the factor f.

– $p_s(x)$ is x pitch-shifted by s semitones.

On the following algorithms, "train and validate N on D" means applying gradient descent on neural network N by passing once over the whole dataset D then estimating the performance on a validation set to track the training progress.

Algorithm 1: training without data augmentation

```
initialize model N
for a number of iterations do
    train and validate N on D
end
```

Note that on Algorithm 2, for each iteration of training, we apply random changes to every example on the dataset, so the neural network barely sees the exact same example twice.

Algorithm 2: training with stochastic continuous augmentation

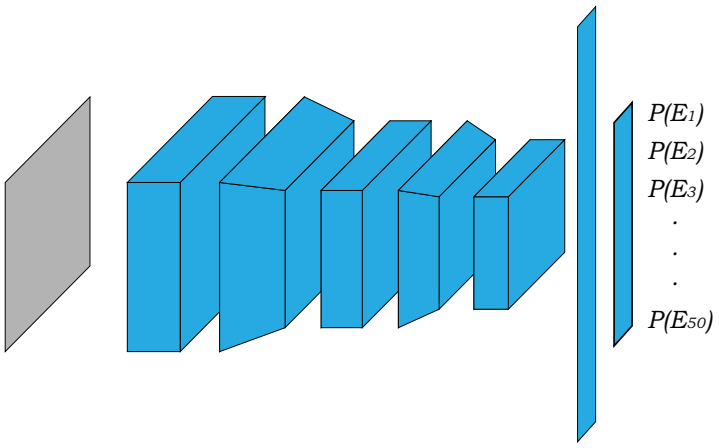
```
initialize model N
for a number of iterations do
    D' ← ∅
    for each example x in D do
        f ← sample from U(0.8, 1.2)
        s ← sample from U(−2, +2)
        x' ← p_s(t_f(x))
        D' ← D' ∪ {x'}
    end
    train and validate N on D'
end
```

4.2.3 Transfer Learning

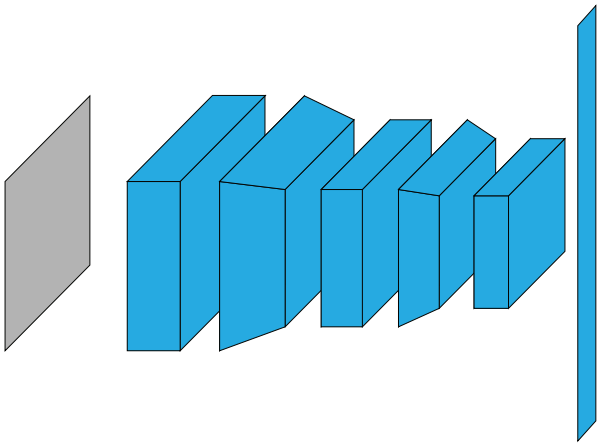
Convolutional neural network learn features extraction in their first layers (convolutional + pooling), and it turned out that these first layers tend not to extract features that are specific only to the dataset they learned from[33]. For e.g, in images, they learn features such as edges and color blobs[33]. Transfer learning consists in transferring the knowledge learned by a neural network from a particular dataset to train it on a different dataset. But how does this exactly work? This is better explained by how transfer learning was used in this work:

– We trained a neural network to classify data into 50 environmental sounds classes using the ESC-50 dataset (stochastic continuous data augmentation was used).

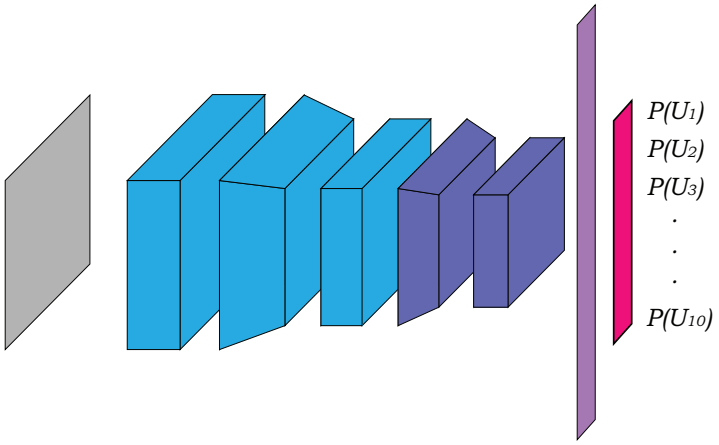
– Then, instead of re-training from scratch another neural network that classifies data into 10 classes using the UrbanSounds8K (no augmentation used), we kept the knowledge (feature extraction) learned from the previous experiment and we only reset the final layer of the neural network to classify into 10 classes. So, we kept the whole previous network and only substituted the classification output layer with a new, randomly initialized, one. The knowledge learned from the ESC-50 dataset is transferred to perform on another dataset.



(a) CNN trained on the ESC-50 dataset. $P(E_i)$ is the probability of the i-th class in the ESC-50 dataset.



(b) The layers from the trained ESC-50 CNN to be used with UrbanSound8K dataset.



(c) CNN trained on the UrbanSound8K dataset using layers trained on ESC-50. $P(U_i)$ is the probability of the i-th class in the UrbanSound8K dataset. Notice after introducing a new (randomly initialized) final layer for classiciation, training the network then would barely influence early layers as they're already capable of detecting environmental sounds features, meanwhile, later layers would be fine-tuned to the new dataset.

Figure 4.3: Transferring the knowledge (feature detection) learned from the ESC-50 dataset to use it with the UrbanSound8K dataset

Chapter 5

Experiments and Results

5.1 Datasets used and pre-processing

5.1.1 ESC-50 dataset

The ESC-50 (ESC = Environmental Sounds Classification) dataset [24] is a dataset of 2000 5-seconds audio recordings. Audio recordings are organized into 50 classes of environmental sounds (meaning 40 example for each class). The 50 classes can be loosely separated into 5 superclasses as illustrated in Table 5.1 as illustrated on its Github repository[1].

Given its limited size, this dataset is only suitable for benchmarking purposes.

Animals	Natural soundscapes & water sounds	Human, non-speech sounds	Interior/domestic sounds 2.5cm	Exterior/urban noises
Dog	Rain	Crying baby	Door knock	Helicopter
Rooster	Sea waves	Sneezing	Mouse click	Chainsaw
Pig	Crackling fire	Clapping	Keyboard typing	Siren
Cow	Crickets	Breathing	Door, wood creaks	Car horn
Frog	Chirping birds	Coughing	Can opening	Engine
Cat	Water drops	Footsteps	Washing machine	Train
Hen	Wind	Laughing	Vacuum cleaner	Church bells
Insects (flying)	Pouring water	Brushing teeth	Clock alarm	Airplane
Sheep	Toilet flush	Snoring	Clock tick	Fireworks
Crow	Thunderstorm	Drinking, sipping	Glass breaking	Hand saw

Table 5.1: ESC-50 dataset classes arranged into 5 superclasses

5.1.2 UrbanSounds8K dataset

The UrbanSound8K dataset [25] is a dataset 8732 audio recordings ($\leq$ 4 seconds) of urban sounds (outdoor). Audio recordings are drawn from the following 10 classes:

- Air conditioner
- Car horn
- Children playing
- Dog bark
- Drilling
- Engine idling
- Gunshot
- Jackhammer
- Siren
- Street music

[1]ESC-50 on Github : github.com/karoldvl/ESC-50

5.1.3 Pre-processing

For both dataset, we applied the same pre-processing:

- Resample the audio signals to 16000Hz.
- Cut them into overlapping segments of 2 seconds while discarding silent segments.

So we end up having two datasets of uniform audio signals (2 seconds sampled at 16000Hz), this increased considerably the sizes of our dataset:

- the ESC-50 dataset from 2000 to 5134 examples,
- the UrbanSounds8K dataset from 8732 to 14838 examples.

An example of why was it useful to do this segmentation: we have audio clips that contain more than one dog bark sound, let's say 3 dog barks, by considering 2s segments and discarding silent segments we end up isolating each one of these barks as a single example on its own. Note that we must keep segments originating from the same audio clip together either in the training set or the test set, because if we don't we may have the overlapped parts in both sets which makes it easy for our classifier.

5.2 Experimental setup

5.2.1 Pre-processing

Pre-processing data from datasets consists in two operations: resampling to 16KHz and segmenting the audio recordings into 2-seconds-long segments while discarding silent segments. Libraries used to perform such tasks are:

- **Librosa** [36] was used for audio files I/O and for the resampling as well.
- **NumPy** [37] (numerical Python), offering a set of mathematical and numerical features in a MATLAB-like fashion, was used to cut the audio signals into segments of 2s as well as detecting the silence using a thresholding strategy.

As mentioned before, we use mel-spectrograms representations as inputs to our neural network. Generating mel-spectrogram representations for the audio signals was also achieved through the use of the previous libraries, given NumPy offers a FFT implementation, and Librosa offers a Hertz-to-Mels function.

5.2.2 Data augmentation

The two effects applied on audio signals for data augmentation were time-stretch and pitch-shifting. We used the **PySndFx**[2] (Python Sounds Effects) library which is a wrapper of the command line utility **SOX**[3] (SOund eXchange) that offers a wide range of audio conversion features as well the effects used in this work.

The stochastic continuous data augmentation strategy applied consists in running every audio signal through the both effects (with random parameters within a safe interval) before feeding it into the neural network.

[2]PySndFx on Github: github.com/carlthome/python-audio-effects
[3]SOX official website : sox.sourceforge.net

5.2.3 Neural network implementation

Writing a neural network implementation from scratch is time-consuming, given you have to account for computational efficiency. Designing a deep learning solution requires a lot of experimentation and prototyping. Given the nature of this work, we chose to use the deep learning framework **Keras** [38] for the following reasons:

- Modularity: it offers a wide range of layers types, activation functions, performance metrics, ... through a modular Python API.
- Efficiency and support: Keras is optimized and continuously maintained to work on both CPUs and GPUs.

Having such features makes the perfect environment for experimenting with the different possibilities and parameters. It also helped with saving the models into files, and re-using trained layers easily into a new model.

5.2.4 Technical details

CPU	Intel Core i5 2.4GHz
Memory	8Gb 1600MHz DDR3
Operating system	Mac OS X, version = 10.12.2
Programming language	Python, version = 3.6.4
Libraries and frameworks	NumPy, version = 1.14 Librosa, version = 0.5.1 PySndFx, version = 0.2.0 Keras, version = 2.1.3 with Tensorflow 1.4 backend

5.3 Results

Since we are working with class-balanced datasets (in other words, there isn't a class that has remarkably more examples than the other class) to solve a classification problem, the metric we're gonna use to compare models is the classification accuracy (or simple accuracy for short), it is defined as follows:

$$\textbf{accuracy} = \frac{\text{number of correctly classified examples}}{\text{number of all classified examples}}$$

So if we want to compute the accuracy of a model on a dataset of 100 examples that we already know their respective classes, and the model classifies 60 of them correctly, we divide 60 by 100 to get 60% as our model's accuracy on this particular dataset.

To estimate accuracy using k-fold cross validation we take the average over the accuracies we get from the k experiments.

5.3.1 On the ESC-50 dataset

As mentioned earlier, this dataset contains only 2000 examples (40 example for each class) which makes it a small dataset. Small datasets are known to be prone to overfitting, since there won't be much data for the learning algorithm to generalize to unseen example.

We will call the first setup without stochastic continuous augmentation ESCNN. Figure 5.1 shows how accuracy on the validation set fails to catch up with that on

the training set, which is a mild form of overfitting since. In this particular fold, the model managed to reach $\sim$ 65% accuracy. Upon evaluating the model using 5-fold cross validation, we get an estimated average accuracy of 64.1%.

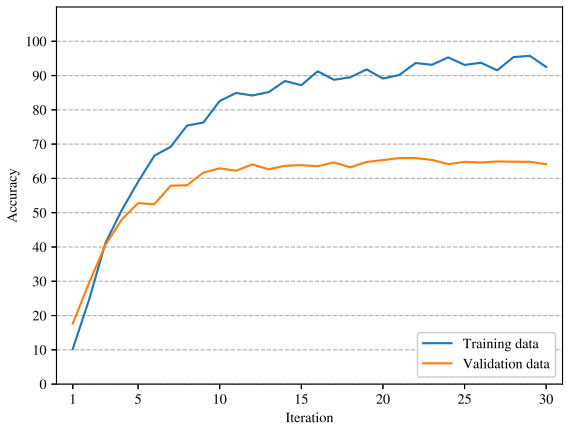


Figure 5.1: Accuracy history during training on ESC-50 without stochastic continuous augmentation

The other setup, ESCNN+, was the one where we employed stochastic continuous augmentation, where instead of reusing each example during the whole training, we continuously introduce random slight modifications to help our neural network gain invariance towards small changes that may exist outside. While this approach increases the training time due to the processing required at each step, it also requires more training iterations to bring value with regard to the accuracy of the model on unseen data.

In Figure 5.2, we can see how slow the ESCNN+ was to reach the 65% region, but later managed to escape that region, as opposed to ESCNN, and reach up to 70% accuracy on unseen data.

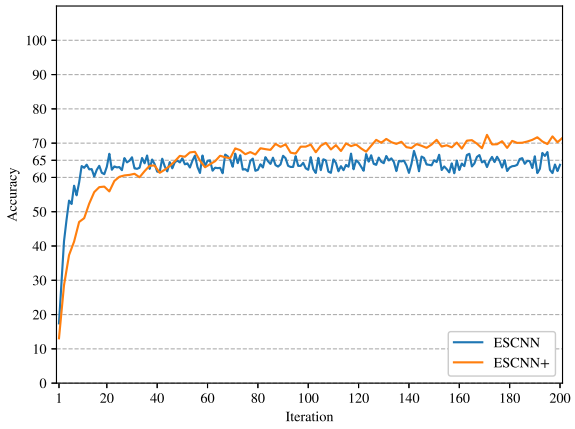
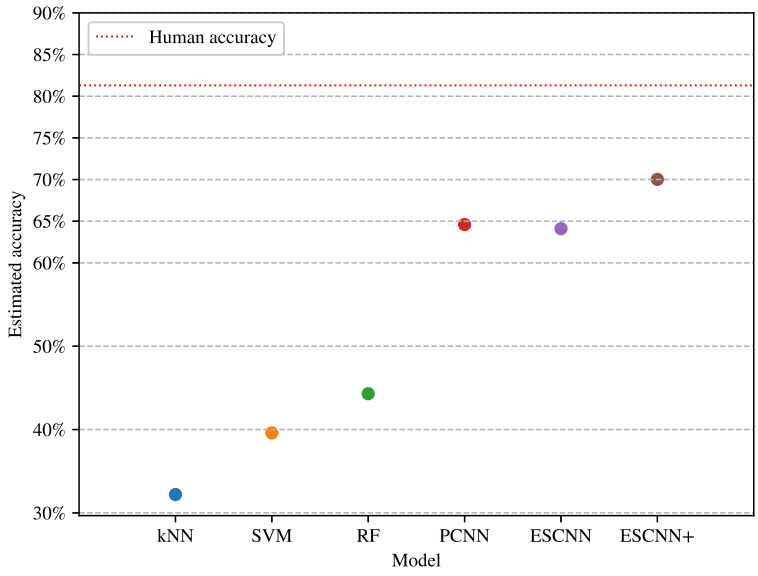


Figure 5.2: Validation accuracy history during training with (ESCNN+) vs. without (ESCNN) stochastic continuous augmentation

The author of the ESC-50 dataset [24] contributed, along with dataset, with 3

simple baseline models (kNN: k-Nearest Neighbor, SVM: Support Vector Machines, and RF: Random Forest) that, unlike deep neural networks, use hand-crafted features as inputs (zero-crossing rate and MFCC) and conducted a crowdsourced experiment on human subjects to estimate the human accuracy on this dataset. Piczak also proposed a CNN model (with simple data augmentation) that outperformed baseline models (with manually engineered features) [30], it is referred to as PCNN in Figure 5.3.



Model	Human	kNN	SVM	RF	PCNN	ESCNN	ESCNN+
Accuracy	81.3%	32.2%	39.6%	44.3%	64.6%	64.1%	70.02%

Figure 5.3: Results summary on the ESC-50 dataset

5.3.2 On the UrbanSound8K dataset

With this dataset, we experimented with 2 different setups:

- U8K-CNN: the previously described CNN model without transfer learning. In other words, trained from a random initialization.
- U8K$^{\mathrm{T}}$-CNN: the same model with transfer learning. Within this setup, the CNN is trained after being already trained on the ESC-50 data (with data augmentation).

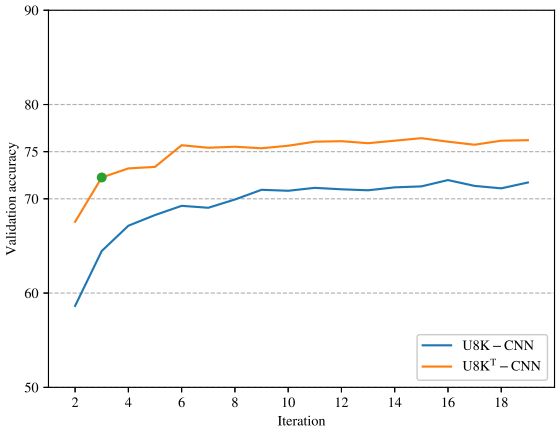


Figure 5.4: Validation-set accuracy during training under both setups.

In Figure 5.4, we can see the impact of transferring previously-learned knowledge from the ESC-50 dataset. Over the entire training process, the U8K$^{\mathrm{T}}$-CNN outperforms U8K-CNN, in this particular experiment, U8K$^{\mathrm{T}}$-CNN surpasses the peak accuracy (see the green dot) of U8K-CNN as early as the 3rd iteration thanks to the knowledge learned previously. Thus, transfer learning did not only improve the accuracy of the model, but also the time required to reach the maximum accuracy the model would reach when trained from scratch.

Evaluating both models using 10-fold cross validation yielded the following estimated average accuracies: 72.5% for U8K-CNN, and 77.3% for U8K$^{\mathrm{T}}$-CNN. The authors of the UrbanSound8K dataset [25] also proposed a convolutional neural network model (with data augmentation) for this dataset [29].

Model	Salamon et al. CNN	Salamon et al. CNN+aug	Ours U8K-CNN	Ours U8K$^{\mathrm{T}}$-CNN
Accuracy	73%	79%	72.5%	77.3%

Table 5.2: Results summary on the UrbanSound8K dataset

Chapter 6

Conclusion

In this work, we investigated the use of deep neural networks, specifically convolutional neural networks, for the problem of environmental sounds recognition. In addition to the neural network architecture we proposed for this problem, we also proposed a data augmentation technique called stochastic continuous data augmentation that has been experimentally proven to be beneficial for the learning model to generalize better by randomly exploring potential slight variations in data, but experimental results also show how it negatively affects the training time. Along with that, we have shown that transferring knowledge learned from one environmental sounds dataset to use it with another not only offers a slight improvement to the accuracy of the model, but also reduces training time since the model is already familiar with environmental sounds. Results on the datasets we used were compared against initial deep learning attempts by the respective author of each dataset.

Unfortunately, the initially intended data gathering did not take place, but it will be a priority for the future of this project. Since the datasets used in this work were collected from the web [24][25], the future materialization of this work relies on gathering data from a particular environment and conducting real-world experiments to further investigate the performance of the models and the approaches explored in this work in real-world conditions with potential hardware limitations.

Another possible future direction would be a further examination of the proposed stochastic continuous data augmentation on other kinds of data and problems. Intuitively, this method explores slight variations in data to help the learning model become less vulnerable and more resistant to some of the novelty faced on unseen data, but we suggest conducting deeper, both mathematical and empirical, analysis on this approach in the future.

Bibliography

[1] S. Chachada and C.-C. J. Kuo, "Environmental sound recognition: A survey," *2013 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, pp. 1–9, 2013.

[2] World Health Organization, "Noise pollution," tech. rep., Genf, Schweiz, 2001.

[3] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *Ieee Signal Processing Magazine*, no. November, pp. 82–97, 2012.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015.

[6] A. B. Downey, *Think DSP: Digital Signal Processing in Python.* O'Reilly Media, Inc., 1st ed., 2016.

[7] R. Priemer, *Introductory Signal Processing.* River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1991.

[8] D. Forsyth and P. Smaragdis, "CS498 - Lecture 8 - Audio Features."

[9] J. V. S. S. Stevens and E. B. Newman, "A Scale for the Measurement of the Psychological Magnitude Pitch," *Journal of the Acoustical Society of America*, 1937.

[10] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning," *Nature Methods*, vol. 13, no. 1, pp. 35–35, 2015.

[11] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, p. 436, may 2015.

[12] R. H. R. Hahnloser, H. S. Seung, and J.-J. Slotine, "Permitted and Forbidden Sets in Symmetric Threshold-Linear Networks," *Neural Computation*, vol. 15, no. 3, pp. 621–638, 2003.

[13] Y. N. e. a. Peter Eckersley, "EFF AI Progress Measurement Project," 2017.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet

Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[15] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: An ASR corpus based on public domain audio books," *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2015-August, pp. 5206–5210, 2015.

[16] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," *Advances in Neural Information Processing Systems 27*, pp. 2672–2680, 2014.

[17] P. Baldi, "Autoencoders, Unsupervised Learning, and Deep Architectures," *ICML Unsupervised and Transfer Learning*, pp. 37–50, 2012.

[18] C. M. B. Markus Svensen, "Pattern Recognition and Machine Learning Solutions," *Journal of Chemical Information and Modeling*, vol. 53, no. 9, pp. 1689–1699, 2013.

[19] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time-series," in *The Handbook of Brain Theory and Neural Networks* (M. A. Arbib, ed.), MIT Press, 1995.

[20] Y. Kim, "Convolutional Neural Networks for Sentence Classification," 2014.

[21] C. N. dos Santos and M. Gatti, "Deep Convolutional Neural Networks for Sentiment Analysis of Short Texts," *Coling-2014*, pp. 69–78, 2014.

[22] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, "Subject independent facial expression recognition with robust face detection using a convolutional neural network," *Neural Networks*, vol. 16, no. 5-6, pp. 555–559, 2003.

[23] S. Sigtia, A. M. Stark, S. Krstulović, and M. D. Plumbley, "Automatic Environmental Sound Recognition: Performance Versus Computational Cost," *IEEE/ACM Transactions on Audio Speech and Language Processing*, vol. 24, no. 11, pp. 2096–2107, 2016.

[24] K. J. Piczak, "ESC: Dataset for Environmental Sound Classification," in *Proceedings of the 23rd Annual ACM Conference on Multimedia*, pp. 1015–1018, ACM Press.

[25] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in *22nd ACM International Conference on Multimedia (ACM-MM'14)*, (Orlando, FL, USA), pp. 1041–1044, Nov. 2014.

[26] 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.

[27] Y. Xu, Q. Huang, W. Wang, and M. D. Plumbley, "Hierarchical learning for DNN-based acoustic scene classification," no. September, 2016.

[28] S. Adavanne, P. Pertilä, and T. Virtanen, "Sound Event Detection Using Spatial Features and Convolutional Recurrent Neural Network," 2017.

[29] J. Salamon and J. P. Bello, "Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification," no. November, pp. 1–5, 2016.

[30] K. J. Piczak, "Environmental Sounds Classification With Convolutional Neural Networks Karol J . Piczak Institute of Electronic Systems Warsaw University of Technology," 2015.

[31] T. H. Yuji Tokozume, "End-to-End Convolutional Neural Network for Sounds Classification," 2017.

[32] W. Dai, C. Dai, S. Qu, J. Li, and S. Das, "Very deep convolutional neural networks for raw waveforms," *CoRR*, vol. abs/1610.00087, 2016.

[33] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," *CoRR*, vol. abs/1411.1792, 2014.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[35] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012.

[36] B. Mcfee, C. Raffel, D. Liang, D. P. W. Ellis, M. Mcvicar, E. Battenberg, and O. Nieto, "librosa: Audio and Music Signal Analysis in Python," *PROC. OF THE 14th PYTHON IN SCIENCE CONF*, no. Scipy, pp. 1–7, 2015.

[37] T. E. Oliphant, "Guide to NumPy," *Methods*, vol. 1, p. 378, 2010.

[38] F. Chollet *et al.*, "Keras," 2015.