Accent Identification with Neural Networks

Teodor Bucht
teodor.bucht@gmail.com

Arvid Lunnemark
arvid.lunnemark@gmail.com

Linn Rydberg
linn.rydberg99@gmail.com

under the direction of
Sam Edgecombe

Malmö Borgarskola
March 4, 2018

Abstract

In this paper, we present an attempt at accomplishing accent identification with an approach based on neural networks. The architecture of a primitive neural network is thoroughly described. We implemented said network, and trained it using sound files from the two Swedish accents *skånska* and *stockholmska*. The performance was evaluated and turned out to be no better than chance when tested on separate data; nevertheless, albeit not our aim, the network did succeed in recognising voices. We found two plausible explanations for the failure in identifying accents: first, the primitive nature of our neural network, and second, the insufficient training data used.

Contents

1 Introduction

The human brain is unique. It has an outstanding capability of processing images and sounds. When perceiving a landscape, we can, thanks to our efficient brain, immediately understand and classify everything we see. Since this classification is inherently natural to us, we seldom reflect upon how advanced it is. However, given the task to detail precisely *how* we do our classification, we will soon understand its immense complexity.

For instance, how do we distinguish between a female face and a male one? Formulating precise rules is far from trivial. One might try to describe a male face as larger and with coarser eyebrows than a female face, possibly having a beard. But this description has several weaknesses. Firstly, far from all men are included, and secondly, neither eyebrows nor beards are crucial for humans to identify the gender. Moreover, some female faces will be included in the definition.

This is the key problem when programming computers to classify images using an algorithmic approach: it is hard to formulate precise rules for what distinguishes one item from another. The rules easily get too broad and too narrow at the same time, and it is not apparent what to do when the rules contradict each other. Yet, humans can reliably distinguish between a female face and a male one. Perhaps we should model our computer program after a human brain, then?

1.1 General Principles of Neural Networks

The key idea with neural networks is that instead of providing the computer with a set of rules, we provide it with the capability to learn from examples, just as the human brain does. Naturally, we then also need to provide the network with a large set of data. In the case of making a computer determine whether a picture represents a female face or a male one, this means that the computer is shown thousands of pictures, rather than being fed a list of rules. From these pictures, the computer, by itself, chooses which characteristics to take into account. This approach, in which the decision-making is opaque to the programmer, has been highly successful in complex tasks where traditional methods have struggled, such as image and speech recognition. [1]

1.2 Aim

The aim of this project is to create a neural network capable of identifying accents. We limit our scope to distinguishing between two Swedish accents: *skånska* and *stockholmska*. This restriction is mainly done due to limited access to easily handled training data. It is not principally significant, because a neural network that can distinguish between two given accents reliably can probably be trained on data from any other two accents and yield satisfactory results.

Distinguishing between accents is presumably a complex task, since there are significant individual differences. Obtaining perfect accuracy is therefore an unrealistic expectation; even humans fail to identify accents at times. Our aim is to be considerably better than chance, and possibly better than humans.

1.3 Applications

Of what use is a computer that is able to identify accents? We imagine two applications: determination of origin among undocumented immigrants and improved precision in speech recognition.

Accent is correlated to origin, which means that an unidentified person's origin can sometimes be determined using accent identification. The Swedish asylum application process therefore includes an accent analysis as a piece of evidence. Presently, this accent identification is done manually in Sweden. If automated with a computer program the process could possibly become both more accurate and more efficient. [2]

Accent identification can also be used to improve the performance of computerised speech recognition. Many words are pronounced differently depending on accent, and are therefore hard to classify for a generalised speech recogniser. This problem can be remedied by first detecting the accent and then choosing a speech recogniser specialised on the detected accent. [3]

2 Theory

In order to understand subsequent sections, one needs to be familiar with the Fourier transform and have an elaborate knowledge of neural networks.

2.1 The Fourier Transform

What we interpret as sound is simply sections of air alternating in density due to vibrations. This alternating density puts the ear's cochlea in motion, which is perceived as sound by the brain [4]. If we continuously measure the density of the air at a fixed point, we can describe the sound there as air density as a function of time. For a tone with a single frequency this function will be a sine wave. When recording speech, the sound will be composed of several different frequencies, and the obtained air density function will therefore not resemble a sine wave. Nevertheless, by using the Fourier transform, one can decompose the function into its constituent sine waves, and thereby determine which frequencies that the sound consists of.

2.1.1 Definition

We will use the Fourier transform to transform an air density function of time to an amplitude function of frequency. In more general terms, the Fourier transform is a transformation from the time domain to the frequency domain. The mathematics behind the Fourier transform is advanced and we have therefore chosen not to present a complete proof of it, but rather to give an overview and provide a sense of intuition for the formula. The Fourier transform $\hat{f}(\omega)$ of a function $f(t)$ is defined as

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{-2\pi i\omega t}\,\mathrm{d}t, \tag{1}$$

where in our case $f(t)$ is the mentioned air density function of time.

2.1.2 The Fourier Transform on a Sine Wave

In order to understand the Fourier transform, we will begin by examining the transformation of a function consisting of single frequency, i.e., a sine wave. The Fourier transform of a sine wave is easily comprehensible due to its predictable output: when decomposing a function consisting of a single frequency into its constituent frequencies, we will output that single frequency only. Consequently, when applying the Fourier transform to a sine wave, the expected result is a graph with one single peak. This means that the output function should be close to zero for every frequency expect for the frequency of the sine wave.

Multiplying $e^{-2\pi i\omega t}$ to a complex number (which can be visualised as a vector from the origin) rotates it with $2\pi\omega t$ radians in the negative direction. Hence, as we increase t, we can visualise $f(t) \cdot e^{-2\pi i\omega t}$ as winding the function $f(t)$ in a circle around the origin, with ω rotations per unit of time. When winding the function in a circle around the origin, we will obtain differently looking graphs depending on what frequency ω we use.

The next step in the Fourier transform is to calculate the integral of the complex function $f(t) \cdot e^{-2\pi i\omega t}$. When calculating an integral we calculate the sum of all points bounded by the function and the x axis. For most frequencies ω, this sum will be close to zero for a sufficiently large t. In fact, the sum will be close to zero for every frequency expect for the frequency ω_f of the particular sine wave making up $f(t)$. We can think of it as the sine wave being out of phase with the rotation for every frequency except ω_f, and when the sine wave is out of phase all its maxima will be evenly spread out, which implies that the sum will be close to zero. However, when the rotation frequency is equal to the sine wave's frequency, i.e. when $\omega = \omega_f$, we will get positive interference: all the maxima of $f(t) \cdot e^{-2\pi i\omega t}$ will coincide, and consequently, the sum of all the points bounded by the graph and the x axis will be nonzero.

2.1.3 Linearity

We have so far only considered the Fourier transform of a single sine wave, where we have established that its output will consist of a graph that has a single peak at the wave's frequency, and is close to zero elsewhere. What happens when we have a function that is a composition of several sine waves? Conveniently, the Fourier transform is linear, which means that applying it to a sum of two functions yields the same result as applying it to each constituent function individually and then adding the results. More formally, the linearity of the Fourier transform means that

$$\hat{h}(\omega) = \int_{-\infty}^{\infty} (g(t) + f(t))e^{-2\pi i\omega t}\,\mathrm{d}t = \int_{-\infty}^{\infty} g(t) \cdot e^{-2\pi i\omega t}\,\mathrm{d}t + \int_{-\infty}^{\infty} f(t) \cdot e^{-2\pi i\omega t}\,\mathrm{d}t$$

holds. If we let $F(t)$ be the air density as a function of time, we know that it can be written $F(t) = f_1(t) + f_2(t) + \ldots + f_n(t)$ where each $f_i(t)$ is a sine wave with frequency ω_i. We have established that the Fourier transform of each $f_i(t)$ has a single peak at frequency ω_i, and as a consequence of the linearity, we see that $F(t)$ will yield a function that has peaks at each of the frequencies $\omega_1, \omega_2, \ldots, \omega_n$, and is close to zero elsewhere.

Having stated that the air density function of time that we used to represent sound can be transformed into a function that has peaks at the frequencies that are present in the sound, we conclude our brief exposition of the Fourier transform. Note that we have not proved that the Fourier transform works; for a more complete reference, consult [5].

2.2 Neural Networks

A neural network is at its core nothing more than a list of weights used to map an input value to an output value. The weights are adjusted to suit known input-output pairs (a process known as *learning* or *training*) in the pursuit of being able to use those weights later to map an unknown input to its correct output.

In this section, we will thoroughly present a primitive yet fully functioning neural network. It is not state-of-the-art, but it is fundamental in that many of its ideas are reused in more advanced networks. We will use standard terminology whenever applicable.

We assume that the network will receive as input a column matrix x and that its output should be a column matrix y. The *training data* is the collection of pairs (x, y), where x contains an input and y the desired answer for x. Data will need to be preprocessed to fit this format before being used in our neural network.

2.2.1 Sigmoid Neurons

Before we study an entire network, it helps to first consider its fundamental building block: a single neuron. A neuron, and in our case a *sigmoid neuron*, takes as input a column matrix x and produces a scalar y as output, where $0 < y < 1$. Figure 1 shows a visualisation of a sigmoid neuron with three input variables.

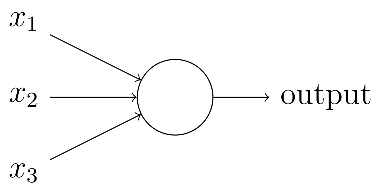

Figure 1: A sigmoid neuron with three input variables.

Each x_{i1} is assigned its own weight w_{1i}, which determines the relative importance of x_{i1} with respect to all other input variables (if the notation seems odd, recall that x is a column matrix and w is a row matrix). All w_{1i} together comprise the weight matrix w, which is a row matrix with the same number of columns as the number of rows in x. The neuron also has a scalar bias b. The neuron's output is determined by the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \tag{2}$$

hence the name *sigmoid neuron*. We let $z = w \cdot x + b$. That is,

$$y = \sigma(z) = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}}.$$

It can be easily verified that this function indeed produces an output in $(0, 1)$ and that it is increasing for all z.

Learning consists of changing the weights w and the bias b such that the output y for every input x corresponds to the given training examples as well as possible. The injectivity of the sigmoid function is therefore critical, as is the fact that a small change in z results in a small change in y.

2.2.2 Network Architecture

A single neuron has a limit on the complexity of the decisions it can make. For nearly all tasks, we need to chain several neurons together in a network in order to get satisfactory results.

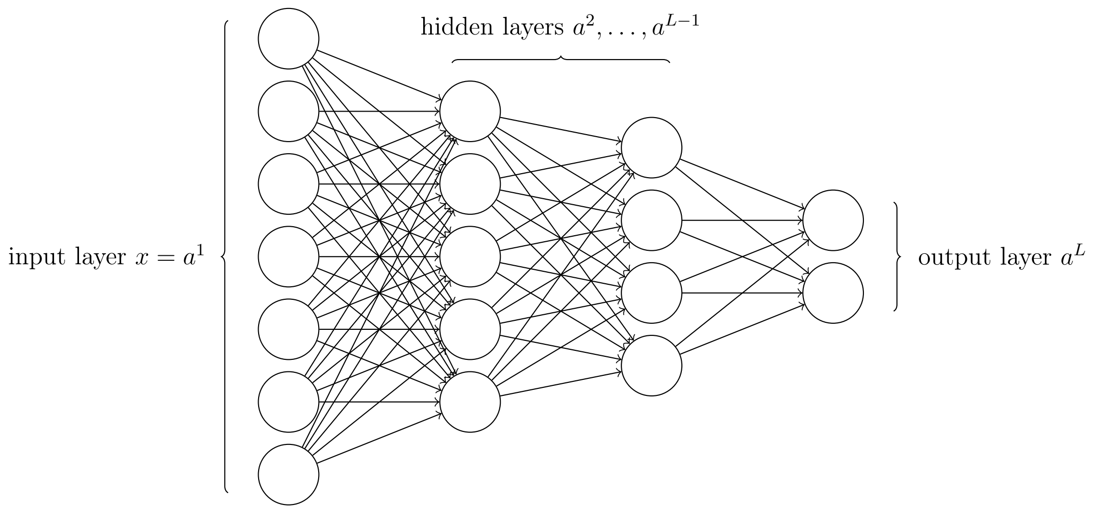

Figure 2: A complete network of sigmoid neurons with $L = 4$ layers (that is, 2 hidden layers).

We structure our network in layers of neurons, as in Figure 2. The first layer is called the input layer, the last layer is called the output layer, and the layers in between are called hidden layers.

Each neuron in layer l takes as input all outputs from the neurons in layer $(l-1)$. For layer l with n_l neurons we will therefore have n_l outputs, which we store in the column matrix a^l (where l is a superscript for indexing, not an exponentiation). The column matrix a^l is called the activations of layer l. We let the activations of our input layer equal our input matrix x. That is,

$$a^1 = x. \tag{3}$$

As outlined in section 2.2.1 each neuron in each layer has a vector of weights and a scalar bias (except for the input layer). For each layer l ($l > 1$) we define w^l to be a matrix of dimension $n_l \times n_{l-1}$ such that the element at position (j,k) contains the weight associated with the edge from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. We also define b^l to be a matrix of dimension $n_l \times 1$ where the element at position $(j,1)$ contains the bias associated with the j^{th} neuron in the l^{th} layer. This slightly cumbersome notation yields the nice relation

$$a^l = \sigma(z^l) = \sigma(w^l \cdot a^{l-1} + b^l), \tag{4}$$

for all $l > 1$. That is, we define $z^l = w^l \cdot a^{l-1} + b^l$. Here, our σ takes as argument a column matrix z^l and outputs a column matrix a^l where $a^l_{i1} = \sigma(z^l_{i1})\ \forall i$.

We now see that equation (4) along with equation (3) gives us a way to compute $y = a^L$, the output of our network, given an input x. Thus, we know now how our network takes decisions given its weights and biases, and all left to do is to find out how to make our network learn.

2.2.3 Cost Function

To be able to learn, we need a measure of how good our network is, with given weights and biases. We do this by introducing a cost function, which should approach zero as our network performs better and better. We let $C(w, b)$ be our cost function, where w and b

represent all the weights and biases in our network, respectively. We define

$$C(w,b) = \frac{1}{n}\sum_x C_x = \frac{1}{n}\sum_x \sum_{i=1}^{s} -(y_{i1} \ln a_{i1}^L + (1-y_{i1})\ln(1-a_{i1}^L)), \qquad (5)$$

where y is the known correct output for input x, and a^L is the output the network produces for input x with weights w and biases b. Note that both y and a^L are column matrices of dimension $s \times 1$. The outer sum is taken over training data tuples (x, y), where we have n such tuples.

The goal of learning is to minimise the cost function. To see why this makes sense, it helps to differentiate the cost function with respect to each a_{i1}. For the sake of simplicity, we ignore the multiple sums for a moment, obtaining

$$\frac{\partial C_x}{\partial a_{i1}^L} = \frac{a_{i1}^L - y_{i1}}{a_{i1}^L(1-a_{i1}^L)}$$

for each i. We see that this derivative will be zero if and only if $a_{i1}^L = y_{i1}$. We also see that it is positive when $a_{i1}^L > y_{i1}$ and negative when $a_{i1}^L < y_{i1}$ (since $0 < a_{i1}^L < 1$ implies $a_{i1}^L(1-a_{i1}^L) > 0$). Thus, the cost function has only one minima for each a_{i1}^L and it will be minimised for that a_{i1}^L if and only if $a_{i1}^L = y_{i1}$, that is, when the network's output a^L is as close to the desired output y as possible.

2.2.4 Gradient Descent

In general, to minimise a function $f : \mathbb{R}^n \to \mathbb{R}$ we can start with an arbitrary v and then successively update it using $v \to v' = v + \Delta v$ such that $f(v') < f(v)$. This will eventually lead us to a global minimum (if one exists). How do we find an appropriate Δv for each v?

It turns out that this is a hard problem, because if we happen to arrive at a local minimum, Δv may need to be very big in order to find v'. Therefore, we restrict ourselves to finding local minima, in which case we know by Theorem 1 that the optimal choice to minimise f is $\Delta v = -\eta \nabla f_v$ where $\eta > 0$ is a small parameter. Thus, the algorithm we use for minimising f, which we call the *gradient descent* algorithm, can be summarised as follows:

1. Choose a random v, and a small η.
2. Update $v \to v' = v - \eta \nabla f_v$.
3. Repeat from 2 as long as v changes substantially.

Theorem 1. Let $f : \mathbb{R}^n \to \mathbb{R}$, and let ∇f_v be the gradient of f at v, i.e., $\nabla f_v = (\frac{\partial f}{\partial v_1}, \dots, \frac{\partial f}{\partial v_n})$. Let $\epsilon > 0$. Then, if $\Delta v = -\eta \nabla f_v$, where $\eta = \frac{\epsilon}{|\nabla f_v|}$ if $|\nabla f_v| \neq 0$ and 1 otherwise, we know that for sufficiently small ϵ

(i) $f(v + \Delta v) \leq f(v)$, and

(ii) $f(v + \Delta v) - f(v) \leq f(v + \Delta v') - f(v)$ for all $\Delta v'$ where $|\Delta v'| \leq \epsilon$.

Proof. We define

$$\Delta f = f(v + \Delta v) - f(v), \tag{6}$$

where we want to prove that $\Delta f \leq 0$. For sufficiently small Δv, it follows from the definition of the gradient that

$$\Delta f \approx \nabla f_v \cdot \Delta v. \tag{7}$$

We now see that

$$\Delta f \approx \nabla f_v \cdot \Delta v = -\eta |\nabla f_v|^2 \leq 0 \tag{8}$$

since $\eta = \frac{\epsilon}{|\nabla f_v|} > 0$. With (6) this concludes our proof of (i).

From Cauchy-Schwarz we see that

$$|\nabla f_v \cdot \Delta v'|^2 \leq |\nabla f_v|^2 |\Delta v'|^2 \leq |\nabla f_v|^2 \epsilon^2 = (\eta |\nabla f_v|^2)^2 = |\nabla f_v \cdot \Delta v|^2 \tag{9}$$

Now, assume that $f(v + \Delta v) - f(v) > f(v + \Delta v') - f(v)$. Then, $\nabla f_v \cdot \Delta v > \nabla f_v \cdot \Delta v'$ by (6) and (7). Since $\nabla f_v \cdot \Delta v$ is nonpositive by (8), this implies that $|\nabla f_v \cdot \Delta v'| > |\nabla f_v \cdot \Delta v|$.

But this is a contradiction, because in (9) we had $|\nabla f_v \cdot \Delta v'| \leq |\nabla f_v \cdot \Delta v|$. Thus, our assumption that $f(v+\Delta v) - f(v) > f(v+\Delta v') - f(v)$ is false, which concludes our proof of (ii). $\square$

To minimise our cost function we will use gradient descent. More explicitly, we will modify each weight by

$$w^l_{jk} \to {w^l_{jk}}' = w^l_{jk} - \eta \frac{\partial C}{\partial w^l_{jk}}, \tag{10}$$

and similarly each bias by

$$b^l_{j1} \to {b^l_{j1}}' = b^l_{j1} - \eta \frac{\partial C}{\partial b^l_{j1}} \tag{11}$$

for each layer l, where η, called the learning rate, is defined beforehand. Since these updates will decrease the cost C as per Theorem 1, this repeated updating of the weights and biases is what constitutes our learning.

In equation (5) we defined the cost to be a sum over all training examples. Since the number of training examples can be very large it is not efficient to compute the partial derivative of every weight and bias with respect to the cost for all training examples. Instead, we employ stochastic gradient descent, in which we randomly choose K examples from the training data and approximate the gradient using

$$\nabla C \approx \nabla C_K = \frac{1}{K} \sum_{i=1}^{K} \nabla C_{x_i}. \tag{12}$$

If we have N training examples in total, we say that our randomly chosen K of these constitutes a *mini-batch*. After having completed $\frac{N}{K}$ mini-batches, we say that we have completed one *epoch* of learning.

2.2.5 Backpropagation

Since we learn using gradient descent, we need to compute gradients. In other words, we need to compute the partial derivative for every weight and bias with respect to the cost C_x. We can then use equation (12) to find ∇C, and finally update the weights and biases according to equations (10) and (11).

Proposition 1. Let $\frac{\partial C_x}{\partial z^L}$ denote the column matrix containing the partial derivatives $\frac{\partial C_x}{\partial z^L_{i1}}$ for every i, where x is a training input and y is its corresponding output, and y and z^L both have dimension $s \times 1$. Then,

$$\frac{\partial C_x}{\partial z^L} = \left(a^L_{11} - y_{11}, \ldots, a^L_{s1} - y_{s1}\right)^T.$$

Proof. Note that

$$C_x = \sum_{i=1}^{s} -(y_{i1} \ln a^L_{i1} + (1 - y_{i1}) \ln(1 - a^L_{i1})) \tag{13}$$

from equation (5). By differentiating with respect to each a^L_{i1} we get

$$\frac{\partial C_x}{\partial a^L} = \left(\frac{a^L_{11} - y_{11}}{a^L_{11}(1 - a^L_{11})}, \ldots, \frac{a^L_{s1} - y_{s1}}{a^L_{s1}(1 - a^L_{s1})}\right)^T. \tag{14}$$

Applying the chain rule yields us

$$\frac{\partial C_x}{\partial z^L_{i1}} = \frac{\partial C_x}{\partial a^L_{i1}} \cdot \frac{\partial a^L_{i1}}{\partial z^L_{i1}}. \tag{15}$$

Recall that $a^l = \sigma(z^l)$ for all l. Thus,

$$\frac{\partial a^L_{i1}}{\partial z^L_{i1}} = \sigma'(z^L_{i1}) = \sigma(z^L_{i1})(1 - \sigma(z^L_{i1})) = a^L_{i1}(1 - a^L_{i1}), \tag{16}$$

by using the definition of $\sigma(z)$ from equation (2) and some algebraic manipulations.

It is now clear that our proposition follows from equations (14), (15) and (16). □

Proposition 2. Let $\frac{\partial C_x}{\partial z^l}$ be a column matrix containing the partial derivatives $\frac{\partial C_x}{\partial z^l_{i1}}$ for every i, and define $\frac{\partial C_x}{\partial z^{l+1}}$ similarly. Then, with $\odot$ denoting the Hadamard product (i.e., element-wise multiplication),

$$\frac{\partial C_x}{\partial z^l} = \left((w^{l+1})^T \cdot \frac{\partial C_x}{\partial z^{l+1}}\right) \odot \sigma'(z^l)$$

Proof. Applying the chain rule yields us

$$\frac{\partial C_x}{\partial z^l} = \frac{\partial C_x}{\partial a^l} \odot \frac{\partial a^l}{\partial z^l}. \tag{17}$$

Since $a^l = \sigma(z^l)$ we see that

$$\frac{\partial a^l}{\partial z^l} = \sigma'(z^l). \tag{18}$$

We now want to find $\frac{\partial C_x}{\partial a^l}$. Recall that $z^{l+1} = w^{l+1}a^l + b^{l+1}$. From this relation we see, with the help of Figure 3, that

$$\frac{\partial C_x}{\partial a^l} = (w^{l+1})^T \cdot \frac{\partial C_x}{\partial z^{l+1}}. \tag{19}$$

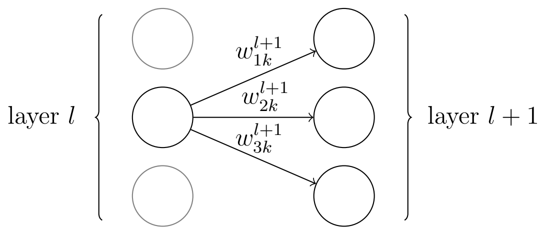

Figure 3: A visualisation showing how the activation from one neuron k in layer l affects all neurons in layer $l+1$.

It is now clear that our proposition follows from equations (17), (18) and (19). □

Proposition 3. Let $\frac{\partial C_x}{\partial b^l}$ be a column matrix containing the partial derivatives $\frac{\partial C_x}{\partial b_{i1}^l}$, and define $\frac{\partial C_x}{\partial z^l}$ similarly. Then,

$$\frac{\partial C_x}{\partial b^l} = \frac{\partial C_x}{\partial z^l}$$

Proof. The proposition follows immediately from $z^l = w^l a^{l-1} + b^l$. □

Proposition 4. Let $\frac{\partial C_x}{\partial w^l}$ be a matrix containing the partial derivatives $\frac{\partial C_x}{\partial w_{ij}^l}$, and define $\frac{\partial C_x}{\partial z^l}$ similarly. Then,

$$\frac{\partial C_x}{\partial w^l} = \frac{\partial C_x}{\partial z^l} \cdot (a^{l-1})^T.$$

Proof. The proposition follows immediately from $z^l = w^l a^{l-1} + b^l$. □

Using the results from propositions 1 through 4 we can efficiently compute the partial derivatives of C_x with respect to every weight and every bias in the network. We can then use stochastic gradient descent to make our network learn.

2.2.6 Hyperparameters

We know now how the neural network learns and takes decisions. We have left one thing unsaid, however: how do we choose the number of layers, the number of neurons in each layer, the learning rate η, the number of training samples K in each mini-batch, and the number of epochs? Since these parameters control how the network learns, they cannot themselves be learned by the network. Therefore, we call them *hyperparameters*, and while there are various heuristics for setting them, we mostly rely on trial and error.

2.2.7 Training Data, Test Data and Validation Data

The neural network learns by minimising the cost function defined on the training data. However, we need to remember that the cost function is not necessarily directly corresponding to being good at the intended task. Having a minimised cost function only tells us that the network fares well on the particular examples included in the cost function definition. Ideally, we would need to compute the cost function on all possible data that the network is supposed to work on; however, that is unfeasible. Instead, we can take a random sample from all possible data and train our neural network using that data. However, when we evaluate the quality of neural network we *cannot* use the same samples that we used for training, since our network might in that case learn the peculiarities of our training data, and it would then perhaps not fare well on a sample chosen randomly from all possible data.

Therefore, to evaluate the performance of our network, we need a separate data set. We call it the *test data*, as opposed to the *training data* that was used in our gradient descent. That is, the test data is used only for evaluating the cost, not for learning. To actually evaluate our neural net, we observe what fraction of the test cases in the training data the neural network got correct.

However, when we set the hyperparameters, we cannot consider the performance of the network based on the test data, because then we commit the same error as if we had trained our network on the test data. Thus, we need yet another data set, which we call the *validation data* and use for setting the hyperparameters.

3 Method

3.1 Gathering Data

We downloaded podcasts with different people talking from [6] and files from different advertisements from [7]. Half of the downloaded files were of people talking *skånska* and half of them of people talking *stockholmska*. We then used Audacity to remove the music from the audio files, as this would elsewise be a source of error [8].

3.2 Converting Audio Files to Numbers

The audio files were first split into multiple audio files, each with a length of five seconds. We also converted all the files into raw waveform files (in *.wav* format). This was done using `splitfiles.py` which can be found in [9]. The goal is to obtain a matrix representing the audio file that we can use as input for the neural network.

The first step in converting the waveform files is to use the Fourier transform. The Fourier transform enables us to analyse the sound by breaking it down to the different frequencies that it consists of. Since our brains perceive sound as amplitudes of different frequencies, applying the Fourier transform seems like a good strategy for making sounds interpretable to a neural network [4]. Therefore, we applied the Fourier transform on every $\frac{5}{214}$ second fragment of the five second file, i.e., we made a spectrogram of each five second file.

We now have matrix A where each column represents a $\frac{5}{214}$ second fragment of the original five second audio file. The data was then compressed in order to enable faster training of the net. This was done in multiple steps. Firstly, the information about frequencies higher than 10 kHz was removed. We now have a 300×214 matrix where the j^{th} column represents the j^{th} $\frac{5}{214}$ second fragment out of the five second file and the i^{th} element in each column represents the sound intensity in the interval $((i-1)\frac{100}{3}, i\frac{100}{3})$ Hz. Secondly, A was converted to B, a 30×22 matrix, using the following rule:

$$b_{ij} = \sum_{k=10(i-1)+1}^{min\{10i,300\}} \sum_{l=10(j-1)+1}^{min\{10j,214\}} \frac{a_{kl}}{min\{10, 300-10(i-1)\} \cdot min\{10, 214-10(j-1)\}}.$$

This basically means that we average over 10×10 squares (and rectangles of varying size when we do not have enough space for 10×10 squares). The next step is to normalise B by finding the greatest element b_M in the matrix and divide every other element by that value. That is, for the new value b'_{ij} at position (i, j) we get $b'_{ij} = \frac{b_{ij}}{b_M}$. Next, we take minus the logarithm of each element in B, meaning that for the new value b''_{ij} at position (i, j) we get $b''_{ij} = -\log(b'_{ij})$. This is done in order to get the sound intensities in dB. To finally obtain the matrix used as input to the network we normalise B once again.

The entire code used for converting the five second audio files is `spectrogram.py` and it can be found in [9]. What we end up with is a 30×22 matrix where the j^{th} column represents the j^{th} $\frac{5}{22}$ second fragment of the five second file and the i^{th} value in each column represents the sound intensity in the interval $((i-1)\frac{1000}{3}, i\frac{1000}{3})$.

For making the neural network easier to implement we converted the 30×22 matrices into column matrices with $30 \cdot 22 = 660$ elements. We can now create a tuple for each five second file where the first element is the column matrix that was just described and the second element is a column matrix containing the answer for that audio file. The second element in the tuple is $(0, 1)^T$ if the person is talking *skånska* and $(1, 0)^T$ if the person is talking *stockholmska*. We can now add the tuples into lists containing training data, validation data and test data, which concludes our preprocessing.

3.3 Training the Neural Network

We can now set up our actual neural network and begin training. This is done as described in section 2. We pass the training data to the net and perform our stochastic gradient descent algorithm. To keep track of all files and data sets, the script `coordinator.py` from [9] was used.

3.4 Experimenting with Hyperparamaters

For training data, the podcasts were used. For validation data and test data the files from advertisements were split evenly. We then experimented with different values on the hyperparamaters and chose the best ones by training the network and evaluating against the validation data.

4 Results

From experimenting with the hyperparamaters we found that 200 epochs, a mini-batch size of 200, $\eta = 5.0$ and a network with one hidden layer and 200 neurons in that layer gave us the best results. The training data contained roughly 10 000 five second files from 24 different voices.

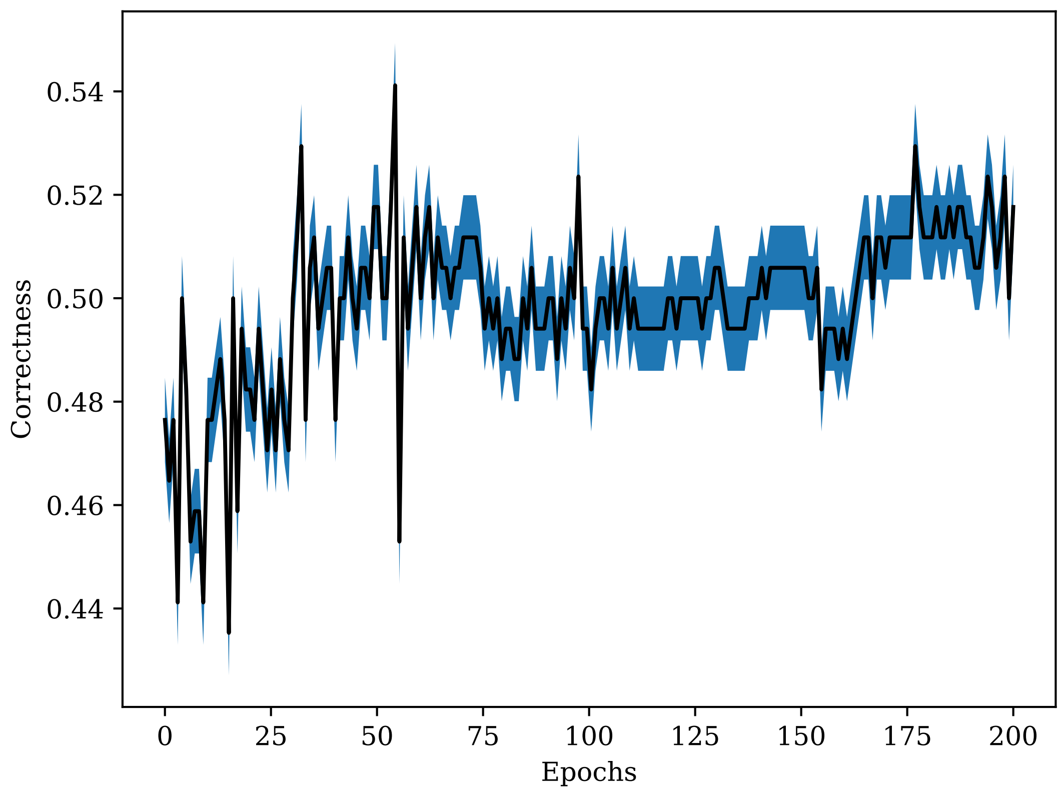

Figure 4: The Black line represents the fraction of correct answers when the net was evaluated on the test data and the blue area around the black line represents a confidence interval with a 5% margin of error. The test data in this diagram come from the advertisement files and the test data contained 170 audio files in total. The code used for plotting is `plotter.py` and can be found in [9].

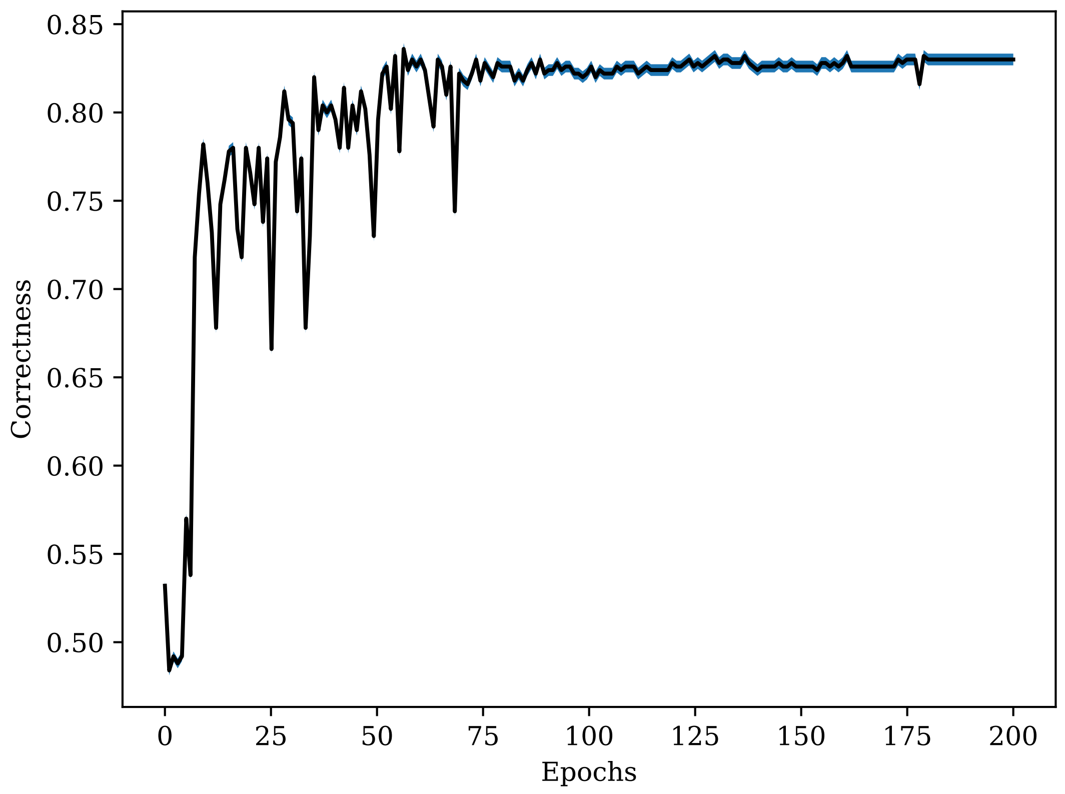

Figure 5: The Black line represents the fraction of correct answers when the net was evaluated on the test data and the blue area around the black line represents a confidence interval with a 5% margin of error. The test data in this diagram come from the same voices as the training data, but from different files. The test data contained 500 audio files in total. The code used for plotting is `plotter.py` and can be found in [9].

5 Discussion

In this section, the results will be interpreted. Moreover, possible improvements will be investigated.

5.1 Interpretation of Results

If we would guess the accent randomly, the probability of achieving the right answer would approach $\frac{1}{2}$, since there only are two possible alternatives (*skånska* and *stockholmska*). Therefore, in order to be useful, a program for accent identification needs to have a correctness that is significantly higher than 50%. The correctness obtained from our program when the training data and the test data were separated (where there were no recordings of the same person in both the test data and the training data) is presented in Figure 4, and it shows that the correctness is stabilising just over 50%, and we can thereby draw the conclusion that the program fails to fulfil its assignment. Using test data that is separated from training data is the only reasonable way of measuring the capability of our accent identification program, since the training data very rarely will coincide with the audio files we wish to classify when using the program for real life applications.

However, the results when the training data and the test data were not separated (that is, the test data and the training data contained recordings from the same persons, but the recordings from the test data and the training data were distinct) looks more promising. As shown in Figure 5 the correctness is stabilising just over 80% which would be a satisfactory result. A reasonable explanation for this significantly higher accuracy is that the program recognises the voices, rather than the accents.

The fact that the voices are recognised instead of the accents is a good illustration of a weakness inherent to neural networks. When learning from example data, the network's definition of *skånska* and *stockholmska* will be solely based on the training data. Therefore, there is a risk that the definitions adapted from the training data will differ from the common definitions of the corresponding items or categories. In this case, the definitions of the accents *skånska* and *stockholmska* are based on the characteristics of the voices of the speakers, rather than the distinguishing features of the accents. Generally, one can state that a neural network will never be better than the data it is trained on. Therefore, it is of the highest relevance that the training data is large, representative and diverse. Otherwise, the definitions of the categories used by the network will differ from their common definitions, which is undesirable since we are interested in the common definitions for real-world applications.

5.2 Possible Improvements

One suggestion for improving the accent identification is to implement and train a more advanced neural network than the one used in this study. It is reasonable to believe that a convolutional neural network would be a good choice, since they are frequently and successfully used for image recognition. Image recognition is a similar task to accent recognition, since the spectrograms that we use to represent our audio files can be represented as pictures. Furthermore, the special and most important property of convolutional neural networks is that they can disregard the location of features in a picture, in our case implying that it would not matter when and with what pitch a person made an accent-specific sound. [1]

Another possible improvement is to use a better set of training data. The podcasts that were used in this study are suboptimal from several points of view. Firstly and most importantly, relatively few persons are speaking in the audio files. This is problematic since only a few different voices are introduced to the neural network. In order to understand this problem, it is important to keep in mind that the neural network is not aware that it is looking for accents. It is only assigned to identify characteristics that are shared by the examples of the same accent, but not shared by the examples of the other accent. One of the most tangible characteristics that the program can identify is which frequencies that are dominating in each audio file. Unfortunately, the dominating frequencies are to a large extent affected by the voice rather than the accent. This means that a large number of different voices are required to identify the relatively subtle differences between the accents. If the task instead would be to differ between two categories where the difference in dominating frequencies is major, for example the task of differing between male and female voices, it would be a minor problem for the training data to only contain a few different voices.

Furthermore, a possible flaw in our test data is that the audio files used are only five seconds long. One can imagine that a person speaking *skånska* does not speak with a sufficiently distinct accent in every five second interval. In order to estimate how large this source of error might be, we gave humans the task of determining the accent from five second audio files. In most cases, however, the humans found it easy to determine the accent, making the short audio files a minor source of error.

6 Conclusion

We may conclude that the neural network used in this report is not capable of solving the complex task of accent identification; however, it is capable of solving simpler tasks such as voice recognition. A possible reason for this is that the training data used was neither sufficiently numerous nor sufficiently diverse. Another possibility is that weak results were caused by the primitiveness of the neural network's architecture. Whether accent identification would be possible with a more advanced neural network cannot be concluded from this attempt. However, there is, as described in the introduction, several interesting applications of accent identification, and we therefore encourage further investigation.

References

[1] Nielsen MA. Neural networks and deep learning. Determination Press; 2015.

[2] Språkanalyser. Migrationsverket; 2017. Accessed 25 February 2018. `https://www.migrationsverket.se/Om-Migrationsverket/Pressrum/Fokusomraden/Sprakanalyser.html`.

[3] Huang C, Chang E, Chen T. Accent Issues in Large Vocabulary Continuous Speech Recognition. Microsoft Research China; 2001.

[4] Björndahl G, Castenfors J. Spira 2. Liber; 2012.

[5] Bracewell RN. The Fourier transform and its applications. McGraw-Hill; 1986.

[6] Alla avsnitt från programmet Sommar & Vinter i P1. Sveriges Radio; 2018. Accessed 14 January 2018. `http://sverigesradio.se/sida/avsnitt?programid=2071`.

[7] Worldwide Voice Archive. TMP Voices; 2013. Accessed 2 March 2018. `http://www.tmpvoices.com/sv/voices/160`.

[8] Audacity. Audacity Team; 2018. Accessed 25 February 2018. `https://www.audacityteam.org`.

[9] Bucht T, Lunnemark A, Rydberg L. Accent Identifcation with Neural Networks. GitHub; 2018. `https://github.com/ArVID220u/accentrecognition`.