NTNU

Norwegian University of Science and Technology

Using Deep Learning in Hearing Aids

Ida Husby Swendgaard

Master of Science in Electronics
Submission date: June 2016
Supervisor: Odd Kr. Pettersen, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Preface

This thesis constitutes the final part of a Master's Degree in Electronics (MTEL) at the Department of Electronics and Telecommunications (IET) at the Norwegian University of Science and Technology (NTNU). The thesis was written during the spring semester of 2016 in cooperation with the research organisation SINTEF.

My specialization is in Signal Processing and Communications thus the thesis is written for people with the same background as myself. The workload corresponds to 30 ECTS.

Trondheim, 17.06.2016

Ida Husby Swendgaard

Acknowledgment

First of all, I would like to thank my supervisor Odd Kr. Pettersen (Professor at NTNU and Research Director at SINTEF ICT) for helping me choose the subject for this thesis which I have found truly interesting. I also thank him for giving me feedback on my project thesis which have been very useful.

I also want to thank my co-supervisors Femke Gelderblom and Tor André Myrvoll (Reseach Scientists at SINTEF ICT) for their guidance and feedback throughout this spring. We have had multiple meetings and discussions I have greatly appreciated, and I have also learnt a lot from all the e-mails we have sent during this spring. Lastly I want to give a thanks to Olav Kvaløy (Reseach Scientist at SINTEF) for his useful input during meetings. He was my co-supervisor during my project thesis and has taught me a lot about hearing and hearing aids.

I.H.S.

Summary

The aim of this masters thesis is to investigate if deep neural networks and deep learning can work as an alternative for existing hearing aids. Traditional hearing aids are limited to compensate for attenuation losses and do not seem to work for losses containing distortion. As deep learning has successfully been used for other speech related tasks such as speech-in-noise and speech recognition, it seems natural to look at the use of deep learning also in hearing aids. Maybe deep neural networks can be trained to pre-compensate for more complex losses than the ones being covered by today's hearing aids.

The work done in this thesis includes setting up a complete system including a deep neural network and a hearing loss simulator in addition to some more general signal processing blocks. First the system is trained without any hearing loss in order to obtain an optimized system for the speech data. Then, some simple hearing losses are applied such as damping, expansion and quantization.

As a hearing loss will be applied after any hearing aid, information that is lost will not be possible to restore using pre-compensation. The results shows that the deep neural network can be trained to compensate for both damping and expansion, but as expected it does not work when the hearing loss contains quantization, that is, loss. The system is not tested for more complex hearing losses, thus this should be done in order to further study the true effect of using deep learning in hearing aids. Designing functions for complex hearing losses is the most challenging task that remains to be done. When such an advanced hearing loss simulator is obtained, the system must be trained using this loss in order to see if it is capable of learning in a satisfactory way.

Sammendrag

Målet med denne masteroppgaven er å undersøke om dype neurale nett og dyp læring kan brukes som et alternativ til eksisterende hørselhjelpemidler. Tradisjonelle hørselhjelpemidler er begrenset til å kompensere for tap med demping og ser ikke ut til å fungere for tap som inneholder forvrenging. Fordi deep learning med hell har blitt brukt i andre oppgaver som inkluderer tale, for eksempel tale-i-støy og talegjenkjenning, virker det naturlig å se på dyp læring også i hørselhjelpemidler. Kanskje kan dype neurale nett bli trent til å prekompensere for mer komlpekse hørseltap enn dagens hørselhjelpemidler.

Arbeidet som er gjort i denne oppgaven inkluderer å sette opp et system bestående av et dypt neuralt nett og en hørseltapssimulator, i tillegg til mer generelle signalbehandlingsblokker. Først er systemet trent uten hørseltap for å finne et optimalt system for taledataen. Senere påføres noen enkle hørseltap som demping, ekspansjon og kvantisering.

Fordi et hørseltap vil komme etter et eventuelt hørselhjelpemiddel vil ikke informasjon som blir tapt være mulig å gjenskape ved bruk av prekompensasjon. Resultatene viser at dype neurale nett kan trenes opp til å kompensere for både demping og ekspansjon, men som forventet fungerer det ikke når hørseltapet består av kvantisering, altså tap av informasjon. Systemet er ikke testet for mer komplekse hørseltap, men dette burde gjøres for å videre kunne studere den virkelige effekten av dyp læring i hørselhjelpemidler. Design av funksjoner for å gjengi komplekse hørseltap er den mest krevende oppgaven som gjenstår. Når en slik hørseltapssimulator er funnet må systemet videre trenes med dette for å se om det er kapabelt til å lære på en tilfredsstillende måte.

Contents

Chapter 1

Introduction

1.1 Background and motivation

The human brain is immensely complex. It is capable of processing input from its surroundings and learn through experience. Neural networks seeks to imitate the brain's learning capability, and has successfully been used in multiple speech problems the last couple of years. It is becoming a mainstream technology for use in speech recognition (Schmidhuber (2015)) and has also been used for improving speech intelligibility in real acoustic environments (Xu et al. (2015)). It is thus natural to look at the use of deep learning in other speech related tasks, as for example hearing aids.

Newer, traditional hearing aids consists of complex signal processing circuits including wide dynamic range nonlinear amplifiers, multichannel devices, fully digital systems, and special features such as noise reduction algorithms and dual-microphone directionality (Walden et al. (2000)). The limitations of such hearing aids includes difficulties in compensating for complex losses, for example containing distortion in time. Maybe deep neural networks can be trained to compensate for such losses.

Fitting strategies change input speech signals to compensate for hearing loss. These strategies output frequency gains and compressor settings for hearing aids based on a persons audiogram as input. Traditionally these strategies are designed by combining understanding of hearing loss, speech intelligibility and feedback of hearing aid users. Deep Learning may provide an alternative method for obtaining a fitting strategy.

1.2 Problem formulation

The scope of this masters thesis is to study the use of Deep Learning for use in hearing aids. First, a suitable structure of the whole system should be determined. Then, the deep neural network should be set up and trained with different inputs as well as different parameters in order to determine which ones gives the best results. At last, hearing losses should be applied in order to test the system could be of use in a hearing aid.

1.3 What remains to be done

The most important task that remains to be done is making a complex hearing aid simulator to imitate realistic hearing losses. As the losses are often very complex, the functions can also be made advanced and Chapter 3.4 describes some alternatives for realizing such a hearing loss simulator. It would have been very interesting to see how the neural network responded to more complex loss functions but the time was unfortunately limited.

In addition to the hearing loss simulator, the error function should be considered updated and optimized. The implemented error function is the squared difference between the predicted value and the true value but others should be tested in order to try to minimize the losses. Another way of improving the training results is to add more input data. This thesis uses a maximum of around 2.6 hours of training data but this could advantageously be increased further. The more data, the better the results (Xu et al. (2015)).

Lastly, a good way of determining the quality of the output should be found. An example is the use of PESQ (Perceptual Evaluation of Speech Quality) in addition to using the training and test results. In the results found in this thesis, the quality was measured by looking at the training and validation losses in addition to listening to the output. Testing on actual hearing impaired persons should eventually be done, in order to evaluate if the system has the desired effect as a hearing aid. This is also discussed in Chapter 5.

A hearing aid used for pre-compensating for hearing losses will not be of use if there is a total loss of information due to sensorineural damages. If certain frequencies are not perceived, a solution might be to move these frequencies up or down to a range where they can be perceived. This is not taken into consideration in this thesis, but can be used for later studies.

1.4 How the thesis is structured

The thesis is structured in the following way: Chapter 2 introduces some theory used when designing the system, including theory on hearing losses and machine learning. Some of the theory on hearing losses are taken from my project thesis which also dealt with hearing aids. Chapter 3 presents how the system is realized. It includes information on the utilized training and validation data as well as a detailed description of the neural network and other applied signal processing blocks. The chapter also introduces possible structures of the hearing loss simulator, even though they were not implemented in the real system due to time limitations. The results are shown in Chapter 4 and discussed in Chapter 5. Conclusions are made in Chapter 6.

Todo list

Chapter 2

Theory

This chapter presents some theory used during the implementation of the system. As stated in the introduction, the system consists of a deep neural network and a hearing loss simulator in addition to some signal processing blocks for doing necessary changes to the signal. This chapter ellaborates these issues.

2.1 Hearing

This subchapter will describe some details on hearing that is of use when designing hearing aids in addition to some theory on hearing losses and traditional hearing aids. Complementary theory on how the hearing works can be found in for example Chapter 12 from Purves et al. (2004) or Chapter 11 from Kinsler et al. (1999).

2.1.1 Threshold curves

The threshold of audibility is the minimum perceptible of a tone that can be detected at each frequency over the entire range of the ear (Kinsler et al. (1999), p.316). A threshold curve thus shows the sensitivity of the ear over the range of audible frequencies. By studying the curves in figure 2.1 one sees that the ear has a maximum sensitivity around 4kHz, which is expected because of the amplification in the outer ear. The remaining frequencies inside the audible range requires much more power to be perceived. The range of frequency humans can hear lies around 20 Hz up to 20 kHz, where the upper limit decreases as we age.

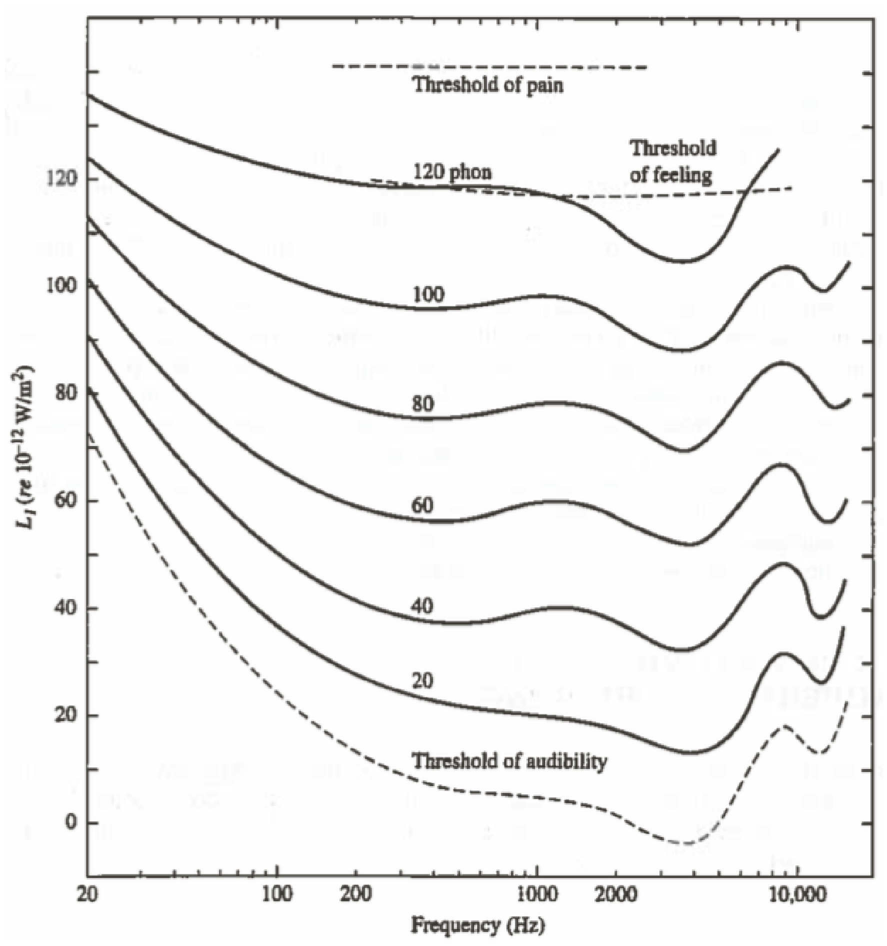


Figure 2.1: Different threshold graphs. (Figure 11.7.1 from Kinsler et al. (1999))

The threshold of audibility for a young undamaged ear is shown as the lower curve in Figure 2.1, and the higher the curve the worse the hearing. The curves for different listeners varies most in the high frequency region and particularly for listeners over 30 year (Kinsler et al. (1999), p. 317). In the figure, L_1 is the intensity level with a reference of $10^{-12} W/m^2$ (IL re $10^{-12} W/m^2$).

The threshold curves are also referred to as *equal loudness curves*, as they show the sound pressure needed to perceive a constant loudness for all frequencies. For information on how the threshold curves can be made, see for example Green (1993).

2.1.2 Audiogram

An audiogram shows the hearing level needed to hear sound for different frequencies. The horizontal axis shows frequencies typically ranging from 125 Hz to 8 kHz which are the most important frequencies of speech. The vertical axis represents the hearing level in decibels (dB HL). 0 dB HL corresponds to the threshold for a person with normal hearing. The vertical axis does not represent the sound pressure levels as for the threshold curve, but the deviation from this threshold.

Figure 2.2 presents the audiogram for two types of hearing loss; noise-induced hearing loss and age-related hearing loss. The audiograms are included to show examples on how the curves can look like for different types of hearing loss and different ages shown in parenthesis. The numbers to the right can be ignored.

2.1.3 Hearing Loss

Damages on the auditory system are normally divided into *conductive* and *sensorineural* damages. The conductive damages are destruction of mainly the outer and middle part of the ear caused by an explosion or other very loud sounds, while sensorineural, or *recochlear*, damages happens over time and usually destroys the neural parts of the inner ear, typically the outer hair cells. The latter damage can be partially fixed using hearing aids and is thus most interesting in this masters thesis.

When the cochlea is exposed for very high sound the hair cells might be broken and in some cases they die. It is the outer hair cells that are most exposed, and if they are dead it leads to a hearing loss of 50-60 dB. If a cell is just hurt, it can still send signals to the brain, but requires a higher sound level in order to work. This can be referred to as a mild hearing loss and is normal for high frequencies when aging. Increasing the quality of sound perception in this case can be done by amplifying the frequencies in the damaged area with a value between 0 and 60 dB.

When a larger amount of hair cells are damaged or dead, one is said to have a moderate hearing loss . When a hair cell is dead it will not send nerve signals to the brain, thus the corresponding frequencies will not be perceived. Once a cell is dead it cannot be recovered, and if a large amount of cells are dead severe cochlear damage occurs and one will not experience an increase of quality by turning up the volume. With severe cochlear damage a Cochlear Implant (see for example Zeng et al. (2008)) can be the solution, replacing the function of the hair cells by sending electrical impulses that are relayed to the hearing nerve.

Two types of sensorineural hearing loss; noise-induced HL and age-related HL are described

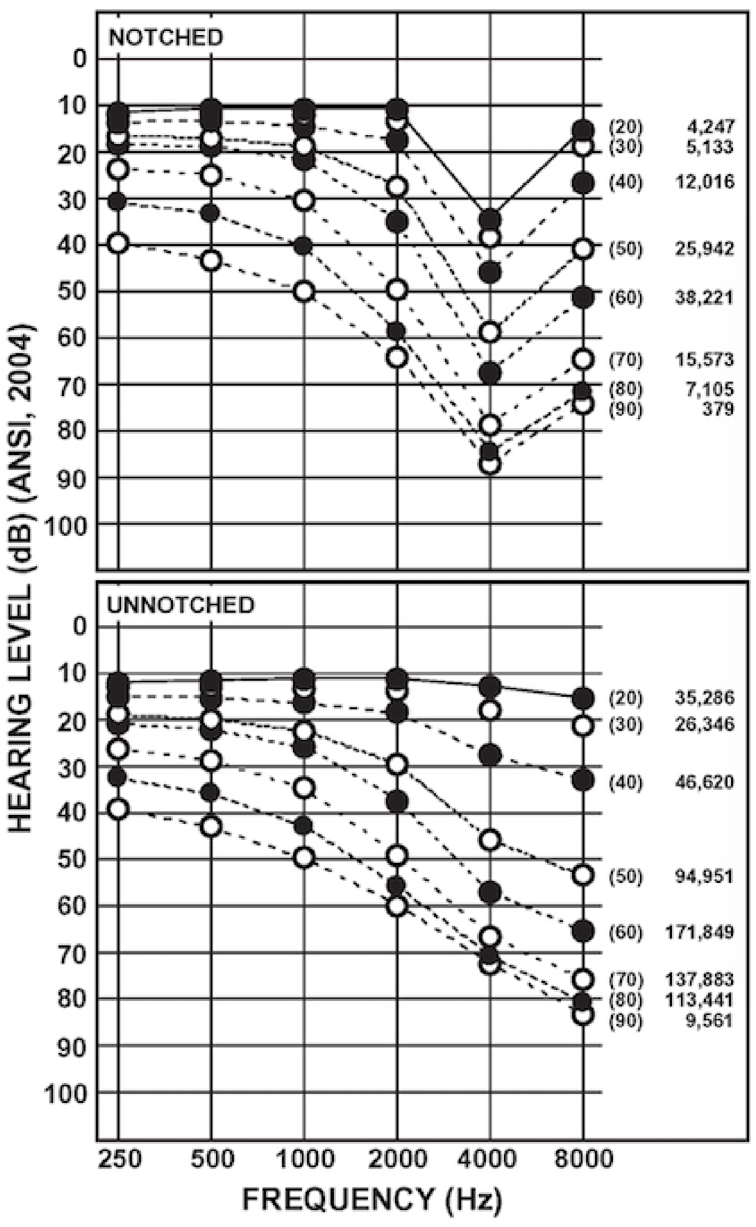


Figure 2.2: Example on audiograms. (Wilson and McArdle (2013))

in the following two sections, in addition to a description on the subjective perception of sound pressure level.

Noise-induced hearing loss

When the ear is exposed for high sound levels the hair cells may be broken, as described earlier. Because of the resonance in the auditory channel the frequencies around 4 kHz are amplified naturally by the ear, thus resulting in a higher sound pressure level into the ear. This means that when a high sound enters the ear it will result in more damage in this specific frequency area. This can be seen from the upper audiogram in Figure 2.2, where the intensities required around these frequencies are much higher than the rest.

Age-related hearing loss

Age-related hearing loss, or *presbycusis*, is a type of hearing loss that occurs naturally when aging. When suffering from presbycusis, the upper range of frequencies are attenuated as can be seen from the lower audiogram in Figure 2.2. From this figure one can also see that the hearing gets worse with the age shown in parenthesis, and that the high frequencies are particularly affected.

Subjective perception of sound pressure levels

Another way to look at hearing loss is to study which sound pressure levels are needed in order to perceive sound. Figure 2.3 illustrates this, where the blue line shows the line for a normal hearing, where input sound pressure level equals the subjective perception at all times. The red line shows the line for a hearing impaired and as can be seen there exists a lower sound pressure level, T1, in order to perceive sound. This is due to the outer hair cells not working properly thus affecting the total perception of weak signals as stated earlier. As is also seen from the figure, the hearing impaired will start hearing as normal when the sound pressure reaches a higher limit, T2. In other words, a hearing impaired will have a narrower dynamic SPL-range compared to the one of a normal hearing.

This can also be explained by the term *loudness recruitment*, which is described as an unusually rapid growth of loudness-level (Allen (1998)). The red-dotted line in Figure 2.3 can be explained by the term *over-recruitment*, where the sound is perceived as louder than normal (Moore (2007)). This needs to be taken into consideration when designing a hearing aid.

Remark: There are reasons to believe that sound pressures over the upper limit, T2, will increase the subjective pressure more than normal like the red dotted line in Figure 2.3, because the outer hair cells do not longer attenuate the strong signals. It is known that older people reacts on high sound pressures, and this might be the reason. These are however just theories.

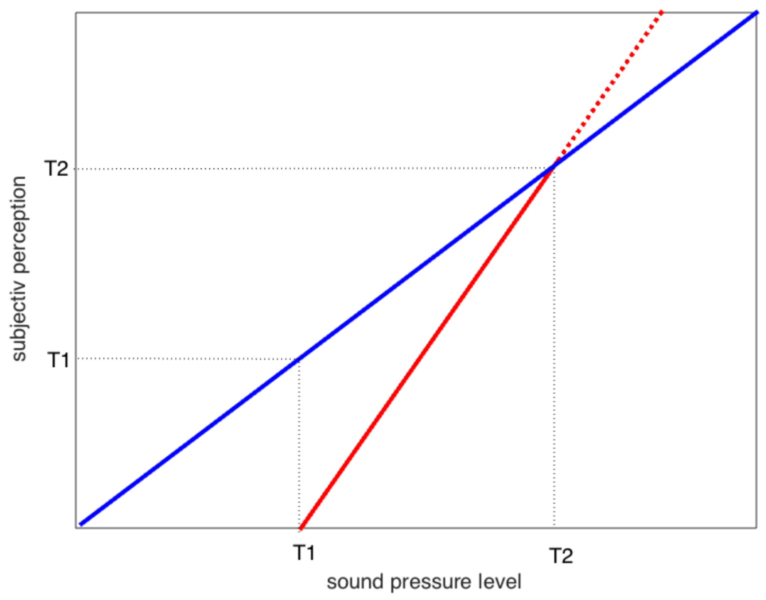


Figure 2.3: Illustration of the subjective perception of sound pressure for a normal (blue line) and a hearing impaired (red line) person. The dotted line illustrates a theory on perception of high sound pressure levels.

2.1.4 Traditional Hearing Aids

Newer hearing aids consists of complex signal processing circuits including wide dynamic range nonlinear amplifiers, multichannel devices, fully digital systems, and special features such as noise reduction algorithms and dual-microphone directionality (Walden et al. (2000)). The performance of hearing aids can further be customized by predicting the optimal frequency response from the users audiogram, where (Byrne and Dillon (1986)) has determined a procedure for this purpose, called NAL. For a thorough explanation on how digital hearing aids works it is referred to Kates (2008).

Plomp (1978) states that every hearing loss can be interpreted as the sum of an attenuation loss (class A) and a distortion loss (class D), where loss A is characterized by a reduction of the levels of both speech signal and noise, and loss D is comparable with a decrease in speech-to-noise ratio. The loss of class D is about one-third of the total hearing loss (A+D) and is the type of loss that cannot be compensated for using a hearing aid. This distortion loss gives difficulties primarily in noise. The article concludes that the most important change for improving the auditory handicap of the hearing impaired listener is to improve the speech-to-noise ratio, and proposes the following ways of doing so:

- Reduction of noise levels
- Reduction of reverberation
- Use of visual speech perception (lipreading)
- Separate microphone near to the speaker
- Application of directional microphones
- Individual fitting of hearing-aid frequency responses
- Compensation of the ear's distortion

where the latter point is the best way to improve the intelligibility by neutralizing the ear's hearing loss of class D. Maybe Deep Learning can provide an alternative to these suggestions.

2.2 Machine Learning

As a big part of this thesis is about designing a deep neural network and training it, some of the utilized machine learning theory is included. Some basics about neurons which are later used in the backpropagation algorithm is presented, as backpropagation is the core of making the neural network *trainable*. The term machine learning can be described as a field of study that gives computers the ability to learn without being explicitly programmed (Simon (2013), p.89).

2.2.1 Neural Networks

Schmidhuber (2015) describes a standard neural network as a network consisting of many simple, connected processors called neurons, each producing a sequence of real-valued activations. Input neurons get activated through sensors perceiving the environment, other neurons get activated through weighted connections from previously active neurons. The next subsection will present some theory on neurons while later in this chapter the activation functions will be covered.

Neurons

As stated, neural networks are made of many connected neurons. A neuron take output from other neurons as input and its output depends on a decision function, y, which in turn depends on the sum of the weighted inputs to the neuron, z. Such a neuron is shown in Figure 2.4.

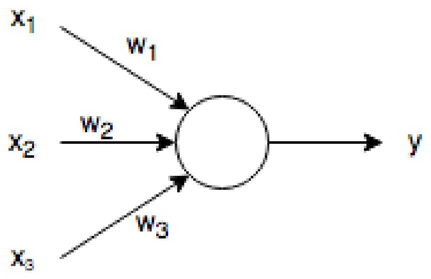


Figure 2.4: A simple neuron.

Here, the weighted inputs can be described as in Equation 2.1, and then the output, y, from the neuron can be described using a decision function such as the one in Equation 2.2, which is a sigmoid function often used because of its nice derivatives. functions are often used because of its nice derivatives making learning easy. The weights, w_i, are being updated during training using the backpropagation algorithm.

$$z = b + \sum_i x_i w_i \tag{2.1}$$

$$y = \frac{1}{1+e^{-z}} \tag{2.2}$$

2.2.2 Deep Learning

Deep learning is becoming a mainstream technology for speech recognition and has won numerous contests in pattern recognition and machine learning (Schmidhuber (2015)). Xu et al.

(2015) shows that deep neural networks gives better results than single neural networks when used in noisy speech, even with the same number of hidden units. If deep learning can work for such speech tasks there are also reasons to believe that it may also be used in hearing aids as an alternative to the traditional methods mentioned in Chapter 2.1.4.

The following subsections will describe some algorithms and functions used when working with deep neural networks and deep learning. For a thorougher description on deep learning and neural networks it is referred to for example Schmidhuber (2015) or Bengio (2009).

Backpropagation algorithm

The backpropagation algorithm is used to estimate the error gradient with respect to the weights in a neural network as shown in Figure 2.5, where all circles corresponds to neurons or "units". The sum of the inputs to a unit in the jth layer, the output layer, is shown as z_j in the figure and is the weighted sum shown in Equation 2.1 earlier in this chapter. In order to achieve the most efficient neural network with ideal weights, one must find the weights that minimizes the error, that is $\partial E/\partial w_{ij}$, where E is a predefined error function.

To compute the error gradient with respect to the weights in the network, one "backpropagates" from the output layer and through the hidden layers. First, the loss function describing the error of the computed output values compared to the desired output must be defined. An example of such an error function is shown in Equation 2.3. Then, one must compute the gradient of this function with respect to the output y_j, as in Equation 2.4.

$$E = \frac{1}{2} \sum_{j \in output} (\hat{y}_j - y_j)^2 \tag{2.3}$$

$$\frac{\partial E}{\partial y_j} = -(\hat{y}_j - y_j) \tag{2.4}$$

Thereafter, the error gradient with respect to z_j is found, describing how the error in one unit changes when the inputs to the unit are changed:

$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1 - y_j)\frac{\partial E}{\partial y_j} \tag{2.5}$$

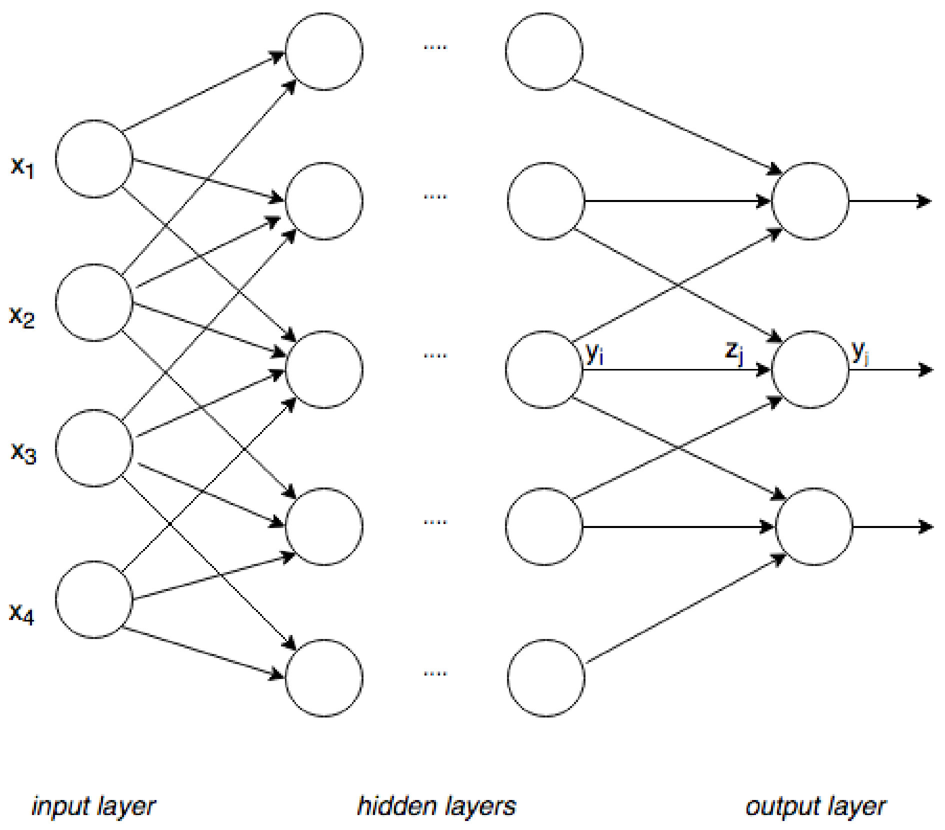


Figure 2.5: Small neural network

The next step is then to find the error gradient with respect to y_i, that is, how the error changes with the layer below. :

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j} \tag{2.6}$$

Then, finally, one can find the error gradient with respect to the weight. The subscripts j and i in these equations corresponds to the layers:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j} \tag{2.7}$$

The computations are made to all the units in each layer, and eventually one will have expressions for the error gradients with respect to all the weights in the network. When the computations are done it is possible to see how changing the weights affects the output error, which is the purpose of this algorithm. For more details on how the backpropagation algorithm works it is referred to Huang et al. (2001).

Dropout

In order to prevent overfitting when using limited training data, dropout can be used. Srivastava et al. (2014) describes the term "dropout" as dropping out units in a neural network, that is, temporarily removing it from the network along with all its incoming and outgoing connections. The dropout should be set as a float, say *p*, with a value between 0 and 1, corresponding to the probability of a unit being dropped. Figure 2.6 shows that training with dropout results in a lower classification error than without.

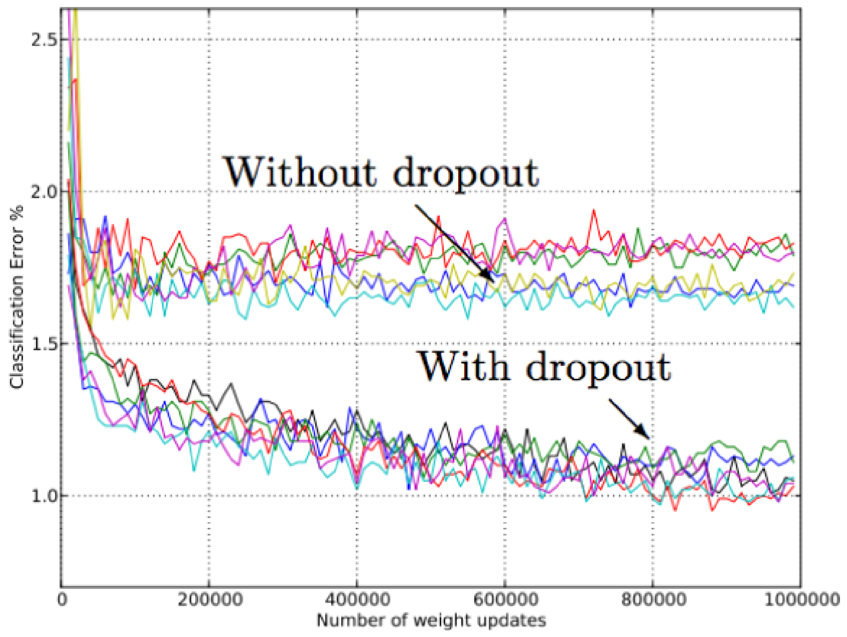


Figure 2.6: Training with and without using dropout (Srivastava et al. (2014))

Activation functions

When designing a neural network one needs to define which activation functions should be used for each layer. A simple function is the linear function where g(x)=x. Other types includes the Sigmoid function shown in Equation 2.8 and Figure 2.7, and the Rectified Linear Unit (ReLU) function shown in Equation 2.9 and Figure 2.8.

$$g(x) = \frac{1}{1+e^{-1}} \tag{2.8}$$

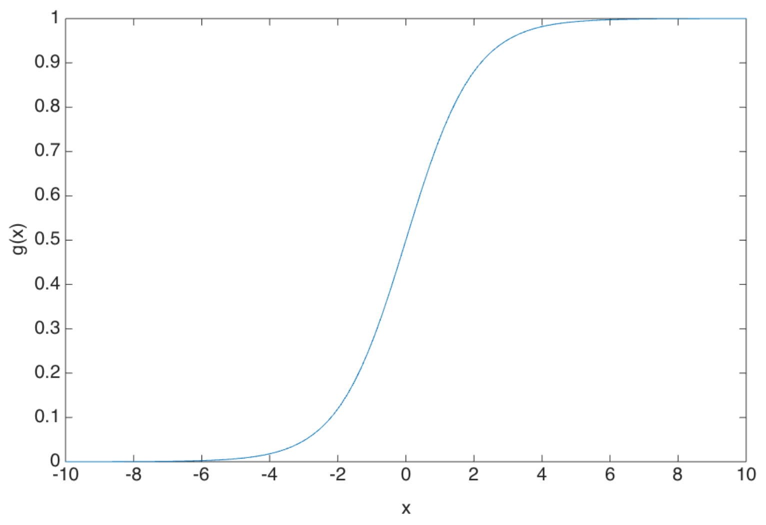


Figure 2.7: Sigmoid function

$$g(x) = \begin{cases} 0 & ,x < 0 \\ x & ,x > 0 \end{cases} \tag{2.9}$$

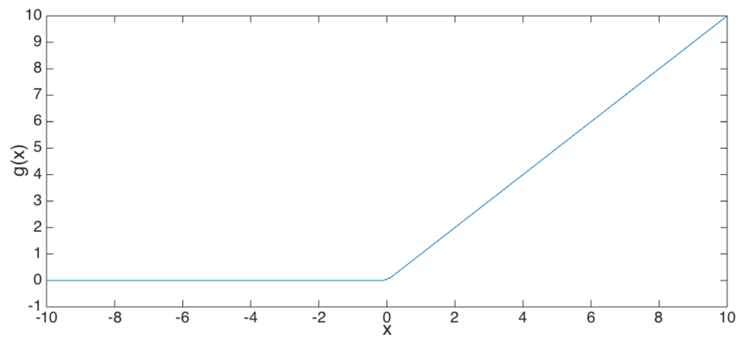


Figure 2.8: Relu function

2.3 Signal Processing

This chapter includes some used theory on general signal processing. The fourier transform is used for representing the signal in frequency domain rather than time domain, and the windowing is used to extract small batches of the signal to input to the neural network.

2.3.1 The Discrete Fourier Transform (DFT)

In order to express a signal in the frequency domain, a Discrete Fourier Transform (DFT) can be used. The DFT of a finite-length sequence of length N is defined by

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}, k = 0, 1, ..., N-1 \tag{2.10}$$

If the signal x[n] is real and consists of k=256 samples, N is also 256 and we can compute some values of X[k]:

$$X[0] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi 0n/N} = \sum_{n=0}^{N-1} x[n], k = 0, 1, ..., N-1 \tag{2.11}$$

$$X[1] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi n/N}, k = 0, 1, ..., N-1 \tag{2.12}$$

$$X[128] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi 129n/N} = \sum_{n=0}^{N-1} x[n](-1)^n, k = 0, 1, ..., N-1 \tag{2.13}$$

$$X[255] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi(256-1)n/N} = \sum_{n=0}^{N-1} x[n]e^{j2\pi n/N}, k = 0, 1, ..., N-1 \tag{2.14}$$

Equation 2.14 uses the fact that $e^{-j2\pi n} = 1$. This means that X[1] is the complex conjugate of X[255] as is also the case for X[2] to X[127] and X[254] to X[129]. Thus, in order to restore x[n] there is only need for X[1] to X[127] in addition to X[0] and X[128] because of their inequality. This gives a total of 129 variables.

The inverse DFT (IDFT) is given by

$$x[n] = \frac{1}{N}\sum_{n=0}^{N-1} X[k]e^{j2\pi kn/N}.n = 0, 1, ..., N-1 \tag{2.15}$$

From Equation 2.10 and 2.15 one can see that the only difference is the negative sign on the exponent of *e* and the scaling factor 1/N. This means that the inverse DFT can also be described as the complex conjugate of the DFT scaled by 1/N. This can be exploited to make faster computations as is done in the Fast Fourier Transform algorithm.

2.3.2 The Fast Fourier Transform (FFT) algorithm

A direct computation of the N-point DFT requires computational cost proportional by N^2.This cost can be minimized to be proportional by $Nlog_2N$ by using the Fast Fourier Transform (FFT) algorithm. The FFT algorithm exploits the periodicity and complex conjugate symmetry properties of W_N^{kn}, shown in Equation 2.16 and 2.17, respectively.

$$W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n} \tag{2.16}$$

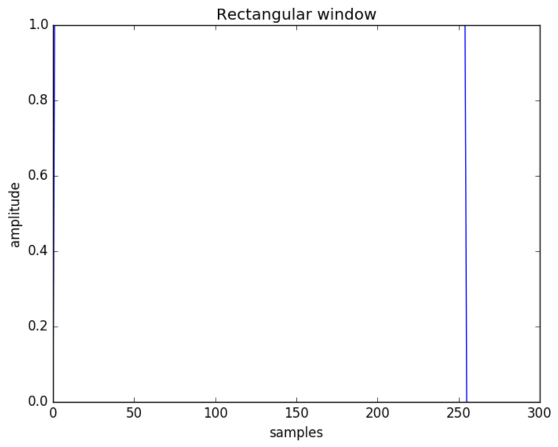
$$W_N^{k(N-n)} = W_N^{-kn} = (W_N^{kn})^* \tag{2.17}$$

As can be seen from these equations, the FFT separates x[n] into even and odd subsequences. As for the Direct Fourier Transform, the Fast Fourier Transform will also return complex conjugates if the input is real. Further information on the fourier transform can be found in Manolakis and Ingle (2011).

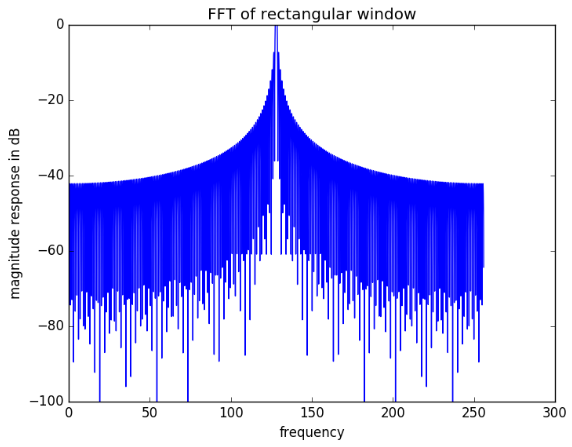
2.3.3 Windowing

Short time spectral analysis is the most common way to characterize a speech signal because its spectrum is nearly constant for small time sequences. When working with speech signals, time periods between 5 and 100 ms are small enough and should be used, because longer time periods makes the signal characteristics change to reflect the different speech sounds being spoken (Hasan et al. (2004)).

Thus, in order to get good results from the FFT, the algorithm should be applied on small batches of the signal. This can be done by extracting frames, and apply the FFT on each frame. However, if one were to extract small batches of the input signal before taking the FFT, the resulting spectrum will be smeared due to spectral leakage. This is because extracting small batches of the signal corresponds to multiplying the signal with a rectangular window, such as the one shown in Figure 2.9 with the attached frequency response.



(a) Rectangular window

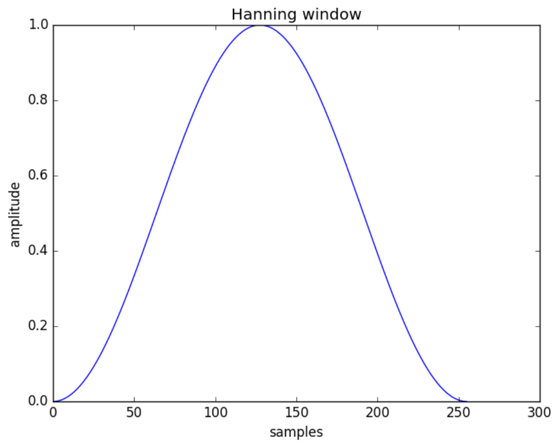


(b) FFT of rectangular window

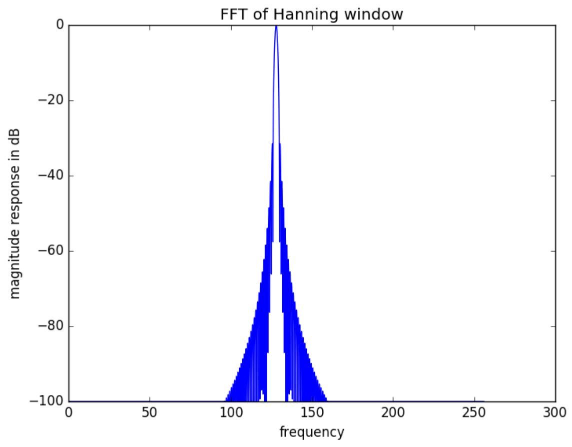
Figure 2.9: Rectangular window

As is known, multiplication in the time domain corresponds to convolution in the frequency domain, which means that the frequency spectrum of the input signal would be convolved with the frequency response of the rectangular window. Instead, one can use a different window function with a better frequency response, such as the Hanning window. The Hanning window is widely used and is defined as:

$$w(n) = 0.5 - 0.5cos(\frac{2\pi n}{M-1}) \quad ,0 < n < M-1 \tag{2.18}$$



(a) Hanning window



(b) FFT of Hanning window

Figure 2.10: Hanning window

Figure 2.10 shows the Hanning window and its frequency response, and one can clearly see that the side lobes are much smaller than the ones of the rectangular window. This means that the smearing effect when it is convolved with the frequency spectrum of the input signal will be much smaller.

In order to preserve the complete signal, overlapping Hanning windows can be used as shown in Figure 2.11, where the size of each window is 256 samples.

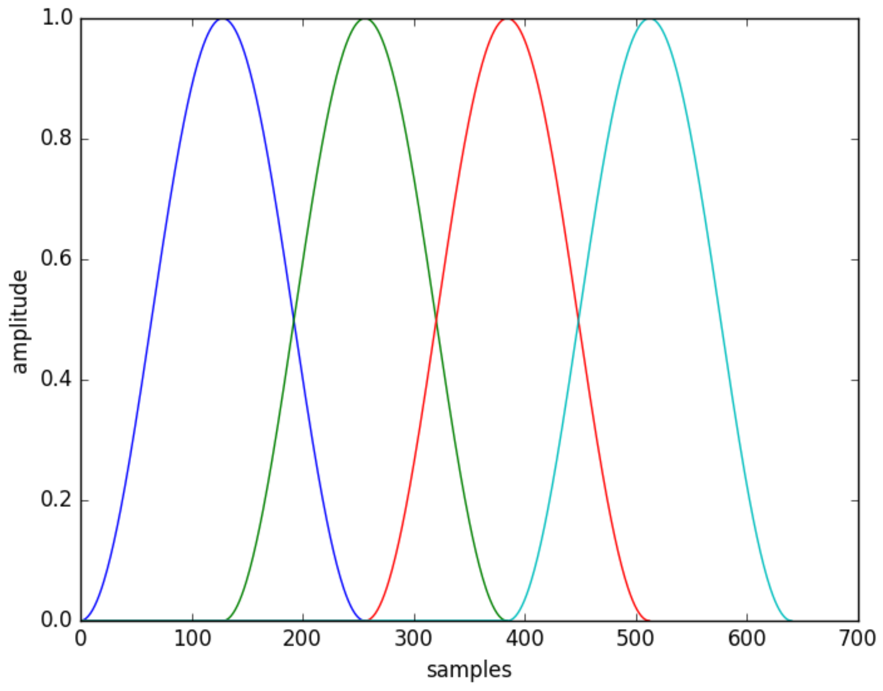


Figure 2.11: Overlapping Hanning windows

Chapter 3

Realization

The system is run on a Unix-machine with 12 processing cores and 32 GB of random access memory (RAM). The programming language used is Python version 3.5 together with NumPy and SciPy. For making the deep neural network a library called Keras, developed by Chollet (2015), is used on top of Theano, developed by Theano Development Team (2016). As is stated in the Keras documentation, "Keras is a minimalist, highly modular neural networks library, written in Python". From the software section of Jones et al. (2016) Theano is described as a Python library that lets you define, optimize and evaluate mathematical expressions easily, especially numpy arrays which is used a lot in this thesis.

The utilized speech files are found from the TIMIT database, where all 4620 files from the training set are used to create the speech files as described in Chapter 3.2.1. The program Audacity is used when looking and listening to the speech signals. All the files mentioned throughout this chapter are to be found in the appendix.

3.1 Overall system

The system contains a deep neural network and a hearing loss simulator, in addition to multiple signal processing blocks. The multiple signal processing blocks before the deep neural network have the task to extract the short time spectrum of the speech signal, because of the small variations compared to the waveform of the signal. This was described in Chapter 2.3.3. The hearing loss simulator is applied after the deep neural network is because a hearing loss in reality comes *after* any hearing aid, thus the hearing aid should *pre-compensate* for the hearing loss happening inside the ear and brain.

The following two subchapters presents the system design when used for training of the neural network and as thought implemented in a hearing aid.

3.1.1 System used when training the neural network

The system during training is shown in Figure 3.1, where all the blocks are described later in this section. The most important blocks are the DNN-block and the HLS-block, which describes the deep neural network and the hearing loss simulator, respectively. The first trainings are done without the HLS-block, that is, without an applied hearing loss in order to optimize the different system parameters for recovering the clean speech data. When these parameters are found, the system is advanced by adding hearing losses to the signal.

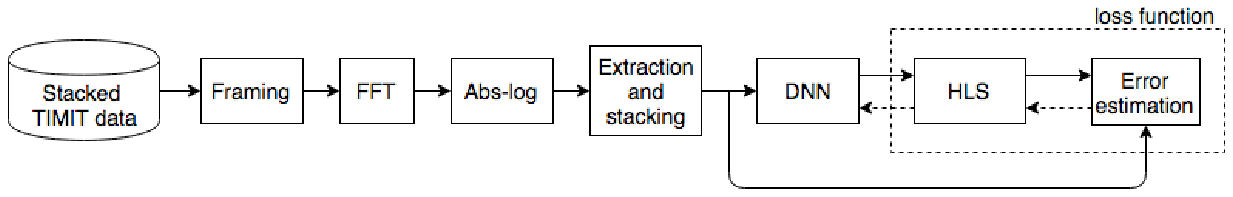


Figure 3.1: The system during training of the deep neural network.

`loadFiles.py` is used for creating input files to the system, that is, for stacking TIMIT data in multiple ways as is further described in Chapter 3.2.1. The script `main.py` is where all the blocks are called from. Each block will be further described later in this chapter.

During the training of the network it is useful to listen to the results in order to do subjective quality tests in addition to looking at the validation losses. To do this, the structure shown in Figure 3.2 is used.

3.1.2 System for use as a hearing aid

Figure 3.3 shows the system if it were to be used as a hearing aid. The difference from the training structure is that the block consisting of the deep neural network, the DNN-block, is already

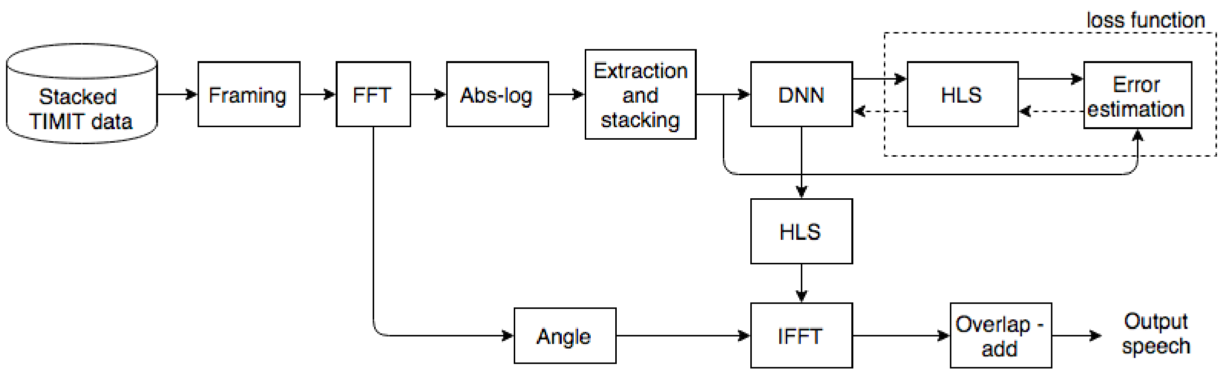


Figure 3.2: The system used when listening to the output

trained on the users specific hearing loss resulting in the loss function being removed. In order to restore the signal, the phase is being stored from the FFT computation and used before computing the IFFT, described in Chapter 3.2.3 and 3.2.6. The output speech in the figure should then be the precompensated speech in accordance with the listeners subjective hearing loss.

As this thesis is just a small part of a bigger research on deep learning for use in hearing aids, this implementation was not tested on real persons because the hearing loss simulator was not finished.

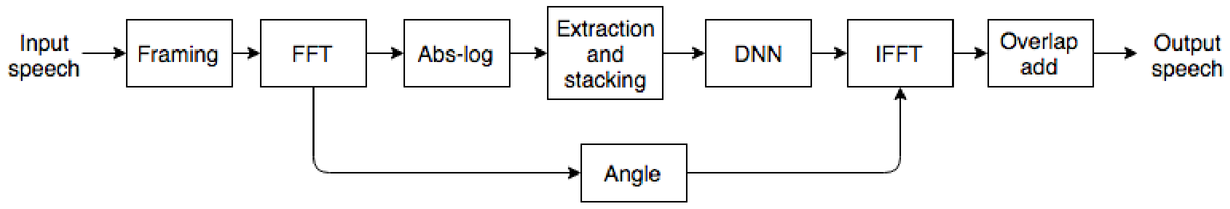


Figure 3.3: The system working as a hearing aid

3.2 General Signal Processing

The following sections describes how the TIMIT speech are processed before used in the deep neural network. This includes how the files are stacked to form one input file as well as how the waveform of the input speech is converted to the short time spectrum.

3.2.1 Preprocessing of speech files

The speech files used for training and testing are from the TIMIT database and contains a number of small phrases spoken by different persons. The training data set consists of 4620 waveform audio file format (.wav) files and are sampled with a sample frequency of 16 kHz. The data files are concatenated before use, so that one big input file can be read into the neural network instead of many small ones, saving some time. As stated earlier in this chapter, the script `loadFiles.py` is used for the preprocessing of the speech files.

When working with python and the package *scipy.io.wavfile*, one must keep in mind that the function *wavfile.write(filename, fs, file)* assumes that the input type is an array of integers. For audio signals, the type *int16* should be used, and the numbers can then range from the minimum and maximum int value. Such a signal is shown in Figure 3.4, and one can see that all information is maintained (note that Audacity scales the signal in order to be in the range from -1 to 1).

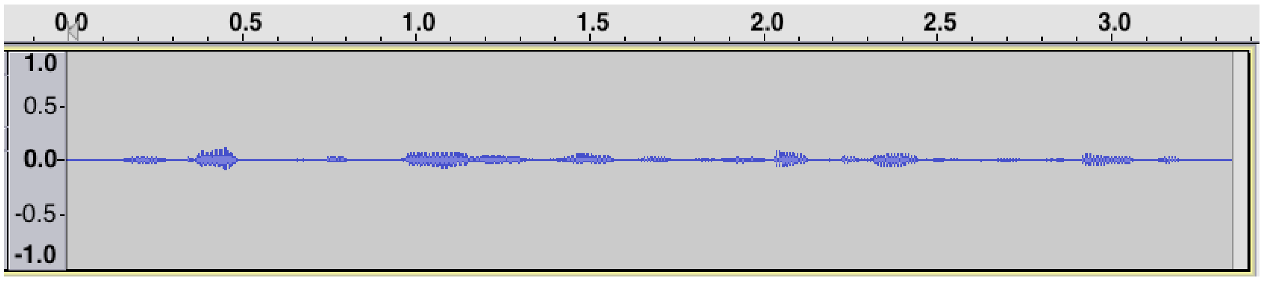


Figure 3.4: The file "mgak0_sx226.wav" from the TIMIT training data

However, if the type of the input file is *float wavfile.write(filename, fs, file)* assumes that the numbers lies in the range between -1 and 1, thus clipping all values with bigger absolute value than 1. This is shown in Figure 3.5, where the input type to *wavfile.write* is float. When listening to this signal one can clearly hear that it is noisy and much louder than the original signal.

As Keras defines the input to the deep learning model as a *float32*, all signals should be transformed to *int16* before written to any file. The *scipy.io.wavfile* function is not very well documented, but for a closer look one can read the source code available on Github which can be found from Jones et al. (2001).

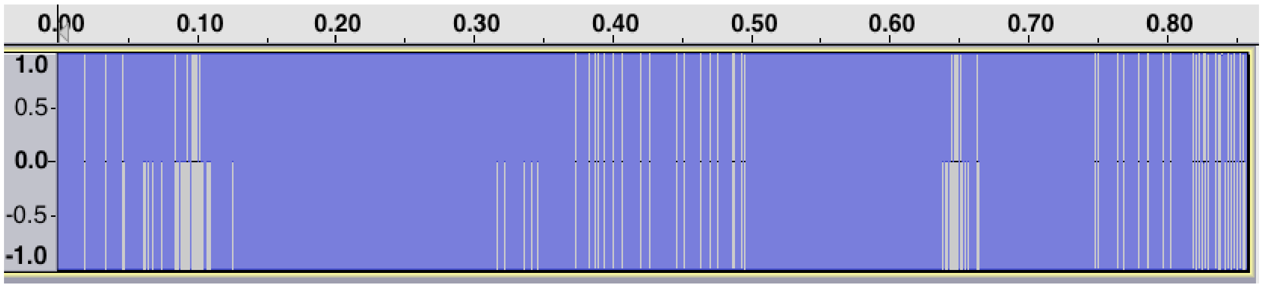


Figure 3.5: Resulting signal of wavfile.write when float type is used on the file "mgak0_sx226.wav"

Silence in signal

As one can see from Figure 3.4, the speech signal consists of silence between the utterances. A possible error source could thus be that the deep neural network is trained for the silence rather than for the actual speech because of biased data. To test if this is true the signal, which contains values from around -4000 to 4000, is manipulated: First, it is *trimmed* for amplitudes smaller than 30, that is, these values are removed from the beginning and end of the signal. As an example, the signal from Figure 3.4 is trimmed and the result is shown in Figure 3.6, where 3.765 of 53.658 samples are removed (7%). Values under 30 was tested but did not remove sufficient silence.

Then, all the amplitudes smaller than a value of 5 was removed. This was done because of the silence between every utterance could also possible cause biased data. It was also tested for some amplitudes greater than 5 without any good results. The signal in Figure 3.7 shows the resulting file, where 11.168 of 53.658 samples are removed (20%).

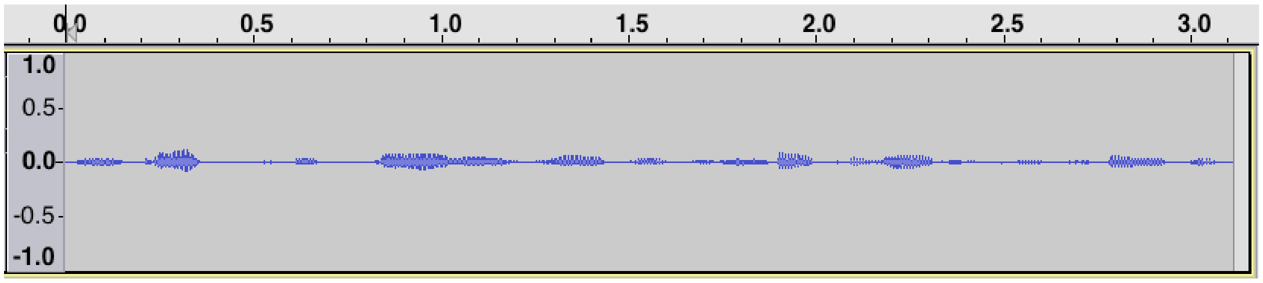


Figure 3.6: The file "mgak0_sx226.wav" trimmed for amplitudes smaller than 30

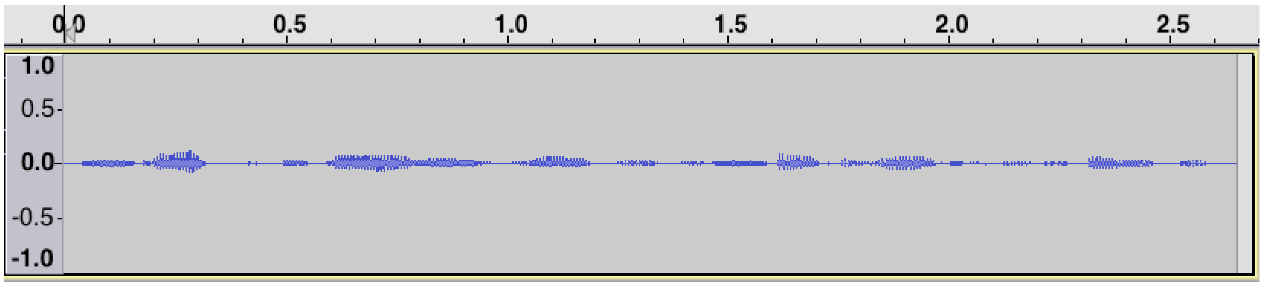


Figure 3.7: The file "mgak0_sx226.wav" without amplitudes smaller than 5

3.2.2 Framing

In order to avoid spectral leakage as described in Chapter 2.3.3, a window function needs to be applied when extracting batches of the data. The same chapter also describes the good frequency response of the Hanning window, which is the reason for why it was chosen in the implementations.

The input files are sliced in order to be a multiplication of the number of samples in the Hanning window, denoted N. Thereafter, the 1-dimensional sliced array is arranged into a 2-dimensional array with the number of columns being equal to N before each row is multiplied with the Hanning window described in Chapter 2.3.3. This is done in the function `applyHanning (inputSignal, N, windowLength)` in the functions.py script in the appendix. It should be mentioned that the input samples from 0 until N/2 will only be a part of one window resulting in some damping under the reconstruction of the signal. This yields also for the last N/2 samples.

Using a window of length 256 samples when the sampling frequency is 16 kHz corresponds to around 16 ms. In order to test how changing the length of this window affects the results, both 128 and 512 samples, corresponding to 8 and 30 ms, are tested.

3.2.3 FFT

The FFT algorithm is applied to the 2-dimensional array using the numpy fft package. In order to do the calculations efficient on every row, NumPys *apply_along_axis* is used. The resulting array is complex, but as the ear is not very sensitive to small phase changes, the phase can be left out when training the deep neural network. Then it should be re applied before taking the IFFT for restoring the signal.

As the input signal is real, the output of the FFT will be mirrored as explained in Chapter 2.3.1. This means that if the input length to the FFT is 256 samples, the result will be a mirrored signal around 128 and only the columns from 0 to 128 should be stored.

3.2.4 Abs-log

Because the ears perception of sound is logarithmic, meaning we can distinguish low frequencies from each other better than high frequencies, the signal is changed to be logarithmic as well. After the FFT-block the absolute values are found, and the signal will then range from 0 up to around 4000. Because the logarithm of 0 is undefined, 1 is added to the signal before applying *numpy.log10* in order to avoid errors.

3.2.5 Extraction and stacking

The output of the abs-log block is now a 2-dimensional array of real positive signals with the number of columns corresponding to the Hanning window length. When used in the deep neural network each row is input at a time, resulting in the number of training and validation samples being the total number of rows in the different arrays. In order for the network to "see" more of the signal than only the window it shall be trained on, each row is stacked with the 4 surrounding frames resulting in a total of 5 frames in each row, where the frame to train on is in the middle.

The first and last row in the signal will have no surrounding frames at the left and right side, respectively, and the second first and second last row will just have one frame here. This means that these four rows must be padded differently and an array of zeros is thus used for stacking.

3.2.6 IFFT

In order to obtain the output speech rather than the 2-dimensional array which is output from the deep neural network, the inverse fast fourier transform (IFFT) is applied in addition to the overlap-add block as shown in Figure 3.2 and 3.3. Before taking the IFFT, the signal is first mirrored to reconstruct the second half of the signal which was removed in the FFT-block.

Second, the amplitude is found by taking the inverse of the logarithm as shown in Equation 3.1. Third, the full signal is obtained by including the phase as was mentioned in Chapter 3.2.3. This is done as in Equation 3.2. The *numpy.ifft* can then be applied, and the real part of the output will then correspond to the input to the FFT.

$$amplitude = 10^{abslog} \tag{3.1}$$

$$angleAndAmplitude = amplitude \cdot e^{j \cdot phase} \tag{3.2}$$

3.2.7 Overlap-add

The last step in order to construct the output speech is to overlap the rows, that is, doing the opposite of what is done in the framing-block. This is done in the function `do_ifft(abslog, storedPhase)` in the recoverSignal.py script. As mentioned in Chapter 3.2.2, the first and last N/2 samples will only be included in one window resulting in a somewhat damped signal here.

3.3 Deep Neural Network (DNN)

The performance of the deep neural network depends on various parameters such as the batch size, the number of epochs and the number of classes in addition to the design of the neural network. The network itself depends on its width and depth which are set by the number of layers and the number of nodes in each layer, and the activation functions in each layer. The training is done using 30 epochs and 10 classes. The rest of the parameters are tested with different values in order to find the optimized ones, and the results are shown in Chapter 4. Dropout is not applied to all tests because its importance as described in Chapter 2.2.2 was not realized until later.

The deep neural network is created in the script called main.py as a `model`. `Dense` is used when creating a new layer in a sequential model, and as an example, `Dense(512, activation ='sigmoid')` will create a layer consisting of 512 nodes where the activation function is of the type sigmoid, described in Chapter 2.2.1. Sigmoid is used in the hidden layers because it was used in Xu et al. (2015), as well as a linear activation function for the output layer. `Dropout` is used for adding a dropout function to this layer, where `Dropout(0.2)` will drop 0.2 percent of the input units as is also described in Chapter 2.2.1.

In the function `model.compile` one decides which loss function to use by setting the `loss` parameter. Keras calls these functions *objectives* and examples on such functions are `mean_squared_error`, `mean_absolute_error` and `binary_crossentropy`. However, in this thesis a new loss function is defined in order to being able to apply hearing losses inside this function. This function is further described in Chapter 3.4.

The `model.fit` function is where the model is trained. In the function call the training data, validation data, batch size and number of epochs are defined. In addition to this it is possible to add a list of callbacks to be executed at different times of the training. In this thesis the callback `printOutput` is defined in order to being able to look at and listen to the output of the model, as described in Chapter 3.1.1 and Figure 3.2. The callback is then called after the `model.fit` function is done, but it is also possible to make more automatic executions by using quantities from the `model.fit` log. Examples on such quantities are `on_epoch_end, on_batch_begin` and `on_batch_end`.

3.4 Hearing Loss Simulator

The hearing loss inside the loss function from Figure 3.1 and 3.2 is being defined in the `model.compile` function as the `loss` parameter. The self defined function being used is defined as `lossFunction` inside the ownObjectives.py script. This function includes the hearing loss to apply in addition to a "standard" loss function. The standard loss function used is the squared difference between the predicted and the true value.

First the system is run without applying any hearing losses in order to find which parameters and structures described in Chapter 3.3 optimizes the system performance when trained on the speech data. Later, some simple hearing losses are applied to this optimized system. More losses was not applied due to the time limitations and the limited knowledge on the Theano framework.

The following two subsections describes possible designs of the hearing loss simulator. There were only time to implement one of these and the latter design was prioritized and tested because of its somehow simpler implementation.

3.4.1 Simulation in time domain

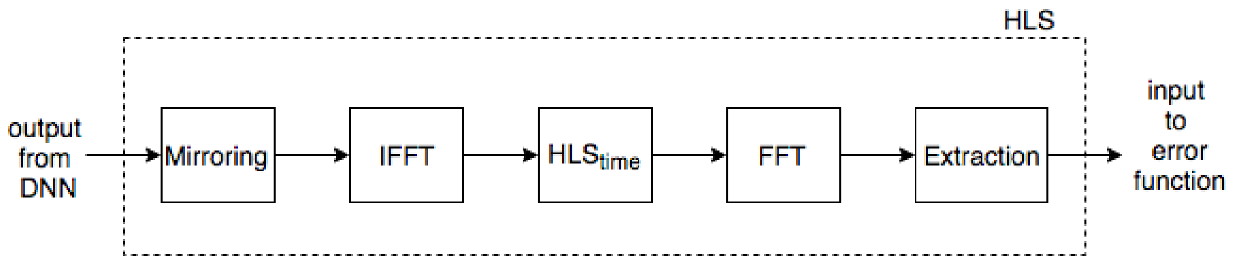


Figure 3.8: Hearing loss simulator in time domain.

Figure 3.8 shows a possible realization of the HLS block when working in the time domain. Such a hearing loss simulator can be made by doing the opposite of what is done in the project "Hearing Aid Processes Simulated in Matlab" (Swendgaard (2015)). In this project, a compressor is made in addition to a parametric equalizer, thus in order to simulate a hearing loss rather than a hearing aid the compressor should be changed to an expander. The parametric equalizer is very flexible and can be used as is, by changing the gain parameters to negative values as described in the thesis.

A more complex hearing loss simulation is described in the masters thesis "Real-time Simulation of Reduced Frequency Selectivity and Loudness Recruitment Using Level Dependent Gammachirp Filters" (Berheussen (2012)). As the title implies, the simulation is done in real-time and is written using the C programming language, so in order to implement it to this system it should be re-written to python.

As is seen from Figure 3.8 the structure requires additional IFFT and FFT blocks in order for the HLS$_{time}$ block to work in the time domain. In order to take the IFFT the signal should first be mirrored as described in Chapter 3.2.6, and after taking the FFT half the signal should be extracted as described in Chapter 3.2.3 and 2.3.1.

3.4.2 Simulation in frequency domain

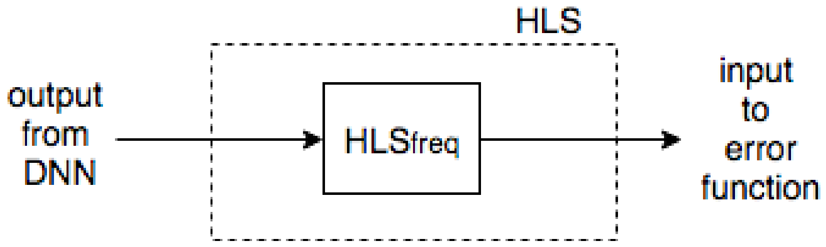


Figure 3.9: Hearing loss simulator in frequency domain.

Figure 3.9 shows the structure of the HLS block if it is to be used in the frequency domain. Compared to the structure in Figure 3.8, this requires less blocks and no transformation of the signal from frequency to time and vice versa. Because of its somewhat simpler form, this is the structured implemented in the system and tested with some simple hearing losses, such as damping and expansion. As mentioned there was unfortunately not enough time to develop more complex functions.

Chalupper and Fast (2000) shows how simulation of hearing loss can be done in the frequency domain by using expansion in addition to quantization. In this thesis the hearing loss is applied after the signal has passed the deep neural network so by quantifying the signal at the end, the output will also be quantized no matter how good the pre compensation by the network is. The expansion is however useful to test. Figure 2.3 from Chapter 2.1.3 shows how the expander should work, where pressures below a threshold T1 should output zero and the pressures within the thresholds T1 and T2 should be expanded. However, because of the time limitations the expander is just realized as $y = 2 \cdot x$ to begin with. In addition to the simple expander, a uniform damping of the signal is also tested.

Chapter 4

Results

In this chapter, the results of the different trainings are presented. All trainings were run with 30 epochs, and the result for epoch 1, 15 and 30 are included in the tables, where "loss" is the training loss and "val_loss" is the validation loss. The reason for why the training losses are in general higher than the testing loss is because the training loss is the average loss over each batch of training data. As the model changes over time, the loss over the first batches of an epoch is generally higher than over the last batches. The testing loss is on the other hand computed using the model as it is at the end of the epoch. In addition to this, regularization mechanisms are turned off at testing time.

4.1 Preprocessing of the input file

The following sections describes how the results are affected when changing the input file regarding both content and length, in addition to changes of the window length used when extracting batches of the signal into arrays as described in Chapter 3.2.2.

Pre-stacking of files

The speech files used in this thesis are as stated in Chapter 3.2.1 from the TIMIT-database. The used training set of the database consists of 4620 files of around 3-5 seconds. Before being used in the system, the TIMIT files are concatenated to one file in multiple ways: The first file consists of the files stacked consecutively, while the second file stacks them in random order to spread the speakers. The third file consists of trimmed data as explained in Chapter 3.2.1 and the last file removes all silence as was also explained in Chapter 3.2.1. The two last files was used in order to test if the network was trained on silence rather than speech.

For these tests, it was used 40.000.000 training samples and 5.000.000 validation samples. The model consisted of two hidden layers, each containing 512 units and the sigmoid activation function. The batch size was 128 samples, the number of classes was 10, the number of epochs was 30 and the window length was 256 samples. The output layer consisted of 129 units. There was used no dropout function, but it should not affect the relative result between the different input files. The results are shown in Table 4.1.

Table 4.1: Training results using different input data

	Epoch 1		Epoch 15		Epoch 30	
Input data	loss	val_loss	loss	val_loss	loss	val_loss
Original	0.2233	0.1388	0.0571	0.0591	0.0345	0.0391
Randomized order	0.2202	0.1701	0.0569	0.0537	**0.0345**	**0.0339**
Trimmed	0.2278	0.1414	0.0557	0.0577	0.0347	0.0443
No silence	0.2059	0.1294	0.0656	0.068	0.0439	0.039

Training and validation size

When testing the number of training and validation samples used, the file consisting of data in randomized order was used due to the results in the previous subsection. The rest of the model data was the same as for the previous test. The results are shown in Table 4.2.

Table 4.2: Training results using different input size

		Epoch 1		Epoch 15		Epoch 30	
Training samples	Validation samples	loss	val_loss	loss	val_loss	loss	val_loss
2.000.000	2.000	0.5864	0.722	0.1529	0.3568	0.1201	0.3622
40.000.000	200.000	0.2206	0.1657	0.0571	0.0561	0.0356	0.0385
70.000.000	5.000.000	0.1845	0.108	0.0459	0.0397	0.0284	0.0274
150.000.000	10.500.000	0.1411	0.0906	0.0279	0.0302	**0.0169**	**0.0186**

Window Length

The window length is changed and the results are shown in Table 4.3. 128, 256 and 512 samples were tested, corresponding to 8, 16 and 32 ms when using an input sampling rate of 16 kHz. When using 128 samples the training duration was the longest, where each epoch used around 49 seconds compared to 19 seconds for 512 samples and 26 seconds for 256 samples. Another approach could have been to change the sampling frequency of the input signal but this was not tested.

Table 4.3: Training results using different window lengths

	Epoch 1		Epoch 15		Epoch 30	
Window length	loss	val_loss	loss	val_loss	loss	val_loss
256	0.2206	0.1657	0.0571	0.0561	0.0356	0.0385
512	0.3485	0.2387	0.1055	0.1137	0.0884	0.1195
128	0.1403	0.0699	0.0162	0.0191	**0.0076**	**0.0084**

4.2 Deep neural network

This section presents the results from the trainings done with different model parameters. The "standard" model consisted of two hidden layers with 512 units in each layer, 129 units in the output layer, a batch size of 128 and a window length of 256. The activation functions were "sigmoid" for the hidden layers and "linear" for the output layer. There was not used a dropout function but this should be considered done in future trainings due to what was described in Chapter 2.2.2. The number of training samples was 150.000.000 and the number of validation samples was 10.500.000. This corresponding to around 2.6 hours and 10 minutes, respectively.

Batch size

The batch size is changed and the results are shown in Table 4.4. It should be mentioned that the batch size of 128 samples used 26 seconds per epoch while a size of 64 samples used 48 seconds per epoch. 129 samples was tested because that was the same as the number of output samples.

Table 4.4: Training results using different batch sizes

	Epoch 1		Epoch 15		Epoch 30	
Batch size	loss	val_loss	loss	val_loss	loss	val_loss
128	0.2206	0.1657	0.0571	0.0561	0.0356	0.0385
129	0.2273	0.2344	0.0596	0.0625	0.0415	0.0554
64	0.1779	0.1145	0.0432	0.0668	**0.0266**	**0.0364**
256	0.2901	0.1864	0.0744	0.0952	0.0585	0.0714

Width and depth of neural network

Some different structures of the neural network was tested. Deeper and wider network could have been tested, but due to the limit of the training data this was not done. There is not use for a very big network when training on small data sets. During this testing, the dropout was set to 0.2, that is, 0.2 percent of the input units was dropped randomly during training. The results are shown in Table 4.5.

Table 4.5: Training results using different neural networks

	Epoch 1		Epoch 15		Epoch 30	
Layers and units	val_loss	loss	val_loss	loss	val_loss	loss
512-512-129	0.1679	0.1208	0.0726	0.0585	0.0641	0.0515
1024-512-129	0.153	0.0901	0.0642	0.0614	0.0568	0.0465
1024-1024-129	0.1532	0.089	0.0614	0.0547	**0.0515**	**0.0382**
512-512-512-129	0.2153	0.1381	0.0912	0.0765	0.0808	0.0671

Activation functions

Other activation functions than sigmoid and linear was tested in the beginning of the work due to a mistake, and the results were not good. The sigmoid function should be used in the hidden layers and the linear function in the output layer (Xu et al. (2015)).

Dropout

The network was trained using different dropouts and the result is shown in Table 4.6. The used model consisted of 2 hidden layers with 1024 units in each layer. The output layer had 129 units.

In order to see the affection over time, the results were plotted where the training losses are shown in Figure 4.1 and the validation losses in Figure 4.2.

Table 4.6: Training results using different dropouts

	Epoch 1		Epoch 15		Epoch 30	
Dropout	loss	val_loss	loss	val_loss	loss	val_loss
None	0.13	0.0748	0.0195	0.026	0.0091	0.0058
0.1	0.1418	0.1012	0.0485	0.0436	0.0334	0.0202
0.2	0.196	0.1232	0.086	0.0692	0.077	0.058
0.5	0.1532	0.089	0.0614	0.0547	0.0515	0.0382

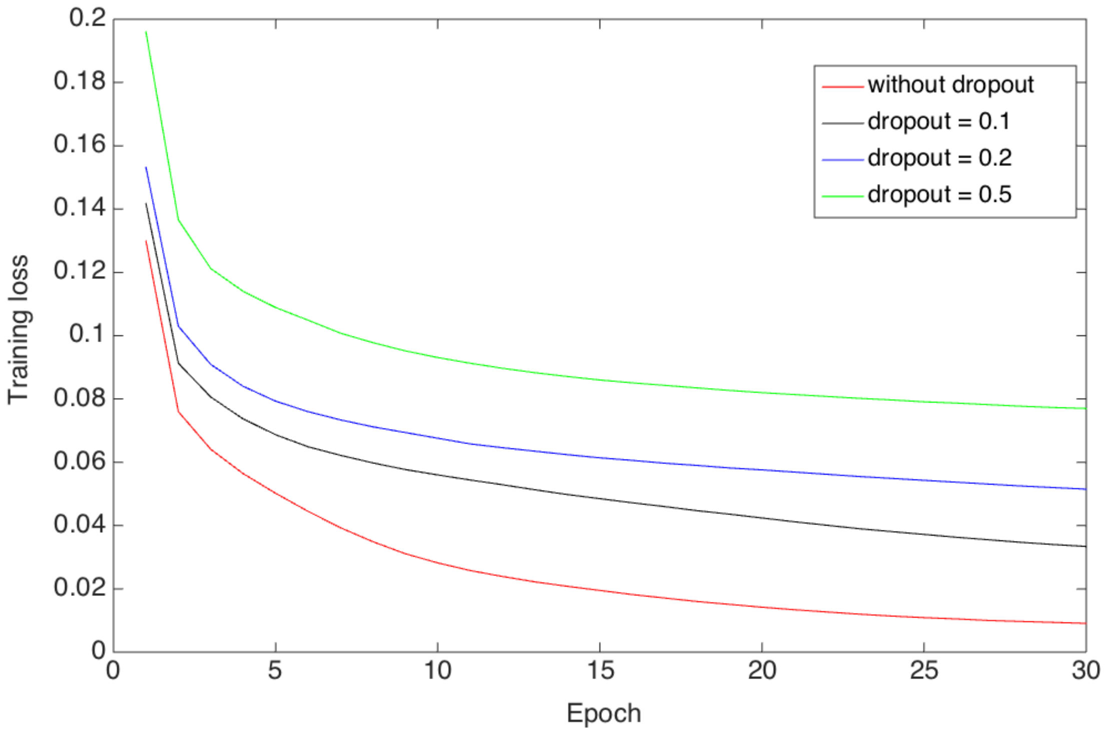


Figure 4.1: Training losses for different dropout values

4.3 Hearing loss functions

As mentioned in the previous chapter, there was not time to test the loss function sufficiently. An uniform damping was applied, and a simple expansion. Both these functions are just a scaling

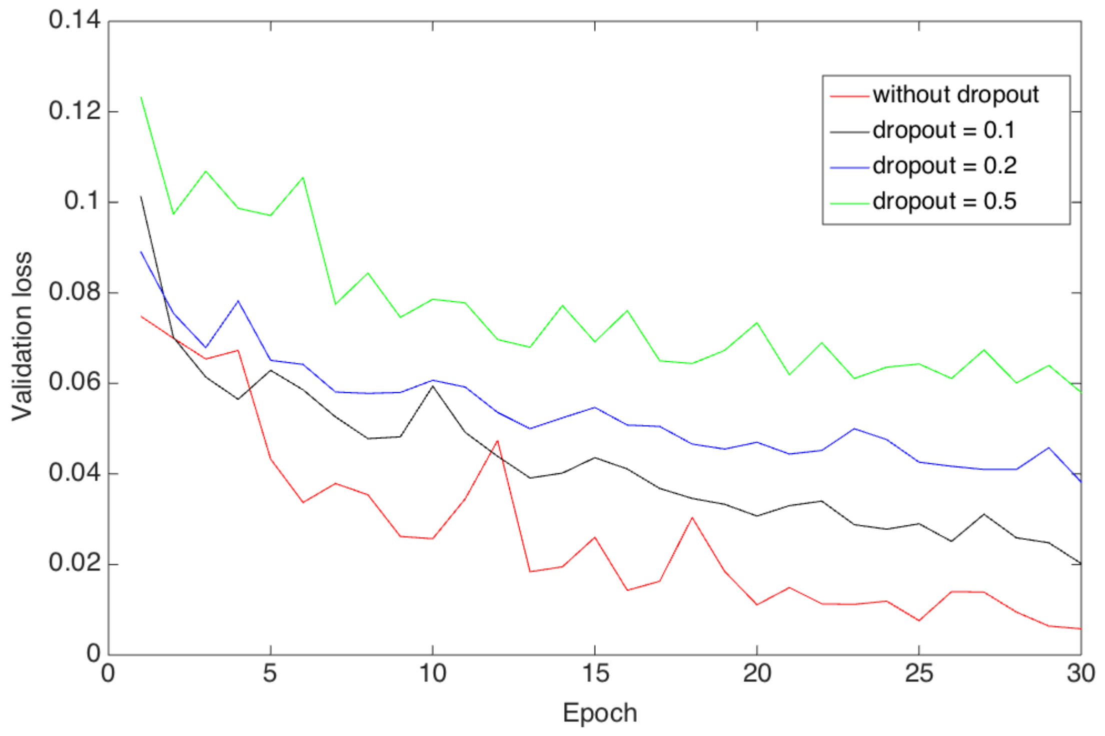


Figure 4.2: Validation losses for different dropout values

of the signal and should therefore not be very difficult for the network to learn. It was also tested to quantize the signal, to confirm what was described in Chapter 3.4.2.

During these tests, a model with 2 hidden layers, each with 1024 nodes was used along with a dropout of 0.1. The number of training samples was 150.000.000 and the number of validation samples was 10.500.000. The results are shown in Table 4.7 and Figure 4.3. The result from the quantization is not plotted as it would just show a straight line.

Table 4.7: Training results using different loss functions

	Epoch 1		Epoch 15		Epoch 30	
Loss	val_loss	loss	val_loss	loss	val_loss	
y = 0.2 x	0.1592	0.0963	0.0548	0.0526	0.0431	0.0341
y = expand(x)	0.1526	0.1019	0.053	0.0508	0.0365	0.0227
y = quantize(x)	2.9006	2.822	2.9007	2.822	2.9008	2.822

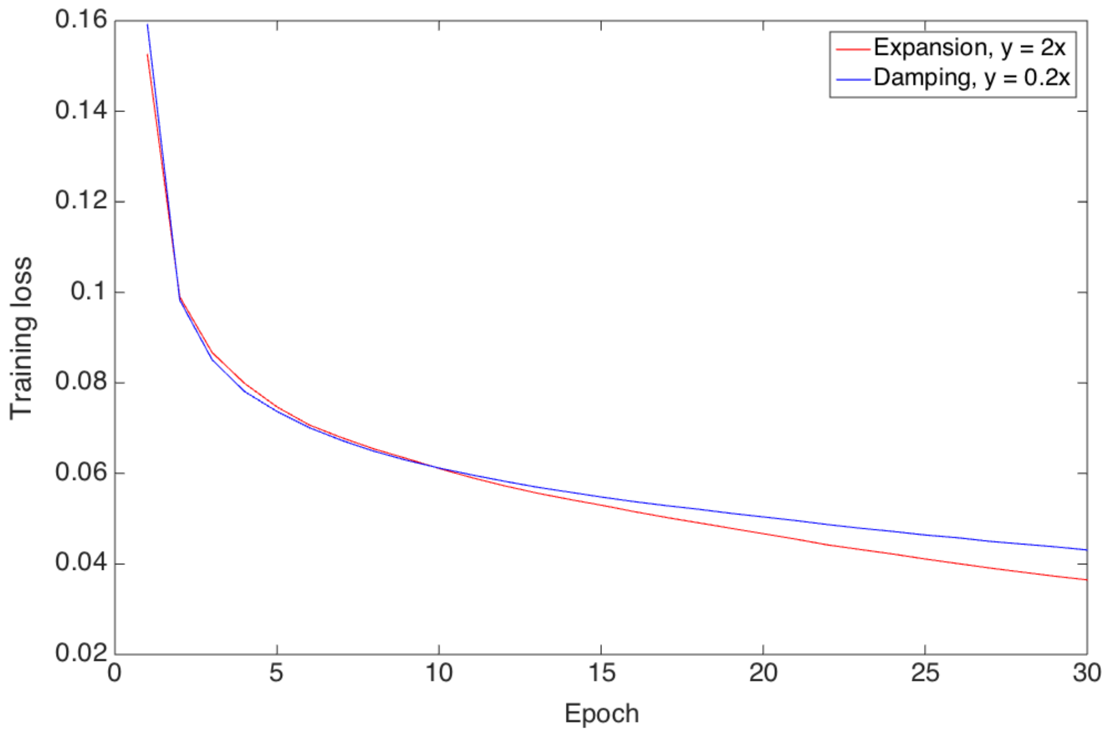


Figure 4.3: Losses for damping and expansion

Chapter 5

Discussion

5.1 Preprocessing of the input file

Table 4.1 contains the results from the trainings with different types of input data. As is seen from this table, the file using the original speech files in randomized order gives the smallest losses. Regarding the file containing "no silence", it should be mentioned that it does not render the original speech in a very good way as it also removes the natural silence between the spoken words. A better approach would have been removing only the silence when it exceeds a given number of samples, but as both the trimmed and the no-silence file results in worse losses than the original files, it might not be of use.

Further, as is seen from table 4.2, the more data used into the model, the better are the results in both training loss and validation loss as is to be expected. Xu et al. (2015) found that the performance increased a lot when going from 1 to 5 hours of training data, and because 150.000.000 training samples corresponds to around 2.6 hours of data, the losses will propbably be smaller by increasing the number of input samples even more.

When training the model for different training and validation sizes, there was not used a constant ratio between the training and validation samples so this might have affected the results regarding the training and validation losses relative to each other.

Framing and stacking

The Hanning window is applied as described in Chapter 3.2.2. As mentioned, the input samples from 0 until N/2 will only be a part of one window resulting in some damping under the reconstruction of the signal, as is also the case for the last N/2 samples. This can be seen from Figure 2.11 from Chapter 2.3.3, where the first 128 samples will be multiplied with the window which is increasing from 0 to 1. An alternative could have been to add N/2 zeros to the signal at the beginning and the end, which would result in all the samples from the original signal being part of two Hanning windows.

The result from testing the different window lengths is shown in Table 4.3, where the window length of 128 samples, that is 8 ms, gives the smallest losses. The smaller the window the more stationary will the different FFT's be, and maybe this had an affection on the training of the neural network. The main reason for this result is most likely due to the amount of training samples, or rows, into the neural network being bigger for smaller window lengths. If the window length is 128 samples, this corresponds to 2.343.750 rows compared to 1.171.875 rows when using 256 samples. The system will then have more data to train on, even though the length of each row will be smaller. One must also keep in mind that the duration of the training will be longer the smaller the window, where the epoch duration is almost doubled when going from 256 to 128 samples of window length. The cost thus increases for smaller window lengths.

Another alternative for testing the window size of 512 samples would have been to downsample the signal to 8 kHz as is done in Xu et al. (2015). They use a window length of 256 samples and a sample frequency of 8 kHz, thus the window length corresponds to 32 ms. They do however not mention why they have chosen that window length.

When padding the signal before stacking the signal as described in Chapter 3.2.5, it is used zeros. The columns could advantageously have been padded with speech instead to avoid the sudden transition from silence to speech. As this will only have affection on four of the rows, that is, four of the input samples to the neural network, the positive effects might be small as there are a total of around 1.170.000 rows when the sample size of the input data is 150.000.000.

5.2 Deep neural network

As is seen from Table 4.4, the smallest batch size tested gives the best result. The smaller the batch size the more updates of the weights, and this is probably the reason why the training takes around twice as long when using 64 samples instead of 128. The difference between a batch size of 64 and 128 is not that big, so one must consider if it is "worth" doubling the training time. Compared to the results from halving the window length from 256 to 128 samples gives a 0.028 smaller training loss, while going from 128 to 64 samples in the batch size results in a 0.009 smaller training loss. Both cases doubles the training duration, so if one were to choose, a smaller window size would be preferred if cost is being considered.

The results from testing the width and depth of the network shows that the network consisting of two hidden layers and 1024 units in each layer gives the smallest loss. One should also note that the deepest network, consisting of three hidden layers with each 512 units gives the worst result. By looking at Table 4.5, it looks like wide networks are preferred over deep ones. One must also keep in mind that adding more units in the network will also increase the costs.

Dropout

The results from using different dropout functions are not as expected. By looking at the graph in Figure 4.1 and 4.2, the losses are smaller the less dropout used, and smallest when no dropout is being used. This is the opposite of what was explained in the theory and the expected results as in Figure 2.6. The results might have been different if more training data was used but there was no time left to test this. However, as described in chapter 2.2.2, multiple studies show that using dropout gives better results when training neural networks. Thus, dropout should be used regardless of the results in this thesis.

The results from the validation losses in Figure 4.2 shows that the losses when dropout is being used gives a smaller variance. It seems like the results are more unstable without the dropout.

5.3 Loss functions

First of all, it should be noted that the system does not compensate for signals where information is lost, such as for quantization. From the results in Table 4.7 one can see that the losses does not get smaller, as was to be expected due to what was described in Chapter 3.1.1: The hearing loss comes *after* the neural network, so the network it will work as a pre-compensator for the hearing loss meaning that the information will be lost as the final step. If however the neural network was to come after the hearing loss, it could have tried to recreate the missed information, but that would not be possible for a hearing aid placed in the ear.

As is expected, the neural network does learn to compensate for the uniform damping as well as for the expansion. However, these functions as they are now do not describe a realistic hearing loss and should be changed for further study. It would have been interesting to see how the system is trained for more complex damping functions, where the damping looks more like the audiogram of an hearing impaired as explained in Chapter 2.1.2 and shown in Figure 2.2. Also, the expansion function should be further developed to depend on the input level, as seen from Figure 2.3 in Chapter 2.1.3. It would also be interesting to combine these two functions and train the system on a more complex hearing loss. One could also take some inspiration from the hearing loss simulator explained in the thesis by Berheussen (2012), mentioned in Chapter 3.4.1, to create even more complex hearing losses.

The system used in this thesis applies the hearing loss inside the loss function of the `model.fit` in Keras, but another possible way could have been to create a new layer without trainable weights. It is not known if it would be a better approach but it should be considered.

5.4 General evaluation

The quality measurement used in this thesis was only to look at the training and validation losses, in addition to listening to some of the signals. In further studies it is recommended to include other quality measurements in order to get a more appropriate evaluation of the results. PESQ (Rix et al. (2001)) is a good alternative which gives a perceptual evaluation of the speech quality. Also, the system should eventually be tested on actual hearing impaired persons in order to see if the system has the desired effect as a hearing aid. This does however require that the neural network is trained for a hearing loss that imitate his or her loss in a satisfactory way, and this will probably be the biggest challenge.

Lastly, as mentioned, a hearing aid used for pre-compensating for hearing losses will not be of use if there is a total loss of information due to sensorineural damages. If certain frequencies are not perceived, a solution might be to move these frequencies up or down to a range where they can be perceived. This is however not taken into consideration in this thesis, but can be used in further studies of hearing aids.

Chapter 6

Conclusion

The aim of this thesis was to investigate if deep neural networks and deep learning could work as an alternative for existing hearing aids. The biggest challenge associated with training these networks is to design functions to imitate real, complex hearing losses. As such, overcoming the aforementioned is the key to being able to create a fully functional hearing aid based on deep neural networks. However, as indicated in this thesis, if the hearing loss contains loss of information, it will not be possible to pre-compensate for it. This is due to that the hearing loss is applied as the final step and will inherently remove the wanted information independently of the signal processing performed beforehand.

By exploiting the knowledge of the signal processing in traditional hearing aids, one can create hearing loss simulators which in general terms does the opposite. Deep learning can thus be used to work the same way as today's hearing aids by learning to do the opposite of such a hearing loss simulator. The traditional hearing aids' capabilities are limited and does not compensate for distortion losses. This may be possible with the use of deep learning. Thus, the true value of using deep learning in hearing aids would be to pre-compensate for such highly complex hearing losses containing dependencies, for example in time.

As discussed and shown in the results, the trainings giving the smallest losses are the ones using the most training data as well as epoch time. Also, the smaller the batch size and window the better in terms of minimizing loss. By using a neural network with 2 hidden layers consisting of 1024 units in each layer instead of 512, the results are improved. In order to further investigate the training when applying hearing losses, these results should be taken into consideration.

Bibliography

Allen, J. B. (1998). Recruitment compensation as a hearing aid signal processing strategy. In *Circuits and Systems, 1998. ISCAS'98. Proceedings of the 1998 IEEE International Symposium on*, volume 6, pages 565–568. IEEE.

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127.

Berheussen, G. (2012). Real-time simulation of reduced frequency selectivity and loudness recruitment using level dependent gammachirp filters. Master thesis done in conjunction with SINTEF.

Byrne, D. and Dillon, H. (1986). The national acoustic laboratories'(nal) new procedure for selecting the gain and frequency response of a hearing aid. *Ear and hearing*, 7(4):257–265.

Chalupper, J. and Fast, H. (2000). Simulation of hearing impairment based on the fourier time transformation. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*, volume 2, pages II857–II860. IEEE.

Chollet, F. (2015). The keras source code. `https://github.com/fchollet/keras`.

Green, D. M. (1993). A maximum-likelihood method of estimating thresholds in a yes-no task. *Journal of the Acoustical Society of America*, 93:2096–2105.

Hasan, M. R., Jamil, M., and Rahman, M. G. R. M. S. (2004). Speaker identification using mel frequency cepstral coefficients. *variations*, 1:4.

Huang, X., Acero, A., Hon, H.-W., and Foreword By-Reddy, R. (2001). *Spoken language processing: A guide to theory, algorithm, and system development.* Prentice Hall PTR.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. [Online; accessed 2016-06-06].

Jones, E., Oliphant, T., Peterson, P., et al. (2016). Deep Learning: Open source scientific tools for Python. [Online; accessed 2016-06-06].

Kates, J. M. (2008). *Digital hearing aids*. Plural publishing.

Kinsler, L. E., Frey, A. R., Coppens, A. B., and Sanders, J. V. (1999). *Fundamentals of Acoustics*. Wiley.

Manolakis, D. G. and Ingle, V. K. (2011). *Applied Digital Signal Processing: Theory and Practice*. Cambridge University Press.

Moore, B. C. (2007). *Cochlear hearing loss: physiological, psychological and technical issues*. John Wiley & Sons.

Plomp, R. (1978). Auditory handicap of hearing impairment and the limited benefit of hearing aids. *The Journal of the Acoustical Society of America*, 63(2):533–549.

Purves, D., Augustine, G. J., Fitzpatrick, D., Hall, W. C., LaManita, A.-S., McNamara, J. O., and Williams, S. (2004). *Neuroscience with CDROM*. Sinauer Associates.

Rix, A. W., Beerends, J. G., Hollier, M. P., and Hekstra, A. P. (2001). Perceptual evaluation of speech quality (pesq)-a new method for speech quality assessment of telephone networks and codecs. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 2, pages 749–752. IEEE.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

Simon, P. (2013). *Too Big to Ignore: The Business Case for Big Data*. Wiley.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Swendgaard, I. H. (2015). Hearing aid processes simulated in matlab. Project thesis done in the course TTT4510- Signal Processing, Specialization Project.

Theano Development Team (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688.

Walden, B. E., Surr, R. K., Cord, M. T., Edwards, B., and Olson, L. (2000). Comparison of benefits provided by different hearing aid technologies. *JOURNAL-AMERICAN ACADEMY OF AUDIOLOGY*, 11(10):540–560.

Wilson, R. H. and McArdle, R. (2013). Characteristics of the audiometric 4000 hz notch (744 533 veterans) and the 3000, 4000, and 6000 hz notches (539 932 veterans). *Journal of Rehabilitation Research & Development (JRRD)*, 50:111–132.

Xu, Y., Du, J., Dai, L.-R., and Lee, C.-H. (2015). A regression approach to speech enhancement based on deep neural networks. *Audio, Speech, and Language Processing, IEEE/ACM Transactions on*, 23(1):7–19.

Zeng, F.-G., Rebscher, S., Harrison, W., Sun, X., and Feng, H. (2008). Cochlear implants: system design, integration, and evaluation. *Biomedical Engineering, IEEE Reviews in*, 1:115–142.

Chapter 7

Appendix: Python code

All the files included in this chapter are to be found at

https://www.dropbox.com/sh/x7hj0os5pxur2gw/AADqVxiA9wiTjA2HS1NIbFaPa?dl=0

7.1 main.py

```
'''
the main script to test deep neural network for different parameters.
'''

from __future__ import print_function
import numpy as np
np.random.seed(1337)  # for reproducibility

from keras.datasets import mnist
from keras.models import Sequential, Model
from keras.layers import Input, Dense
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD, Adam, RMSprop
from keras.utils import np_utils
from keras import backend as K
from keras.callbacks import Callback

from scipy.io import wavfile
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import ownObjectives
import functions
import recoverSignal


#----------------------- define variables -----------------------#
batch_size = 128
nb_classes = 10
nb_epoch = 30
windowLength = 256
N_train = 150000000
N_test = 10500000

#------------------------ load data   ---------------------------#

FsTrain, training_data = wavfile.read("4000mixedOriginalTrainingFiles.wav")
# for simplicity, the validation data is just the latter part of the training data
test_data = training_data[150000001:]

#--------------- process data before neural network  ------------#
```

```
dB_train, phase_train = functions.hanningFFT(training_data, N_train, windowLength)
dB_test, phase_test = functions.hanningFFT(test_data, N_test, windowLength)
# all these are numpy.ndarray
# stack 5 batches pr input row.
# each input row to the model is 645 samples if the windowLength is 256
X_train = functions.stack(dB_train, windowLength)
X_test = functions.stack(dB_test, windowLength)

#--------------- define input and output of model ---------------#

# output of model is 129 samples. this is the true output value
Y_train = dB_train
Y_test = dB_test

#-------------------------- make model --------------------------#
inputShape = int((windowLength/2+1)*5)
outputShape = int(windowLength/2+1)

# this returns a tensor
inputs = Input(shape=(inputShape,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(1024, activation='sigmoid')(inputs)
x = Dropout(0.1)(x)
x = Dense(1024, activation='sigmoid')(x)
x = Dropout(0.1)(x)
predictions = Dense(outputShape, activation='linear')(x)

# create model
model = Model(input=inputs, output=predictions)
```

```
# --------------- define callback in order to be able to -----------#
# ------------- get the output from the model (test data)-----------#
class printOutput(Callback):

    def getOutput(self):
        get_output = K.function([model.layers[0].input], [model.layers[5].output])
        layer_output = get_output([X_test])[0]
        self.outputs = layer_output

po = printOutput()

# ------------------- compile and train the model ------------------#
rms = RMSprop()
model.compile(loss=ownObjectives.lossFunction2, optimizer=rms, metrics=["accuracy"])
# model.fit returns a history object which can be looked at
model.fit(X_train, Y_train,batch_size=batch_size, nb_epoch=nb_epoch, verbose=2,
            validation_data=(X_test, Y_test), callbacks=[po])

po.getOutput()
print("output is updated")
# the output is the result when the TEST signal is sent through the model.

# --------------------- restore signal and write file -------------------------#

true_output = recoverSignal.do_ifft(Y_test, phase_test)
output_model = recoverSignal.do_ifft(po.outputs, phase_test)

int_true = true_output.astype("int16")
```

```
int_model = output_model.astype("int16")


wavfile.write('0706/output29.wav', 16000, int_model)

wavfile.write('0706/true_output29.wav', 16000, int_true)

print("files written")
```

7.2 loadFiles.py

```
'''

script to load TIMIT data into one file

switch which function to use at end of file
'''


from __future__ import print_function

import numpy as np

np.random.seed(1337)  # for reproducibility

import glob

from scipy.io import wavfile

from scipy.signal import decimate



#----------------------- define trimming function --------------------------#


def loadTrim():


    # get all names of files in database

    fileNamesTrain = np.array([])

    for file in glob.glob("../tale/timit/train_wav/*.wav"):
        fileNamesTrain = np.append(fileNamesTrain, file)

    # define how many files to use
    slicedFileNamesTrain = fileNamesTrain[0:2000]

    # get random order to spread the speakers
    np.random.shuffle(slicedFileNamesTrain)

    trimmed_output_train = np.array([])
    for fileName in slicedFileNamesTrain:
        Fs, newFile = wavfile.read(fileName)

        # replace "silence" with 0
        newFile[abs(newFile) < 10] = 0

        # remove zeros at beginning and end of file
        trimmedFile = np.trim_zeros(newFile)
        trimmed_output_train = np.append(trimmed_output_train, trimmedFile)

    # define output as int16 in order to avoid clipping using wavfile.write
    int_array = trimmed_output_train.astype("int16")

    wavfile.write('2000originalTrainingFiles.wav', 16000, int_array)
    print ("file written")

#------------------- define function to remove "silence" -----------------#

def loadNoZeros():

    # get all names of files in database
    fileNamesTrain = np.array([])
    for file in glob.glob("../tale/timit/train_wav/*.wav"):
        fileNamesTrain = np.append(fileNamesTrain, file)

    # define how many files to use
    slicedFileNamesTrain = fileNamesTrain[0:2000]

    # get random order to spread the speakers
    np.random.shuffle(slicedFileNamesTrain)

    output_train = np.array([])
    for fileName in slicedFileNamesTrain:
        Fs, newFile = wavfile.read(fileName)

        # only store the values greater than 10
        noZeroes = newFile[abs(newFile) > 10]
        output_train = np.append(output_train, noZeroes)

    # define output as int16 in order to avoid clipping using wavfile.write
    int_array = output_train.astype("int16")

    wavfile.write('noZeros.wav', 16000, int_array)
    print ("file written")




#--------------- define function to downsample signal ------------------#

def downsample():
    Fs, newFile = wavfile.read("trimmedTestFB_10.wav")
    file = newFile[0:1000000]

    downsampled = decimate(file, 2, n=61, ftype="fir")

    wavfile.write('downsampledTest8000.wav', 8000, downsampled)
    print("file written")

#---------------------- run functions ----------------------#

loadTrim()
#loadNoZeros()
#downsample()
```

7.3 functions.py

```
'''
functions to apply on data before they are used in the neural network
the function "hanningFFT" is the one being called from main.py
'''

import glob
import numpy as np
from scipy.io import wavfile
import math
from scipy.signal import lfilter
from numpy.random import randn
from numpy.fft import fft
```

```
import random

def applyHanning(inputSignal, N, windowLength):

    print("sample length before cut off in 'applyHanning': ", len(inputSignal))

    newN = N-N%windowLength
    numberOfWindows = math.floor(newN/windowLength)*2

    # cut inputSignal to be a multiple of windowLength < N
    inputSignal = inputSignal[0:newN]
    print("sample length after cut off in 'appltHanning': ", len(inputSignal))

    # make hanning window
    window = np.hanning(windowLength)

    # iterate through the whole input signal and copy windowed batches into hanningArray
    hanningArray = np.zeros(shape=(numberOfWindows,windowLength))
    j = 0;
    index = 0
    while (index < newN-windowLength):
        hanningArray[j] = inputSignal[index:int(index+windowLength)]*window
        index += windowLength/2
        j += 1

    return hanningArray

def doFFT(hanningArray):

    # take fft of each row
    fftArray = np.apply_along_axis(np.fft.fft, axis=1, arr=hanningArray)

    # storing phase (in radians) from output of fft
    storedPhase = np.apply_along_axis(np.angle, axis=1, arr=fftArray)

    # extracting amplitude from output of fft
    amplitude = np.apply_along_axis(np.absolute, axis=1, arr = fftArray)

    return [amplitude, storedPhase]

# stacking signal so that the neural network "sees" more than the one being trained
def stack(inputMatrix, windowLength):
    # get shape
    rows = inputMatrix.shape[0]
    columns = inputMatrix.shape[1]

    # should stack 5 rows for the one in the middle.
    zeroVector = np.zeros(columns)
    L = windowLength/2 + 1
    stackedMatrix = np.zeros(shape=(rows, columns*5))

    stackedMatrix[0][0:L] = zeroVector
    stackedMatrix[0][L:2*L] = zeroVector
    stackedMatrix[0][2*L:3*L] = inputMatrix[0]
    stackedMatrix[0][3*L:4*L] = inputMatrix[1]
    stackedMatrix[0][4*L:5*L] = inputMatrix[2]

    stackedMatrix[1][0:L] = zeroVector
    stackedMatrix[1][L:2*L] = inputMatrix[0]
    stackedMatrix[1][2*L:3*L] = inputMatrix[1]
    stackedMatrix[1][3*L:4*L] = inputMatrix[2]
    stackedMatrix[1][4*L:5*L] = inputMatrix[3]

    for i in range (0, rows-3):
        stackedMatrix[i][0:L] = inputMatrix[i-2]
        stackedMatrix[i][L:2*L] = inputMatrix[i-1]
        stackedMatrix[i][2*L:3*L] = inputMatrix[i]
        stackedMatrix[i][3*L:4*L] = inputMatrix[i+1]
        stackedMatrix[i][4*L:5*L] = inputMatrix[i+2]

    stackedMatrix[rows-2][0:L] = inputMatrix[rows-4]
    stackedMatrix[rows-2][L:2*L] = inputMatrix[rows-3]
    stackedMatrix[rows-2][2*L:3*L] = inputMatrix[rows-2]
    stackedMatrix[rows-2][3*L:4*L] = inputMatrix[rows-1]
    stackedMatrix[rows-2][4*L:5*L] = zeroVector

    stackedMatrix[rows-1][0:L] = inputMatrix[rows-3]
    stackedMatrix[rows-1][L:2*L] = inputMatrix[rows-2]
    stackedMatrix[rows-1][2*L:3*L] = inputMatrix[rows-1]
    stackedMatrix[rows-1][3*L:4*L] = zeroVector
    stackedMatrix[rows-1][4*L:5*L] = zeroVector

    return stackedMatrix

# the function called from main.py calls the other functions in this script
def hanningFFT(inputMatrix, N, windowLength):
    hanningArray = applyHanning(inputMatrix, N, windowLength)
    amplitude, phase = doFFT(hanningArray)
    halfAmplitude = amplitude[0:, 0:int((windowLength/2)+1)]
    abslog = np.log10(halfAmplitude+1)
    # because the input to the keras model should be float32
    abslog = abslog.astype('float32')

    # to get the dB-value defined as 10log(x)
    # return (abslog * 10), phase
    return abslog, phase
```

7.4 ownObjectives.py

```
'''
define own objectives in order to add hearing losses to the signal

'''



import numpy as np
np.random.seed(1337)  # for reproducibility

# import hls

# function to round up numbers to the nearest odd number.
# "quantizing"
def roundUp(array):
    return np.ceil(array) // 2 * 2 + 1

# function to expand input signal
def expand(array):
    return (2*array)
    # return [ 2*array if array > 30 else 0]


# defining the loss function which should include the hearing loss
# the inputs to this function are "tensorType(float32, matrix)".
def lossFunction(y_true, y_pred):

    # starting by just attenuating whole signal
    edited = y_pred * 0.2

    # quantize
    #edited = roundUp(y_pred)

    # expand
    #edited = expand(y_pred)

    difference = abs(edited-y_true)
    return difference**2
```

7.5 recoverSignal.py

```
'''
Script to recover speech signal from output of the keras model
The output is the FFT-abslog of the original signal.
```

```
Should "mirror" the signal before taking the IFFT
'''



from __future__ import print_function
import numpy as np
np.random.seed(1337)  # for reproducibility
import glob
from scipy.io import wavfile
from scipy.signal import decimate



def do_ifft(abslog, storedPhase):


    numberOfWindows = abslog.shape[0]
    print("numberOfWindows: ", numberOfWindows)
    windowLength = (abslog.shape[1]-1)*2
    print("windowLength: ", windowLength)

    # get the linear signal
    amplitude = np.power(10,abslog)

    # reversing the amplitude array in order to create mirror
    reversedArray = amplitude[:, ::-1]

    # the stored values are more than half of the original signal
    reversedArray = reversedArray[:, 1:windowLength/2]

    # in order to do the ifft, we must reconstruct the other half of the fft signal
    fullAmplitude = np.concatenate((amplitude, reversedArray), axis=1)
    # need to put together the amplitude and phase, z = Acos(0)+jAsin(0)  (0 is angle)
    angleAndAmplitude = np.multiply(fullAmplitude, np.exp(storedPhase*1j))

    # restoring the original matrix
    ifftArray = np.apply_along_axis(np.fft.ifft, axis=1, arr=angleAndAmplitude)

    # take out the real part of the ifft
    realOutputMatrix = ifftArray.real

    # need to do overlap-add.
    # ifftArray is now a 256x2524 matrix. should be overlapped into one 323301 (323200) arr
    overlap = np.zeros(numberOfWindows*windowLength/2+windowLength/2)

    overlap[0:windowLength/2] = realOutputMatrix[0][0:windowLength/2]
    j=windowLength/2    #current index in overlap array
    for i in range(0,int(numberOfWindows-1)):
        overlap[j:j+windowLength/2] = np.add(realOutputMatrix[i][windowLength/2:],realOutpu
        j += windowLength/2

    # must add the last values
    overlap[j:] = realOutputMatrix[numberOfWindows-1][windowLength/2:]

    return overlap


def mirrorSignal(signal, windowLength):

    # reversing the amplitude array in order to create mirror
    reversedArray = signal[:, ::-1]
    # the stored values are more than half of the original signal
    reversedArray = reversedArray[:, 1:int(windowLength/2)]

    # in order to do the ifft, we must reconstruct the other half of the fft signal
    fullSignal = np.concatenate((signal, reversedArray), axis=1)

    return fullSignal
```

7.6 hlsTime.py

```
'''
function for applying hearing loss to a signal in time domain
'''



from __future__ import division
import math
import regalia_mitra



# def hls(inputSignal, Fs, f1, bw1, k1, f2, bw2, k2, f3, bw3, k3):
def hls(inputSignal):
    # hearing loss parameters (for the hls function)
    Fs = 16000

    # define parameters
    f1 = 2000
    bw1 = 500

    k1 = -10

    f2 = 4000

    bw2 = 800

    k2 = -30

    f3 = 7000

    bw3 = 1000

    k3 = -10


    # scaling parameters

    w1 = 2*math.pi*f1/Fs;

    BW1 = 2*math.pi*bw1/Fs;

    K1 = 10**(k1/20); #db2mag

    w2 = 2*math.pi*f2/Fs;

    BW2 = 2*math.pi*bw2/Fs;

    K2 = 10**(k2/20);

    w3 = 2*math.pi*f3/Fs;

    BW3 = 2*math.pi*bw3/Fs;

    K3 = 10**(k3/20);


    firstSimulation = regalia_mitra.regalia_mitra(inputSignal, w1, K1, BW1);

    secondSimulation = regalia_mitra.regalia_mitra(firstSimulation, w2, K2, BW2);

    simulatedSignal = regalia_mitra.regalia_mitra(secondSimulation, w3, K3, BW3);


    return simulatedSignal
```

7.7 regaliaMitra.py

```
# regalia mitra code
# allpass filtering based on the Regalia Mitra model

# w0 : central frequency[rads/sample]
# k  : gain at f0 for boost/cut. (1 = original signal, 0<k<1 cut, >1 boost)
# bw : bandpass at -3dB [rads/sample]
import math
from scipy.signal import lfilter


def regalia_mitra(inputSignal, w0, k, bw):

    k1 = (1-math.tan(bw/2))/(1+math.tan(bw/2))
    k2 = -math.cos(w0)
    k0 = k/2

    b = [k1, k2*(1+k1), 1]
    a = [1, k2*(1+k1), k1]

    y = lfilter(b, a, inputSignal)
    outputSignal = 0.5*inputSignal + 0.5*y + k0*inputSignal - k0*y

    return outputSignal
```