University of Tartu

Faculty of Science and Technology
Institute of Physics

Karl Ehatäht Hendrik Ehrpais
Kristjan Eimre Ants Remm

Word recognition using neural networks

Instructors: Kaupo Voormansik
Karlis Zalite

Tartu 2014

1 Introduction

The objective of this group work was to create a word recognition program, that would work with our voices and some specific words. We created feature vectors from databases of four different words using signal processing algorithms, trained an artificial neural network to be able to detect these words and later tested the neural network by recording and testing new words. The neural network was trained using a genetic algorithm. At the time of writing, our program is able to recognize the given words with about 87% accuracy. Hopefully we can increase the accuracy in the future by creating a larger database for words and improving the algorithms used to make the feature vectors.

2 Method description

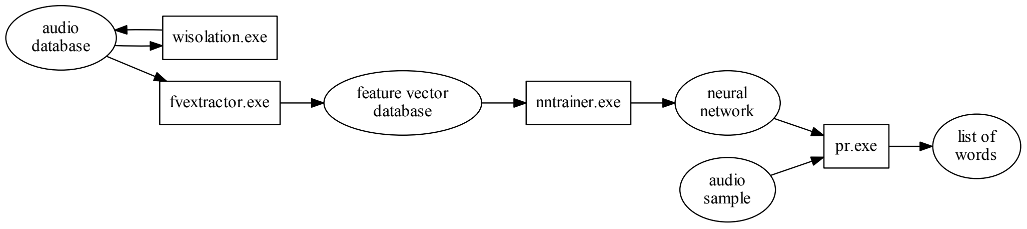

Figure 1: The flowchart of the overall process

To solve the given problem for a set of words, we constructed an audio database, which contains different pronunciations of said words, and 4 different programs (as shown in figure 1). The first program, *wisolation.exe* performs word isolation on the audio database. *fvextractor.exe* turns the audio database into a feature vector database. The feature vector database is used by *nntrainer.exe* to train a neural network. This neural network is used by the final program *pr.exe*, which takes an audio sample as an input and outputs words corresponding to the sample. All of these processes will be discussed in the following sections.

2.1 Word isolation

This section's objective is to take the input from the files, distinguish and separate the part of the audio file that contains the word we want to recognize. To achieve this we cut off signals below a certain threshold volume value.

To make sure that we are also able to recognize words that may have gaps in them, we increment a variable that checks if we have not been over the threshold for some time. Because we have a controlled environment where we have control over the input audio files and can set them at a required volume, we don't need to create any additional features for the first version of this program.

2.2 Feature vectors

This section considers feature extraction. Its implementation is mainly based on [1] and some concepts are also adapted from [2] and [3].

The main idea of feature extraction is to transform input data into a reduced representation set of features called feature vectors. In speech recognition the typical features are mel frequency cepstral coefficients (MFCCs). They reflect the dynamic change of a speech signal. The method for finding those coefficients is inspired by human auditory perception. Feature vectors may be calculated from audio samples of any length (phoneme, word, phrase). We use a collection of feature vectors, because our aim is to recognize and differentiate a certain set of words. Each such collection should roughly be the same for identical words pronounced by different speakers. The algorithm of transforming audio samples into MFCCs is quite straightforward. Some constraints are applied due to the peculiarities of the neural network used in the next stage of program. The algorithm goes as follows:

1. *Preemphasis.*
   Pre-emphasis filter is used to eliminate –6dB per octave[1] decay of spectral energy. It increases spectral energy of signal at higher frequencies. Given original series of samples $s(n)$, pre-emphasis filter is done with

   $$\hat{s}(n) = s(n) - \alpha s(n-1)\,,$$

   where α alpha is some constant between 0 and 1. Recommended value for α is 0.97, but it's adjustable by the user. First sample from original series is lost at this point.

2. *Framing.*
   The resulting signal is then chopped into a number of speech segments (from now on: frames), each having an overlap with its neighboring frame. The width w of each frame should be 25...40 ms and recommended overlap w_o is around 10 ms. If speech signal is sampled with frequency f_s then each frame (overlap) must contain $N = f_s \cdot w$ ($N_o = f_s \cdot w_o$) samples. A feature vector is calculated for each frame.

[1]Number of octaves between frequencies f_1 and f_2 is $\log_2\left(\frac{f_2}{f_2}\right)$.

3. *Energy.*
   Signal energy is one component of feature vector. It's simply computed from the sum of squares of samples:
   $$e = \sum_{n=0}^{N-1} \hat{s}^2(n)\,.$$

4. *Windowing.*
   Each frame is multiplied with a windowing function $W(n)$, thus cutting out some portion of samples from the beginning and the end of frame. This is justified since we have an overlap between successive frames and no information is lost. Our implementation includes two windows functions: Hann window
   $$W_{\text{Hann}}(n) = \frac{1}{2}\left(1 - \cos\left(2\pi\frac{n}{N}\right)\right)$$
   and Hamming window
   $$W_{\text{Hamming}} = 0.53836 - 0.46164\cos\left(2\pi\frac{n}{N}\right)\,.$$
   The aim of window function is to eliminate artifacts in the spectrum produced by discrete Fourier transform (DFT).

5. *STFT.*
   This stage may as well be named DFT, because STFT (short-time Fourier transform) is combination of windowing and DFT-ing. We have implemented brute-force DFT and its inverse (which we don't use in our program, just coded for completeness). We've also written Cooley-Tukey and Bluestein FFT algorithm. The use of Cooley-Tukey is optional, and is set by the user, because it modifies recommended values of frame width w such that each frame contains power-of-two number of samples. Explicit formulas for DFTs are given in our documentation of the code. We'll denote the resulting spectrum with $\hat{S}(\omega)$.

6. *Power spectrum.*
   Here we just take modulus squared on each element in spectrum series:
   $$P(\omega) = \operatorname{Re}\left(\hat{S}(\omega)\right)^2 + \operatorname{Im}\left(\hat{S}(\omega)\right)^2\,.$$

7. *Mel filter bank.*
   The reason mel filter bank is used is that human ear is less sensitive at higher frequencies. The relationship between sensitivity and frequency is thought to be logarithmic — the human ear has high resolution at lower frequencies, and vice versa. A common way to mimic this is to weigh each frame with mel filter bank. It can be constructed as follows:

a) Fix the number of mel filters $\mathcal{K}$ you want to use. Recommend values vary, but it's typically 40. The user of our program may set other, desired value. If the sampling frequency of original audio file is f_s, then we'll take $f_g = f_s/2$ as our maximum frequency.

b) Compute the maximum mel frequency m corrsponding to f_g as given by

$$m(f) = 2592 \cdot \log\left(1 + \frac{f}{700\text{ Hz}}\right) . \tag{1}$$

c) Set the average spacing in mel scale to be $\Delta_m = \frac{m}{\mathcal{K}+1}$. Compute center frequencies

$$m_c(k) = (k+1) \cdot \Delta_m ,$$

where $k = 0 \ldots \mathcal{K} - 1$. Revert back to frequency scale by computing frequencies $f_c(k)$ with the inverse of (1).

d) Find nearest indices n_c from $P(\omega)$ that correspond to $f_c(k)$.

e) With those indices, compute weighting matrix

$$H_{\omega k} = \begin{cases} \frac{\omega - n_{c_k}}{n_{c_{k+1}} - n_{c_k}} & \text{for} \quad n_{c_k} \geq \omega < n_{c_{k+1}} \\ \frac{n_{c_{k+2}} - \omega}{n_{c_{k+2}} - n_{c_{k+1}}} & \text{for} \quad n_{c_{k+1}} \geq \omega < n_{c_{k+2}} \\ 0 & \text{otherwise} \end{cases} .$$

$H_{\omega k}$ is then multiplied by power spectrum: $C_k = P(\omega) \cdot H_{\omega k}$. C_k-s are called mel spectral coefficients.

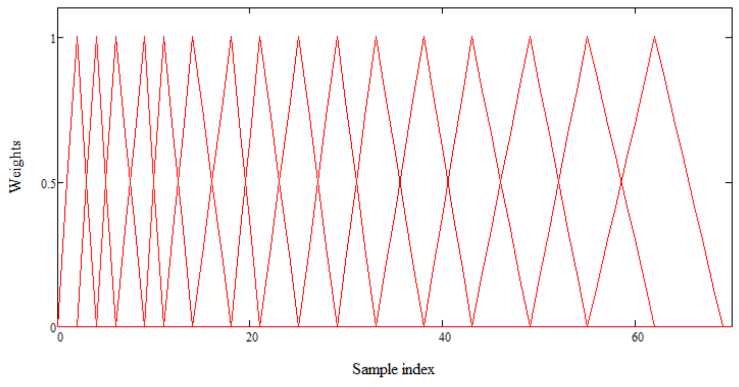

Figure 2: The columns of the weighting matrix $H_{\omega k}$.

8. *Log DCT.*
   The next step is
   $$G(k) = \log\left(|C_k|\right)$$
   and apply DFT on $G(k)$-s. Those will be our MFCCs $\widetilde{G}(k)$. The space where readily obtained MFCCs lie is quefrency domain.

9. *Liftering.*
   Liftering (a reversed „filtering") just emphasizes higher order coefficients. Each MFCC is multiplied by
   $$g_k = \begin{cases} 1\,, & k < 2 \\ (k-1)^{\beta}\,, & k \geq 2 \end{cases}\,,$$
   where we assumed that indices (i.e. mel filter numbering) start from $k = 0$. Default value for β is 0.6 and it's user-adjustable. At this point, energy (obtained in third step) is appended to MFCCs.

10. *Deltas.*
    Deltas are simply differences between frames in the past and in the future. If we have M frames, then deltas for j-th frame are expressed by
    $$\Delta\widetilde{G}_j(k) = \widetilde{G}_{j+\tau}(k) - \widetilde{G}_{j-\tau}(k)\,,$$
    where $\tau = 2\ldots4$ is (user-adjustable) time interval (in frame units). The second order deltas are computed from first order deltas the same way. Deltas are important because they reflect dynamical changes of MFCCs. But they also have a downside — feature vectors that are fed into neural network must be of the same length and we had to throw out first 2τ and last 2τ frames, since it's impossible to calculate derivatives for them.

11. *Interpolation.*
    This part of our algorithm takes care that to each word corresponds the same number of feature vectors. Otherwise the next stage of our program, neural network, won't work. Interpolation is done between the feature vectors of different frames. This is justified when the number of frames in the word doesn't exceed the desired number of feature vectors (again, user-adjustable) by more than two times.

2.3 Neural network

This section and the implementation corresponding to it is mainly based on the reference [4].

We used a feedforward neural network in our work. A feedforward neural network (just neural network from now on) is a computational model, which calculates a specified number of numerical outputs for a given number of inputs. It consists of three types of layers of neurons: the input layer, the hidden layers and the output layer (see figure 3). In our task we used one hidden layer. Every neuron is connected by edges to each neuron of the subsequent layer and every edge has a numerical value or *weight* associated with it.

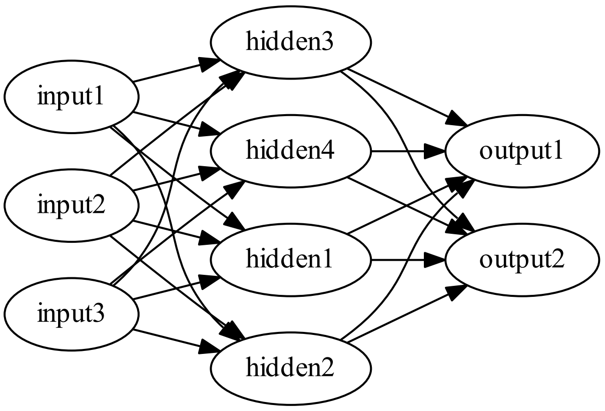

Figure 3: An example neural network.

Input neurons are given values by an external method (in our case, the inputs are words' feature vector[2] coefficients). All other (hidden and output) neurons' values are calculated based on the values of the preceding layer's neurons and the weights of the incoming edges as

$$a = \sum_{i=1}^{n} w_i x_i, \tag{2}$$

where n is the number of neurons in the preceding layer, x_i is the value of the i-th neuron in that layer, w_i is the corresponding weight of the edge that connects the i-th neuron and the currently evaluated neuron and a is the preliminary value of the neuron or *activation*. To get the value of the current neuron, a sigmoid

[2]From this part on, one feature vector will correspond to one pronuncation of a word. Previously it was used in a slightly different meaning (multiple feature vectors corresponded to one pronunciation).

function (see figure 4) needs to be used on a:

$$x = \frac{1}{e^{-a/p} + 1}, \tag{3}$$

where x is the value of the currently evaluated neuron and p is a parameter, which controls the curve. We used the value of $p = 1.0$. The sigmoid function is needed to introduce non-linearity in the network. Otherwise the output could only be a linear combination of the inputs.

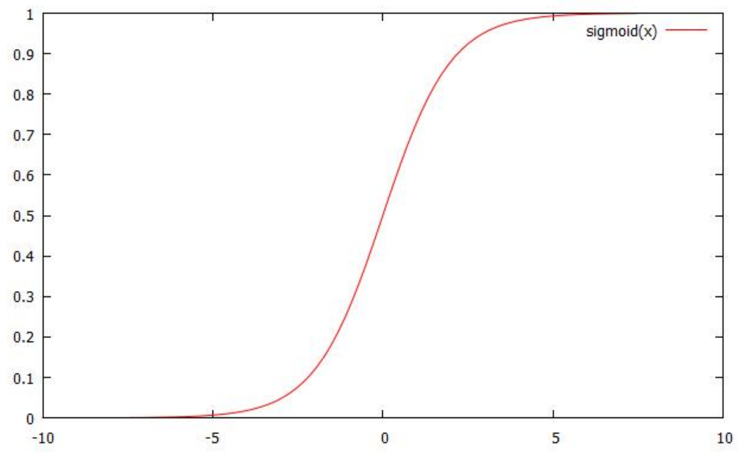

Figure 4: The sigmoid function.

If the number of layers, and neurons per layer is constant and selected as the external parameters require, then the neural network is completely specified by its weights.

Ultimately, what we want the neural network to accomplish is that when we input any of a words' feature vector (different feature vectors represent different pronunciations of the word) into it, then it would output an unique combination of numbers (identifier) specific to the word, or in our case, all zeros except for one 1.0, which corresponds to the word. This means that the neural network needs to have a very specific set of weights, which we will find using a genetic algorithm, discussed in the next section.

2.4 Genetic algorithm

This section is also partially based on the material in the reference [4].

The genetic algorithm we used is described in the flowchart in figure 5. The specimens are neural networks and their genomes (or genes) consist of their weights.

The genetic algorithm needs a way to compare how well different specimen perform in their tasks and based on that, the more successful specimen have a

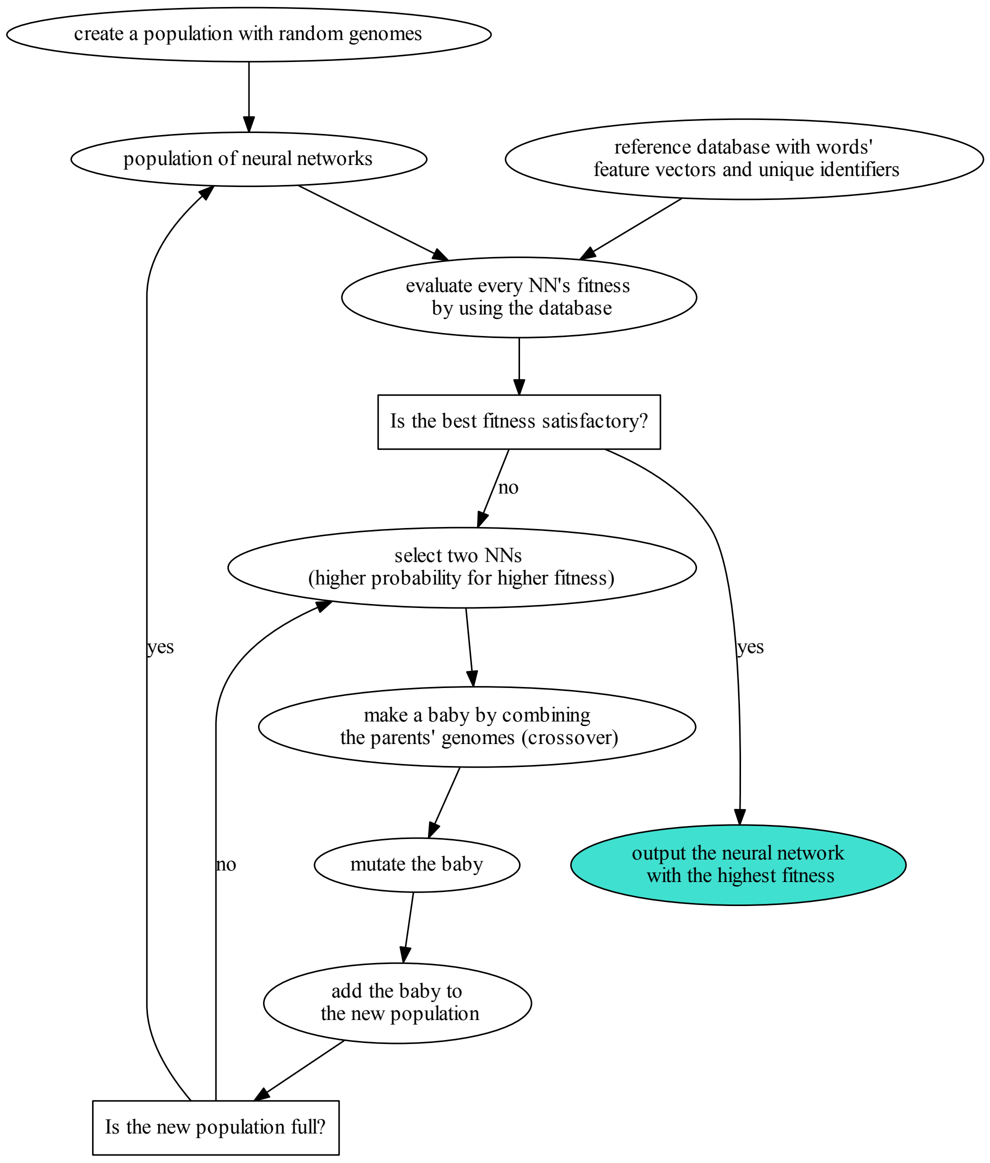
create a population with random genomes
population of neural networks
reference database with words' feature vectors and unique identifiers
evaluate every NN's fitness by using the database
Is the best fitness satisfactory?
no
yes
select two NNs (higher probability for higher fitness)
make a baby by combining the parents' genomes (crossover)
mutate the baby
output the neural network with the highest fitness
add the baby to the new population
Is the new population full?
yes
no

Figure 5: The genetic algorithm we used.

higher probability to pass on their genes to the next generation. To accomplish this, every neural network will be given a fitness, which is found using a premade database, which includes several words and the feature vectors corresponding to each of them. Neural networks' number of outputs matches the number of words. Each word is given a unique combination of outputs (identifier), in our case, all zeros except for the i-th output, which corresponds to the i-th word, and it's value is 1.0.

To calculate a neural network's fitness, we first find the neural network's score s by summing the average of squares of differences of each feature vector's outputs and the words' desired outputs (or their unique identifiers) and divide it by the number of words:

$$s = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{1}{Nfv_i} \sum_{j=1}^{Nfv_i} (\boldsymbol{\omega}_{ij} - \mathbf{d}_i)^2 \right), \tag{4}$$

where N is the number of words, Nfv_i is the number of feature vectors of the i-th word, $\boldsymbol{\omega}_{ij}$ is the neural network's output vector corresponding i-th word's j-th feature vector and $\mathbf{d}_i$ is the i-th word's desired output vector, or its unique identifier. The divisions are needed to normalize the score so it doesn't depend on the number of feature vectors or words.

And because the fitness needs to be higher for a better performing specimen, then we chose to calculated it like this:

$$f = 100e^{-10s}, \tag{5}$$

so it would be 100 at the best score (0) and reasonably low at the worst score.

Now we have a population of neural networks and each of them have a fitness. To find a new generation of networks, we make a roulette-wheel [5] selection to select two parent networks, a mother and a father. Based on these networks' genes, a new baby network is formed. With the probability of p_c, a crossover will occur at a random point in the mother's genes. Until that point, the baby gets the mother's genes (or weights) and from that point on, it gets the father's genes. If no crossover occurs, the baby is its mother's clone.

After that, the baby is mutated. It means that each of its genes have, with a probability of p_m, to change a little. After mutation the baby is put into the new population.

This process is repeated until a new population is formed and the cycle starts again until a neural network with a satisfactory fitness is found.

We also implemented an option to use elitism. It transfers a specified number of a generation's best networks into the next one without changing them.

2.4.1 Reference fitness

We had a problem with the previously described algorithm. The output neural network was too specialized and it had trouble detecting different pronunciations of the words from the ones we used to train it with.

Therefore we made a reference database. It consists of the same words but the feature vectors are different from the ones in the main database. When we calculated the fitness according to the main database, we also calculated a reference fitness (which did not affect the training in any way) according to the reference database. And when the improved algorithm finished training, it outputted the neural network, which had the highest sum of both, the normal fitness and the reference fitness.

3 Results

To test the resulting program, we made a testfile (with the same voices as the program was trained with) and ran the program on it. Out of 46 words, it correctly guessed 40. That is about a 87% success rate.

References

[1] B. Plannerer. *An Introduction to Speech Recognition*. 2005.

[2] Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. "Comparative evaluation of various MFCC implementations on the speaker verification task". In: *in Proc. of the SPECOM-2005*. 2005, pp. 191–194.

[3] Zhenhao Ge. *Development of Automatic Speech Evaluation System*. 2008.

[4] *ai-junkie*. 2014. URL: `http://http://www.ai-junkie.com/`.

[5] *Wikipedia - Fitness proportionate selection*. 2014. URL: `http://en.wikipedia.org/wiki/Fitness_proportionate_selection`.