Case study on practical applications of neural networks

Csaba Botos
Pázmány Péter Catholic University

***Abstract*—Theoretical analysis of (single) *Perceptron* and *MLP* models on binary and N-ary classifying, supported by practical demonstration. In this study intuitive experiments are presented and followed by suggestive descriptions.**

Keywords*—*Voice recognition, Perceptron, MLP, Case study

I. Introduction

Today's neural networks work on algorithmically very complex, yet intuitive tasks. These objectives can be categorized into two main classes: *supervised* and *unsupervised*. The trend shows that labeling slavery becomes unfavored therefore classical supervised learning is getting less attention over time. Still understanding how these models (should) work is essential and demonstrating via the simplest examples ends in some interesting case studies. In the following an informal introduction presents results of such experiments.

II. Technical background

For the sake of readibility the detailed implementation of the perceptron framework is excluded from this writing (although can be found at github). The writing assumes basic knowledge in linear algebra, the model will be described in the following manner:

For a given network N and fixed length input x the system's assumption, the output is computed as $y = N(x)$, where y is a fixed length vector. N can be seen as a *black box*, that takes $dim(x)$ information, somehow processes it and - if N is trained correctly it returns the class of x. Practically $dim(y) = |\{\text{classes}\}|$ - therefore y represents the network's confidence on each class.

A. Introductory example

For example if a small picture is given to a person, he or she guesses which it could and could not be. If that person is told that there are four classes, like *Car, Plane, Cat, Kid*, he can tell how likely it is that the given picture falls in that class, so he can give a so-called *confidence parameter*.

Patterns which are associated with cats may be associated with kids too, but is unlikely to be associated with planes. Deciding whether a pattern improves the confidence on each class or not, yields a **sign** for the given pattern, and the measure how strongly it influences the likelihood is called the **weight** of the pattern. In *neural networks* there are small nodes based on the biological model of neurons, that is responsible for recognizing and weighting the patterns. These nodes are called **perceptrons**.

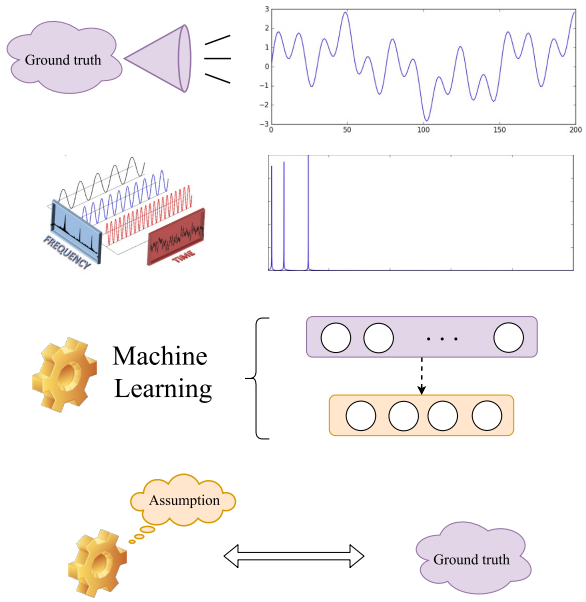


Fig. 1. The basic conception of voice recognition: the original message is first expressed in physical waveforms which may introduce a lot of noise in real life. After recording the wave from time domain is transferred to *frequency domain* that tells which frequencies were dominant in the captured signal sample. According to nature's biological systems in this form the same information can be better processed.

B. Formal definition

Let x be a preprocessed information, the perceptron $\mathbf{P}$ decides which of them are important in recognizing a cat: having a corresponding weight with large magnitude, and which of them are irrelevant: having a weight with magnitude close to zero. Therefore a *perceptron* has a weight w_i for each input x_i and its transient state can be now formally written:

$$\mathbf{P}_{transient} = \sum_i x_i w_i$$

Which can be also rewritten as the inner product of $\mathbf{x}$ and $\mathbf{w}$

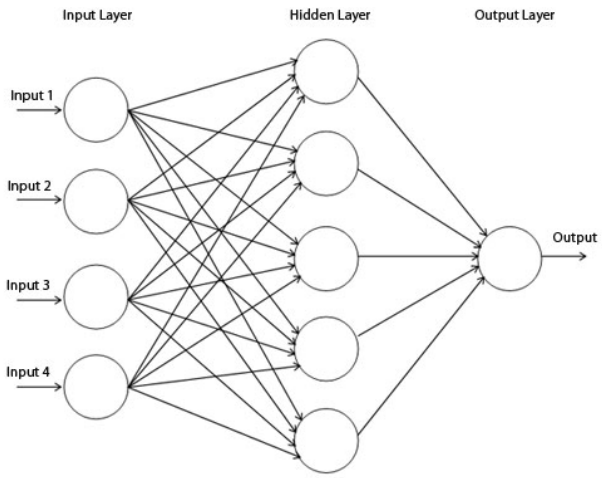


Fig. 2. A small feedforward network with one hidden layer, and single output perceptron. Source: Codeproject

$$\mathbf{P}_{transient} =< \mathbf{x}, \mathbf{w} >= \mathbf{x}^T\mathbf{w}$$

To enhance the stability of N *non-linearities* may be introduced as activation functions. Without further investigation, accept the fact, that wiring many perceptrons together results in a numerically unstable system, caused by the lack of any restriction on the magnitude of the weights. In the following examples perceptrons will be arranged in a special way, to form a **Feed Forward** network, meaning that the information flow will occur in a direct way, without feedbacks, see figure 2.

C. Learning, Training

After describing the model, the first question is:
what are the correct values for $\mathbf{w_P}$ *for each* $\mathbf{P}$ *in network* N*?*

There are a lot of intuitive explanation how this problem should be approached and solved. For exhaustive investigation of the topic, read the chapters from Michael Nielsen's site. Anyhow, in general this is still the most studied question in machine learning. Luckily if N's performance can be measured, thanks to its feed-forward structure a *numerical suggestion* can be defined as well, which tells how to change $\mathbf{w_P}$ to improve efficiency of N. In practice an $\mathbf{E}$ error function is defined, and the objective of the training is to reduce it. Without digging deep in math we can find an intuitive situation with the same results.

Take a perceptron $\mathbf{P}$ with input $\mathbf{x}$ of objects on a given picture of a cat. Suppose that this perceptron *fires* (has high output value) when wheels are on the picture. It should remain silent when there are no round objects listed in $\mathbf{x}$, still it turns on, ruining the output of N, producing high $\mathbf{E}$. Since we know the original label of the picture, we can tell which assumptions were wrong, and which were good - which to decrease, which to increase if the next time N is given the same input. This information is distributed between the previous perceptrons which caused the actual to fail by simply multiplying the error with the weight corresponding to the previous node. Also with the information of how much the output of the actual node influenced the error, and with its input values, the significance of wrong $\mathbf{x_i}$s can be reduced, and important features' weights can be increased, which is actually finding a better $\mathbf{w_P}$. For further intuitive examples on distributing the error, and finding the so called *gradient* for each weight visit Andrej Karpathy's guide.

The example concluded to the practical application of the so called *Stochastic Gradient Descent*. Originally every samples in the training set should be introduced to the system before updating its weights and that would be the *Gradient Descent*, but in the hope that the samples falls in a subset of the whole space of all possible inputs, the algorithm uses mini-batches for the learning process. When the number of samples in the batches reduces to 1, the training is called on-line training. And if it happens on N containing a single perceptron it is called *Rosenblatt perceptron learning algorithm.*

III. Separating sine waves

After defining the model, the first implementation should be able to classify the low- and high-pitch samples. For starting let the low-pitch sample, $\mathbf{x^1}$ be a sine function with frequency $F1$, and high-pitch sample $\mathbf{x^2}$ a sine with frequency $F2$. For mimicking the nature's living systems, the sample is transformed to frequency domain with an algorithm called *N-point Fast Fourier Transform*. Without further explanation it just tells which frequencies were dominant in the time domain, which is a better clue for both the living, and the machines too. The architecture will be two perceptrons in the same layer. For each $\mathbf{x_i}$ they have different weights, which helps them decide when to turn on, and when to output lower value. After training them the results are promising. The training set is totally fed to the network. For details see Figure 3. For the script which was used to produce the figures check out demo

A. Varying the precision of the Fourier Transform, N

For such arbitrary cases, where no noise is added to the signal, larger precision is unnecessary, for validation check Figure 4, 5. However the importance of chosing the right N should not be underestimated: in environments with high *Signal-to-Noise ratio* it is advised to use high *sampling rate*, and large N, however for real-time applications it is a trade-off between quality and speed, because the computation complexity for this architecture is $O(N^2)$, therefore choosing too large N results in slow processing.

B. Introducing regularization

An unknown term was used in the previous description, which now will be revealed. The regularization is simply a practical implementation of Occam's razor, meaning that the unimportant features of $\mathbf{x}$ will not be taken into account by the perceptron. On Figures 3, 4, 5 the weights were originally

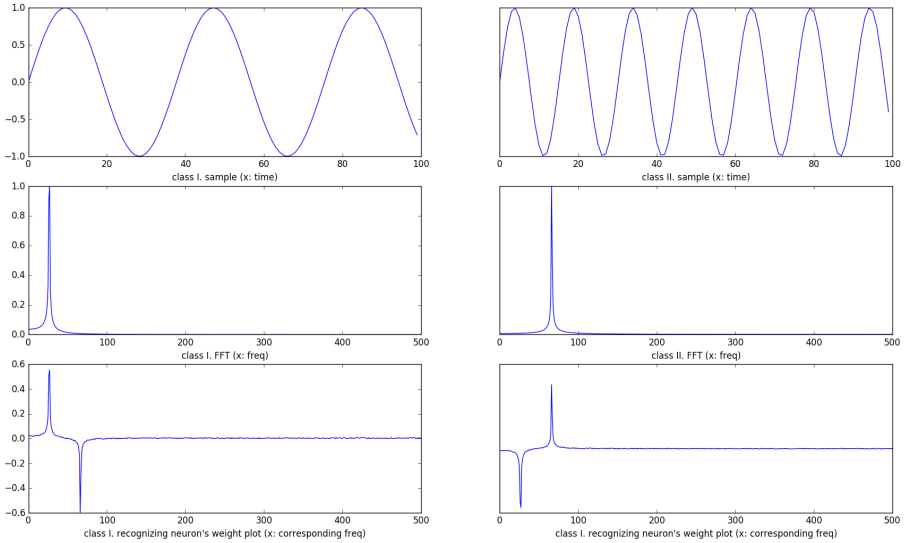


Fig. 3. On the first row the original sample is shown in time domain. On the second row the N-FFT of it. On the last row the plot of the weights of the corresponding perceptron. The first feature to recognize, that each perceptron learned that not just to turn on for the corresponding input sample, but to output negative values when samples of the opposite class are shown to the network. The network was initialized with Gaussian random noise with mean at 0 and deviation $1/2$, and trained with mini-batch size 1, learning rate $\eta = 0.5$, *regularization* factor $\lambda = 0.1$ for 10 epoch with *cross-likelihood* error function

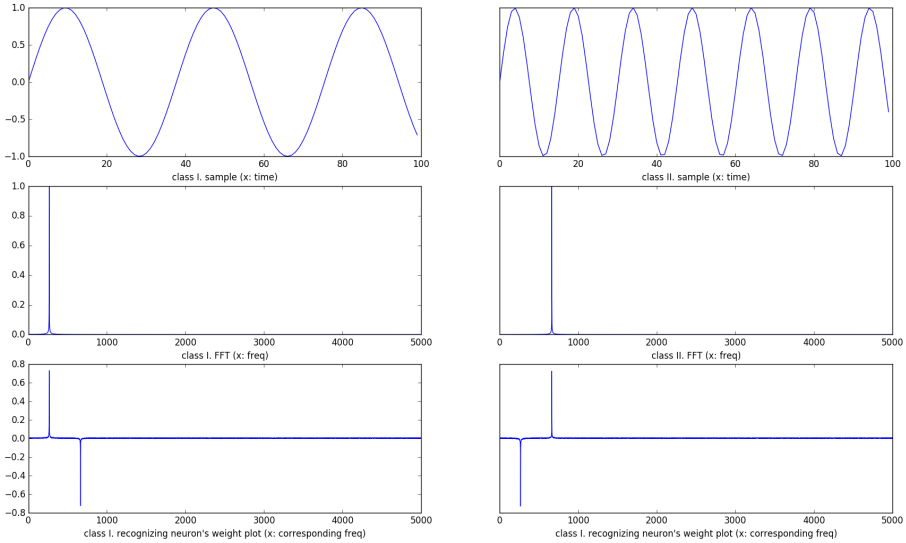


Fig. 4. The network's initialization and training parameters are identical to the network depicted on Figure 4, only the transform precision is increased from $N = 500$ to $N = 5000$ resulting in more narrow spikes corresponding to the recognized frequency

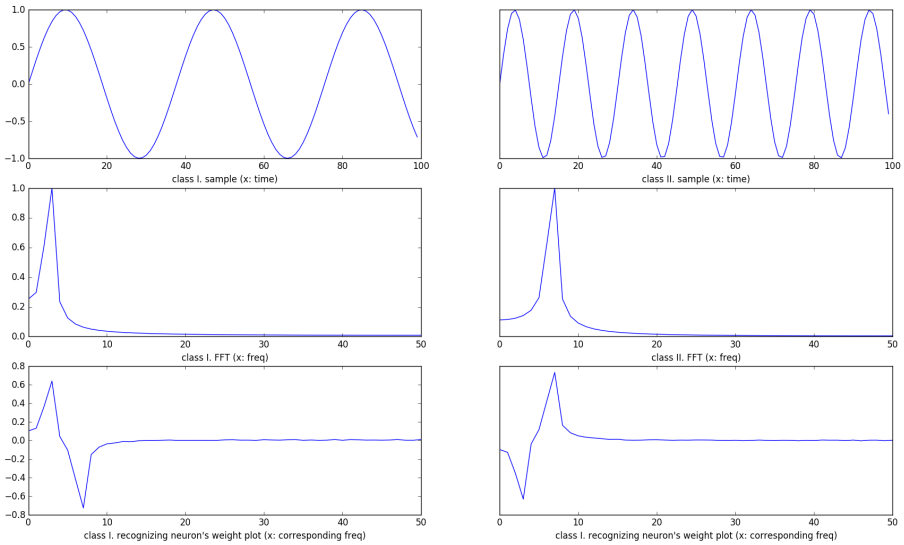


Fig. 5. The network's initialization and training parameters are identical to the network depicted on Figure 3, only the transform precision is decreased from $N = 500$ to $N = 50$ resulting in more shallow spikes corresponding to the recognized frequency

initialized with random Gaussian noise, but after training only the important frequencies' weights stayed significant. If the regularization factor is reduced the noise wouldn't disappear and the perceptron would still take irrelevant informations into account. This phenomena is shown on Figure 6. However instead of regularizing we can *overtrain* our networks, increasing the number of epochs, the number of times we show the same samples in the training set. Noise reducing via **overfit** can be seen on Figure 7.

C. Dealing with multiple waves

Approaching to real life problems, more parallel signals should be introduced to the network. In this subsection the case of recognizing overlapping frequencies and common frequencies are examined. The networks were initialized and trained identically to the network in previous experiment with $\lambda = 0.5$. In the first case six different base functions were used to compose the samples. On Figure 8 it is really trivial that the network is not just capable of separating low-pitch samples from high-pitch samples, but can recognize different patterns, even if they are scattered throughout the state space of the input. In the second case the conlusion is that the perceptrons (if well trained) will look only for relevant signals and patterns in the input. See Figure 9

IV. Samples from outside

For acquiring samples from the environment the architecture should be wired to a physical sensor which will transfer signals from analog to digital. Practically outer libraries are applied to perform the capturing, the only thing remaining is to write an adapter to make the information format compatible with the network's architecture.

A. Capturing sound

For the following experiments an algorithmic smart capturing is used, which enables us to record multiple sound waves with different length, without pressing a single key. The main idea is that the noise of the environment can be monitored, and in the background mean averaged. A practical treshold is set for recognising significant audio inputs, where the magnitudes of the sampled waves reach the critical. After the recording starts, a fixed size time-frame will cancel out further activities of the capturing system. Another option is to end the capturing process when the magnitude of the input falls below a treshold. Either way, the samples are gathered in a list, and each of them are transformed to frequency domain with *N-FFT*, producing equal length inputs $\mathbf{x^i}$ for the network.

B. Defining the classes

For human interpretation it is useful, that the separated classes are labeled, while from the aspect of the network it is totally irrelevent whatever these classes are. That is an important thing to understand, both in supervised and in unsupervised learning. In the next cases separation sound waves of *Hello* will be separated from samples of *Goodbye*.

C. Binary classifiers

D. First greeting recognizer

The first model trained on greetings and farewells is working as expected. In the demo (so.py) the user is asked to input

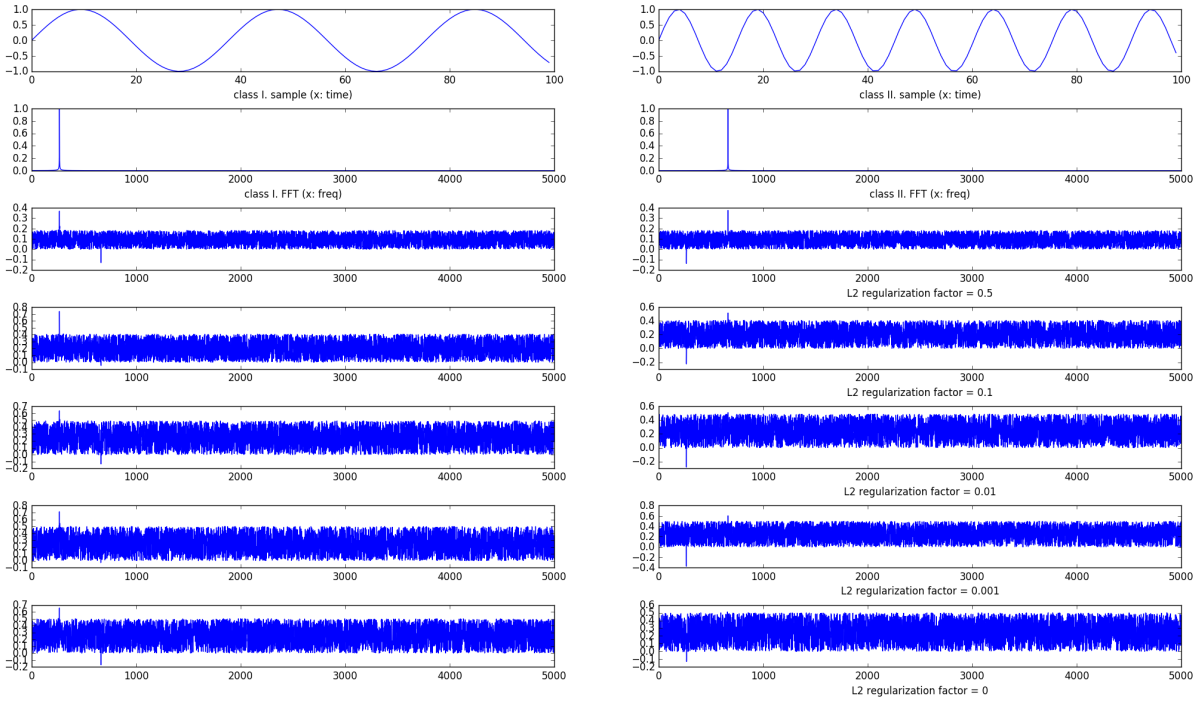


Fig. 6. After the third row the figure depicts independently, identically initialized and trained networks detailed in Figure 3, only the regularization factor λ is decreased from 0.5 to 0.0001 resulting in more noisy weight plot. Still the corresponding weights' magnitude are outstanding

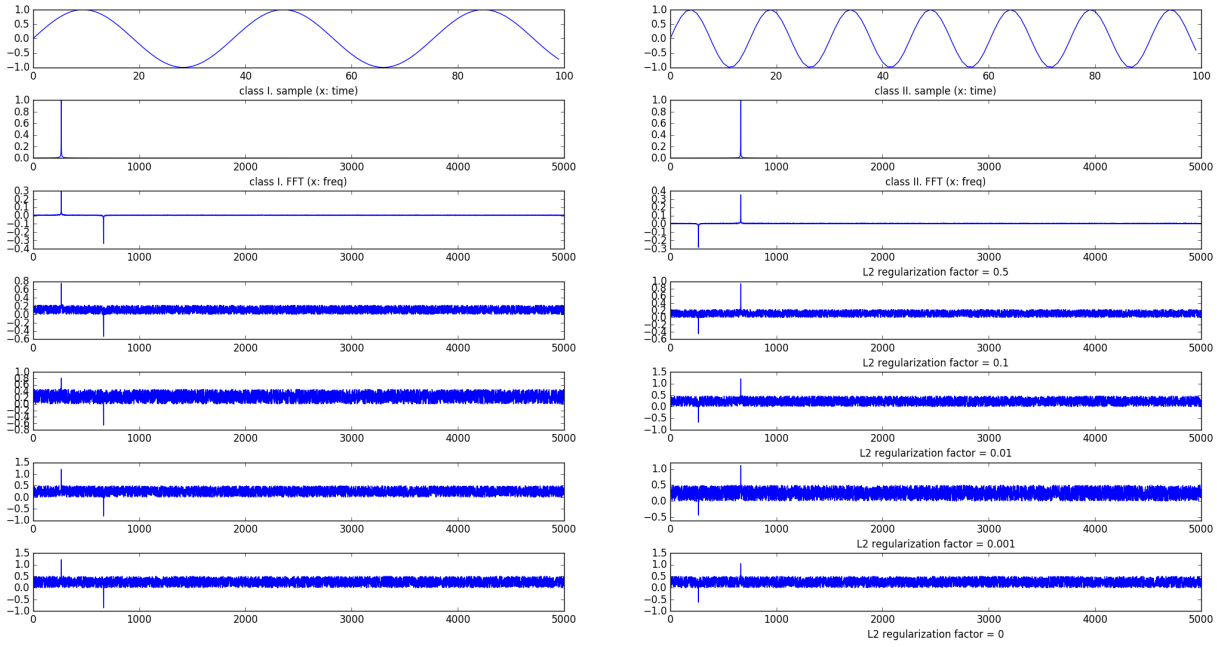


Fig. 7. The network was trained with the same parameters as the one depicted on 6, except that it was trained for five times longer. Despite the result is quite the same as the result achieved with greater regularization factor, overfitting should be avoided. Overfitting occurs when the network N tends to perform well on the training set, but fails to recognize unmet samples, because either the training samples were not general enough, either the network was just large enough to memorize each one of the samples, or both.

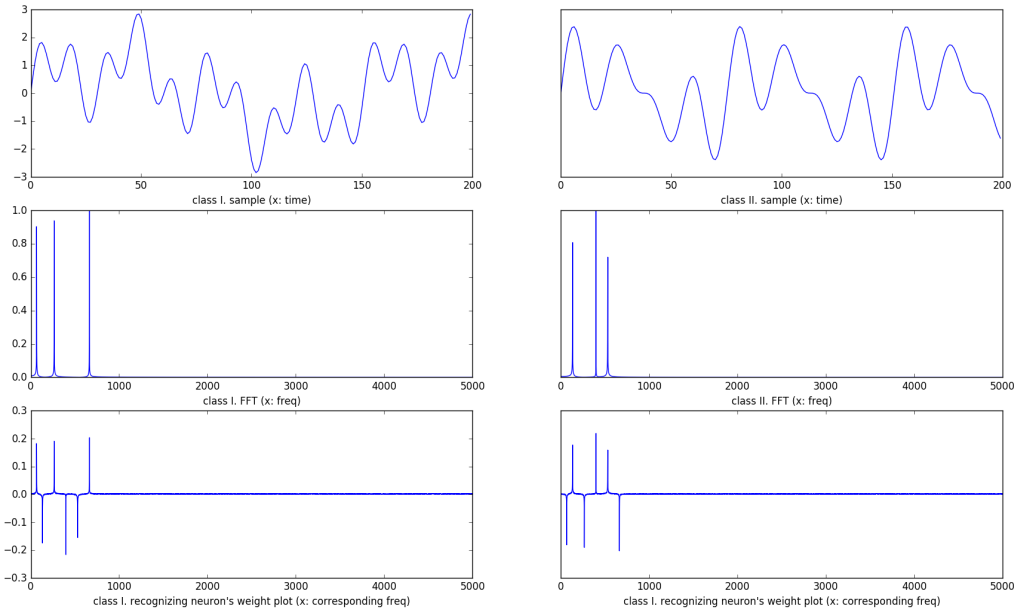


Fig. 8. Each sample is an equal combination of 3 independent sine function. The frequency of the waves are not separated to low and high frequencies, both can be found in each sample, but it is important to notice, that the two samples do not have any common component.

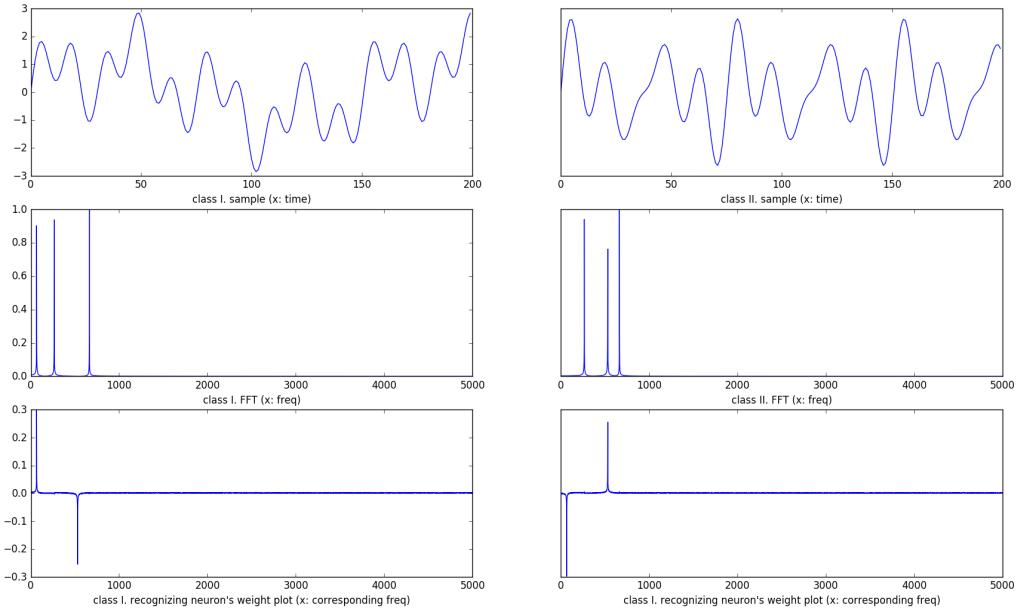


Fig. 9. In opposite to the case depicted on Figure 8 here the samples have two common component, meaning that in the base functions there are two common frequency and only differs in one harmonics. The result is that the common features extracted by the FFT are discarded, and the perceptrons only learn the difference.

number of N samples representing class I, and the same number from class II which will be labeled with the assumption that the samples do not overlap. The length of the samples can vary in time, because the smart capturing takes care of the preprocess. For ending the recording session produce some loud noise, a clap should do. The labeled samples are then taken into account as the training set for the network, and the process adjusting the parameters begins. After 30 epochs the network waits for new $\mathbf{x}$s to classify. A few tests can be made to ensure the network is working well. My results with N=1000, on 3-3 input sample is performing around 91% (100 tests were performed totally). Sending a **SIG-TERM** will end the testing session and a summary window will pop up, probably something like on Figure 12.

V. Multiclass classification

A. Recognizing greetings in multiple languages

Currently the model somehow works like the one depicted on Figure 10. A real task can be defined, where the network $\mathbf{N}$ has to classify that in which language the user said hello. Unfortunately there are more languages than two, but neural networks are flexible enough to be able to recognize and separate the inputs into N number of classes. For the improved model see Figure 11

The experiments were done with 5 different languages: *French, Arabic, Spanish, German, Hungarian*. The training scene is the same as in the previous cases: input sequentially N number of samples for each language, end the recording with a clap. For a demonstration script check out demo (nclass.py) The network trains on the given set, and the system is ready for testing. End the testing session by sending a **SIG-TERM** signal. With the following hyperparamethers: 1-1 number of samples in each language, *mini-batch*$= 1$, *number of epochs*$= 15$, $\lambda = 0.005$, from random Gaussian initialization a network with 60% performance can be obtained. With larger training set, 3 samples for each language the result raised over 85%. Comparing to industrial networks the training set used to adjust the parameters is very small in size. For larger projects usually the training set contains thousands and millions of samples, and overfitting is less likely to occur. For such a narrow network trained with a poor training set it can be said that 85% is not that bad at all. Although the result still depends on who tests the configured network.

VI. Conclusion

So far it is shown how single layer neural networks work. How they should be thought of, what difficulties rises during their training. The capabilities of shallow networks were met during tests with the N-multiclass classifier. The next topic of the investigation is that how can these perceptron layers be combined to each other, for example to recognise whether a user said yes or no, without explicitly telling which language he or she was speaking.

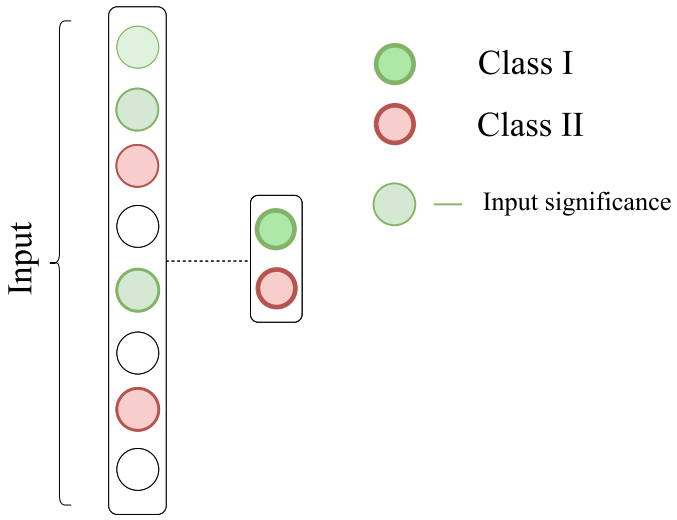


Fig. 10. A basic perceptron model, operating on 8 clues of the input. The perimeter of each input node describes how much it increases the output of the perceptron with the same colour and decreases the neuron with opposite colour. Input nodes which are not colored holds irrelevant informations for both perceptrons, therefore discarded.

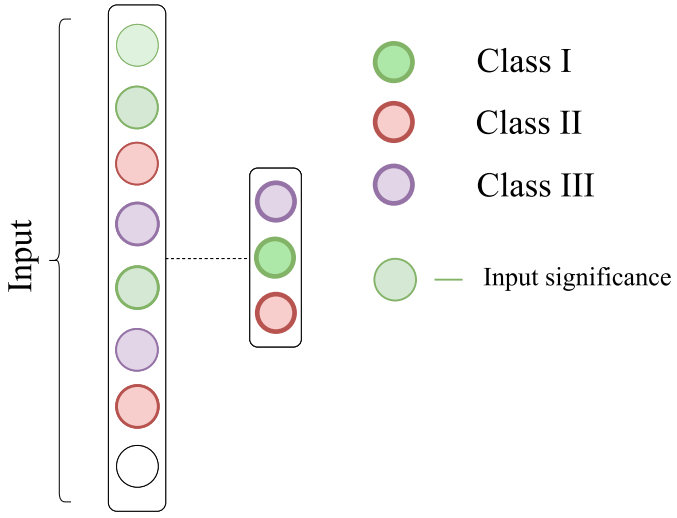


Fig. 11. An improved perceptron model, operating on 8 clues of the input $\mathbf{x}$, deciding which class it belongs to. The weight anti-symmetry is not so trivial in this case because perceptrons of multiple classes can be turned on by one single feature while other perceptrons are inhibited by the same feature or perceptrons may even discard it. Meaning that colours on this figure are a bit tricky, because each input node can be shared between the classes.

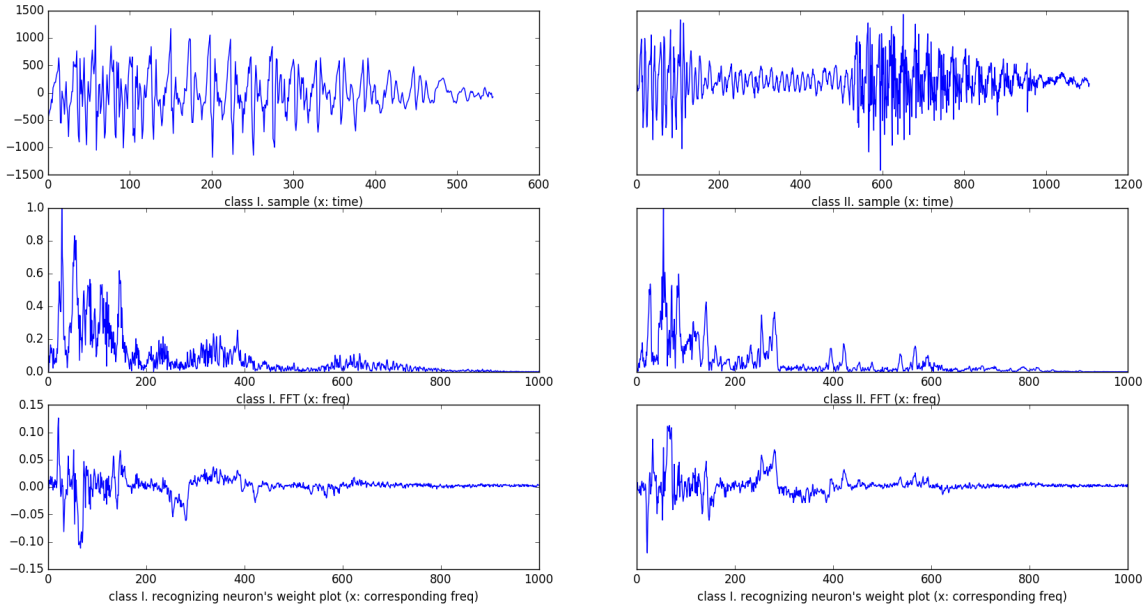
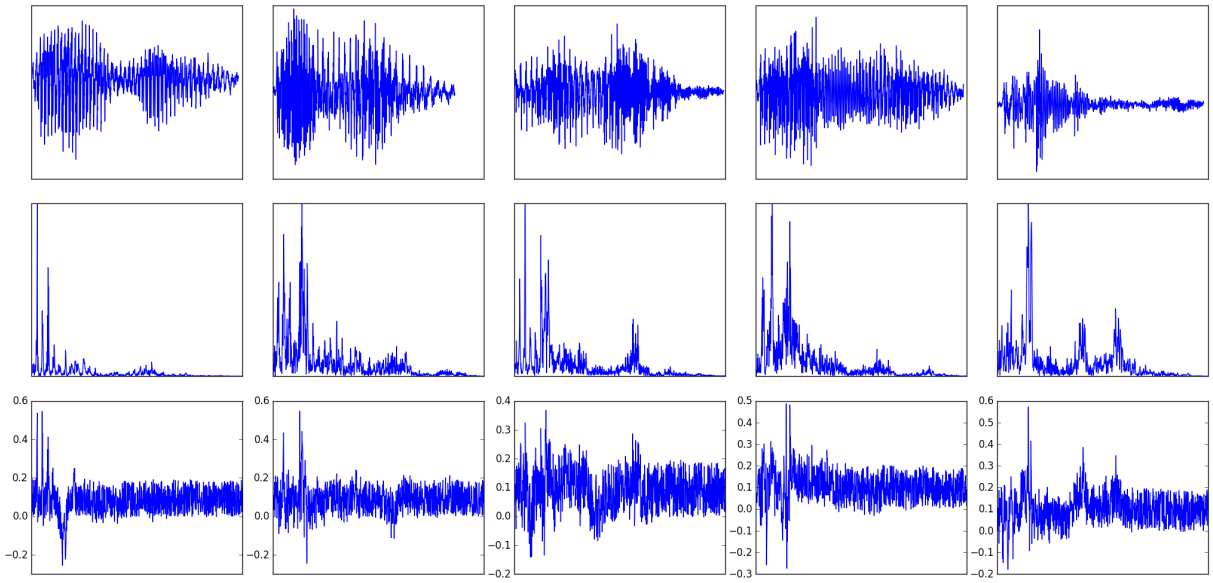


Fig. 12. On the left: the first example of the whole training set, showing the wave and frequency plot of a *Hello*. On the right: the same for the word *Goodbye*. As experienced on Figure 3, an anti-simmetric pattern arises in the weight plot of the classifier perceptrons. One pattern in the frequency domain of *Goodbye* samples is very dominant and the weights of the *Goodbye*-recognizing perceptron fits on it very well, while the same pattern can be found in the opposite perceptron's weight plot with negative sign.



S

Fig. 13. On this figure the sample time, frequency and the perceptrons' weight plots are depicted of the network that recognizes greetings in 5 languages. For human eyes it is hard to tell from either the time and frequency domain plot the corresponding language. The recorded greetings were sampled by me saying *Bonjour, Salem, Hola, Halo* and *Csá*