Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Building, testing and visualizing neural networks from scratch

Csaba Botos

2016

A dissertation submitted for the Council of Scientific Students' Associations

Advisor: PhD. István Z. Reguly
External supervisor: PhD. Csaba M. Józsa

Abstract

Recently a special branch of Machine Learning, a model based on living organic systems called Deep Neural Networks is overtaking previous paradigm of algorithmic problem solving. It gained larger attention when better results were achieved than task-specific, handcrafted models in feature extraction. The reason behind its success is its scalability: the latest architectures are able to exploit the capacity of cutting-edge GPU hardware since the abstraction of the data is accomplished by succeeding neural nodes performing elementary operations, which can be easily parallelized.

It is of the utmost importance to understand the main concept of such networks to contribute to the breakthroughs of the fourth industrial revolution. To this purpose, building a framework from the base unit blocks of the newest models is the best introduction to Machine Learning. In my research I have disassembled black-box represented networks to the very basic, intuitive level and reorganized it in object-oriented manner, where each neural layer is treated as an entity derived from a common ancestor, therefore information flow and processes of the system are easily traced. My design and implementation is based on the principals of the components used by networks built for ImageNet classification, such as Convolutional, ReLU, Max-Pooling, Fully-Connected, Dropout, DropConnect, Softmax and k-Winner-Takes-All layers. Furthermore, the following training methods and policies were adapted: cross validated, mini-batch, on-line, L_p regularized and basic Stochastic Gradient Descent training.

For testing the framework, the parameter space of Fully Connected networks was exhaustively explored. After training and evaluating sessions - mainly performed on the MNIST and self-acquired datasets - the results were gathered to analyze the performance of different architectures. For further investigation the best performing models were compared to each other to find pros and cons of different capacity, layout and training of networks.

Besides architectural experiments, a non-trivial task targeted by many recent research of visualizing the inner representation of information, understanding transient activation patterns was studied as well. Previously mentioned candidate networks were also visualized individually to retrieve information about characteristics of the processes in their Hidden Layers. My implementation proposes a simplification of the DeconvNet derived from Gradient Ascent, an efficient algorithm to reveal patterns recognized by nodes in the hidden layers of Neural Networks, to produce adversarial input samples.

Declaration

I, Botos Csaba, declare that this thesis titled 'Building, testing and visualizing neural networks from scratch', and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signature

Contents

1. Introduction

1 Overview

> *No one knows what the right algorithm is, but it gives us hope that if we can discover some crude approximation of whatever this algorithm is and implement it on a computer, that can help us make a lot of progress.*
>
> — Andrew Ng

Every time we are interacting with our environment, we get closer to understanding it. However, as a byproduct, further questions arise, for which pure logical (mathematical?) solution is not available, or the problem is larger than to be solved by current ones. Luckily we can always turn back to nature for inspiration: biological systems have proven their efficiency, therefore their functions are worth to be further analyzed – even if there could be theoretically a better way to tackle obstacles. One of the many branches of artificial intelligence, Neural Network is based on the nerve systems of living organisms which is capable of self-learning.

Such a simple paradigm is playing a fundamental role in boosting the industry and researches of today, because introducing Machine Learning to many fields of life seems to results in great leap forward. Thanks to the recent technological advancements, even with the current computational capacity of a personal computer one can get encouraging results by simply exploring the core principles of the topic.

Main motivation. By using external libraries [1, 3, 11], without diving deep into mathematical proofs, and treating neural networks as black-boxes, thousands of useful applications [9] are made. On the other hand, building such architectures from the very basics helps to clarify how simple units might be organized, taught [20], and function [10] as a large system. As complexity arises, the processes in the network becomes unclear and brings the question: what is the purpose of each node? Methods to visualize how activation patterns formulate, and what information is held within them has been already investigated [21], and applied to improve performance [22]. My studies mainly rely on recent publications, and research in field of computer vision. The design of my own Deep Learning framework library is influenced by off-the-press tutorials [8, 4, 18, 15, 7] and open-source libraries [1, 3, 11] available to anyone.

Primary goal. With my work I intend to bring closer, and demystify cutting-edge concepts of applied neural networkings for larger audiences. On related works and case studies I want to show

what can be done by simply going back to the drawing board

2 Related Works and Literature

3 Thesis outline and contributions

2. Literature

Considering course slides [15, 5] and textbooks [8, 20, 2] of established universities, popular websites [4, 19], blogs [7, 12] and vlogs [16] on Artificial Neural Networks one might find it hard to find a good point to start. Some content may offer formal description of the general machine learning problem, others try to clarify through analogies with the biological nerve system. Tutorials, which I have found interesting, and the most helpful, had several common features which are important to adapt, when designing a neural network library. Generally these common attributes are the following: they intend to be *simple* as possible, have many *intuitive* examples and analogies, their *interpretations* are not restricted to either universal approximators, or nervous systems of living organisms, but combines both aspect.

3. Design and Implementation

> *... where in this snippet $W1$ and $W2$ are two matrices that we initialize randomly. We're not using biases because meh.*
>
> — Andrej Karpathy

After researching literature on building libraries from scratch, analyzing pet project source-codes [13, 6] and dozens of implementations of professional libraries [1, 11, 3], I concluded that the best way to acquire an in-dept understanding of neural networks is to build my own Deep Learning framework. This way I had a chance to understand why novel solutions in Machine Learning are formed the way they are. I also gained insight into what main paradigms are popular libraries based on, such as *computational graphs, parallel processing*, and what trade-offs can be made between *computational cost* and *memory usage*, between robustness and plasticity. Most thankfully, by starting from scratch I have faced situations when theoretical formulas had to be translated into exact working code, which was an important challenge.

Goals. My main objectives when writing code was the following:

to make such a library that is able to be extended further

open-source, so it can be forked by anyone interested in developing it

to make it modular therefore make its usage independent of the task

use the fewest possible technical tricks for sake of simplicity

to stay as close to pure mathematical formulation of the classical paradigms as possible

put emphasis on ease of use and understanding

Disclaimer. Apart from **NumPy** and its complementary package **SciPy**, no external libraries and dependencies are used in the implementation. I want to emphasize that this work was not written to compete with contemporary state-of-the-art frameworks, rather to help perceive the general ideas behind novel researches, and to encourage interested fellows to carry out researches on their own.

1 Choice of design pattern and language

The formulas of classical neural networks, whose nodes are organized into lattices forming a Directed Acyclic Graph, shows minor variance in contents of independent sources. Still their common point

that they rely on basic vector algebra and functional analysis, therefore considering array represented general mapping functions, as their core building units is a paradigm which would not interfere with the current formal and informal descriptions of deep learning.

1.1 Disassembling a universal approximator

Thinking of a neural net as a function $\mathcal{F}$, is implicitly a black-box representation of the paradigm. Users of different applications which offers feature detection, segmentation, prediction, etc. are using this function without knowing what exactly happens behind the curtains. Further investigating $\mathcal{F}$ we can constrain it to have a DAG computational graph, also to have the nodes of the graph arranged to lattices. Practically it means that the perceptrons making up the layer l are strictly projecting F_l their **inputs** $x_l \in \mathbb{R}^N$ *forward* to scalars, which if all perceptron is evaluated parallel, forms the **output** $y_l \in \mathbb{R}^M$, have no feedbacks and loops. This projection of layer l can be written as $y_l = F_l(x_l)$. Intuitively the input of the next $(l+1)^{th}$ layer will be the output of the previous layer: $x_{(l+1)} = y_l$. For the sake of simplicity I excluded Recurrent Networks from the space of $\mathcal{F}$, but later the definition can be extended for vanilla recurrent networks and LSTM networks as well.

1.2 Language

Keeping in mind, that using a general high-level language (like MATLAB) can yield poor computational efficiency, making benchmarks, testing and applications impossible. Also considering a low-level language (like C) would distract us from the main goal, always optimizing further and further the basic algorithms – else resulting in boilerplate codes. Either way it would make the implementation unclear for those, who did not participate in the designing of the library. I wanted to chose a lagnuage which offers an optimal solution for this challenge, is flexible, well documented and simple enough for newbies to catch up. Because of the support of both object-oriented and procedural approaches and offering the above, I choose Python.

1.3 Design pattern

While carrying out blueprints of the implementation, I examined the Neural Networks as universal approximators in a TOP-DOWN manner. I have disassembled them to basic blocks, abstracting the function of each level. Later I used these units in BOTTOM-UP approach to create an object-oriented hierarchic model that realize simple operations and is well defined on every level: granting a universal interface to be further extended.

2 Single Layered networks

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

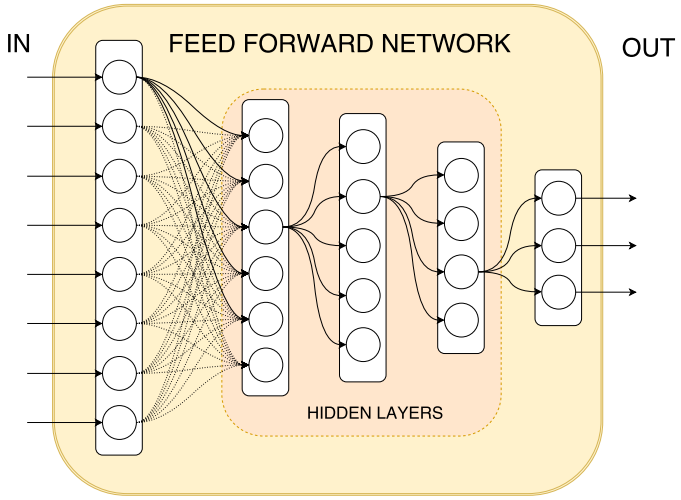


Figure 3.1: A small feed forward network with three hidden layer composed of Fully Connected layers.

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**.

2.1 Defining the Perceptron

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 $\mathbf{w}$

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

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 3.1.

2.2 Basics of Learning

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, see [18]. Anyhow, in general this is still the most studied ques-

tion 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 $\mathbf{w_P}$.

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*.

3 Inference

Let us suppose that for a particular task an $\mathcal{F}$ is given, first what we have to understand is how input data $x \in \mathbb{X}$ (interchangeably x_1) is inferred. First, assume that the data can be expressed as multi-dimensional matrix, like RGB pictures, audio recordings, gene maps. Some networks preserve the spatial information i.e. feature extraction performed on images, while other instances operate on the whole input data i.e. processing audio samples in frequency domain. Either way the output y (or y_L) can be obtained by feeding x to the network: $y = \mathcal{F}(x)$ The core concept is that we can compose such an $\mathcal{F}$ function by applying multiple projections to x. Practically that means sending input through the first layer, the second and all the way through to the last layer L^{th}, which output would be the value of $\mathcal{F}(x)$, the response of the network. Therefore in terms of evaluating $\mathcal{F}(x)$ layer by layer, actually translates to a single function call, which can be unfolded to a sequence of embedded projections:

$$\mathcal{F}(x) = F_L\left(x_L\right) = F_L\left(y_{(L-1)}\right)$$

$$F_L\left(y_{(L-1)}\right) = F_L\left(F_{(L-1)}\left(x_{(L-1)}\right)\right) = F_L\left(F_{(L-1)}\left(\cdots F_1(x)\right)\right)$$

Using the function composition operator $\circ$, rewritten in the classical notation:

$$\mathcal{F}(x) = F_L \circ F_{(L-1)} \circ \cdots \circ F_1(x) \tag{3.1}$$

The 3.1 equation is the most fundamental idea behind feed-forward neural networks, namely the

inference or *forward-propagation* As mentioned above, every layer l is represented by an F_l. The most basic layers are the *Fully Connected* and *Activation* layers.

3.1 Fully Connected layer

These layers carry out the heavy-lifting of inference by performing linear projection and translation transformations. The operations are following the rules of basic linear algebra, where the input $x_{FC} \in \mathbb{R}^N$ and the output $y_{FC} \in \mathbb{R}^M$ are specified as real valued vectors. The parameters of the layer $\phi_{FC} = (W, b)$ are the corresponding weights and biases of each perceptron node in the layer forming a *weight matrix* $W \in \mathbb{R}^{M \times N}$ and a *bias vector* $b \in \mathbb{R}^M$ respectively. Therefore evaluating the output of the Fully Connected layer is the defined by the following:

$$\begin{aligned} y_i &= \left(\sum_j W_{i,j}\ x_j \right) + b_i \\ y &= W \cdot x_j + b \\ [M] &= [M \times N] \cdot [N] + [M] \end{aligned} \tag{3.2}$$

3.2 Activation layer

Nodes in activation layers are introducing non-linearity to the network, by applying the same non-linear activation function to the corresponding output of the previous layer, performing element-wise operation. Let F be an activation layer with activation function f after a fully connected layer with 3 neurons:

$$F(x) = \begin{pmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \end{pmatrix}$$

These functions are essential for the network, since they increase the numerical stability: they *squeeze* or *mitigate* the input preventing the network from *saturation* or *explosion* (numerical of course). Conventionally the following functions are applied most often as activation function:

$$\text{RectifiedLinearUnit(ReLU)} := max\{0, x\} \tag{3.3}$$

$$\text{HyperbolicTangent(TanH)} := \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.4}$$

$$\text{SoftPlus(SP)} := \ln(1 + e^x) \tag{3.5}$$

$$\text{Logistic(Log)} := \frac{1}{1 + e^{-x}} \tag{3.6}$$

The only constraint on these functions that they have to keep the dimension of the input, namely $F_{act} : \mathbb{R}^N \mapsto \mathbb{R}^N$. *Note:* These functions do not have any variable parameters, therefore activation layers cannot be trained.

4 Measuring efficiency

If the function $\mathcal{F}$ mentioned above is given, and satisfies our needs, then we are done. However this is usually not the case, and finding the optimal $\mathcal{F}^*$ is the main challenge targeted by many branches of Machine Learning. Despite it was proven, that standard multilayer feed-forward networks are capable of approximating any measurable function to any desired degree of accuracy [10] if the goal function $\mathcal{F}^*$ is unknown, or too abstract to be *measurable* (i.e. telling how funny a picture is), we cannot utilize the universal approximator.

4.1 Loss

By reformulating the objective, we can define a Loss function $\mathcal{L} : \mathcal{F}(\mathbb{X}) \mapsto \mathbb{R}$, which maps our candidate $\mathcal{F}$ network to a scalar field, that represents the general correctness of $\mathcal{F}$ over the space of possible inputs $\mathbb{X}$ – the lower its value the better $\mathcal{F}$ is performing. By doing so we may apply Machine Learning algorithms that would *minimize* the Loss, therefore $\mathcal{F}$ would converge towards $\mathcal{F}^*$ implicitly.

Note: In practical implementations $\mathcal{F}$ is not evaluated over the whole space of possible inputs, instead in the hope that a small subset of both *training* and *validating* samples called a mini-batch will approximate $\mathcal{L}(\mathcal{F})$ as well. Useful practices for reducing computing complexity and improving stability, and rate of convergence will be covered later.

4.2 Supervised learning

In cases where the parameters of $\mathcal{F}^*$ is not known but we know how it would map the input space $\mathbb{X} \mapsto \mathbb{Y}$, e.g. which character appears on the input image x, then we can define a set of previously *labeled* pairs of input - solution sample which could be used later on for training, and evaluating performance of the network.

4.3 Unsupervised learning

When no labeled dataset is available, the network still can be used for extraction of hidden structure of the unlabeled samples. Later these instances are used as density estimators, or adapted as feature extractors for larger networks. **Generative Networks.** Training such architectures can be done by feeding networks random noise as input and training them to reproduce given samples: $\mathcal{F} : \mathbb{R}^k \mapsto \mathbb{X}$, hence the name *Generative Networks*.

Auto-encoder Networks The other frequently applied paradigm is setting the objective task to compress the input sample into a $h \in \mathbb{R}^k$ hidden representation vector that is able to preserve the key information about the original input, and either by symmetric (Restrictive Boltzmann Machines) or independent (Deep Belief networks) operations decompress the data.

In both cases hyper-parameter k is intuitively the number of unlabeled features in the *latent space* that the network will be able to categorize, i.e. correlation between different color intensities on pictures taken of stained brain samples.

5 Adjusting parameters

Generally speaking, applying Machine Learning algorithms boils down to the process of iteratively altering parameters ϕ of $\mathcal{F}$ to optimize the Loss. Once $\mathcal{L}(\mathcal{F})$ is obtained, we can evaluate how changing the parameters would influence it – evaluate the gradient $\nabla_\phi \mathcal{F}$ of the parameters with regards to $\mathcal{L}$.

5.1 Gradient Descent

Updating ϕ by descending on the gradient slope with a small step size ϵ will decrease $\mathcal{L}$. Enucleating the general form of Gradient Descent depends on the architecture of the network, but there is a main concept for doing so, called Backpropagation described by Werbos *et al.* [20].

Train policies

5.2 Networks in practice

Though networks can vary in shape and function, the representation of the succeeding layers, evaluated by embedded functions described in 3.1 is a common feature. Each of these layers serve as nodes of the computational graph of the network. The type of the layer determines whether it can be updated or not: for *Fully Connected* layers $\nabla_\phi F_{FC}$ is well defined, explained in the following example.

Toy example. Assume $\mathcal{L}$ is given, and we have a fully connected network with 2 hidden layers, i.e. $L = 3$, which maps a N dimensional input vector x to a M dimensional output vector y, namely $\mathcal{F} : \mathbb{R}^N \mapsto \mathbb{R}^M$. The constraint on the parameters are:

W^1 of the first layer must have N columns

W^3 and the bias b^3 of the last layer must have M rows, dimensions respectively.

Number of rows of W^l must match $dim(b^l)$ 3 dimensional

Number of columns of W^l must match $dim(b^{(l-1)})$

Then the evaluation unfolded would look like the following:

$$y = \mathcal{F}(x) = F_3 \circ F_2 \circ F_1(x) = W^3(W^2(W^1(x) + b^1) + b^2) + b^3$$

For further usage and simplicity, I would like to fix these numbers. Let $\mathcal{F}$ be a network with the following *shape*: $[5, 4, 3]$ meaning that in each layer there are 5, 4, 3 neurons respectively, $M = 3$ and all nodes are connected to the previous layer. The width of the input layer is not yet defined, let it be $N = 10$. *Note:* It is usually distracting and redundant to explicitly write the width of the outermost layers when testing different networks, because the input and the output layers must have fixed dimension for the same task, while the width of the hidden layers are varied. Define an L_2 Loss

function on the toy example. For one sample-label pair (x, y^*) the L_2 Loss is:

$$\mathcal{L}_2(\mathcal{F}) = \frac{1}{2}\sum_i (y_i^* - \mathcal{F}(x)_i)^2 = \frac{1}{2}\sum_i (y_i^* - y_i)^2 \tag{3.7}$$

L_2 is a universal Loss function, which is used in cases where the label space $\mathbb{Y}$ is continuous (e.g. floorspace, consumption and height of a house, based on the price: $\mathbb{R}^1 \mapsto \mathbb{R}^3$).

6 Differentiation

In the above evaluating of $\nabla_\phi \mathcal{F}$ can be done in two ways, namely by numerical approximation, or by analytical derivation, in the following I will discuss both.

6.1 Numeric differentiation

Evaluating the numerical gradient (or difference) is an elementary, yet powerful operation, in which we would *perturb*, or modify one parameter ϕ of our system $\mathcal{F}$ at once. That is done by first adding ϕ^+ and after subtracting ϕ^- a little amount $d\phi$ from the original ϕ and evaluate $\mathcal{L}^\pm = \mathcal{L}(\mathcal{F}_{\phi^\pm})$, namely the *Loss* of the system in the modified state, yielding the numerical gradient in the following equation:

$$\frac{d\mathcal{L}(\mathcal{F})}{d\phi} = \frac{\mathcal{L}^+ - \mathcal{L}^-}{2d\phi} = \frac{\mathcal{L}(\mathcal{F}_{\phi^+}) - \mathcal{L}(\mathcal{F}_{\phi^-})}{2d\phi} \tag{3.8}$$

Summary. In a nutshell value of $\frac{d\mathcal{L}(\mathcal{F})}{d\phi}$ tells how changing the parameter ϕ by $d\phi$ would change the performance of the network. If it is positive then updating $\mathcal{F}$ by adding $d\phi$ to ϕ would result in higher Loss value, which is the opposite of our goal, so we just subtract it, if it is negative, than trivially we should add $d\phi$ to ϕ since it is making some good progress.

6.2 Complexity

Though letting the computer do the hard work seems to be a good idea, it worths considering that the simple method above will be applied to every ϕ of $\mathcal{L}$. It means that for the network in the toy example, we need to evaluate $\mathcal{L}(\mathcal{F})$ two times for each parameter in the weight matrices and the bias vectors of the network, totaling in

$$\#(\phi) = 2 \times \sum_{i=1}^{3} N_{(i-1)} \cdot N_i + N_i = 2 \times (10 \cdot 5 + 5 ... + 4 \cdot 3 + 3) = 188$$

Even if $\mathcal{F}$ is approximated by using k-sized mini-batches for evaluation it is still a computationally very expensive function, because the inference would result in the following number of operations of addition and multiplication:

$$\#(\text{operations}) = k \times \sum_{i=1}^{3} 2 \cdot N_{(i-1)} \cdot N_i + N_i = k \times 176$$

Therefore approximated with $k = 10$ mini-batches would a single parameter update of a very tiny network would require total operations of:

$$\#(\text{total}) = \#(\phi) \times k \times \#(\text{operations}) = 188 \cdot 10 \cdot 176 = 330880 \tag{3.9}$$

Because both $\#(\phi)$ and $\#(\text{operations})$ has complexity of $\mathcal{O}(N^2)$, one update will yield complexity of

$$\#(\text{total}) = \mathcal{O}(N^4) \tag{3.10}$$

We can see that even for a shallow and relatively small network (industrial AI networks has billions of parameters, and uses much larger batches) described in the toy example the method is really costly. That encourages us to derive our differentials on paper first, and use *numerical gradient approximation* for checking our solution. Using *Gradient Check* is essential when implementing new architectures, because it is a very efficient tool for debugging in comparison with updating. The method in a few words is about setting an error rate ϵ, and decrease $d\phi$ until the numeric solution does not match the analytic solution with $1 - \epsilon$ significance. If the analytic solution is incorrect the cycle will not terminate.

6.3 Analytic differentiation

Deriving the update by hand requires basic knowledge in calculus extended to multivariate cases, though since the operations are elementary, in general we must understand only three basic definitions to do so. Some formality before starting: we have three independent variables x, y, z and functions f, g, h. The result of operations performed on variables, i.e. $x + 2y$ can be represented by a function $f = x + 2y$. If the value depends on a variable then it can be written explicitly, passing the variable as *the argument* of the function $f(x, y) = x + 2y$. For the sake of simplicity assume that the variables are not general objects from an abstract space, they are only real values: $x, y, z \in \mathbb{R}$. However the following description could be extended for the above-mentioned variables as well. For any one-dimensional function $f(x) : \mathbb{R} \mapsto \mathbb{R}$ we say that the value represented by the function depends on the variable by the extent of its derivative. The derivative (or differential) of the function can be seen as an ideal case of 3.8 where the perturbation would approach zero, namely:

$$\frac{\partial f}{\partial x} = \lim_{dx \to 0} \frac{df(x)}{dx} = \lim_{dx \to 0} \frac{f(x + dx) - f(x - dx)}{2dx} \tag{3.11}$$

Multiplication rule. Consider a value $x \cdot y \cdot z$ represented by $f(x, y, z)$. f is now depending on three variables, we can define the measure of this dependency on one variable by the formal equation:

$$\begin{aligned}\frac{\partial f}{\partial x} = \lim_{dx \to 0} \frac{f(x + dx, y, z) - f(x - dx, y, z)}{dx} &= \lim_{dx \to 0} \frac{((x + dx) \cdot y \cdot z) - ((x - dx) \cdot y \cdot z)}{2dx} \\ = \lim_{dx \to 0} \frac{(x + dx) - (x - dx)}{2dx} yz &= \lim_{dx \to 0} \frac{2dx}{2dx} yz = yz\end{aligned}$$

We can apply the same method for each variable, the result will be the elements of the *gradient* ∇f

$$\frac{\partial f}{\partial x} = yz \qquad \frac{\partial f}{\partial y} = xz \qquad \frac{\partial f}{\partial z} = xy \tag{3.12}$$

The important thing to understand that in a computational graph, a multiplicative node, which takes N arbitrary parameters (or arguments), will have a *partial derivative* for each variable its output is depending on. In general if these derivatives are represented as a vector, then it is called the gradient ∇f of f. Also the value of the derivative will be the product of all variables except the one of which we are computing the influence of on the output.

Addition rule. Consider a value $x \cdot y + x \cdot z$ represented by $g(x, y, z)$. The change of g with respect to x is defined with the following shortened equation:

$$\frac{\partial g}{\partial x} = \lim_{dx \to 0} \frac{((x + dx)y + (x + dx)z) - ((x - dx)y + (x - dx)z)}{dx} = y + z \tag{3.13}$$

Notice that – in the terms of computational graphs – if a node contributes to other different operations (namely $x \cdot y$ and $x \cdot z$), than the derivative of each occurrence in *later* values will be summed up.

Chain rule. Let $f(x) = 2x + 3$ and $g(f) = 5f$. Suppose that we would like to know the derivative of g with respect to x. At first we cannot do so, but there are two options: in the hope that substituting the value represented by f into g would not make the equation too complex we can unroll the references and rewrite $g(x) = 5 \cdot (2x + 3)$, or we could use the chain rule:

$$\frac{\partial g}{\partial x} = \lim_{dx \to 0} \frac{g(f(x + dx)) - g(f(x - dx))}{2dx}$$

$$\text{Assume that } f(x + dx) - f(x - dx) \neq 0.$$

$$\lim_{dx \to 0} \frac{g(f(x + dx)) - g(f(x - dx))}{f(x + dx) - f(x - dx)} \cdot \frac{f(x + dx) - f(x - dx)}{2dx}$$

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x} \tag{3.14}$$

Which is a formula of the products of partial derivative of g, that treats f like a variable, and f explicitly operating on variable x. The derivative of g with respect to variable x is the product of the *local derivative* of g is $\frac{\partial g}{\partial f} = 5$ and $\frac{\partial f}{\partial x} = 2$ which equals $\frac{\partial g(x)}{\partial x} = \frac{\partial (5 \cdot (2x+3))}{\partial x} = 10$, the function strictly depending on x. The important message is that we can interchangeably use function values and variables with a constraint that at a point, in an arbitrary depth there must be a real variable. *Note:* the statement above stands for computations with *Acyclic Graph*, meaning that there should not be any feedbacks or loops – no definitions like $f(g), g(f)$ or $f(f)$. We will see that these rules play a very fundamental role in training networks.

Vector Notation. Before drilling deep into mathematical equations, a small reminder: the following vector and matrix formulation is just a special annotation, using the rules above, which helps to make clear both the definition, and the computations done by the network when it is implemented. The vectors with partially derivatives inside are just representing *real values*, arranged in a fancy way. Every vector and matrix defined in forward propagation, has its corresponding derivative w.r.t. the Loss. More trivially, if any value is depending on a list of variables $f(x_1, x_2 \cdots x_n) = f(\mathbf{x})$ (a vector) then there is a list of *partial derivatives* w.r.t. to f – forming the gradient $\nabla f = \left(\frac{\partial f}{x_1}, \frac{\partial f}{x_1} \cdots \frac{\partial f}{x_N}\right)$.

Notation: When the vector notation is emphasized the variable name is conventionally written in bold font $\mathbf{x}$, or is underlined $\underline{x}$. Because later the indexing would become too crowded, we only use the indexed notation when it is necessary, otherwise using x.

The next step is formulating the dependency of multiple functions on multiple variables. As seen above, a multivariate function has its gradient vector – in the same fashion as the list of variables were organized into vectors $\mathbf{x}$, values composed of them can also form a vector $F = (f_1, f_2 \cdots f_M)$, composing a multivalued function depending on the same variables. Doing so yields a first-order derivative matrix, composed of gradient vectors $J = (\nabla f_1, \nabla f_2 \cdots \nabla f_M)^T$, called the *Jacobian of* F. The Jacobian has as many rows as output values F has, and the same number of columns of the variables that f_i is a function of.

$$J(F) = \begin{pmatrix} \nabla f_1 \\ \vdots \\ \nabla f_M \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{pmatrix} \tag{3.15}$$

The matrix and vector operations (such as addition and inner product) that can be performed on the derivative *arrays* are identical defined in the inference section. That is important because product of derivatives introduced by the chain rule, can be applied as well for multidimensional array of derivatives too.

6.4 Fully Connected Layer.

Return to the toy example and begin with the last layer, with 3 nodes. If a sample x is inferred $\mathcal{F}(x) = y$, then the response of the network will be a vector of $dim(y) = 3$. If we took the L_2 *distance* between the response and the goal y^*, then it would tell how far we are from the ideal, by a single scalar value. Since we want to minimize it, we have to adjust the parameters of the network, namely descend on the gradient slope. To get the small extent of the update we have to evaluate $\frac{\partial \mathcal{L}}{\partial \phi}$. Intuitively in the case we would like to correct the weights of a decision, it would require two things:

The original situation (the input of the l^{th} layer x_l), which the decision was made in.

The error on the decision – the derivative δ^l of the Loss with regards to the decision.

Since x_l is obtained via inference, what we have to calculate is δ^l for the l^{th} layer in order to acquire the parameter gradient. The first step is to evaluate the δ_L, or the *error* of the last layer's response y^3, namely $\delta^3 = \nabla_{y^3} \mathcal{L}$. Begin with the first element:

$$\delta_1^3 = \frac{\partial \mathcal{L}}{\partial y_1} = \frac{\partial}{\partial y_1} \frac{1}{2} \left((y_1^* - y_1)^2 + (y_2^* - y_2)^2 + (y_3^* - y_3)^2 \right) = y_1 - y_1^*$$

Expanding it to the whole array:

$$\delta^3 = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial y_1} \\ \\ \frac{\partial \mathcal{L}}{\partial y_2} \\ \\ \frac{\partial \mathcal{L}}{\partial y_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial y_1} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \\ \\ \frac{\partial}{\partial y_2} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \\ \\ \frac{\partial}{\partial y_3^3} \frac{1}{2} \sum_i (y_i^* - y_i)^2 \end{pmatrix} = \begin{pmatrix} y_1 - y_1^* \\ \\ y_2 - y_2^* \\ \\ y_3 - y_3^* \end{pmatrix}$$

Consider the following notation: $\delta_i^3 = (y_i^* - y_i)$, called parametric vector notation. Writing arrays in this way, saves a lot of space. However, when this notation gets jammed with indexes, it is useful to write down explicitly the whole array for clarification.

The last layer Now we have exact values of δ^3 and x^3, so we can calculate how should the weights in W^3 be changed in order to get a better network. Applying the differentiation rules for each weight (forming a matrix) of the layer will result in a derivative for each weight (also forming a matrix). Taking the first perceptron of the layer, it has a weight $W_1^3 = (W_{1,1}, W_{1,2}, W_{1,3}, W_{1,4})$ for each output of the previous layer.

Suppose that this neuron had to tell how rounded is the object on an image sample, and the i^{th} neuron of the 2^{nd} layer fires when it recognizes sharp edges. Of course it would be bad if our neuron had a large weight on x_i^3. If this node performs poorly because of x_i^3, then it would contribute a lot to the Loss function while inferring sharp objects, with its output y_1 being far away from y_1^*, resulting in a positive δ_1^3. In case of $x_i^3 = 0$, the weight $W_{1,i}$ has nothing to do with the error δ_1^3 of the neuron. Notice that the error of each weight (w.r.t $\mathcal{L}$) should be proportional to the error and the input as well: $\frac{\partial \mathcal{L}}{\partial w_{1,i}} = x_i^3 \cdot \delta_1^3$. However it can be also derived in terms of the differentiation rules:

$$\frac{\partial \mathcal{L}}{\partial W_{1,i}} = \frac{\partial \mathcal{L}}{\partial y_1^3} \cdot \frac{\partial y_1^3}{\partial W_{1,i}} = \delta_1^3 \cdot x_i^3$$

Considering the parameter update. assume that a picture of an origami sculpture (a very edgy one) was inferred and the i^{th} node of the second layer worked correctly. Both x_i^3 and δ_1^3 is positive, however the weight should be decreased: that is why we will take the negative of the derivative for updating $w_{1,i}$. Substituting $i = 1, 2, 3, 4$ into $\frac{\partial \mathcal{L}}{\partial w_{1,i}}$ yields a gradient of $\mathcal{L}$ with regards to the weights of the first neuron of the last layer. Doing so for each neurons in the layer would result in 3 gradient vectors $\nabla_{W_1}\mathcal{L}$, $\nabla_{W_2}\mathcal{L}$ and $\nabla_{W_3}\mathcal{L}$ which is practically stacked to make a matrix which can be later added element-wisely to the weight matrix W.

$$\nabla_W \mathcal{L} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial W_{1,1}} & \frac{\partial \mathcal{L}}{\partial W_{1,2}} & \frac{\partial \mathcal{L}}{\partial W_{1,3}} & \frac{\partial \mathcal{L}}{\partial W_{1,4}} \\ \frac{\partial \mathcal{L}}{\partial W_{2,1}} & \frac{\partial \mathcal{L}}{\partial W_{2,2}} & \frac{\partial \mathcal{L}}{\partial W_{2,3}} & \frac{\partial \mathcal{L}}{\partial W_{2,4}} \\ \frac{\partial \mathcal{L}}{\partial W_{3,1}} & \frac{\partial \mathcal{L}}{\partial W_{3,2}} & \frac{\partial \mathcal{L}}{\partial W_{3,3}} & \frac{\partial \mathcal{L}}{\partial W_{3,4}} \end{pmatrix} = \begin{pmatrix} x_1\delta_1 & x_2\delta_1 & x_3\delta_1 & x_4\delta_1 \\ x_1\delta_2 & x_2\delta_2 & x_3\delta_2 & x_4\delta_2 \\ x_1\delta_3 & x_2\delta_3 & x_3\delta_3 & x_4\delta_3 \end{pmatrix}$$

The last part of the equation can be also expressed as:

$$\nabla_W \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial W_{i,j}} \right) = (x_j \delta_i) = x \wedge \delta \tag{3.16}$$

Where $\wedge$ denotes the *outer product* operator. The gradient of the bias is simply $\nabla_{b^L}\mathcal{L} = \delta^L$.

The $(L-1)^{th}$ layer. Gradient descent can be applied on networks with more than one layer, however continuing the example of the image descriptor network requires a bit more abstraction. In the previous explanation we assumed that the i^{th} neuron of the second layer worked properly. In general cases this assumption is incorrect, if the whole network is initialized at once. If we think

about that also the mentioned neuron in the *last hidden layer* should be trained, then we can apply the same method with a 4 dimensional $\delta^{(L-1)}$ and with a 5 dimensional $x^{(L-1)}$ input.

Note: Capital L is representing the number of layers. In the toy example $L = 3$.
Acquiring the $\delta^{(L-1)}$ is where the chain rule (3.14) steps into the scene. While in the example before we could find an intuitive workaround, in this case it would be quite strained, since $\mathcal{L}$ does not depend directly on $y_{(L-1)}$. Utilizing the chain rule:

$$\delta^{(L-1)} = \frac{\partial \mathcal{L}}{\partial y^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial y^L} \cdot \frac{\partial y^L}{\partial y^{(L-1)}} = \delta^L \cdot J(F_L)$$

Where $J(F_L)$ denotes the *Jacobian* (3.15) of the function representing the projection of the last layer $F_L : \mathbb{R}^4 \mapsto \mathbb{R}^3$. It is a map of how each input variable affects the output of the layer. In general: the Jacobian numerically can be represented as a 3×4 matrix, analytically as a *local derivative* of function F. In case of *Fully Connected* layers the Jacobian is simply the weight matrix $F(F_l) = W_l$ (the bias drops out here).

For derivative of scalar values ($\mathbb{R}$) the *product* operator is well defined, and it can be expanded the same way to derivatives of multidimensional values as regular *inner product* of the values they are composed of. *Note*: In order to stay consistent with the dimensions of the computation, we have to switch sides of the matrix multiplication defined in (3.2):

$$\begin{aligned} \delta_j^{(L-1)} &= \sum_j \delta_i^L \, W_{i,j}^L \\ \delta^{(L-1)} &= \delta^L \cdot W^L \\ [4] &= [3] \cdot [3 \times 4] \end{aligned} \tag{3.17}$$

6.5 Backpropagation

The backpropagation algorithm, first described by Werbos *et. al* [20]

The l^{th} layer. The contribution to the Loss of the general l^{th} layer δ^l can be retrieved by unfolding $\frac{\partial \mathcal{L}}{\partial y^l}$ applying the chain rule, namely the backpropagation:

$$\begin{aligned} \delta^l = \frac{\partial \mathcal{L}}{\partial y^{(L-1)}} &= \frac{\partial \mathcal{L}}{\partial y^L} \cdot \frac{\partial y^L}{\partial y^{(L-1)}} \quad \cdots \quad \frac{\partial y^{(l+1)}}{\partial y^l} \\ \delta^l &= \delta^L \cdot J(F_L) \cdot J(F_{(L-1)}) \quad \cdots \quad J(F_{(L-1)}) \\ [dim(l)] &= [dim(L)] \cdot [dim(L) \times dim(L-1)] \cdots [dim(l+1) \times dim(l)] \end{aligned} \tag{3.18}$$

Note: the order of evaluating these derivatives is theoretically irrelevant, however computationally there is an opportunity to implement it in two ways [14]:

forward-mode differentiation: evaluating in the order of layers is efficient in cases where the output of the network is much larger than the input

reverse-mode differentiation: evaluating in reversed order for networks with fewer outputs than inputs.

As pointed out in the thesis, the former would require $(L - l)$ times evaluating a *matrix-matrix* product and one *vector-matrix* operation, while the latter would require $(L - l)$ times evaluating a *vector-matrix* product and one *matrix-matrix* at the end. Using forward-mode differentiation does not require keeping transient activations y^l, however it is computationally costly. Using backward-mode differentiation does not strain the CPU, but the memory. It is because if $J(F_l)$ is not linear, then it requires the input x_l to evaluate the first order derivatives.

6.6 Activation Layer

Though the activation layer operates on the input, it has no adjustable parameter. Since we have to evaluate δ for layers behind activation layers, the error must pass through $J(F_{activation}$ as well. Luckily activation layers applies a scalar function element-wise on the inputs, the Jacobian of the composite function F has a special attribute: it is **diagonal**, meaning that:

$$J(F) = \frac{\partial F_i}{\partial x_j} = \begin{cases} \frac{\partial f(x_i)}{\partial x_i} & i = j \\ 0 & i \neq j \end{cases}$$

We can exploit this function when implementing activation layers, by simply using element-wise product $\frac{\partial f(x_i)}{\partial x_i}$ instead of a matrix multiplication. The mentioned activation functions (3.3, 3.4, 3.5, 3.6), are differentiable *almost everywhere*, the corresponding derivatives:

$$(\text{ReLU})' := \text{Heaviside}(x) \tag{3.19}$$

$$(\text{TanH})' := 1 - \tanh(x)^2 \tag{3.20}$$

$$(\text{SP})' := \frac{1}{1 + e^{-x}} \tag{3.21}$$

$$(\text{Log})' := \text{Logistic}(x) \cdot (1 - \text{Logistic}(x)) \tag{3.22}$$

4. Case studies and Results

1 Voice recognition

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 sample 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.

Technical background For the sake of readability 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.

1.1 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 4.2. For the script which was used to produce the figures check out demo

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 4.3, 4.4. However the importance of choosing 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

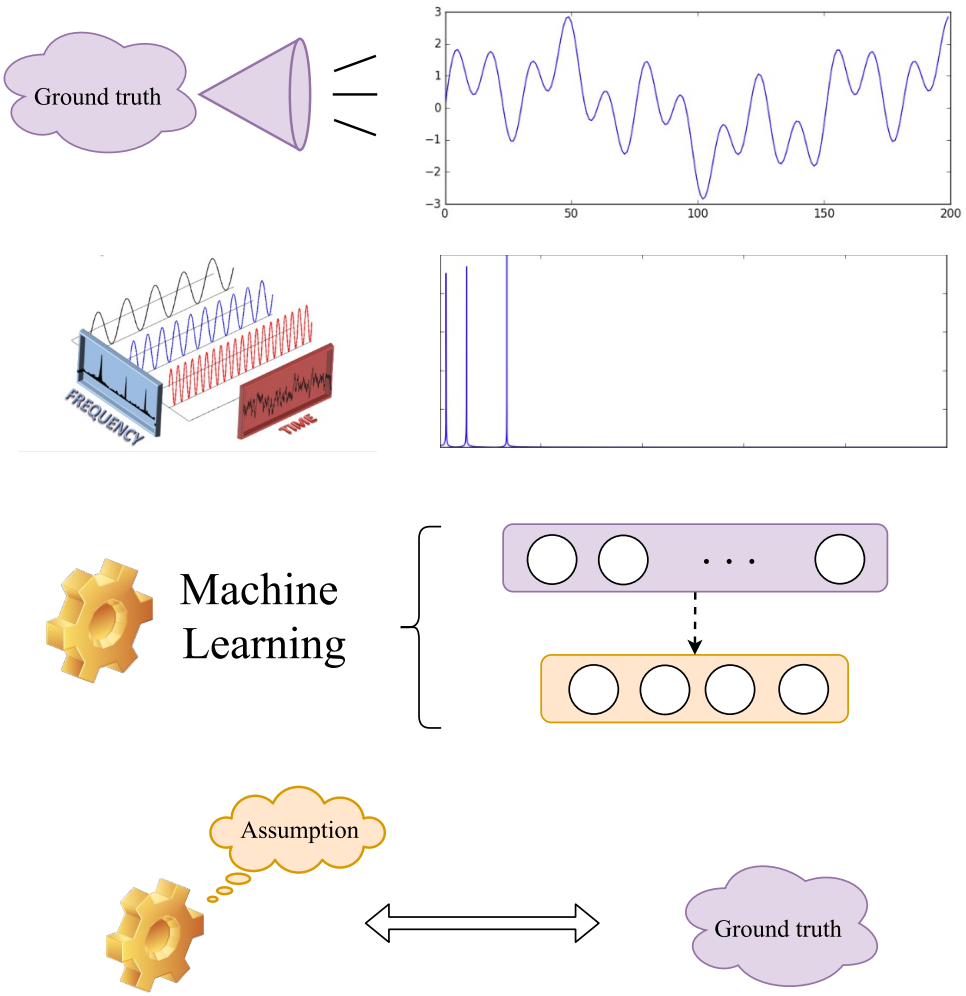


Figure 4.1: The basic conception of voice recognition: the original message is first expressed in physical waveforms which may introduce a lot of noise in real life. After recording, the wave from time domain is transferred to *frequency domain* that tells which frequencies were dominant in the captured signal sample. According to nature's biological systems in this form the same information can be better processed.

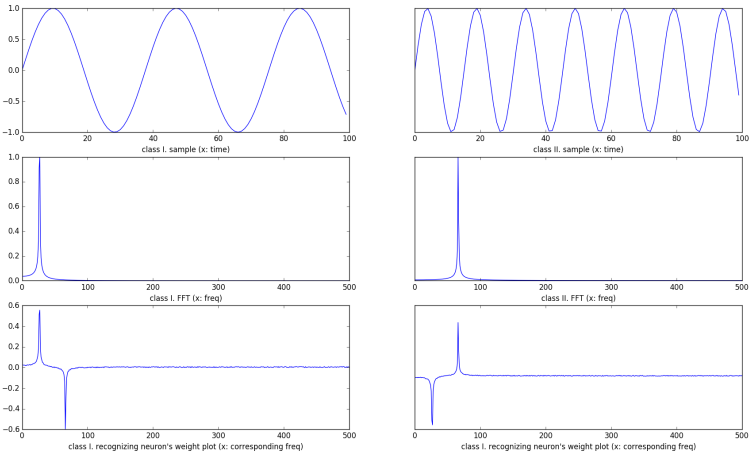


Figure 4.2: On the first row the original sample is shown in time domain. On the second row the N-FFT of it. On the last row the plot of the weights of the corresponding perceptron. The first feature to recognize, that each perceptron learned that not just to turn on for the corresponding input sample, 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

applications it is a trade-off between quality and speed, because the computation complexity for this architecture is $\mathcal{O}(N^2)$, therefore choosing too large N results in slow processing.

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 4.2, 4.3, 4.4 the weights were originally 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 4.5. However instead of regularizing we can *over train* 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 4.6.

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 4.7 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 conclusion is that the perceptrons (if well trained) will look only for relevant signals and patterns in the input. See Figure 4.8

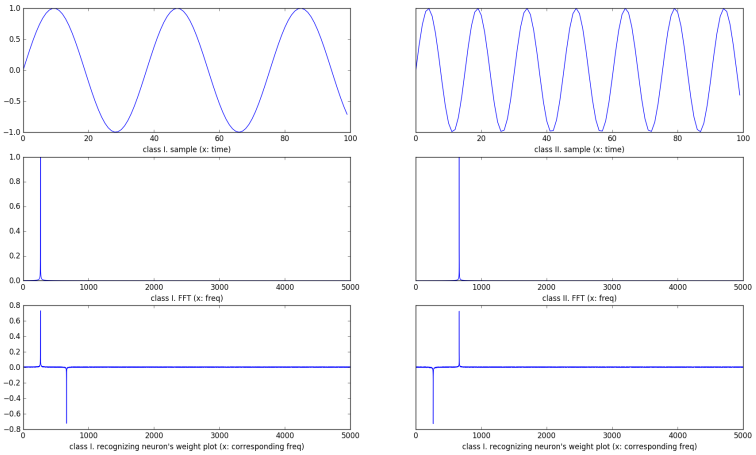


Figure 4.3: The network's initialization and training parameters are identical to the network depicted on Figure 4.3, only the transform precision is increased from $N = 500$ to $N = 5000$ resulting in more narrow spikes corresponding to the recognized frequency

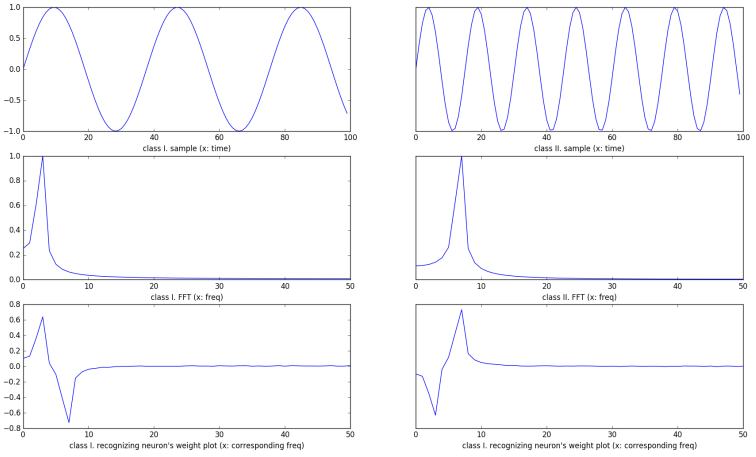


Figure 4.4: The network's initialization and training parameters are identical to the network depicted on Figure 4.2, only the transform precision is decreased from $N = 500$ to $N = 50$ resulting in more shallow spikes corresponding to the recognized frequency

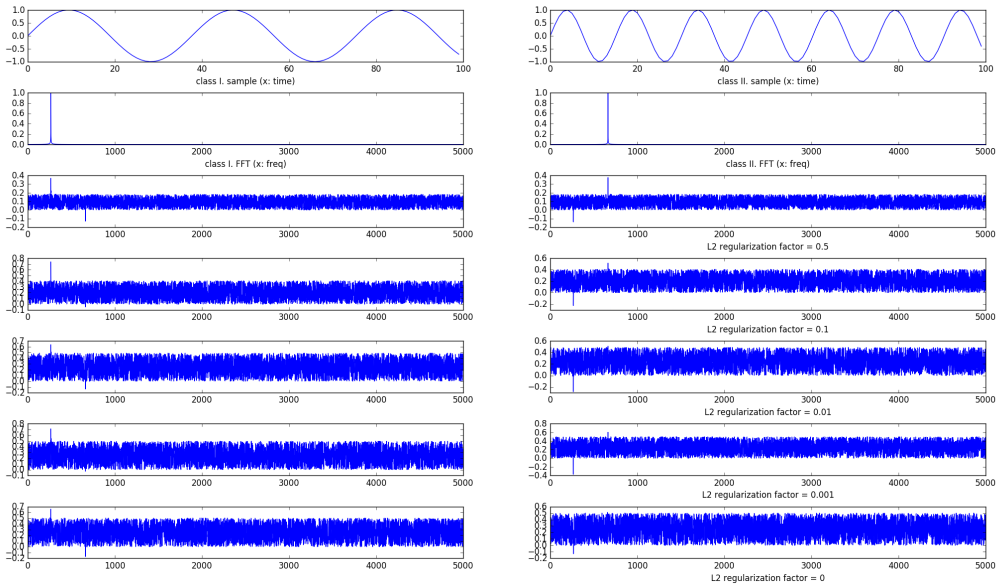


Figure 4.5: After the third row the figure depicts independently, identically initialized and trained networks detailed in Figure 4.2, only the regularization factor λ is decreased from 0.5 to 0.0001 resulting in more noisy weight plot. Still the corresponding weights' magnitude are outstanding

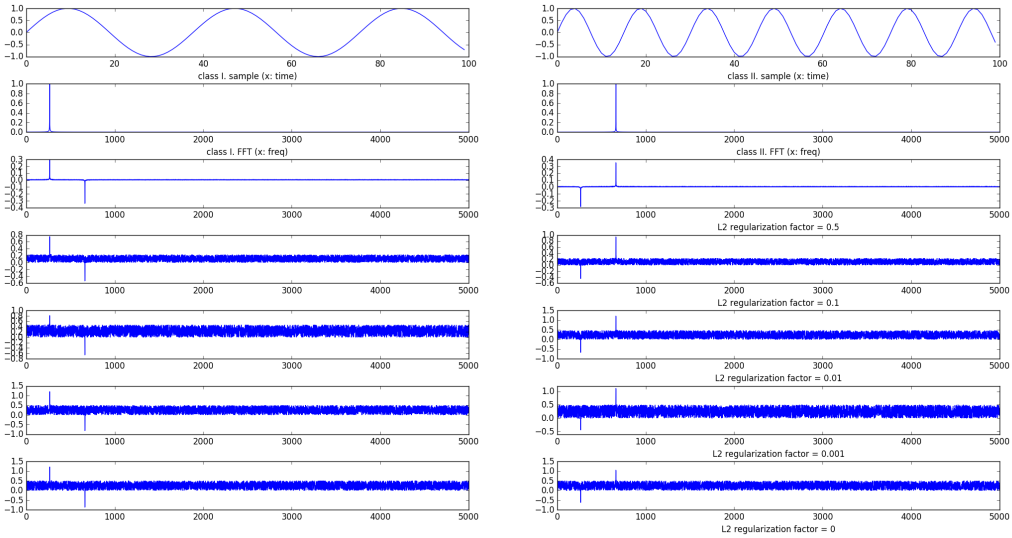


Figure 4.6: The network was trained with the same parameters as the one depicted on 4.5, except that it was trained for five times longer. Despite the result is quite the same as the result achieved with greater regularization factor, overfitting should be avoided. Overfitting occurs when the network 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.

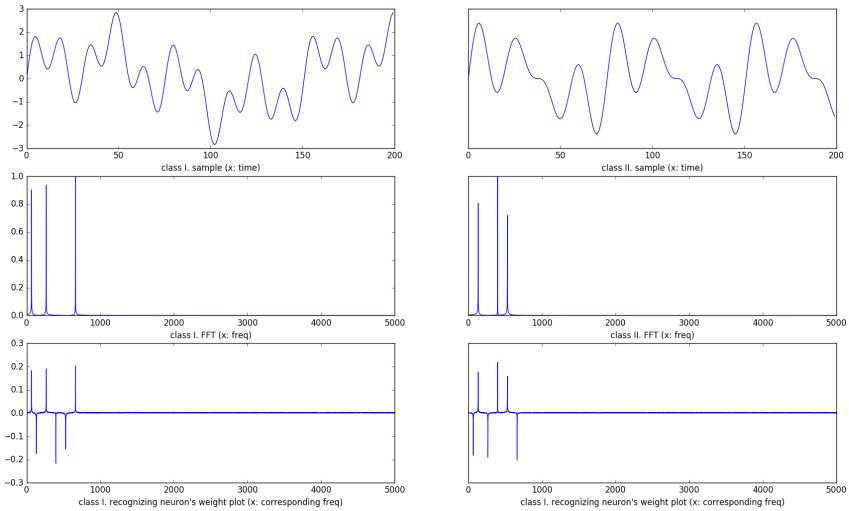


Figure 4.7: Each sample is an equal combination of 3 independent sine function. The frequency of the waves are not separated to low and high frequencies, both can be found in each sample, but it is important to notice, that the two samples do not have any common component.

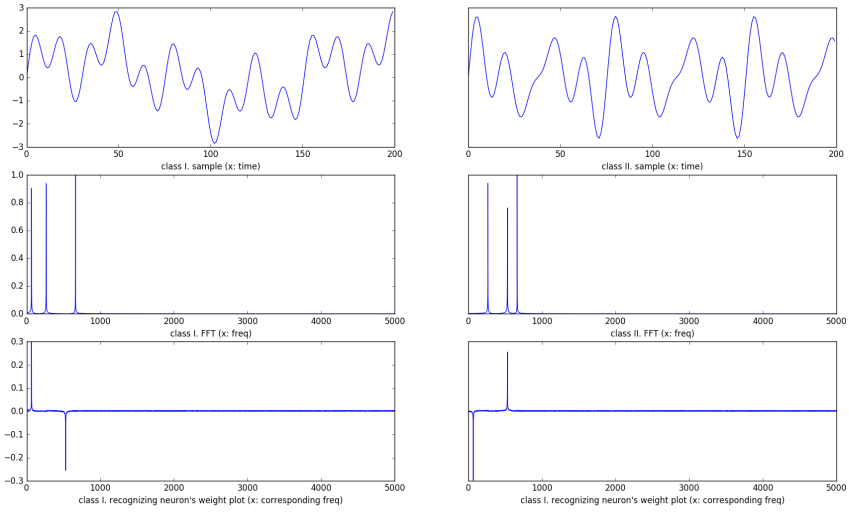


Figure 4.8: In opposite to the case depicted on Figure 4.7 here the samples have two common component, meaning that in the base functions there are two common frequency and only differs in one harmonics. The result is that the common features extracted by the FFT are discarded, and the perceptrons only learn the difference.

1.2 Real Samples

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.

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 threshold is set for recognizing 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 threshold. 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.

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 irrelevant 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*.

Binary classifiers

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 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 4.11.

1.3 Multiclass classification

Recognizing greetings in multiple languages Currently the model somehow works like the one depicted on Figure 4.9. 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 4.10

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

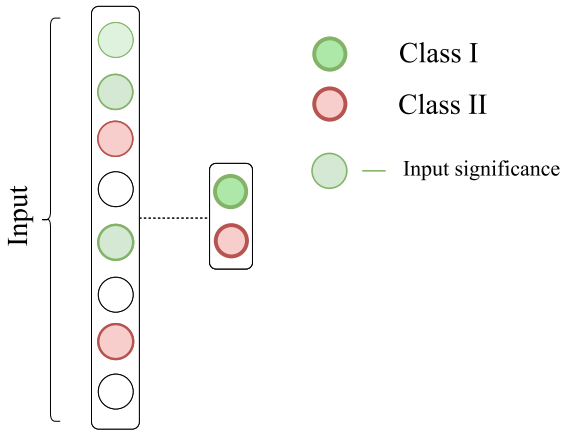


Figure 4.9: A basic perceptron model, operating on 8 clues of the input. The perimeter of each input node describes how much it increases the output of the perceptron with the same color and decreases the neuron with opposite color. Input nodes which are not colored holds irrelevant informations for both perceptrons, therefore discarded.

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 hyperparameters: 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.

1.4 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 recognize whether a user said yes or no, without explicitly telling which language he or she was speaking.

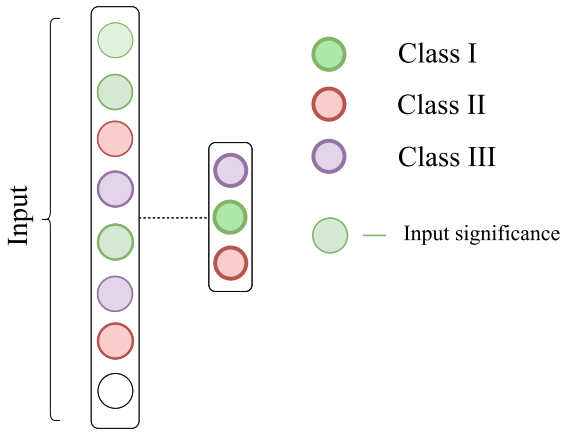


Figure 4.10: An improved perceptron model, operating on 8 clues of the input $\mathbf{x}$, deciding which class it belongs to. The weight anti-symmetry is not so trivial in this case because perceptrons of multiple classes can be turned on by one single feature while other perceptrons are inhibited by the same feature or may even discard it. Meaning that the colors on this figure are a bit tricky, because each input node can be shared between the classes.

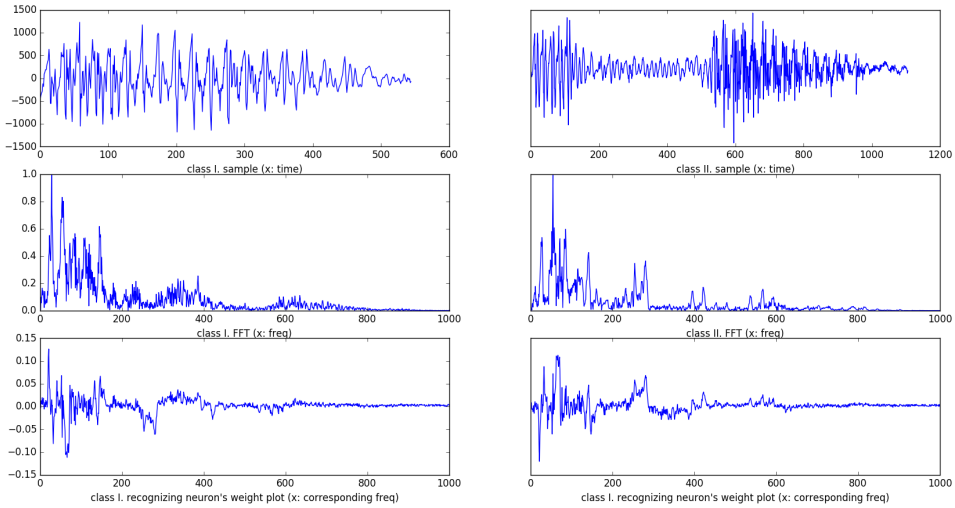


Figure 4.11: On the left: the first example of the whole training set, showing the wave and frequency plot of a *Hello*. On the right: the same for the word *Goodbye*. As experienced on Figure 4.2, an anti-symmetric pattern arises in the weight plot of the classifier perceptrons. One pattern in the frequency 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.

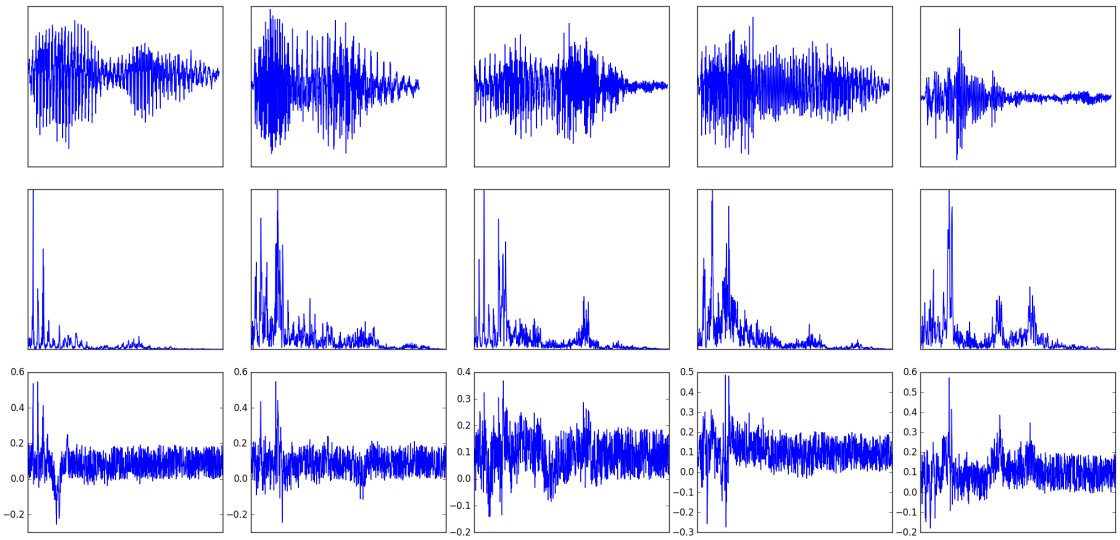


Figure 4.12: On this figure the sample time, frequency and the perceptrons' weight plots are depicted of the network that recognizes greetings in 5 languages. For human eyes it is hard to tell from either the time and frequency domain plot the corresponding language. The recorded greetings were sampled by me saying *Bonjour, Salem, Hola, Halo* and *Csá*

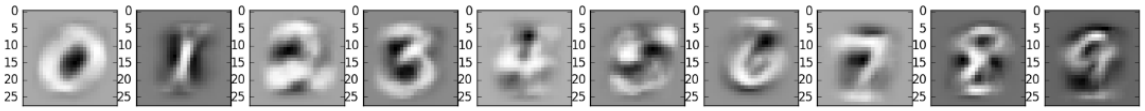


Figure 4.13: Logistic regression: single layered network (without any hidden layers) trained with L_2 regularization can be easily visualized, because the classifier neurons are strictly connected to the input pixels. Therefore by just simply reshaping their weights to make a 28×28 image will exactly show, what each neuron is looking for on the pictures.

2 Visualizing handwritten digit recognizer

First layer can be easily visualized by just simply reshaping the weights (see Figure 4.13), but extracting patterns which are recognized by further layers are not trivial. The main goal is to find an algorithm which with the input images can be enhanced to highlight the patterns that causes strongest activation in each neuron.

2.1 Training

The networks were trained on the MNIST dataset [17], with varying the following hyperparameters:

- Number of total neurons, with different distribution between layers
- Number of hidden layers
- Learning rate - constant during epochs
- Regularization - L_p

The experiments width shapes and corresponding results are shown on *Figure (4.14, 4.15, 4.16)*. In the very beginning it turned out that networks with 3 layer are the most stable for MNIST character recognition. Shallow networks, with fewer layers that had theoretically larger capacity than 3 layered ones, were unable to generalize the digits. Deeper networks, with more layers either were numerically unstable, and exhibited vanishing gradient, or overfitting occurred early in the training process. The 3 layered networks were trained with the following constraints: Stochastic gradient descent with $mini - batch = 64$, learning rate $\epsilon = 0.05$ and total number of $epochs = 25$. Despite the varying capacity, the results rather depend on the shape of the network. The best of the three networks are compared on Figure 4.17.

2.2 Candidates for enhancing

The most basic method for visualizing a perceptron is cherry-picking those pictures which results in the highest output. Formally: for a given network and set of input samples, a set of candidates can be found, which yield the highest output w.r.t. each neuron. For a given layer, the search can be done parallel, and for more general results, not only the best samples are taken into account, but the top T inputs yielding the highest activation of all samples from the given search set, see Figure 4.18.

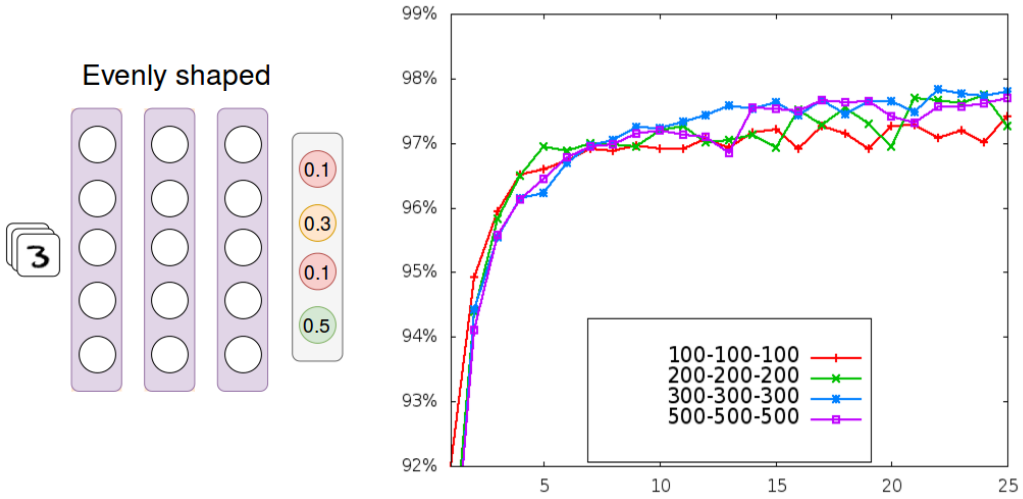


Figure 4.14: Evenly distributed networks don't the input, even the last layers tend to activate on various input patterns. During backpropagation the gradient is also more likely to disappear.

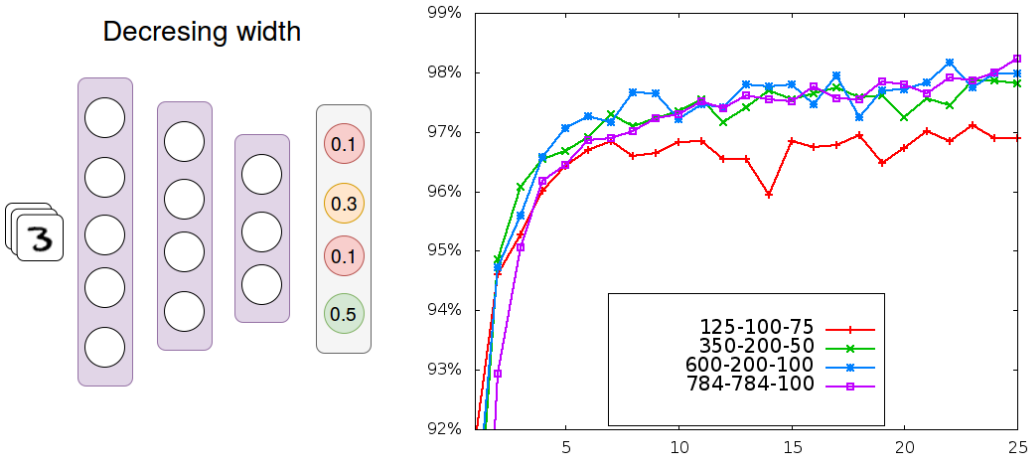


Figure 4.15: Decreasing networks tend to compress data through layer to layer, highest result was achieved with decreasing width network

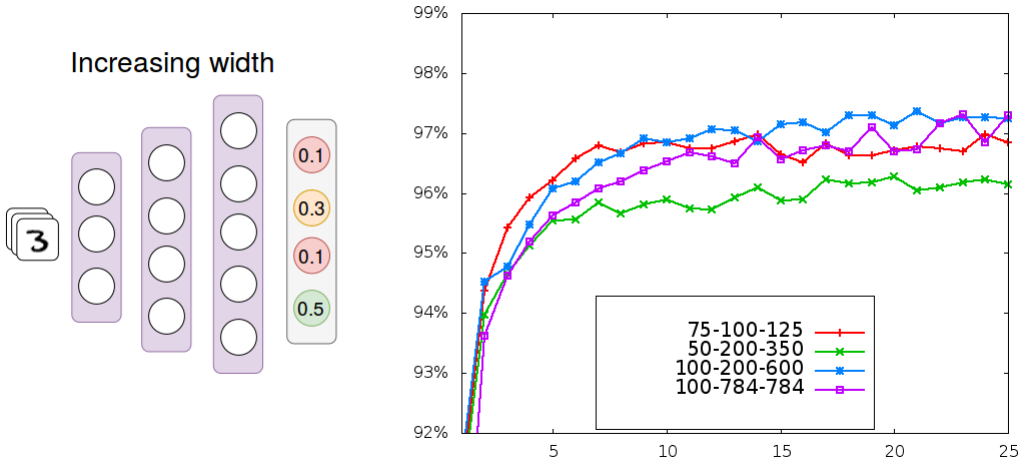


Figure 4.16: During forward propagation the data is compressed in early layers, the later neurons rely on activation patterns of the coding layer. Networks which width of layer increase in order tends learn slower, the process is inefficient, since information is lost in the beginning.

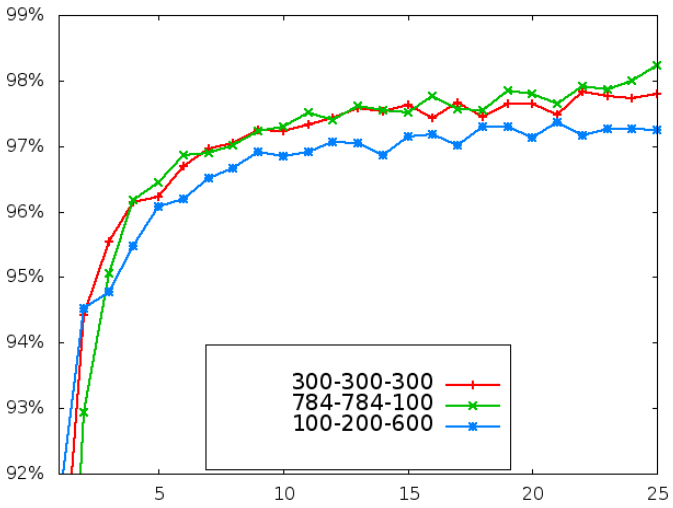


Figure 4.17: Comparison of the three type of distribution of neurons between layers

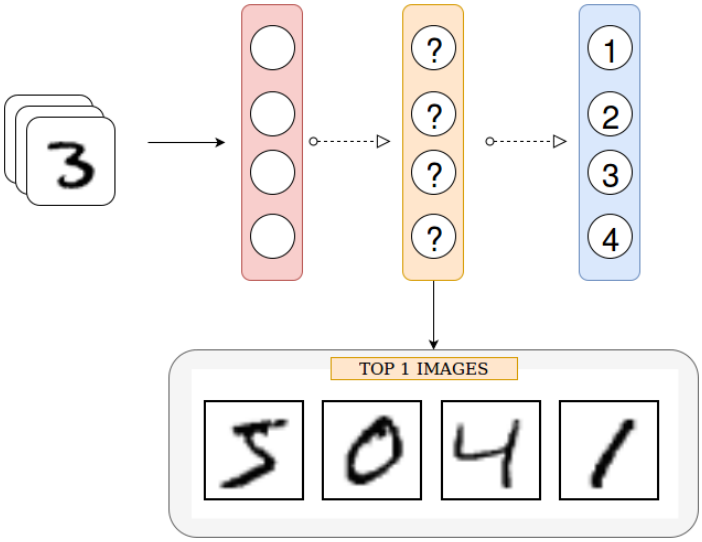


Figure 4.18: First step of visualizing the l^{th} layer: selecting each neuron's most confident choice from the image set

2.3 Enhancing - gradient ascent

Simply mean averaging the candidates would not reveal the true signals which each neurons look for, see Figure 4.19

Gradient ascent is an iterative method for enhancing the input to emphasize the patterns that are recognized by the neurons. A method which utilizes a network on top of the original to extract the recognition patterns, called DeConvnet, is described by Zeiler *et al.* [22]. The mentioned algorithm can be reduced to the following, by leaving out the sparsity encouraging regularization term - making parallel processing more efficient: for fully connected layers the method boils down to backward propagating an *arbitrary* delta vector (instead of calculating the gradient of the loss function described in [22]): $\delta_l = \mathbf{e}$ or $\delta_l = \mathbf{b}$ from the l^{th} layer toward the input layer to get gradient of the input w.r.t the i^{th} neuron's activation of the l^{th} layer, and iteratively update the original sample by the gradient multiplied with *modification rate*. The method is depicted on Figure 4.18 and 4.20, for visual guide for understanding how the applied gradient looks like, see Figure 4.21.

By choosing $\mathbf{e}_i = [0, 0 \cdots 1 \cdots 0]^T$ the enhancement would not take other activations into account, while letting $\delta_l = \mathbf{b}_i = [-1, -1 \cdots 1 \cdots -1]^T$ would result in activation where other neurons' output would decrease over amplification. Both methods were tested by the following procedure: The top 1 digit was selected for each neurons in the last layer of the best performing network, totaling in 10 samples. Skipping the candidate search of the top 10 samples were enhanced by the GA algorithm, with different number of iterations and with $\mathbf{e}$ and $\mathbf{b}$. The results are shown on Figure 4.22.

It is important to notice, that the algorithm initialized with different samples will introduce artifact features on the samples. These features will bring closer the input towards an artificial sample which will much more likely be recognized as a sample from the class of the corresponding neuron. The method of distortion can be exploited as well, which is called *generating Adversarial* samples (detailed explanation in the end of the section). However applying GA on the sample with the corresponding neuron (samples in the green box on Figure 4.22) results in much better visualization of *what each node is looking for* on the input. By qualitative analysis we concluded

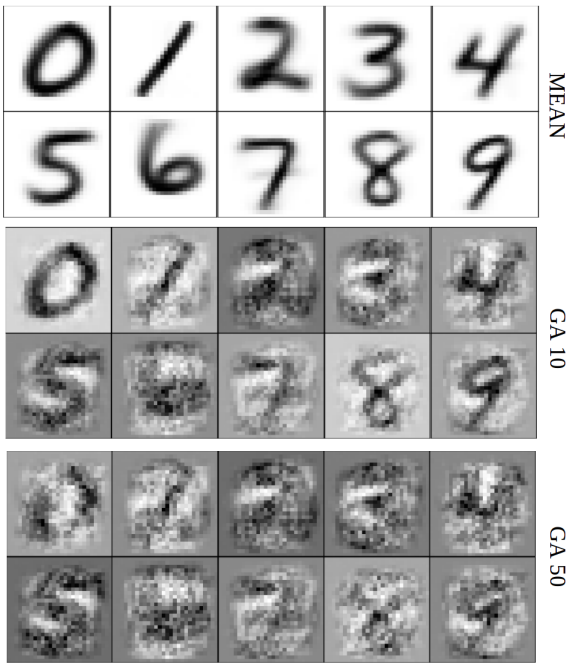


Figure 4.19: On the first set of pictures the mean of the $T = 100$ candidates are shown, they are the most preferred inputs of the candidate set for the last layer's neurons. On the set depicted below the enhanced images are shown

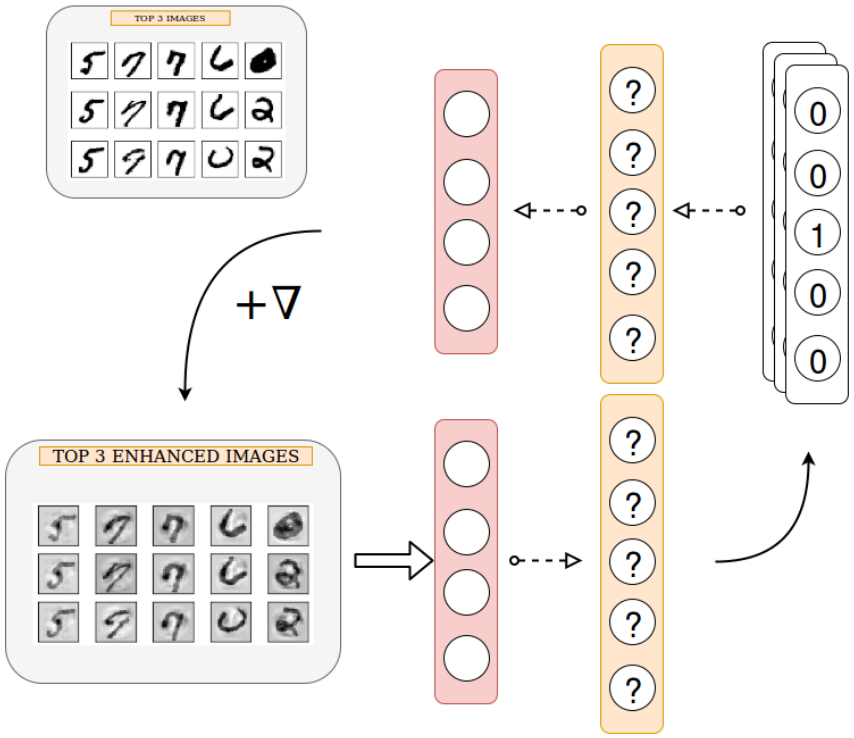
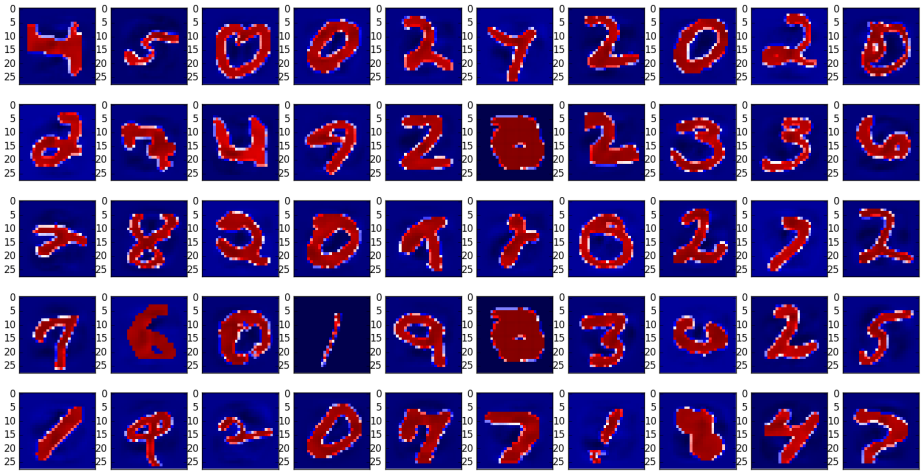
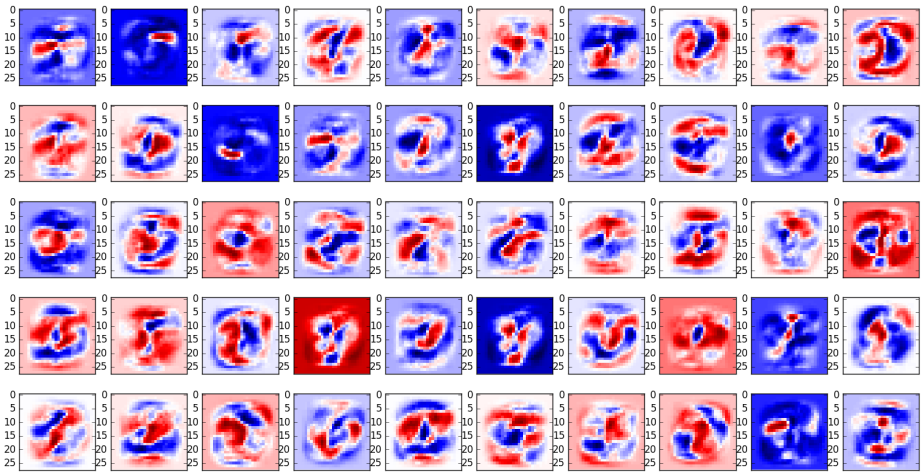


Figure 4.20: One iteration shown: a hidden layer's weights cannot be reshaped into images, but applying their activation gradient on the input images will emphasize, and amplify the patterns which yields the highest output w.r.t each neuron



(a) $x + \nabla$



(b) ∇

Figure 4.21: After one iteration the enhancement is evanescent, still the gradient holds important information of the layer. By looking at the gradient, we can tell which parts will be emphasized and which will be diminished during the process. The depicted gradients are extracted from the randomly picked neurons from the first hidden layer of a two layered network. For the smooth image result `bicubic` interpolation was used, with `seismic` color map.

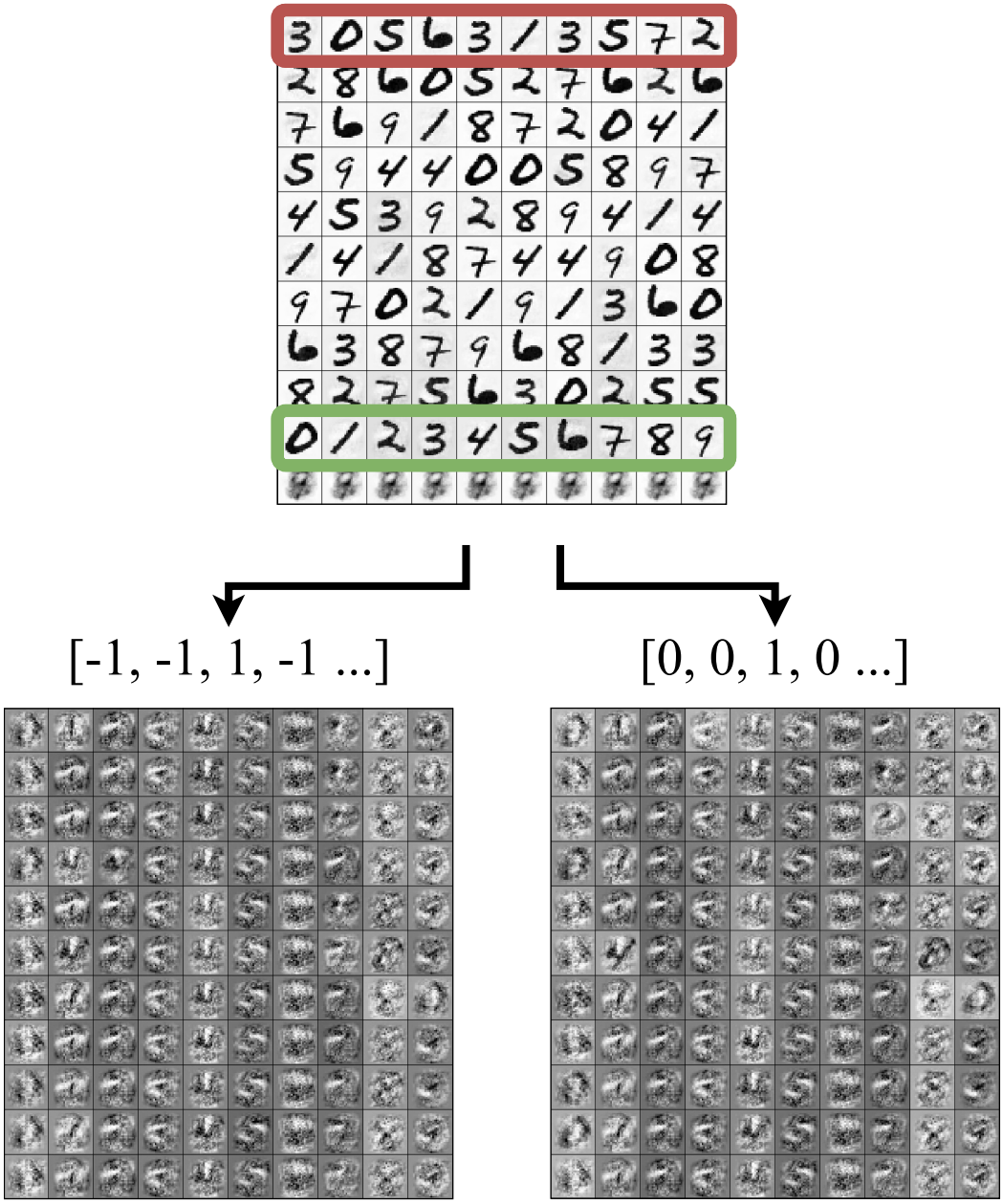


Figure 4.22: The different results of visualizing the last layer of a 3 layered network with 40 iterations of *Gradient Ascent*. Samples highlighted in green are the samples which were recognized with the highest confidence by each classifier perceptron in the last layer. These samples are organized in increasing order w.r.t. response of each neuron resulting in a 10×10 figure. *Top diagram (highlighted in red)*: are the samples of the 10 candidate sample which caused the lowest activation. *Bottom of the diagram (below the green box)*: there is a row of mean average of the samples above it, for each neuron. The arrows are pointing to diagrams, organized in a similar fashion, depicts samples enhanced by G.A. applied with the corresponding gradient on the latent layer. *Left diagram*: samples enhanced by **biased** GA 40, with $rate = 0.05$. *Right diagram*: samples enhanced by **unbiased** GA 40, with $rate = 0.05$.

that biased Gradient Ascent takes less iterations to transform the sample, however applying it should be only restricted in the classifying layer, since it is diminishing activation of neighboring neurons. A comparison of how the algorithms work iteration by iteration, on noise and misleading samples is shown on Figure 4.23

Therefore for further analysis $\mathbf{e_i} = [0, 0 \cdots 1 \cdots 0]^T$ is used.

$$x = x + \eta \cdot \nabla_l^i x$$

Applying the gradient to the input x is the first step of the iteration (see Figure 4.20). Next step is forwarding the enhanced input, applying the δ on the l^{th} layer and backwarding, and applying it on the input again makes a full cycle. The hyperparameters of the method are the *rate of amplification* η and *number of iterations* I

2.4 Processing enhanced samples

The results of amplifying the top T samples for n_l number of neurons in the l^{th} layer are stored in a $d + 2$ dimensional matrix:

$$shape(\mathbf{R}) = (T, n_l, d_1, d_2, \cdots)$$

where d is the dimension of the original input. For compact visualisation, the results are aggregated, mean averaged in the first dimension of $\mathbf{R}$ which highlights the main parts of the input, that took role in increasing the activation of the given neuron.

2.5 Results of visualization

A pre-trained network holds generalized information in the weights and biases, and the approach mentioned in *subsection 2.2*, produce great amount of possible representations of the networks' inner state. In this subsection MNIST samples enhanced by parameters derived from the *MLP*s trained in *subsection 2.1* are analyzed and compared - *absolutely* non exhaustively, following intuitions like:

- Where, and how does abstraction occur, from combining previous layers' activation patterns
- How does differently distributed nodes affect the above.
- Do fully connected networks converge to recognizing *localized* patterns, like convolutional networks do

2.6 Challenges

Because the parameter space is too wide, the main obstacle in visualization is to find out what to look for. For example visualizing the current best network that has the following dimensions:

- for each node in the network there is a $T \times 28 \times 28$ sized input set
- for each layer there is a given set of nodes $N = [784, 784, 100, 10]$ respectively

Totaling in $T \times (N_l)$ picture per layer, which are hard to deal with, *(see Figure 4.25)*. Especially when the neuron's favored patterns can be matched to different digits, the T should be increased in

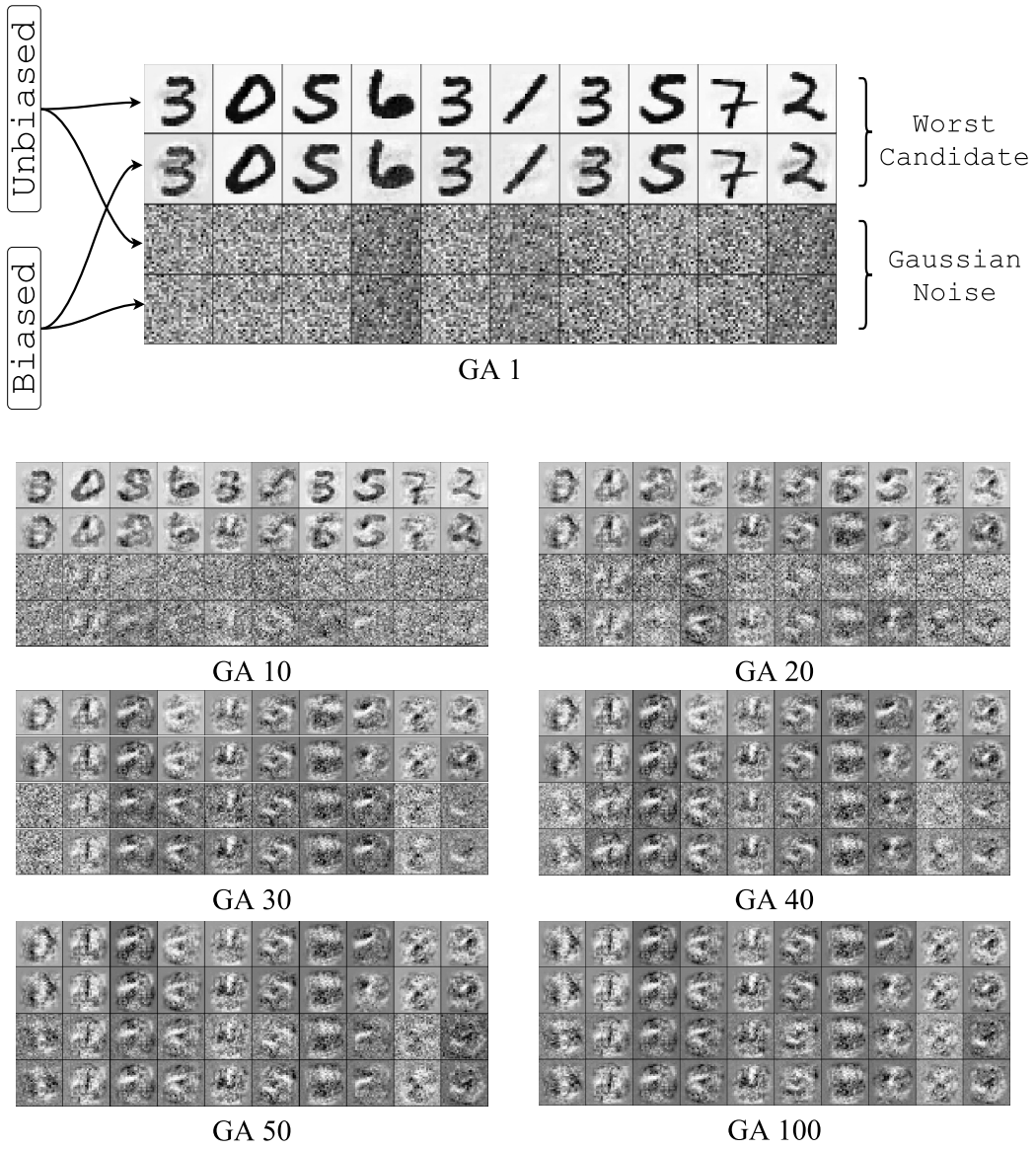


Figure 4.23: Generating *adversarial* samples with biased and unbiased Gradient Ascent. In the first two rows, the images were enhanced by the *'wrong'* gradient, making them more likely to be recognized by the corresponding classifier neuron $0, 1...9$.

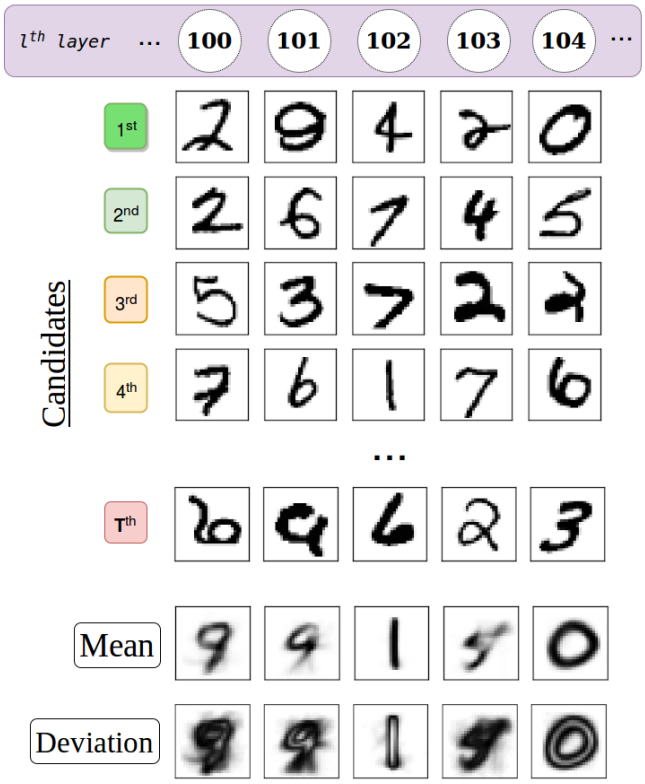


Figure 4.24: The naive way to extract interpretable patterns is to gather those inputs which excites the most the given layer, and **mean average** it. The case depicted on the figure, is that the ideal input is not trivial as the listed candidates shows, especially in the first layers. By aggregating

order to get a more general picture when aggregating the amplified pictures. On the other hand, the gradient ascent optimization cannot terminate on *convex functions*, which is actually true for the activation of neural node's activation w.r.t to its input - meaning that the original image can be amplified as many times as possible, the output of the corresponding neuron y_l^i will always increase with the number of iterations I. An example for applying the same method, but with 5 times the original number of iterations depicted on *Figure 4.25* is shown on *Figure 4.26*. Therefore finding the best I is also essential for extracting important information from the network.

***Note**: Deviating, or averaging the input samples is an important step to strengthen the features which the given node truly recognize, and drop out those which it does not (see Figure 4.24). For the rest of the case studies mean averaging is used for aggregation*

Remainder: For the following examples, a small subset of samples are shown, which were selected for the most revealing clues. Still, they may not represent all features of the network.

2.7 Analyzing the best performing network

The best result belongs to a network with hidden layer width: 784-784-100. The layers were visualized with the following parameters: $T = 100$ $\eta = 0.1$ $I = 10, 50$ and the training set was used for choosing candidates.

The suggestion while looking for evidences is that the reason behind the good performance of

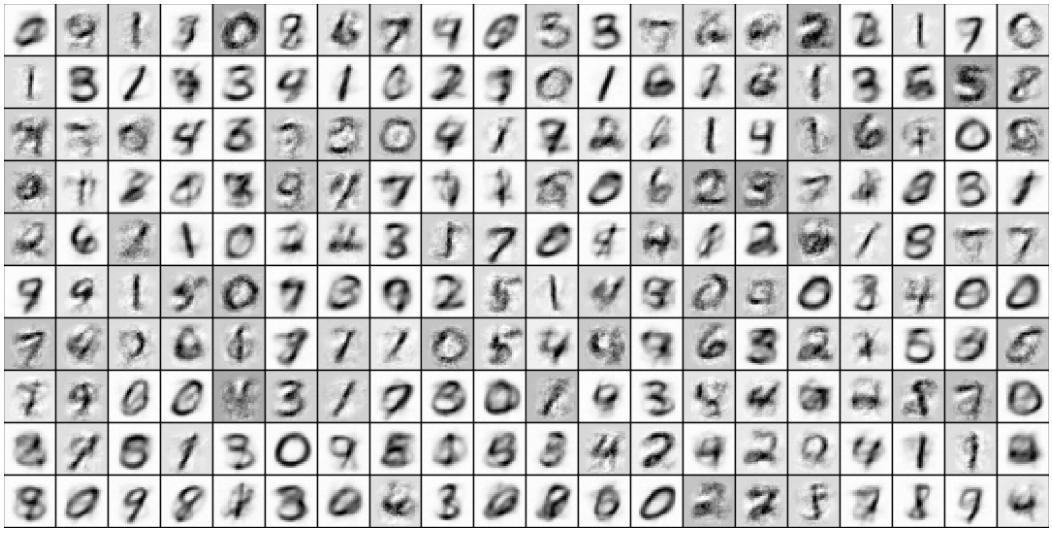


Figure 4.25: Iteratively enhanced pictures of the first hidden layer of a **[300-300-300]** network (first 200 are shown in *row-major order*). The main question is, which of them holds important features, or needs further analysis.

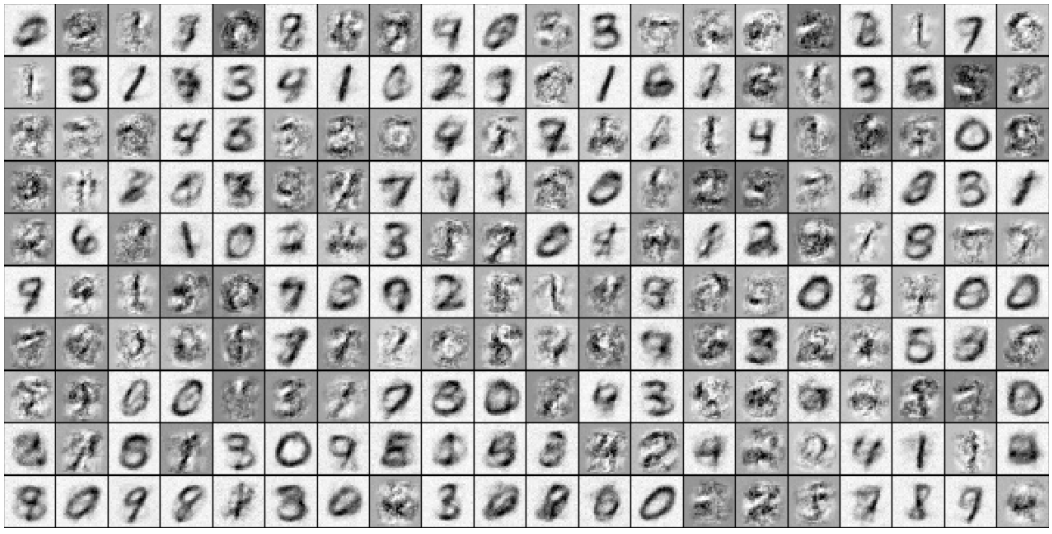


Figure 4.26: After increasing T, compared to fig. 4.25 some of the amplified pictures gets noisy, which is not a problem in the first case. It is a trend in the early layers, to only focus on small, localized type of features, that yields different candidates, which makes the mean of the outcome noisy overall. Especially this is one important feature of *MLPs* that initialized with random weights they converge towards recognizing localized features.

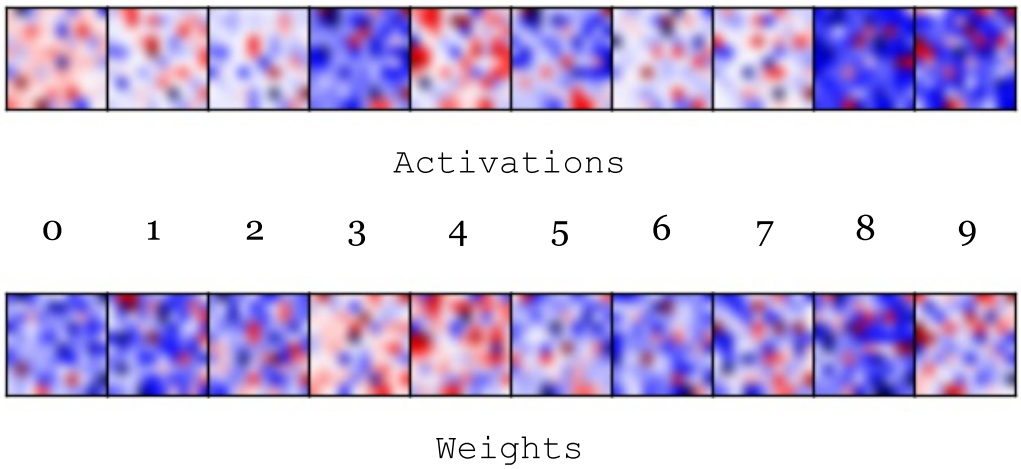


Figure 4.27: *On top*: Every sample translates to some response signal (like the depicted maps) before reaching the final classifier layer. Though we cannot understand it, for the network the activation patterns holds all information about the corresponding input. *On bottom*: Each map is processed by the weights of the last layer, which is also essential for the network, but does not exhibit any intuitive feature. Think about the analogy: we cannot simply define what happens inside by just looking at brains of creatures. Even if we had a high resolution microscope, revealing the fine structure of the nervous tissue would not suddenly illuminate everything.

the network is relying on different levels of abstraction. I visualized all three layers in order to get a better grasp what happens inside of a network, which can recognize digits with 98% success. For conviction, that we cannot simply read the values out from neither the activation, or the weights from the hidden layers see Figure 4.27.

5. Conclusion

6. Summary

7. Future

List of Figures

Bibliography

[1] Martin Abadi et al. *TensorFlow: A system for large-scale machine learning*. Tech. rep. arXiv preprint. Google Brain, 2016. URL: https://arxiv.org/abs/1605.08695.

[2] Yoshua Bengio. "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1 (2009), pp. 1–127.

[3] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. "Torch7: A matlab-like environment for machine learning". In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376. 2011.

[4] "Deep Learning moving beyond shallow machine learning since 2006". Accessed: November 14, 2016. 2016. URL: http://deeplearning.net/tutorial/.

[5] Nando de Freitas. "Machine Learning". Accessed: November 14, 2016. 2015. URL: https://www.cs.ox.ac.uk/people/nando.defreitas/machinelearning/.

[6] Andrew Gibiansky. "Neural networking code snippets and sources". Accessed: November 14, 2016. 2012. URL: https://github.com/gibiansky/experiments/tree/master/neural-network.

[7] Andrew Gibiansky. "Math to Code". Accessed: November 14, 2016. 2016. URL: http://andrew.gibiansky.com/.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Deep Learning". Book in preparation for MIT Press. 2016. URL: http://www.deeplearningbook.org.

[9] Simon Haykin and Neural Network. "A comprehensive foundation". In: *Neural Networks* 2.2004 (2004).

[10] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[11] Yangqing Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[12] Andrej Karpathy. "Andrej Karpathy blog Hacker's guide to Neural Networks". Accessed: November 14, 2016. URL: http://karpathy.github.io/neuralnets/.

[13] Andrej Karpathy. "ConvNetJS Deep Learning in your browser". Accessed: November 14, 2016. URL: http://cs.stanford.edu/people/karpathy/convnetjs/.

[14] Andrej Karpathy. "Connecting Images and Natural Language". PhD thesis. Stanford University, 2016.

[15] Andrej Karpathy, Fei-Fei Li, and Justin Johnson. "CS231n Convolutional Neural Networks for Visual Recognition." Accessed: November 14, 2016. 2016. URL: http://cs231n.github.io/.

[16] Hugo Larochelle. "neural network tutorial". Accessed: November 14, 2016. URL: https://youtu.be/SGZ6BttHMPw.

[17] Yann LeCun, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

[18] Michael Nielsen. "Neural Networks and Deep Learning". Accessed: November 14, 2016. 2016. URL: http://neuralnetworksanddeeplearning.com/.

[19] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of Machine Learning Research* 12.Oct (2011), pp. 2825–2830.

[20] Paul John Werbos. *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*. Vol. 1. John Wiley & Sons, 1994.

[21] Jason Yosinski et al. "Understanding neural networks through deep visualization". In: *arXiv preprint arXiv:1506.06579* (2015).

[22] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European Conference on Computer Vision*. Springer. 2014, pp. 818–833.