Artificial Neural Networks

Jan Feldmann

January 4, 2021

1 Motivation

In this talk, our goal is to improve on linear models for classification and regression by adapting to more complex data, say, data which is not linearly separable.

This is done by learning the non-linear basis function. We will be concerned with Neural Networks which are built from multiple perameterised, non-linear basis functions that are adapted to the data by gradient descend.

2 Perceptrons

A perceptron is a function $\mathbb{R}^n \to \{-1, 1\}$ of the form

$$x \longmapsto \operatorname{sgn}(w \cdot x + b),$$

where $w \in \mathbb{R}^n$, $b \in \mathbb{R}$ and sgn denotes the step function

$$\operatorname{sgn}(\xi) = \begin{cases} 1 & \xi \geq 0, \\ -1 & \xi < 0. \end{cases}$$

The components of the vector w are called the *weights* of the perceptron, the quantity b is called the *bias* or *threshold*.

2.1 Linear Separability

A perceptron $f : \mathbb{R}^n \to \{-1, 1\}$ labels a given input according to its *decision boundary*, which is given by the affine hyperplane defined by the equation $w \cdot x + \theta = 0$. Thus, the weight vector w determines the orientation of the decision boundary and $\theta / \|w\|$ determines its distance from the origin.

The perceptron will then label input on the "positive" side of the decision boundary with 1 and input on the "negative" side with -1. Only functions $\mathbb{R}^n \to \{-1, 1\}$ under which the preimages of 1 and -1 are separated by a hyperplane can therefore by accurately modelled by a perceptron. We call such a function linearly separable.

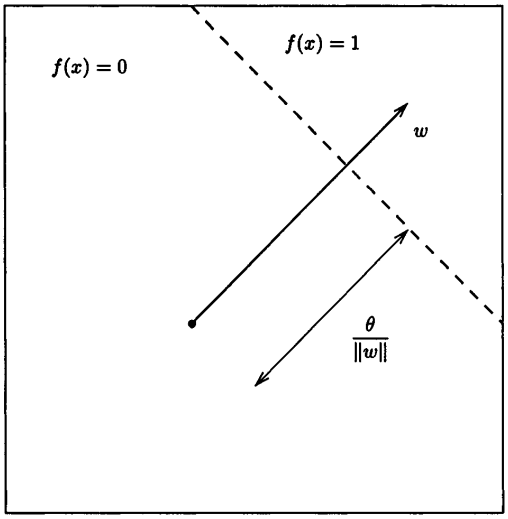


Figure 1: Decision boundary of a perceptron $f : \mathbb{R}^2 \to \{-1, 1\}$ with bias θ.

2.2 The Perceptron Algorithm

Given any linearly separable classification $X \times Y \subseteq \mathbb{R}^n \times \{-1, 1\}$ of a finite set $X \subseteq \mathbb{R}^n$, any perceptron f can be trained to fit that classification.

We choose a positive constant η and iterate through the data $(x, y) \in X \times Y$. If x is misclassified by f, we replace the weight vector w by $w + \eta(y - f(x))x$ and the threshold θ by $\theta + \eta(f(x) - y)$.

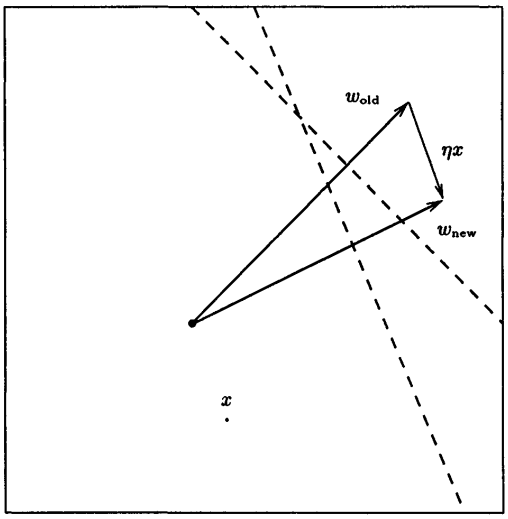


Figure 2: The point x is misclassified, weight and threshold are modified.

After a finite number of iterations, the perceptron f will correctly classify X. A rigorous proof of that claim can be found in [2], chapter 24.

2.3 Some History

- The idea of a perceptron is originally conceived by Frank Rosenblatt in the late 1950s, cf. [1].
- In 1958, a first mechanical machine implementing a perceptron is presented to the public. Endowed with 400 light sensors, it was trained with the algorithm of the previous section to recognize written characters.

- In 1968, Marvin Minsky and Seymour Papen publish with [3] a mathematical account of perceptrons and show, that perceptrons can solve only linearly separable classifications and give with the *exclusive or* an elementary example of a function, which can not be modelled by a perceptron.
- As a result, interest in neural networks and funding of their research decreases significantly.

3 Forward Feeding Neural Networks

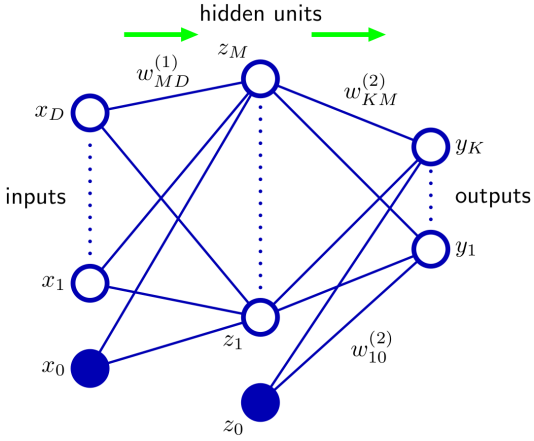
In some sense, neural networks can be thought of as a collection of perceptrons stacked in layers. Consider functions $\mathbb{R}^n \times \mathbb{R}^m$ of the form

$$y(x, w) = f\Big(\sum_{j=1}^{m} w_j \phi_j(x)\Big)$$

with some non-linear *activation function* f and *basis functions* (ϕ_j).

Such a function is then called a *neural network*, if each base function ϕ_j is again of the above form or the identity.

The base functions are often differentiable, gradient methods are thus applicable to learn neural networks, which constitutes one of the major differences to perceptrons. It is helpful to introduce a graphical depiction of neural networks by a directed acyclic graph, an example is displayed on the right. Here, we speak of a neural network with a single hidden layer.



This suggest calculating the output (y_k) by means of *forward feeding* the input vector x. First, the *activation* (a_k) of the *hidden units* or *neurons* is calculated:

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}.$$

Then, a differentiable non-linear *activation function* h is applied, to obtain the output

$$z_j = h(a_j)$$

of the basis functions (ϕ_j).

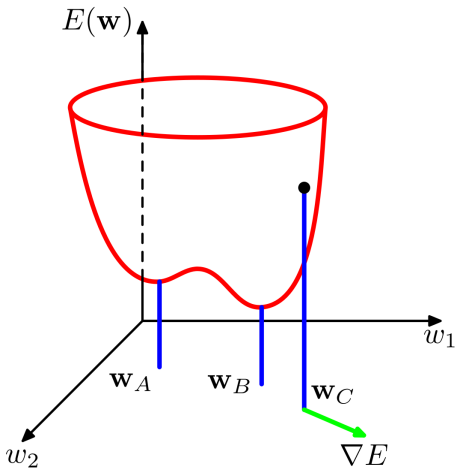
This process is then similarly repeated with the quantities (z_j) to calculate the output of the network. Thus, our example network possesses the form

$$y(x, w) = f\Big(\sum_{j=1}^{M} w_{kj}^{(2)} h\Big(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\Big) + w_{k0}^{(2)}\Big).$$

3.1 Network Training

Given a neural network with an output function $y(x, w)$ and some training data comprising input vectors $(x_n)_{1\leq n\leq N}$ and corresponding target vectors $(t_n)_{1\leq n\leq N}$, we want to find weight matrices which minimize an error function $E(w)$. For example, consider the squared error

$$E(w) = \frac{1}{2} \sum_{n=1}^{N} \|y(x_n, w) - t_n\|^2 .$$



Minimizing $E(w)$ is done algorithmically. Since $y(x, w)$ is differentiable - unlike a perceptron - we can find a local minimum of $E(w)$ by *steepest descend*, or *gradient descend*.

The error function $E(w)$ will in general not be convex (unlike in the case of *support vector machines*). Therefore, multiple local minima can exist, as it is depicted in the figure on the right. There is no sensible way to determine whether a local minimum is a global minimum, but for practical purposes, it suffices to choose an adequate weight among a few numerically computed local minima.

Any local minimum w_{min} satisfies the equation $\nabla E(w_{min}) = 0$ while for any w, which is not a local extremum, $\nabla E(w)$ points in the direction of steepest ascend of the error function.

Choosing a learning rate $\eta > 0$ and some initial weight $w^{(0)}$, we can successively move through the space of weights by setting

$$w^{(r+1)} = w^{(r)} - \eta \nabla E(w^{(r)})$$

and eventually find an approximation of a local minimum.

3.2 Error Backpropagation

The training technique described in the section above requires computing the gradient of the error function. This can be done by *error backpropagation*. The chain rule provides us the equation

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}},$$

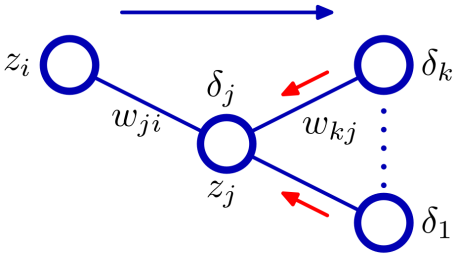
where a_j is the activation of the neuron j. We now note

$$\frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \Big(\sum_k w_{jk} z_k \Big) = z_i$$

and introduce the notation $\delta_j = \frac{\partial E_n}{\partial a_j}$. The (δ_j) are called the *errors*. We thus obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i.$$

The errors (δ_j) can be calculated successively following a scheme suggested by the figure on the right. While the output of a neural network is calculated feeding the input through the neurons in a forward fashion, the errors are calculated starting with the output layer and then moving backwards.



Assume that the errors $\delta_1, \ldots, \delta_k$ as in the figure have already been calculated. Then, using again the chain rule, we calculate

$$\delta_j = \sum_{l=1}^{k} \frac{\partial E_n}{\partial a_l} \frac{\partial a_l}{\partial a_j} = h'(a_j) \sum_{l=1}^{k} w_{lj} \delta_l,$$

where h is the activation function appearing in the calculation of the activations

$$a_l = \sum_i w_{li} z_i = \sum_i w_{li} h(a_i).$$

The errors corresponding to the output units can be directly computed to start the backpropagation.

4 Other Kinds of Neural Nets

Different architectures of neural networks, which are specialized to certain applications and part from the feed forward structure discussed in the previous sections, are now abundant. We shall briefly discuss recurrent neural networks and convolutional neural networks. Many more architectures are discussed in great detail in [5], which we recommend for further reading.

4.1 Recurrent Neural Networks

Suppose we want to train a neural network to classify film reviews (*sentiment analysis*) as favourable or unfavourable, or to predict the next word given a part of a sentence. In each case, we are given a text and it seems reasonable to break it apart into single words to feed it to a neural network.

Forward feeding neural networks as presented in the previous section are not well suited for that approach: The number of input nodes is fixed while the length of the texts will vary and the sequential information of the sentence structure is lost by reducing the text to a set of single words.

This issue is remedied by *recurrent neural networks*. The number of layers of such a network varies with the length of the input by stacking copies of a fixed layer (hence the designation *recurrent*). Each instance of that fixed layer takes a single input and propagates its computation forward to the next layer. In the graphical depiction below, we find a recurrent neural network being trained to predict the next word from its input by being fed the sentence *"The cat chased the mouse."*

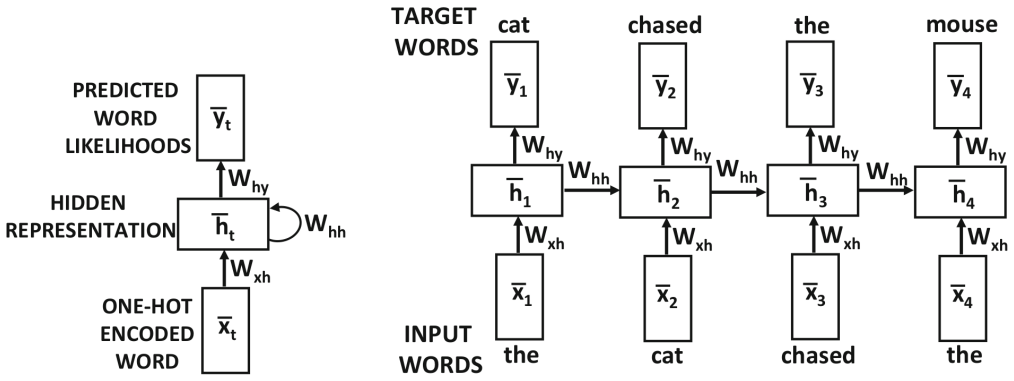


Figure 3: A recurrent network with a time-layered representation on the right.

The parameters of the repeated layer are retained throughout the process. The number of weights of the network is thus fixed as well and the computation at each sequence point is modelled the same way.

Every time some sequential aspect of a data set is to be emphasised, recurrent neural networks can be used. For example, recurrent neural networks find as well application in the financial sector to analyse stock markets or to monitor credit card transactions for fraudulent behaviour.

4.2 Convolutional neural network

Convolutional neural networks were originally inspired by the study of the visual cortex of cats and now find application in the field of computer vision for image classification or object detection.

The layers of a convolutional network are three dimensional. For example, the input layer could organise its units among axes representing hight and width of an image and the RGB colour channels. Among the layers, we again discern *convolutional* and *sampling* layers. The sampling layers are processed by sampling operations, simply averaging over small local regions of units to obtain a more compressed successive layer.

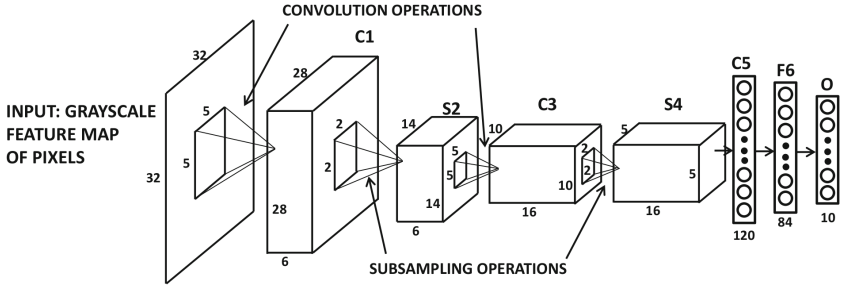


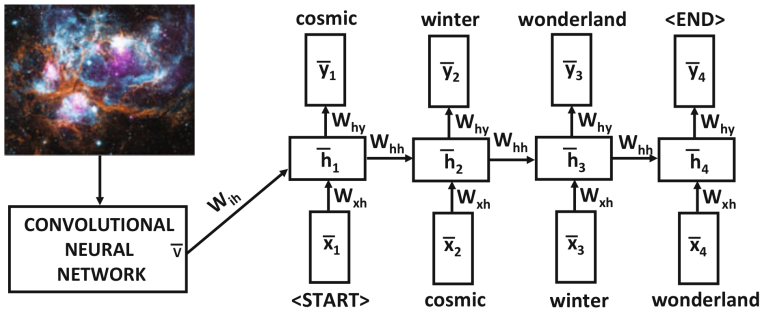
Figure 4: LeNet-5: An early convolutional neural network.

The convolution operation convolves a small *filter* or *kernel* with the preceding convolutional layer. Learning this filter is significantly less work than learning weights between all pairs of neurons between two layers.

The hidden layers of the network are thought to obtain semantic information from the input data. The first hidden layers might recognize lines or other primitive shapes in the input while the successive layers will capture increasingly complex shapes. It is also possible to cut a trained convolutional neural network off at the penultimate layer to obtain abstract representations of images for further processing.

4.3 Combining different architectures

It is also possible to combine different types of neural networks. We want to give an example of combining a convolutional neural network with a recurrent neural network.



We are looking for a neural network which assigns a caption to an image which should be trained with sets of pairs of images with captions. In this case, a convolutional neural network can first be used to obtain some abstract representation of an image which is then fed into the first input unit of a recurrent neural network, followed by the caption. A schematic depiction is found on the previous page.

Upon completion of the training, the network can predict a caption for an image by feeding the output of each recurrent layer as new input to the following one.

References

[1] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6), 386.

[2] Anthony, M., & Bartlett, P. L. (2009). Neural network learning: Theoretical foundations. Cambridge university press.

[3] Minsky, M., & Papert, S. A. (1988). Perceptrons: An introduction to computational geometry. MIT press.

[4] Bishop, C. M. (2006). Pattern recognition and machine learning. Springer.

[5] Aggarwal, C. C. (2018). Neural networks and deep learning. Springer.