# Data Analysis and Machine Learning: Elements of machine learning

Morten Hjorth-Jensen[1,2]

Department of Physics, University of Oslo[1]

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University[2]

Sep 28, 2018



## Neural networks

Artificial neural networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable.

## Artificial neurons

The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general. A model of artificial neurons was first developed by McCulloch and Pitts in 1943 to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output.

This behaviour has inspired a simple mathematical model for an artificial neuron.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{1}$$

Here, the output $y$ of the neuron is the value of its activation

## Neural network types

An artificial neural network (NN), is a computational model that consists of layers of connected neurons, or *nodes*. It is supposed to mimic a biological nervous system by letting each neuron interact with other neurons by sending signals in the form of mathematical functions between layers. A wide variety of different NNs have been developed, but most of them consist of an input layer, an output layer and eventual layers in-between, called *hidden layers*. All layers can contain an arbitrary number of nodes, and each connection between two nodes is associated with a weight variable.

Neural networks (also called neural nets) are neural-inspired nonlinear models for supervised learning. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods.

## Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of NN devised. In this network, the information moves in only one direction: forward through the layers.

Nodes are represented by circles, while the arrows display the connections between the nodes, including the direction of information flow. Additionally, each arrow corresponds to a weight variable, not displayed here. We observe that each node in a layer is connected to *all* nodes in the subsequent layer, making this a so-called *fully-connected* FFNN.

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation.

## Recurrent neural networks

So far we have only mentioned NNs where information flows in one direction: forward. *Recurrent neural networks* on the other hand, have connections between nodes that form directed *cycles*. This creates a form of internal memory which are able to capture information on what has been calculated before; the output is dependent on the previous computations. Recurrent NNs make use of sequential information by performing the same task for every element in a sequence, where each element depends on previous elements. An example of such information is sentences, making recurrent NNs especially well-suited for handwriting and speech recognition.

## Other types of networks

There are many other kinds of NNs that have been developed. One type that is specifically designed for interpolation in multidimensional space is the radial basis function (RBF) network. RBFs are typically made up of three layers: an input layer, a hidden layer with non-linear radial symmetric activation functions and a linear output layer ("linear" here means that each node in the output layer has a linear activation function). The layers are normally fully-connected and there are no cycles, thus RBFs can be viewed as a type of fully-connected FFNN. They are however usually treated as a separate type of NN due the unusual activation functions.

Other types of NNs could also be mentioned, but are outside the scope of this work. We will now move on to a detailed description of how a fully-connected FFNN works, and how it can be used to interpolate data sets.

## Multilayer perceptrons

One use often so-called fully-connected feed-forward neural networks with three or more layers (an input layer, one or more hidden layers and an output layer) consisting of neurons that have non-linear activation functions.

Such networks are often called *multilayer perceptrons* (MLPs)

## Why multilayer perceptrons?

According to the *Universal approximation theorem*, a feed-forward neural network with just a single hidden layer containing a finite number of neurons can approximate a continuous multidimensional function to arbitrary accuracy, assuming the activation function for the hidden layer is a **non-constant, bounded and monotonically-increasing continuous function**. Note that the requirements on the activation function only applies to the hidden layer, the output nodes are always assumed to be linear, so as to not restrict the range of output values.

We note that this theorem is only applicable to a NN with *one* hidden layer. Therefore, we can easily construct an NN that employs activation functions which do not satisfy the above requirements, as long as we have at least one layer with activation functions that *do*. Furthermore, although the universal approximation theorem lays the theoretical foundation for regression with neural networks, it does not say anything about how things work in practice: A neural network can still be able to approximate a given function reasonably well without having the

## Mathematical model

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(u) \tag{2}$$

In an FFNN of such neurons, the *inputs* $x_i$ are the *outputs* of the neurons in the preceding layer. Furthermore, an MLP is fully-connected, which means that each neuron receives a weighted sum of the outputs of *all* neurons in the previous layer.

## Mathematical model

First, for each node $i$ in the first hidden layer, we calculate a weighted sum $u_i^1$ of the input coordinates $x_j$,

$$u_i^1 = \sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1 \tag{3}$$

This value is the argument to the activation function $f_1$ of each neuron $i$, producing the output $y_i^1$ of all neurons in layer 1,

$$y_i^1 = f_1(u_i^1) = f_1\left(\sum_{j=1}^{2} w_{ij}^1 x_j + b_i^1\right) \tag{4}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation $f_l$

$$y_i^l = f_l(u_i^l) = f_l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{5}$$

## Mathematical model

The output of neuron $i$ in layer 2 is thus,

$$y_i^2 = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right) \tag{6}$$

$$= f_2\left[\sum_{j=1}^{3} w_{ij}^2 f_1\left(\sum_{k=1}^{2} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{7}$$

where we have substituted $y_m^1$ with. Finally, the NN output yields,

$$y_1^3 = f_3\left(\sum_{j=1}^{3} w_{1m}^3 y_j^2 + b_1^3\right) \tag{8}$$

$$= f_3\left[\sum_{j=1}^{3} w_{1j}^3 f_2\left(\sum_{k=1}^{3} w_{jk}^2 f_1\left(\sum_{m=1}^{2} w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{9}$$

## Mathematical model

We can generalize this expression to an MLP with $l$ hidden layers. The complete functional form is,

$$y_1^{l+1} = f_{l+1}\left[\sum_{j=1}^{N_l} w_{1j}^3 f_l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^2 f_{l-1}\left(\ldots f_1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right] \tag{10}$$

which illustrates a basic property of MLPs: The only independent variables are the input values $x_n$.

## Mathematical model

This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\vec{x} \in \mathbb{R}^n \to \vec{y} \in \mathbb{R}^m$. In our example, $n = 2$ and $m = 1$. Consequentially, the number of input and output values of the function we want to fit must be equal to the number of inputs and outputs of our MLP.

Furthermore, the flexibility and universality of a MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$h(x) = c_1 f(c_2 x + c_3) + c_4 \tag{11}$$

where the parameters $c_i$ are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

**Matrix-vector notation.** We can introduce a more convenient notation for the activations in a NN.

Additionally, we can represent the biases and activations as layer-wise column vectors $\vec{b}_l$ and $\vec{y}_l$, so that the $i$-th element of each vector is the bias $b_i^l$ and activation $y_i^l$ of node $i$ in layer $l$ respectively.

We have that $\mathrm{W}_l$ is a $N_{l-1} \times N_l$ matrix, while $\vec{b}_l$ and $\vec{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum in becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 in

$$\vec{y}_2 = f_2(\mathrm{W}_2\vec{y}_1 + \vec{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{12}$$

**Matrix-vector notation and activation.** The activation of node $i$ in layer 2 is

$$y_i^2 = f_2\left(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\right) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{13}$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\mathrm{W}_l\vec{y}_{l-1}$ we move forward one layer.

**Activation functions.** A property that characterizes a neural network, other than its connectivity, is the choice of activation function(s). As described in, the following restrictions are imposed on an activation function for a FFNN to fulfill the universal approximation theorem

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

**Activation functions, Logistic and Hyperbolic ones.** The second requirement excludes all linear functions. Furthermore, in a MLP with only linear activation functions, each layer simply performs a linear transformation of its inputs.

Regardless of the number of layers, the output of the NN will be nothing but a linear function of the inputs. Thus we need to introduce some kind of non-linearity to the NN to be able to fit non-linear functions Typical examples are the logistic *Sigmoid*

$$f(x) = \frac{1}{1 + e^{-x}}, \tag{14}$$

and the *hyperbolic tangent* function

$$f(x) = \tanh(x) \tag{15}$$

**Relevance.** The *sigmoid* function are more biologically plausible because the output of inactive neurons are zero. Such activation function are called *one-sided*. However, it has been shown that the hyperbolic tangent performs better than the sigmoid for training MLPs. has become the most popular for *deep neural networks*

```
"""The sigmoid function (or the logistic curve) is a
function that takes any real number, z, and outputs a number (0,1).
It is useful in neural networks for assigning weights on a relative sc
The value z is the weighted sum of parameters involved in the learning

import numpy
import matplotlib.pyplot as plt
import math as mt

z = numpy.arange(-5, 5, .1)
sigma_fn = numpy.vectorize(lambda z: 1/(1+numpy.exp(-z)))
sigma = sigma_fn(z)

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(z, sigma)
ax.set_ylim([-0.1, 1.1])
ax.set_xlim([-5,5])
ax.grid(True)
ax.set_xlabel('z')
ax.set_title('sigmoid function')
```

## Setting up a Multi-layer perceptron model

```
from scipy import optimize

class Neural_Network(object):
    def __init__(self, Lambda=0):
        #Define Hyperparameters
        self.inputLayerSize = 2
        self.outputLayerSize = 1
        self.hiddenLayerSize = 3

        #Weights (parameters)
        self.W1 = np.random.randn(self.inputLayerSize,self.hiddenLayer
        self.W2 = np.random.randn(self.hiddenLayerSize,self.outputLaye

        #Regularization Parameter:
        self.Lambda = Lambda

    def forward(self, X):
        #Propogate inputs though network
        self.z2 = np.dot(X, self.W1)
        self.a2 = self.sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W2)
        yHat = self.sigmoid(self.z3)
        return yHat

    def sigmoid(self, z):
        #Apply sigmoid activation function to scalar, vector, or matri
        return 1/(1+np.exp(-z))
```

## Two-layer Neural Network

```
import numpy as np

#sigmoid
def nonlin(x, deriv=False):
    if (deriv==True):
        return x*(1-x)
    return 1/(1+np.exp(-x))

#input data
x=np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])

#output data
y=np.array([0,1,1,0]).T

#seed random numbers to make calculation
np.random.seed(1)

#initialize weights with mean=0
syn0=2*np.random.random((3,4))-1

for iter in range(10000):
    #forward propogation
    l0=x
    l1=nonlin(np.dot(l0,syn0))
    l1_error=y-l1
    #multiply error by slope of sigmoid at values of l1
    l1_delta=l1_error*nonlin(l1,True)
    #update weights
```