

Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning

Morten Hjorth-Jensen^{1,2}

Department of Physics, University of Oslo¹

Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University²

Oct 2, 2018

© 1999-2018, Morten Hjorth-Jensen. Released under CC Attribution-NonCommercial 4.0 license

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.

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

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

Neural network types

An artificial neural network (ANN), is a computational model that consists of layers of connected neurons, or nodes or units. We will refer to these interchangeably as units or nodes, and sometimes as neurons.

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 ANNs 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 we discussed earlier.

Feed-forward neural networks

The feed-forward neural network (FFNN) was the first and simplest type of ANNs that were 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 (figure to come). 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.

Convolutional Neural Network

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. (figure to come)

Convolutional neural networks emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

Recurrent neural networks

So far we have only mentioned ANNs 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 ANNs 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 to the unusual activation functions.

Multilayer perceptrons

One uses 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.

Mathematical model

The output y is produced via the activation function f

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

This function receives x_i as inputs. Here the activation $z = \sum_{i=1}^n w_i x_i$. 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 z_i^1 of the input coordinates x_j ,

$$z_i^1 = \sum_{j=1}^M w_{ij}^1 x_j + b_i^1 \quad (2)$$

Here b_i is the so-called bias which is normally needed in case of zero activation weights or inputs. How to fix the biases and the weights will be discussed below. The value of z_i^1 is the argument to the activation function f_i of each node i . The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^M w_{ij}^1 x_j + b_i^1\right) \quad (3)$$

where we assume that all nodes in the same layer have identical

Mathematical model

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2 \left(\sum_{j=1}^N w_{ij}^2 y_j^1 + b_i^2 \right) \quad (5)$$

$$= f^2 \left[\sum_{j=1}^N w_{ij}^2 f^1 \left(\sum_{k=1}^M w_{jk}^1 x_k + b_j^1 \right) + b_i^2 \right] \quad (6)$$

where we have substituted y_k^1 with the inputs x_k . Finally, the ANN output reads

$$y_i^3 = f^3 \left(\sum_{j=1}^N w_{ij}^3 y_j^2 + b_i^3 \right) \quad (7)$$

$$= f_3 \left[\sum_j w_{ij}^3 f^2 \left(\sum_k w_{jk}^2 f^1 \left(\sum_m w_{km}^1 x_m + b_k^1 \right) + b_j^2 \right) + b_i^3 \right]$$

Mathematical model

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

$$y_i^{l+1} = f^{l+1} \left[\sum_{j=1}^{N_l} w_{ij}^{l+1} f^l \left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l+1} \left(\dots f^1 \left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1 \right) \dots \right) + b_j^l \right) \right] \quad (9)$$

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 $\hat{x} \in \mathbb{R}^n \rightarrow \hat{y} \in \mathbb{R}^m$.

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

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \quad (10)$$

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 an ANN.

Additionally, we can represent the biases and activations as layer-wise column vectors \hat{b}_l and \hat{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 W_l is an $N_{l-1} \times N_l$ matrix, while \hat{b}_l and \hat{y}_l are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 as

$$\hat{y}_2 = f_2(W_2 \hat{y}_1 + \hat{b}_2) = f_2 \left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix} \right) \quad (11)$$

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

$$y_i^2 = f_2 \left(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2 \right) = f_2 \left(\sum_{j=1}^3 w_{ij}^2 y_j^1 + b_i^2 \right) \quad (12)$$

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 $W_l \hat{y}_{l-1}$ 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*

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

and the *hyperbolic tangent* function

$$f(x) = \tanh(x)$$

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 scale. 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')
```

The multilayer perceptron (MLP)

The multilayer perceptron is a very popular, and easy to implement approach, to deep learning. It consists of

- ➊ A neural network with one or more layers of nodes between the input and the output nodes.
- ➋ The multilayer network structure, or architecture, or topology, consists of an input layer, one or more hidden layers, and one output layer.
- ➌ The input nodes pass values to the first hidden layer, its nodes pass the information on to the second and so on till we reach the output layer.

As a convention it is normal to call a network with one layer of input units, one layer of hidden units and one layer of output units as a two-layer network. A network with two layers of hidden units is called a three-layer network etc etc.

For an MLP there is no direct connection between the output nodes/neurons/units and the input nodes/neurons/units. Hereafter we will call the various entities of a layer for nodes. There are also

From one to many layers, the universal approximation theorem

A neural network with only one layer, what we called the simple perceptron, is best suited if we have a standard binary model with clear (linear) boundaries between the outcomes. As such it could equally well be replaced by standard linear regression or logistic regression. Networks with one or more hidden layers approximate systems with more complex boundaries.

As stated earlier, an important theorem in studies of neural networks, restated without proof here, is the [universal approximation theorem](#).

It states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of real functions. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters. It is the multilayer feedforward architecture itself which gives neural networks the potential of being universal approximators.

Deriving the back propagation code for a multilayer perceptron model

Note: figures will be inserted later!

As we have seen now in a feed forward network, we can express the final output of our network in terms of basic matrix-vector multiplications. The unknown quantities are our weights w_{ij} and we need to find an algorithm for changing them so that our errors are as small as possible. This leads us to the famous [back propagation algorithm](#).

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights?

To derive these equations let us start with a plain regression problem and define our cost function as

$$\mathcal{C}(\hat{W}) = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2,$$

Definitions

With our definition of the targets \hat{t} , the outputs of the network \hat{y} and the inputs \hat{x} we define now the activation z_j^l of node/neuron/unit j of the l -th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the forward passes/outputs \hat{a}^{l-1} from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l \hat{a}_i^{l-1} + b_j^l,$$

where b_j^l are the biases from layer l . Here M_{l-1} represents the total number of nodes/neurons/units of layer $l-1$. The figure here illustrates this equation. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\hat{z}^l = (\hat{W}^l)^T \hat{a}^{l-1} + \hat{b}^l.$$

With the activation function \hat{z}^l we can in turn define the output of

Derivatives and the chain rule

From the definition of the activation z_j^l we have

$$\frac{\partial z_j^l}{\partial w_{ji}^l} = a_i^{l-1},$$

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1 - a_j^l) = f(z_j^l)(1 - f(z_j^l)).$$

Derivative of the cost function

With these definitions we can now compute the derivative of the cost function in terms of the weights.

Let us specialize to the output layer $l = L$. Our cost function is

$$C(\hat{W}^L) = \frac{1}{2} \sum_{i=1}^n (y_i - t_i)^2 = \frac{1}{2} \sum_{i=1}^n (a_i^L - t_i)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = (a_j^L - t_j) \frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L} \frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1 - a_j^L) a_k^{L-1},$$

Bringing it together, first back propagation equation

We have thus

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = (a_j^L - t_j) a_j^L(1 - a_j^L) a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1 - a_j^L) (a_j^L - t_j) = f'(z_j^L) \frac{\partial C}{\partial (a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\hat{\delta}^L = f'(\hat{z}^L) \odot \frac{\partial C}{\partial (\hat{a}^L)}.$$

This is an important expression. The second term on the right handside measures how fast the cost is changing as a function of the j th output activation. If, for example, the cost function doesn't depend much on a particular output node j , then δ_j^L will be small, which is what we would expect. The first term on the right, measures how fast the activation function f is changing at a given activation value z_j^L .

Derivatives in terms of z_j^L

It is also easy to see that our previous equation can be written as

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L},$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L , namely

$$\delta_j^L = \frac{\partial C}{\partial b_j^L} \frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial C}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

Bringing it together

We have now three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

The starting equations

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \quad (13)$$

and

$$\delta_j^L = f'(z_j^L) \frac{\partial C}{\partial (a_j^L)}, \quad (14)$$

and

$$\delta_j^L = \frac{\partial C}{\partial b_j^L}, \quad (15)$$

An interesting consequence of the above equations is that when the activation a_k^{L-1} is small, the gradient term, that is the derivative of

Final back propagating equation

We have that (replacing L with a general layer l)

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l + 1$. Using the chain rule and summing over all k entries we have

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

we obtain

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l),$$

This is our final equation.

Setting up the Back propagation algorithm

The four equations provide us with a way of computing the gradient of the cost function. Let us write this out in the form of an algorithm.

First, we set up the input data \hat{x} and the activations \hat{z}_1 of the input layer and compute the activation function and the pertinent outputs \hat{a}^1 .

Secondly, we perform then the feed forward till we reach the output layer and compute all \hat{z}_l of the input layer and compute the activation function and the pertinent outputs \hat{a}^l for $l = 2, 3, \dots, L$.

Thereafter we compute the output error $\hat{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L) \frac{\partial \mathcal{C}}{\partial (a_j^L)}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \dots, 2$ as

Setting up a Multi-layer perceptron model for classification

We are now going to develop an example based on the MNIST data base. This is a classification problem and we need to use our cross-entropy function we discussed in connection with logistic regression. The cross-entropy defines our cost function for the classification problems with neural networks.

In binary classification with two classes $(0, 1)$ we define the logistic/sigmoid function as the probability that a particular input is in class 0 or 1. This is possible because the logistic function takes any input from the real numbers and inputs a number between 0 and 1, and can therefore be interpreted as a probability. It also has other nice properties, such as a derivative that is simple to calculate.

For an input \mathbf{a} from the hidden layer, the probability that the input \mathbf{x} is in class 0 or 1 is just:

$$P(y = 0 \mid \mathbf{x}, \theta) = \frac{1}{1 + \exp(-\mathbf{a}^T \mathbf{w}_{out})},$$

Defining the cost function

Our cost function is given as (see the Logistic regression lectures)

$$\mathcal{C}(\theta) = -\ln P(\mathcal{D} \mid \theta) = -\sum_{i=1}^n y_i \ln[P(y_i = 0)] + (1 - y_i) \ln[1 - P(y_i = 0)] =$$

This last equality means that we can interpret our cost function as a sum over the loss function for each point in the dataset $\mathcal{L}_i(\theta)$. The negative sign is just so that we can think about our algorithm as minimizing a positive number, rather than maximizing a negative number.

In *multiclass* classification it is common to treat each integer label as a so called *one-hot* vector:

$y = 5 \rightarrow \mathbf{y} = (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)$, and

$y = 1 \rightarrow \mathbf{y} = (0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$,

i.e. a binary bit string of length C , where $C = 10$ is the number of classes in the MNIST dataset (numbers from 0 to 9)..