MULTILAYER FEED-FORWARD NETWORK

Preview

The field of artificial neural networks is often just called neural networks or multi-layer Perceptrons after perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks.

The power of neural networks come from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm.

The predictive capability of neural networks comes from the hierarchical or multi-layered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function.

An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and strength of the output signal.

Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

Traditionally non-linear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Non-linear functions like the logistic also called the sigmoid function were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called tanh that outputs the same distribution over the range -1 to +1.

Figure 1 shows a typical three-layer perceptron. In general, a standard L-layer feed-forward network (we adopt the convention that the input nodes are not counted as a layer) consists of an input stage, (L-1) hidden layers, and an output layer of units successively connected (fully or locally) in a feed-forward fashion with no connections between units in the same layer and no feedback connections between layers

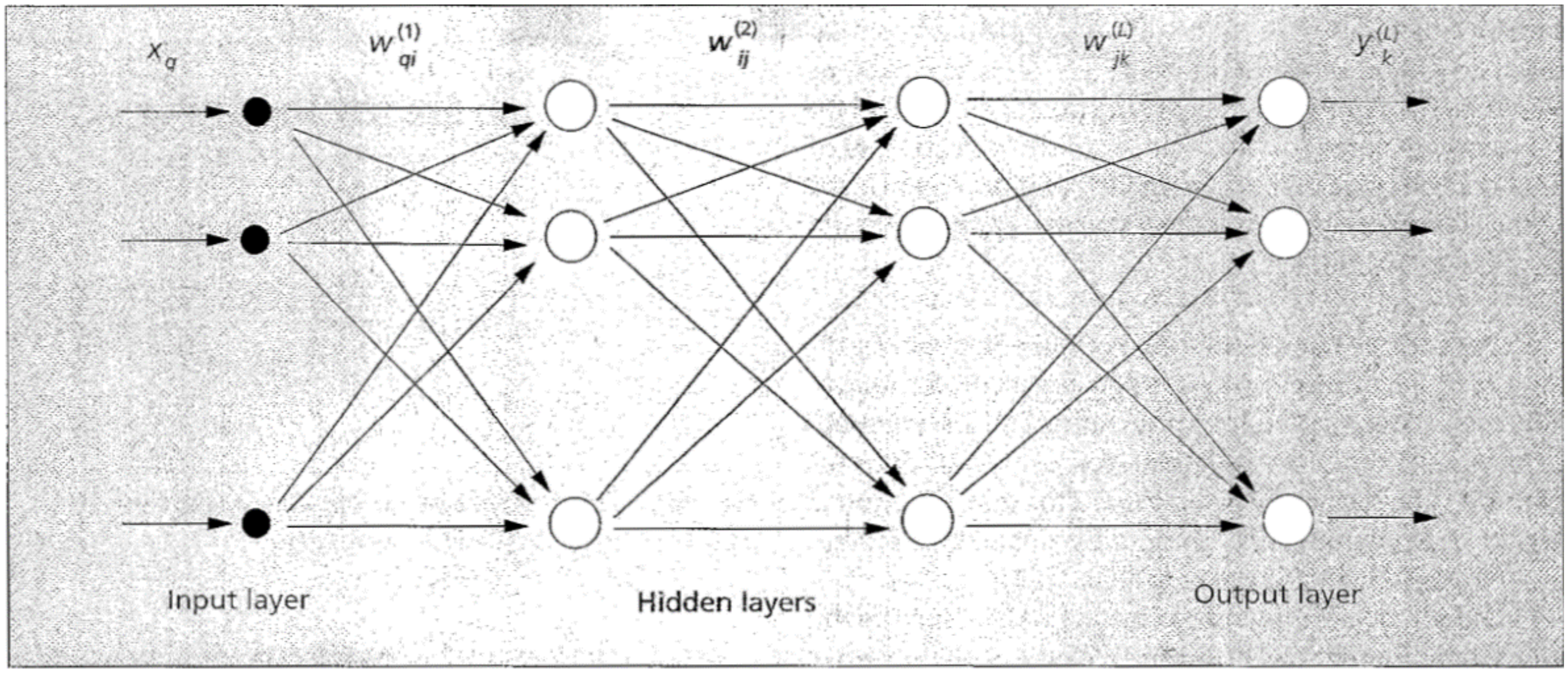

Figure 1: A typical three-layer feed-forward network architecture

Multilayer perceptron

The most popular class of multilayer feed-forward networks is multilayer Perceptrons in which each computational unit employs either the thresholding function or the sigmoid function. Multilayer Perceptrons can form arbitrarily complex decision boundaries and represent any Boolean function. The development of the back-propagation learning algorithm for determining weights in a multilayer perceptron has made these networks the most popular among researchers and users of neural networks.

We denote w_{ij}^{l}as the weight on the connection between the ith unit in layer $(l-1)$ to jth unit in layer l.

Let $\{(\boldsymbol{x}^{(1)}, \boldsymbol{d}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{d}^{(2)}), \ldots, (\boldsymbol{x}^{(p)}, \boldsymbol{d}^{(p)})\}$ be a set of p training patterns (input-output pairs), where $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n}$ is the input vector in the n-dimensional pattern space, and $\boldsymbol{d}^{(i)} \in [0,1]^{m}$, an m-dimensional hypercube. For classification purposes, m is the number of classes. The squared-error cost function most frequently used in the ANN literature is defined as

$$E = \frac{1}{2} \sum_{i=1}^{p} \left\| \boldsymbol{y}^{(i)} - \boldsymbol{d}^{(i)} \right\|^{2}$$

The back-propagation algorithm is a gradient-descent method to minimize the squared-error cost function in the Equation above.

Back-Propagation algorithm

1. Initialize the weight to small random values.
2. Randomly choose an input pattern $\boldsymbol{x}^{(\boldsymbol{\mu})}$.
3. Propagate the signal forward through the network.
4. Compute δ_i^L in the output layer $(o_i = y_i^L)$
$$\delta_i^L = g'(h_i^L)[d_i^{\boldsymbol{\mu}} - y_i^L],$$
Where h_i^l represent the net input to the ith unit in the lth layer, and g' is the derivative of the activation function g.
5. Compute the deltas for the preceding layers by propagating the errors backwards;
$$\delta_i^l = g'(h_i^l)\sum_j w_{ij}^{\boldsymbol{l}+1}\delta_j^{l+1},$$
For $l = (L-1), \dots, 1$.
6. Update the weight using
$$\Delta w_{ij}^l = \eta\delta_i^l y_j^{l-1}$$
7. Go to step 2 and repeat for the next pattern until the error in the output layer is below a prespecified threshold or maximum number of iterations is reached.

A geometric interpretation shown in Figure 2 can help explicate the role of hidden units (with the threshold activation function).

Each unit in the first hidden layer forms a hyperplane in the pattern space; boundaries between pattern classes can be approximated by hyperplanes. A unit in the second hidden layer forms a hyperregion from the outputs of the first-layer units; a decision region is obtained by performing an AND operation on the hyperplanes. The output-layer units combine the decision regions made by the units in the second hidden layer by performing logical OR operations. Remember that this scenario is depicted only to explain the role of hidden units. Their actual behavior, after the network is trained, could differ.

A two-layer network can form more complex decision boundaries than those shown in Figure 2. Moreover, multilayer Perceptrons with sigmoid activation functions can form smooth decision boundaries rather than piecewise linear boundaries.

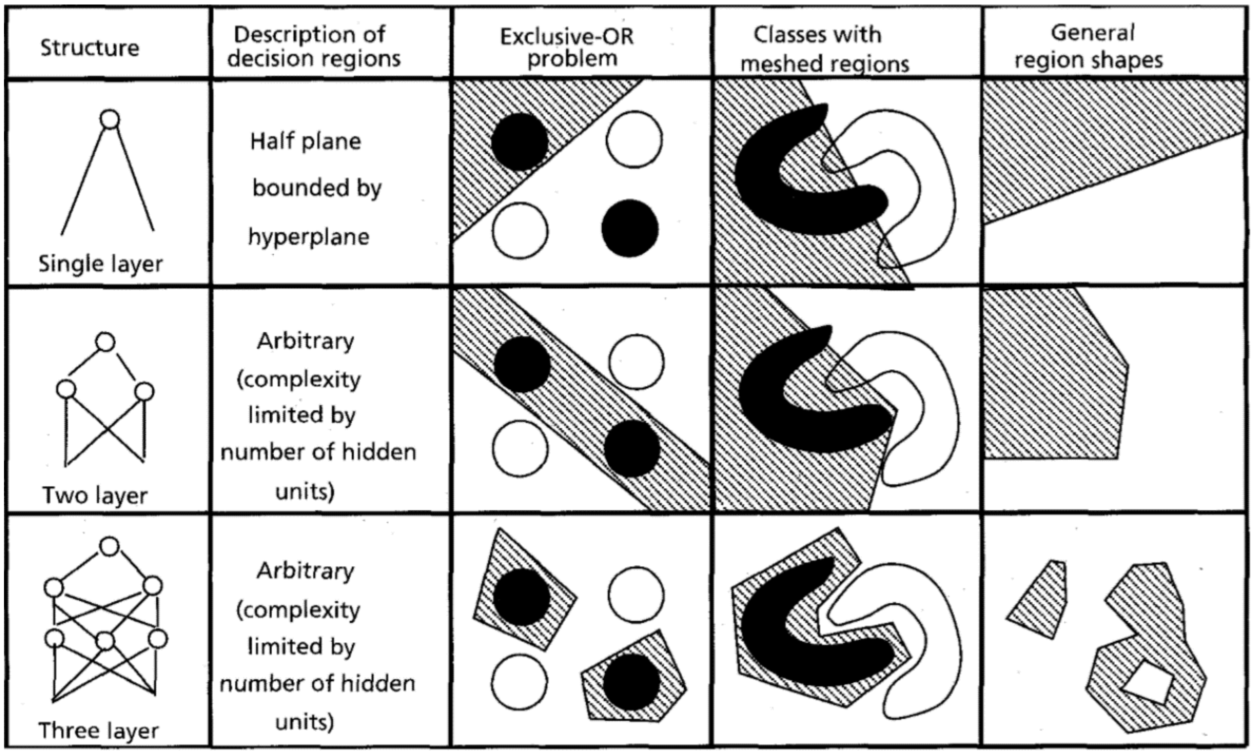
Structure
Description of decision regions
Exclusive-OR problem
Classes with meshed regions
General region shapes
Single layer
Half plane bounded by hyperplane
Two layer
Arbitrary (complexity limited by number of hidden units)
Three layer
Arbitrary (complexity limited by number of hidden units)

Figure 2: A geometric interpretation of the role of hidden unit in a two-dimensional input space.

Local Minima

In gradient descent we start at some point on the error function defined over the weights and attempt to move to the global minimum of the function. In the simplified function of Fig 3a the situation is simple. Any step in a downward direction will take us closer to the global minimum. For real problems, however, error surfaces are typically complex, and may more resemble the situation shown in Fig 3b. Here there are numerous local minima, and the ball is shown trapped in one such minimum. Progress here is only possible by climbing higher before descending to the global minimum.

Momentum

A momentum term was introduced in the BP algorithm by Rumelhart. The idea consists in incorporating in the present weight update some influence of the past iterations. The delta rule becomes

$$\Delta w_{ij}(n) = -\eta \frac{\partial E(n)}{\partial w_{ij}(n)} + \alpha \Delta w_{ij}(n-1).$$

α is the momentum parameter and determine the amount of influence from the previous iteration on the present one. The momentum introduces a "damping" effect on the search procedure thus avoiding oscillations in irregular areas of the error surface by averaging gradient components with opposite sign and accelerating the convergence in long flat areas. In some situations, it possibly avoids the search procedure from being stopped in a local minimum helping it to skip over those regions without performing any minimization there. In summary it has been shown to improve the convergence of the BP algorithm, in general.

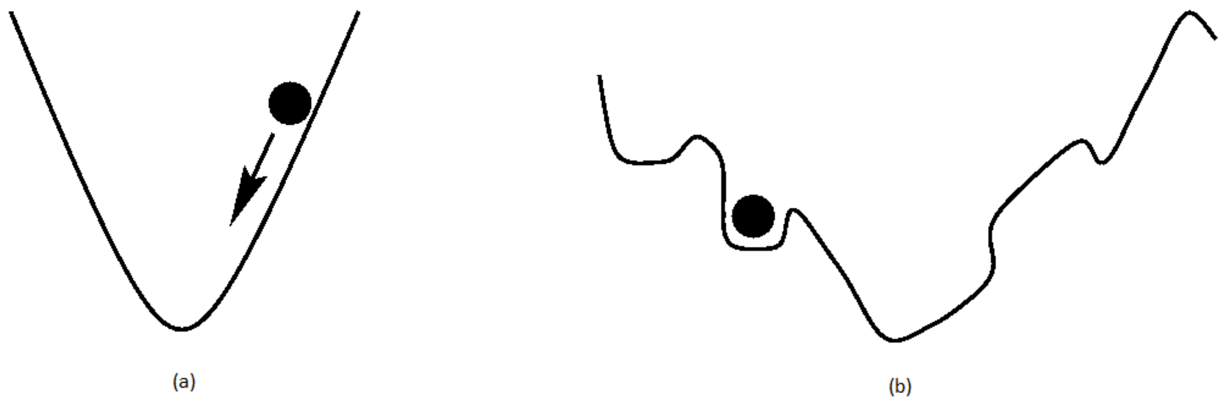

Figure 3: Local Minima and global minimum

Batch Backpropagation

Neural networks are often trained using algorithms that approximate gradient descent. Gradient descent learning (also called steepest descent) can be done using either a batch method or an on-line method. In batch training, weight changes are accumulated over an entire presentation of the training data (an epoch) before being applied, while on-line training updates weights after the presentation of each training example (instance). Another alternative is sometimes called mini-batch, in which weight changes are accumulated over some number u of instances before actually updating the weights.

on-line training learns faster than batch training because it takes many steps per epoch which can follow curves in the gradient. On-line training handles large training sets and redundancy in data well and avoids the need to store an accumulated weight change.

Batch training, on the other hand, can only take a single step per epoch, which must be in a straight line, please see Fig 4. It estimates the gradient only at the starting point in weight space, and thus cannot follow curves in the error surface. As the size of the training set gets larger, batch training must use a smaller learning rate in order for its learning to remain stable.

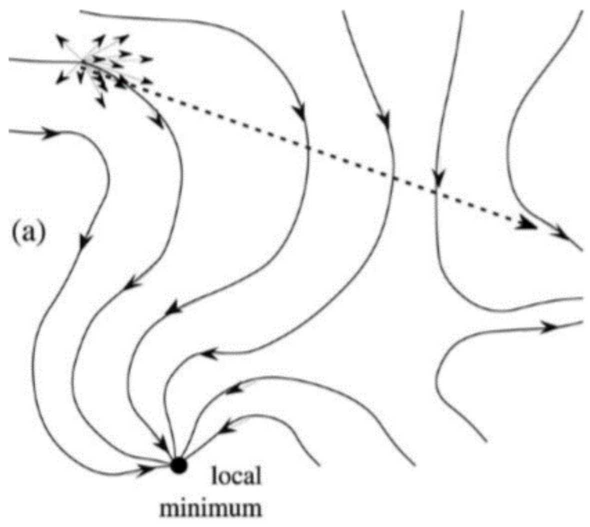

Figure 4: Example of changes in weight space. The directed curves indicate the underlying true gradient of the error surface (a) Batch training. Several weight changevectorsandtheirsum. (b) On-line training. The local gradient influences the direction of each weight change vector, allowing it to follow curves

References and further reading:

Jain, A. K., Mao, J., & Mohiuddin, K. M. (1996). Artificial neural networks: A tutorial. Computer, 29(3), 31-44.

Wilson, D. R., & Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. Neural Networks, 16(10), 1429–1451.

Moreira, M., & Fiesler, E. (1995). Neural networks with adaptive learning rate and momentum terms (No. REP_WORK). Idiap.

https://www.willamette.edu/~gorr/classes/cs449/momrate.html