Chapter 12

Neural Networks

The term "neural network" has come to be an umbrella term covering a broad range of different function approximation or pattern classification methods.

The classic type of neural network can be defined simply as a connected graph with weighted edges and computational nodes. Let us summarize those we have seen up to this point:

- The Linear Neural Net: $\boldsymbol{x} \to A\boldsymbol{x} + \boldsymbol{b}$.
- Kohonen's SOM: The graph is the network topology, and the units undergo competition.
- Neural Gas: Same interpretation as Kohonen's SOM.
- Radial Basis Functions: Each "center" represents a computational unit, and the output of unit i was $\boldsymbol{x} \to \phi(\|\boldsymbol{x} - \boldsymbol{c}^{(i)}\|)$

In the case of the Linear Neural Net and the RBF, network training was cast as solving a least squares problem, so that we could use features of linear algebra to solve for the best parameters.

In this chapter, we look at the classic workhorse of the neural network industry: The three-layer, feed-forward neural network.

As with the RBF, we will see that the three-layer, feed-forward network is capable of performing function approximation arbitrarily well when the domain is a compact subset of $\mathbb{R}^n$.

12.1 Biological Motivation

The neuron is the basic building block of the central nervous system. There are many different types of neurons, but each can be anatomically broken down into three parts: the dendrites, the cell body, and the axon. Information flows from the dendrites to the cell body through the axon to a synaptic junction connecting to the dendrite to the next neuron.

The synaptic junction is made up of the presynaptic node (the end of an axon), the postsynaptic node (the beginning of a dendrite), and the "empty space", which is the synapse.

For the purposes of our basic mathematical model, we will assume the following processing features that a neuron makes to information (we will refer to Neurons as Nodes in what follows):

1. As information flows across a synapse, information can be amplified, inhibited, or re-polarized. If we let x denote the signal, then synaptic processing is represented as a multiple of x:

   $$x \to wx$$

   where w will be referred to as a *weight*. Since there are multiple incoming signals, we denote by w_{ij} the weight connection from Node j to Node i.

2. At the cell body, information from across the dendrites is collected and processed in the following way.

   (a) The incoming signals are summed. Let $j = 1..m$ denote the index of the signal coming to Node i:

   $$\sum_{j=1}^{m} w_{ij}x$$

   (b) This quantity is added to the resting state of the cell, which is assumed to be constant:

   $$\sum_{j=1}^{m} w_{ij}x + b_i$$

   and b_i is called the bias of the Node. The summed quantity is referred to as the Prestate of the Node.

   (c) Biologically, we assume an "all or nothing" response to the incoming signal. That is, if the signal strength is above a certain threshold, the neuron fires. If not, then the information is not passed along. Mathematically, we use a step function, with the step at 0. This is not a differentiable function, however. We substitute for the step function a "sigmoidal" function, $\sigma(x)$. This is defined to be a monotonically increasing function so that

   $$\lim_{x\to-\infty} \sigma(x) = A \qquad \lim_{x\to\infty} \sigma(x) = B$$

   where A and B are constants. Common choices for the sigmoidal function include:

   i. The inverse tangent:

   $$\sigma(x) = \text{atan}(x)$$

ii. The exponential sigmoidal function:

$$\sigma_\beta(x) = \frac{1}{1 + \mathrm{e}^{-\beta x}}$$

We will use the latter choice, since the computation of its derivative is especially easy to implement on the computer:

$$\sigma'_\beta(x) = \frac{\beta \mathrm{e}^{-\beta x}}{1 + \mathrm{e}^{-\beta x}} = \beta\sigma(x)(1 - \sigma(x))$$

Other functions that transfer the incoming signal out to the axon are possible. At times, we will use the following, denoted by $t(x)$ as transfer functions:

i. The Linear Node

$$t(x) = x$$

ii. The Circular Node (two inputs, two outputs per node):

$$t(x, y) = \left(\frac{x}{\sqrt{x^2 + y^2}}, \frac{y}{\sqrt{x^2 + y^2}} \right)$$

iii. The Spherical Node (three inputs, three outputs):

$$t(x, y, z) = \left(\frac{x}{\sqrt{x^2 + y^2 + z^2}}, \frac{y}{\sqrt{x^2 + y^2 + z^2}}, \frac{z}{\sqrt{x^2 + y^2 + z^2}} \right)$$

See [15, 16] for examples of how to implement the last two transfer function types.

3. The axon sends along the information to the synapse.

In summary, we have the following information processing for the i^{th} single node:

$$x \to \underbrace{\sum_{j=1}^{m} w_{ij}x + b_i}_{\text{Prestate}} \to \underbrace{\sigma\left(\sum_{j=1}^{m} w_{ij}x + b_i \right)}_{\text{State}}$$

In vector form,

$$x \to (\boldsymbol{w}_i)^{\mathrm{T}} x + b_i \to \sigma\left((\boldsymbol{w}_i)^{\mathrm{T}} x + b_i \right)$$

So that, taking $\boldsymbol{x} \in \mathbb{R}^N$, and having k Nodes,

$$\boldsymbol{x} \to \left((\boldsymbol{W}^{(0)})^{\mathrm{T}} \boldsymbol{x} + \boldsymbol{b} \right) \to \sigma\left((\boldsymbol{W}^{(0)})^{\mathrm{T}} \boldsymbol{x} + \boldsymbol{b} \right)$$

where

$$\boldsymbol{W}^{(0)} = (\boldsymbol{w}_1, \boldsymbol{w}_2, \ldots, \boldsymbol{w}_k) \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix}$$

and σ operates on a vector or matrix component-wise. Also note that, if σ were a linear function, then the neural network would be linear.

12.2 The Three Layer Feedforward Neural Network

The usual method of describing a neural network's architecture is to use a connected graph with computational nodes. The basic feedforward neural network uses three layers of "nodes". The first layer is the input layer, so the number of nodes is equal to the dimension of $\boldsymbol{x}$, and the transfer function is linear.

The hidden layer contains the nonlinear transfer function, and there is a choice as to how many nodes one includes in the hidden layer. We assume there are k nodes in the hidden layer.

The output layer again has a linear transfer function, and there are as many nodes here as there are output variables. Each edge of the graph is weighted, and we refer to edges from the input layer to the hidden layer as the matrix

$$\boldsymbol{W}^{(0)} = \left[W_{ij}^{(0)}\right]_{i=1:n(1),\ j=1:n(0)}$$

where $n(i)$ is the number of nodes in Layer i, and

$$\boldsymbol{W}^{(1)} = \left[W_{ij}^{(0)}\right]_{i=1:n(2),\ j=1:n(1)}$$

so that W_{ij}^{k} refers to the weight in the k^{th} layer, going from Node j in the k^{th} layer to Node i in the $k+1^{\text{st}}$ layer.

There are also vectors of biases associated with the hidden layer, and output layer, respectively:

$$\boldsymbol{b}^{(0)} \text{ and } \boldsymbol{b}^{(1)}$$

For the purposes of the backpropagation algorithm given later, note that the biases can also be represented as one additional node in the input and hidden layers, with constant transfer function, $\sigma(x) = 1$. We will adopt this notation, and suppress the bias specification in Section 4.2.1. Figure 12.1 shows the graphic layout of the three layer, feedforward neural network.

The purpose of this network is to provide an approximation to an arbitrary continuous function, $\boldsymbol{F}$, defined on a compact domain X, and we will use the notation $\mathbf{y} = \boldsymbol{F}(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^N$ and $\mathbf{y} \in \mathbb{R}^M$. The network will render the

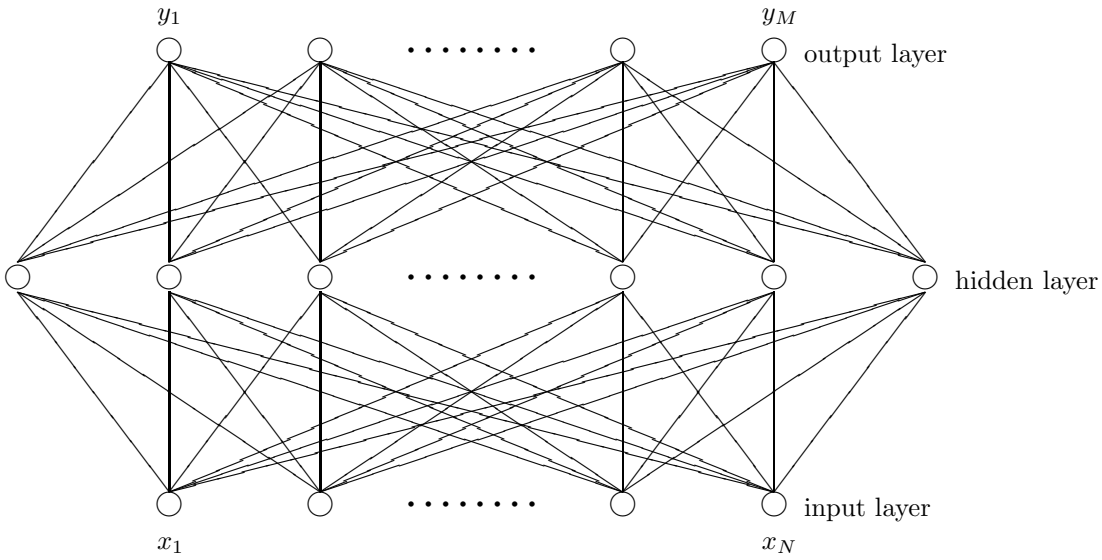

Figure 12.1: The neural network architecture.

approximation in terms of a sigmoidal basis so that the approximation to $\boldsymbol{F}$ is given pointwise by:

$$\boldsymbol{y}^{(i)} = \boldsymbol{F}(\boldsymbol{x}^{(i)}) \approx \left(\mathbf{W}^{(1)}\sigma\left(\mathbf{W}^{(0)} \cdot \mathbf{x}^{(\mathbf{i})} + \mathbf{b}^{(0)}\right) + \mathbf{b}^{(1)}\right) \tag{12.1}$$

In a paper of fundamental importance, Cybenko [7] shows that sums of the form in equation (12.1) are dense in the space of continuous functions defined on the unit cube. In other words, a neural network is capable of approximating any continuous function arbitrarily well by increasing the number of nodes in the hidden layer. As the number of nodes in the hidden layer increases, the error between the network approximation and the continuous function goes to zero. This is very significant for several reasons:

- It gives us a theoretical framework from which to work.
- The proof of this theorem ties neural networks to the classical approaches to function approximation in mathematics.
- It solves one of Hilbert's problems from the turn of the last century. Kolmogorov had given the first proof in the 1950's, but it is still not clear that the functions he used can be implemented in a computer algorithm.
- The proof revitalized neural network research. Prior to this, research had been stagnant after the discovery that the first neural networks (in the 1960's) were incapable of modeling certain standard functions (the XOR function). We know now that this was because the first networks only worked on linearly seperable data.

- Although we have taken outrageous liberties (biologically speaking) in our model, the model shows that a very simple structure is capable of very complex behavior.

In fact, there are more general results that can be found in [13, 14, 11, 12, 9, 10] and more specifically, they show that a neural approximation not only approximates the function, but can also simulatenously approximate the first m derivatives using the same functional parameters.

12.3 Training and Error

Theoretically, we have now seen the neural network as a "Universal Function Approximator". The real question now is, given an input and output set, how does one build the neural net? This question can be reframed as: Find the weights w_{jk}^i and biases b_j^i that perform a desired input-output. This process is also known as network training.

For a fixed value of the parameters in the matrices $\boldsymbol{W}^{(i)}$ and biases $\boldsymbol{b}^{(i)}$, the network outputs a smooth function in $\boldsymbol{x}$. As mentioned previously, we adopt the notation of including the biases in the parameters $\boldsymbol{W}^{(i)}$, and add an extra node on the hidden and input layers.

Define the mean square error of the approximation, given a set of parameters $\{\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}\}$, as:

$$E(\boldsymbol{W}^{(0)}, \boldsymbol{W}^{(1)}) = \frac{1}{2|X|} \sum_{i=1}^{|X|} \| F(\mathbf{x^{(i)}}) - \left(\mathbf{W}^{(1)} \, \sigma \left(\mathbf{W}^{(0)} \cdot \mathbf{x^{(i)}} \right) \right) \|_2^2 \qquad (12.2)$$

where $|X|$ denotes the number of points in the data set X. Then E is a smooth function of the weights and biases, so that it can be minimized. From the method of steepest descent, we update the weights:

$$W_{jk}^{(i)}(t+1) = W_{jk}^{(i)}(t) + \alpha(t) d(t) \qquad (12.3)$$

until the error has become smaller than a prescribed value. For the standard gradient descent, for example, we have

$$d(t) = \frac{\partial E}{\partial W_{jk}^{(i)}}(t)$$

How we find the value of $\alpha(t)$ and the definition of $d(t)$ is where the divergence in training algorithms occurs. There are many algorithms built especially for this purpose, such as Line Minimization. Other update methods can be applied, since training is a general, unconstrained optimization problem:

$$\min_{\boldsymbol{W}^{(0),(1)}} E$$

Most optimization algorithms will require that we obtain expressions to compute the partial derivatives of E with respect to the weights and biases. It is

fortunate that, for large networks, there is a recursive algorithm to accomplish this. This will be discussed in the next section. First, we introduce some notation that will make the algorithm more clear.

Define the prestate for layer i, Node j at time t:

$$P_j^i = \sum_{k=1}^{n(i-1)} W_{jk}^{i-1} S_k^{i-1}, \text{ for } i = 1 \dots L$$

where S_j^i is the State of Node j on layer i, which is

$$S_j^i = \begin{cases} \sigma\left(P_j^i\right) & i > 0 \\ x & i = 0 \end{cases}$$

12.3.1 Backpropagation of Error

We now detail a recursive algorithm, known as the backpropagation of error [1], which gives an efficient method for computing the partial derivatives for training (see Equation (12.3)).

The goal is to compute

$$\frac{\partial E}{\partial W_{jk}^{i-1}}$$

for each i, j, k.

From the chain rule, we can write

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \frac{\partial E}{\partial P_j^i} \cdot \frac{\partial P_j^i}{\partial W_{jk}^{i-1}}$$

which we can re-express, using:

$$\frac{\partial P_j^i}{\partial W_{jk}^{i-1}} = S_k^{i-1} \quad \frac{\partial P_j^{i+1}}{\partial P_k^i} = W_{jk}^i \sigma'(P_k^i)$$

and defining:

$$\delta_j^i = \frac{\partial E}{\partial P_j^i}$$

Then we have:

$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$

So, we need an algorithm to compute δ_j^i.

For the output layer (L), with aribtrary transfer function σ,

$$\delta_j^L = \frac{\partial E}{\partial P_j^L} = \frac{\partial E}{\partial S_j^L} \frac{\partial S_j^L}{\partial P_j^L} = \sigma'(P_j^L) \frac{\partial E}{\partial S_j^L}$$

where $\frac{\partial E}{\partial S_j^L}$ is computed from the error by substituting S_j^L into Equation (12.2):

$$E = \frac{1}{|X|} \sum_{i=1}^{2|X|} \|\boldsymbol{y}^{(i)} - \begin{pmatrix} S_1^{(L)} \\ \vdots \\ S_m^{(L)} \end{pmatrix} \|_2^2$$

which implies that:

$$\frac{\partial E}{\partial S_j^L} = \frac{-1}{|X|} \sum_{i=1}^{|X|} |y_j^{(i)} - S_j^L| \tag{12.4}$$

For the hidden nodes,

$$\delta_j^i = \frac{\partial E}{\partial P_j^i} = \sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}} \frac{\partial P_k^{i+1}}{\partial P_j^i}$$

Noting $\frac{\partial P_k^{i+1}}{\partial P_j^i} = W_{kj}^i \sigma'(P_j^i)$, we obtain:

$$\delta_j^i = \sigma'(P_j^i) \sum_{k=1}^{n(i+1)} \frac{\partial E}{\partial P_k^{i+1}} W_{kj}^i$$

Which, from our definition of δ_j^i, is recursively defined as:

$$\delta_j^i = \sigma'(P_j^i) \sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i \tag{12.5}$$

This can be interpreted as a backpropagating the δ_j^i backwards through the network. The algorithm is summarized as:

Algorithm 12.3.1 *The Backpropagation Algorithm*

1. Forward propagate all the x's, keeping track of the error.
2. Compute δ_j^L, for $j = 0 \ldots M-1$ using:
$$\delta_j^L = \sigma'(P_j^L) \frac{\partial E}{\partial S_j^L}$$
3. Compute all other δ's:
$$\delta_j^i = \sigma'(P_j^i) \sum_{k=0}^{n(i+1)} \delta_k^{i+1} W_{kj}^i$$
4. Compute:
$$\frac{\partial E}{\partial W_{jk}^{i-1}} = \delta_j^i \cdot S_k^{i-1}$$
for $i = 1 \ldots L$, $j = 1 \ldots n(i+1)$, and $k = 1 \ldots n(i)$.

12.3.2 Nonlinear Optimization Techniques

There are many methods to choose from, and some are included below:

- Steepest Descent (with variations on how to compute α)
- Newton's Method (This is indirect - we solve for where $f'(x) = 0$).
- Conjugate Gradient (Search along the eigenvectors of the Hessian).
- Levenburg-Marquardt (A combination of the methods above).

To really study all of these techniques would take us too far afield in this course. Any text on nonlinear optimization (as well as "Numerical Recipes for C") will cover the basic ideas.

The introduction of a nonlinear optimization technique is the primary reason we will use Matlab to perform our neural network training. Programming these techniques can be a delicate task! For our purposes, Levenburg-Marquardt is more than adequate.

12.4 Neural Networks and Matlab

Matlab has set up a very general data structure by which to construct and train almost any type of neural network. Here we focus on the construction and operation of the three-layer feed-forward network.

Because the data structure is so general, building and training a neural network is broken into multiple stages.

- Identify the network architecture. This includes constructing neural layers, connections, identifying the transfer functions for each node. Furthermore, one must define the error measure (or performance index), the training function, and how one wishes to compute a Jacobian matrix. This step would be extremely tedious if we had to do it every time! Matlab has several built in architectures, and we've dealt with a few. For example:
  - `newsom` initializes Kohonen's SOM.
  - `newlin` initializes a linear neural network.
  - `newrb` initializes a radial basis function network.
  - `newrbe` initializes (and trains) an exact RBF.

  The new function here is `newff`.
- Train the network. This is performed (for all network types) by the command: `net=train(net,X)`. The specific functions and training parameters have already been set up by the initialization process.
- Simulate the network with new data. Again, this is performed generically by: `Y=sim(net,X)`. The specific functions and parameters have been set in the network data structure.