6

MULTI-LAYER NETWORKS

Vasant Honavar

Artificial Intelligence Research Group
Department of Computer Science
Iowa State University
Ames, Iowa 50011

1 INTRODUCTION

So far, we have considered single layer networks for pattern classification and function approximation. We have seen that the computational and function approximation abilities of such networks are fairly limited. For instance, threshold neurons and winner-take-all groups can be trained to classify only linearly separable pattern sets. Linear neurons can only find a a linear fit to data on function approximation tasks. However, we know that multi-layer networks of threshold neurons can realize arbitrary boolean functions. However, it is not immediately obvious how such networks could be trained. In particular, since perceptron algorithm requires the desired ouput to be known for each neuron and since we do not know the desired output for all but the output neurons, we need a different approach to training such networks. One approach is to replace thereshold neurons by sigmoid neurons and define differentiable error functions and extend the gradient-based error minimization techniques from the previous chapter to handle multi-layer networks. Similarly, one might ask if there are ways to constuct multi-layer networks of non-linear neurons to approximate any desired function $f : \Re^N \rightarrow \Re^M$ from examples. Other motivations for the study of such networks include: limitations of behaviorist models of learning (based on classical conditioning), in particular, their inadequacy in explaining successful learning of functions such as exclusive OR by animals in laboratory settings; non-linear statistical regression and nonlinear control systems.

The following theorem addresses the function approximation capabilities of multi-layer neurla networks.

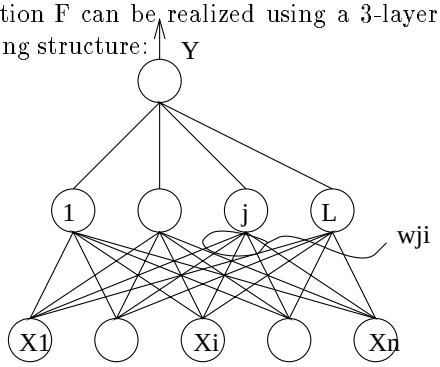
Universal Function Approximation Theorem
Let $\varphi()$ be a non-constant, bounded, monotone, continuous function. Let I_N denote the N-dimensional unit hypercube in Euclidian space $\Re^N$. Let $C(I_N)$ be the set of all continuous function $\{f : I_N \rightarrow \Re\}$

Then for any function $f \in C(I_N)$ and any $\varepsilon > 0$, $\exists$ an integer L and sets of real-values $\alpha_j, \theta_j, w_{ji}$ (j = 1 ... L, i = 1 ... N) such that

$$F(x_1 \ldots x_N) = \sum_{j=1}^{L} \alpha_j \varphi(\sum_{i=1}^{N} w_{ji} X_i - \theta_j) - \theta$$

is a *uniform* approximation of f.
That is, $|F(x_1 \ldots x_N) - f(x_i \ldots x_N)| < \varepsilon \ \ \forall (x_1 \ldots x_N) \in I_N$.

The approximation F can be realized using a 3-layer artificial neural network with the following structure:



The above theorem is easily extended to vector valued functions $f : \Re^N \Rightarrow \Re^M$. It also turns out that similar universal approximation theorems can be proved for different choices of ϕ e.g., radial basis functions which we will examine later.

Note that the universal function approximation theorem guarantees the existence of adequately accurate approximations of continuous functions defined on bounded subsets of $\Re^N$. It does not offer a recipe for finding such an approximation for an unknown function implicitly specified by a set of labeled examples. In other words, it does not give an algorithm for finding suitable values for L, and the parameters α_j, w_{ji}, θ_j. Nor does the theorem imply that a 3-layer network is optimal for approximating a given f to within a desired error bound ε. Thus one is left with task of discovering efficient algorithms for

constructing the approximations whose existence is guaranteed by the universal function approximation theorem.

Suppose $z_j = \frac{1}{1+e^{-n_j}}$ (this is clearly a non-constant, bounded, monotone, continuous function of the inputs), $y = \sum_{j=1}^{L} \alpha_j z_j$.

To simplify notation, we will replace θ_j by $-w_{j0}$ which can be thought of as a weight on a link connected to a constant input of 1. Similarly, we will replace θ by weight $-\alpha_0$.

2 LEARNING A FUNCTION APPROXIMATION FROM EXAMPLES

Given a training set of examples of an unknown function $f : \Re^N \Rightarrow \Re$ and an $\varepsilon > 0$, how can we find an integer L, and the parameters w_{ji} and α_j ($i = 0 \cdots N$, $j = 0 \cdots L$), so that the network provides a uniform approximation of f?

Note that this presents a difficult search problem in the space of network architectures (corresponding to different choices of L) and an infinite choice of parameter settings. To simplify the problem, we will first assume that we can make a reasonable guess for L. This fixes the network architecture, leaving us with the problem of finding suitable values of the parameters for the parameters w_{ji} and α_j. More generally, we can consider approximation of vector valued functions $f : \Re^N \Rightarrow \Re^M$ using networks with M output neurons. We can do the latter extending the gradient based error minimization techniques discussed in the previous chapter to multi-layer networks.

Let us define the following:

$$y_{kp} = \sum_{j=0}^{L} z_{jp} w_{kp} \tag{6.1}$$

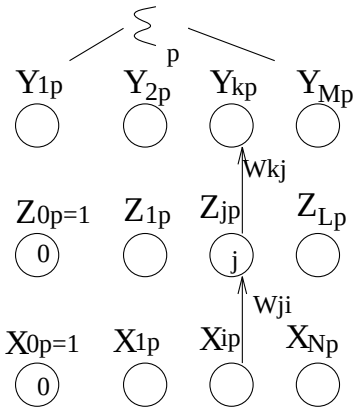
$$z_{jp} = \frac{1}{1 + e^{-n_{jp}}} \tag{6.2}$$

$$n_{jp} = \sum_{i=0}^{N} w_{ji} x_{ip} \tag{6.3}$$

$$d_{kp} = \text{desired output of neuron k for pattern} \mathbf{X}_p \tag{6.4}$$

$$
\begin{aligned}
E_p &= && \text{error on pattern} \mathbf{X}_p \\
&= && \frac{1}{2}\sum_{k=1}^{M} {e^2}_{kp} \\
&= && \frac{1}{2}\sum_{k=1}^{M} (d_{kp} - y_{kp})^2 \qquad (6.5) \\
& && \qquad (6.6)
\end{aligned}
$$

The problem now is to find the weights w_{ji}, w_{kj} $(i = 0 \ldots N, j = 0 \ldots L, k = 1 \ldots M)$ that minimize $E = \sum_p E_p$.



Using the per pattern version of gradient-based error minimization, we can write:

$$w_{kj} \leftarrow w_{kj} - \eta \frac{\partial E_p}{\partial w_{kj}} \qquad (6.7)$$

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_p}{\partial w_{ji}} \qquad (6.8)$$

where $\eta > 0$ is a suitable learning rate.

$$\frac{\partial E_p}{\partial w_{kj}} = \frac{1}{2}\sum_{l=1}^{M} \frac{\partial {e^2}_{lp}}{\partial w_{kj}}$$

$$\begin{aligned} &= \frac{1}{2}\frac{\partial e^2{}_{kp}}{\partial y_{kp}}\frac{\partial y_{kp}}{\partial w_{kj}} \\ &= 2\frac{1}{2}(-)(d_{kp}-y_{kp})z_{jp} \\ &= -(d_{kp}-y_{kp})z_{jp} \\ &= -\delta_{kp}z_{jp} \end{aligned} \tag{6.9}$$

where $\delta_{kp} = (d_{kp} - y_{kp})$ is the error associated with the kth output node.

So we have:

$$w_{kj} \leftarrow w_{kj} + \eta\delta_{kp}z_{jp} \tag{6.10}$$

Now we have:

$$\begin{aligned} \frac{\partial E_p}{\partial w_{ji}} &= \sum_{k=1}^{M}\frac{\partial E_p}{\partial y_{kp}}\frac{\partial y_{kp}}{\partial w_{ji}} \\ &= \sum_{k=1}^{M}\frac{\partial E_p}{\partial y_{kp}}\frac{\partial y_{kp}}{\partial z_{jp}}\frac{\partial z_{jp}}{\partial n_{jp}}\frac{\partial n_{jp}}{\partial w_{ji}} \end{aligned} \tag{6.11}$$

Where:

$$\begin{aligned} \frac{\partial E_p}{\partial y_{kp}} &= -(d_{kp}-y_{kp}) \\ &= -\delta_{kp} \\ \frac{\partial y_{kp}}{\partial z_{jp}} &= w_{kj} && (6.12) \\ \frac{\partial z_{jp}}{\partial n_{jp}} &= \frac{-1(-e^{-n_{jp}})}{(1+e^{-n_{jp}})^2} \\ &= z_{jp}(1-z_{jp}) && (6.13) \\ \frac{\partial n_{jp}}{\partial w_{ji}} &= x_{ip} && (6.14) \end{aligned}$$

Thus, we have:

$$\frac{\partial E_p}{\partial w_{ji}} = -\sum_{k=1}^{M}\delta_{kp}w_{kj}z_{jp}(1-z_{jp})x_{ip} \tag{6.15}$$

We can write the error associated with the hidden node node j as $\delta_{jp} = \sum_{k=1}^{M} \delta_{kp} w_{kj} z_{jp}(1 - z_{jp})$. Then:

$$w_{ji} \leftarrow w_{ji} + \eta \delta_{jp} x_{ip} \tag{6.16}$$

Note that the weight update equation has the general form learning rate multiplied by the error for the node times the input on the link. Since the error for the hidden nodes is computed using the error for the output nodes, this technique is called error backpropagation.

The resulting learning algorithm (also known as the generalized delta rule) is summarized below:

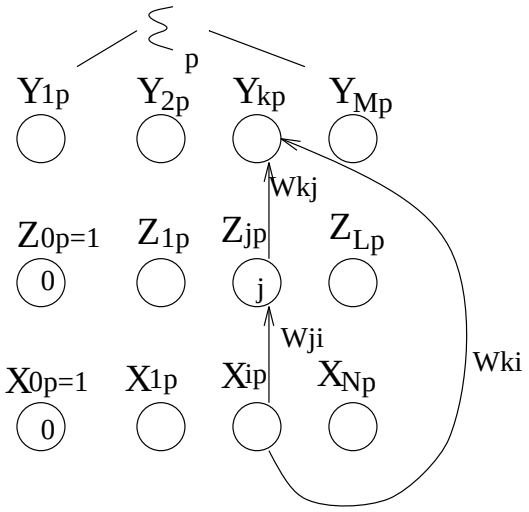
1. Select a network architecture.
2. Initialize the weights to small random values.
3. Present the network with training examples from training set in some order. A complete pass through the training set is called a training epoch. It helps to randomize the order of presentation of examples in each epoch.
   (a) For each training example $((X_p,\ D_p),\ D_p = [d_{1p} \ldots d_{np}]$ and $X_p = [x_{0p} \ldots x_{np}])$, compute the activations and outputs of each of the neurons in the network (e.g., $z_j p,\ y_k p$) (forward pass).
   (b) Compute the errors for each of the neurons in the network (e.g., δ_{kp}, δ_{jp} and $E = \sum E_p$.
   (c) If E $\leq \epsilon$ stop otherwise goto step 3.

When such networks are used for pattern classification, we assign one output neuron per class. Thus we have M output neurons for an M-Category classification problem. When the network is used to classify test patterns, a pattern X_p is assigned to class j if $y_{jp} > y_{kp} \forall_{k \neq j} (k = 1 \ldots M)$ (with the ties broken at random). This ensures optimal classification in the Baysesian sense when linear output neurons are used.

The general strategy used here for deriving the weight learning rules can be extended to networks with an arbitrary number of layers, using any neuron functions that are differentiable with respect to modifiable parameters (e.g.,

weights), and arbitrary feedforward connectivity. Furthermore, other differentiable error functions besides the squared error function can be used when appropriate. A variety of techniques can be used to speed up learning. These include the use of momentum modification discussed in the previous chapter.

As an exercise, consider a 3-layer network with direct links from input to output neurons in addition the links from inputs to hidden neurons and from hidden neurons to output neurons.



As before,

$$
\begin{aligned}
z_{jp} &= \frac{1}{1+e^{-n_{jp}}} \\
n_{jp} &= \sum_{i=0}^{N} w_{ji}x_{ip} \\
E_p &= \tfrac{1}{2}\sum_{k=1}^{M} e^2{}_{kp} \\
&= \frac{1}{2}\sum_{k=1}^{M}\left(d_{kp}-y_{kp}\right)^2 \\
d_{kp} &= \text{desired output of neuron } k \text{ for pattern } \mathbf{X}_p
\end{aligned}
$$

But now the output is given by:

$$y_{kp} = \sum_{j=0}^{L} z_{jp} w_{kj} + \sum_{i=0}^{N} w_{ki} x_{ip} \tag{6.17}$$

Consider the update equations for the weights w_{kj}, w_{ji}, and w_{ki} where $(i = 0 \ldots N, j = 0 \ldots L, k = 1 \ldots M)$.

We have:

$$w_{kj} \leftarrow w_{kj} - \eta \frac{\partial E_p}{\partial w_{kj}} \tag{6.18}$$

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_p}{\partial w_{ji}} \tag{6.19}$$

$$w_{ki} \leftarrow w_{ki} - \eta \frac{\partial E_p}{\partial w_{ki}} \tag{6.20}$$

It is easy to show that:

$$\frac{\partial E_p}{\partial w_{kj}} = -(d_{kp} - y_{kp}) z_{jp} = -\delta_{kp} z_{jp} \tag{6.21}$$

$$\frac{\partial E_p}{\partial w_{ji}} = -\sum_{k=0}^{M} w_{kj} z_{jp} (1 - z_{jp}) x_{ip} = -\delta_{jp} x_{ip} \tag{6.22}$$

$$\frac{\partial E_p}{\partial w_{ki}} = -\delta_{kp} x_{ip} \tag{6.23}$$

3 NETWORK PRUNING

Note that the backpropagation algorithm does not offer a technique for choosing an appropriate network architecture for a given data set. If the network is too small, it may fail to approximate the function specified by the training set to a desired accuracy. If it is too large, it can overfit the data (or in essence, memorize the training set) and hence generalize poorly on test data. There is considerable theoretical as well as practical justification for prefering simpler approximations (networks) that adequately fit the training data. Generally speaking, all other factors being the same, networks with fewer modifiable parameters tend to yield simpler approximations. Thus, algorithms that adapt

the network topology in addition to the parameters (e.g., weights) to match the complexity of the function being approximated are of interest. The difficulty of course is that we do not, in general, have a-priori knowledge of the complexity of the function to be learned. There are two broad classes of approaches for dealing with this problem:

- Constructive or generative algorithms that incrementally grow networks
- Pruning algorithms that start with a large network and gradually get rid of neurons and/or links

We will examine constructive algorithms in a later chapter. In what follows, we outline some approaches to network pruning using a modified error function which incorporates a cost penalty for weights. Consider a modified error function given by $R(\mathbf{W}) = \lambda E(\mathbf{W}) + (1 - \lambda)C(\mathbf{W})$ where $0 \leq \lambda \leq 1$. Here, $E(\mathbf{W})$ denotes the error of approximation (e.g., the sum squared error used in deriving the weight update equations in the previous section) and $C(\mathbf{W})$ is a term that penalizes weights λ is a user-defined parameter that trades off the error of the approximation against network complexity (as measured by $C(\mathbf{W})$). The particular form of $R(\mathbf{W})$ ensures that $R(\mathbf{W})$ is convex function of the parameters (e.g., weights) whenever both E and C are convex functions. One possible choice of C is given by $C(\mathbf{W}) = \sum_{\text{over all the weights}}(w_j i)^2$. Note that the negative gradient of C with respect to any particular weight tends to drive the weight towards zero, leading to relatively sparse networks. However, the gradient of E changes the weights so as to minimize E. The resulting network is a compromise between these two requirements.

4 BACKPROPAGATION ALGORITHM IN PRACTICE

Backpropagation algorithm is a useful practical algorithm for function approximation and pattern classification. It has been successfully used on a wide range of applications including handwriting recognition, face recognition, speech recognition, approximating complex and time consuming computations (e.g., numerical solution of differential equations for specific choices of inputs) etc. Successful application of the technique on large scale practical problems often requires additional modifications and the use of domain-specific tricks. There is a significant body of literature that describes such enhancements or

special tricks that make the algorithm more useful for solving real-world problems.

A particularly important issue of practical interest is the possibility of gradient-based learning algorithm (like backpropagation) getting caught in a local minimum of the error surface. Although the sum of squared errors for a single layer network of linear neurons is a quadratic convex function with a unique global minimum, it can be shown that multi-layer networks of non-linear neurons (e.g., sigmoid neurons) can have multiple minima. Despite this, backpropagation algorithm works quite well in practice. The reasons for this are not completely understood. However, there are some plausible explanations:

- The networks are initialized with small random weights. This means that the sigmoid neurons operate in the linear region of their activation function. Thus, during the initial phase of training, region of the error surface that the network is operating with is more or less quadratic. As the training proceeds, the sigmoid neurons might start operating in the nonlinear portion of their activation function. If the weights have moved to the general neighborhood of the global minimum of the error surface by then, backpropagation can converge to weights that correspond to the global minimum of the error surface.
- Local minima in high-dimensional weight spaces require the gradient to be near zero and the second derivative to be positive with respect to each of the dimensions in the neighborhood of the minimum.