Artificial Neural Networks

Neural networks provide a more robust approach to approximating real-valued, discrete-valued and vector-valued target functions. They are built as a dense connection of interconnected neurons, where each neuron takes a real-valued input and produces a real-valued output.

ANNs are applicable to problems with the following characteristics

1. Instances are represented by attribute-value pairs
2. The target function can be real-valued, continuous, vector-valued
3. The training instances may contain errors
4. Long training times are acceptable
5. Fast evaluation of target function may be required
6. Ability of humans to understand the target function is not required

Perceptrons

A perceptron takes a vector of real valued inputs, calculates a linear combination of these inputs, and outputs a 1 if the result is greater than some threshold, and -1 otherwise.

Or,

$$o\left(x_1, \ldots x_n\right) = \begin{cases} 1 & if\ w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1 & otherwise \end{cases}$$

Here, (-w0) represents the threshold and w represents the weight assigned to each input. For brevity, the perceptron function is sometimes written as

$$o(\vec{x}) = sgn(\vec{w}.\vec{x})$$

$$where$$

$$sgn(y) = \begin{cases} 1\ if\ y > 0 \\ -1\ otherwise \end{cases}$$

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances. The perceptron outputs a 1 for instances on one side of the hyperplane and -1 for the other side. The equation of the hyperplane is w.x = 0.

When the data can be separated by a hyperplane, it is said to be linearly separable.

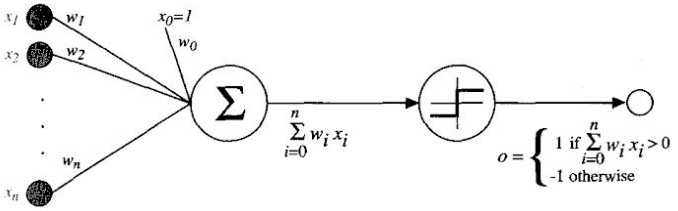


FIGURE 4.2
A perceptron.

The Perceptron Training Rule

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example and modifying it upon misclassification. These weights are modified using the perceptron training rule which is

$$w_i = w_i + \Delta w_i$$

where

$$\Delta w_i = \eta (t - o) x_i$$

Here t is the target output, and o is the predicted output. n represents the learning rate and is a positive constant. This rule can be proven to converge, provided the data is linearly separable and a small value is given to the learning rate.

Gradient Descent and the Delta Rule

The delta rule is used when the data is not linearly separable. The key idea behind the delta rule is to use gradient descent to search the hypothesis space for possible weight vectors. This provides the basis for the backpropagation algorithm.

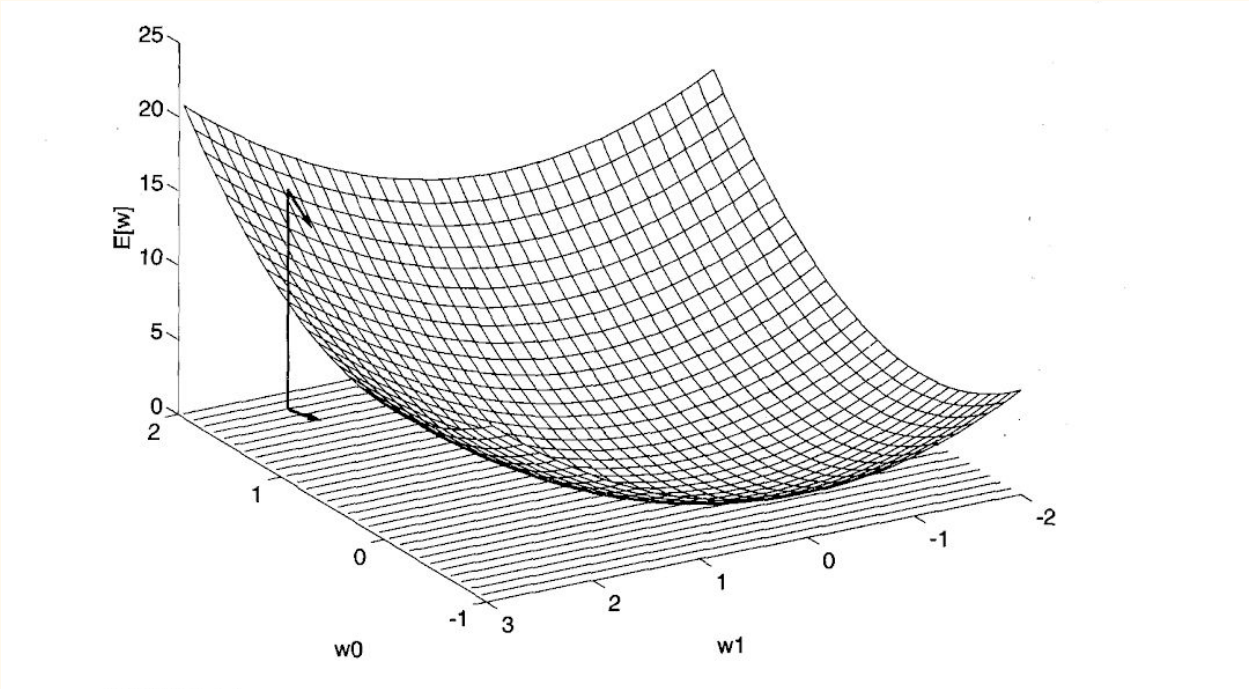
We consider an un-thresholded perceptron, where o(x) = w.x.

The training error is given by

$$E(w) = \frac{1}{2} \sum_{d \in D} \left(t_d - o_d \right)^2$$

Where D is the set of training examples, t is the target output for example d and o is the predicted output for example d.

E is characterised as a function of w because the linear output unit o depends on this weight vector.



Here, w0 and w1 characterise the E(w) vector, which denotes the errors for every weight vector in the hypothesis space. Gradient descent tries to determine the combination of weight vectors for which the lowest point on this curve is obtained.

Derivation of the Gradient Descent Rule

The direction of steepest descent can be found by finding the derivative of E with respect to every weight component. This is called the gradient of E wrt w.

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2} \cdots \frac{\partial E}{\partial w_n} \right]$$

The error vector is interpreted as a vector as it specifies the direction of steepest increase in E. The negative of the vector hence gives the direction of steepest decrease. The gradient descent rule hence becomes

$$w = w + \Delta w$$

where

$$\Delta w = -\eta \nabla E(w)$$

Here, n denotes the step size in the gradient descent search. The negative sign is to move the vector in the direction that decreases w.

The gradient can be simplified as

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\frac{1}{2}\sum_{d \in D}\left(t_d - o_d\right)^2\right)$$

$$= \frac{1}{2}\sum_{d \in D}\frac{\partial}{\partial w_i}\left(t_d - o_d\right)^2$$

$$= \frac{1}{2}\sum_{d \in D} 2\left(t_d - o_d\right)\frac{\partial}{\partial w_i}\left(t_d - o_d\right)$$

$$= \sum_{d \in D}\left(t_d - o_d\right)\frac{\partial}{\partial w_i}\left(t_d - w.x_d\right)$$

$$= \sum_{d \in D}\left(t_d - o_d\right)\left(-x_d\right)$$

Where xid denotes the single input component xi for training example d.

Substituting this in the weight update rule,

$$\Delta w_i = \eta \sum_{d \in D}\left(t_d - o_d\right)x_{id}$$

To summarise, gradient descent is essentially

1. Pick a random initial weight vector
2. Apply the linear unit to all training examples
3. Compute del(w) for each weight
4. Update each weight by del(w)
5. Repeat from step 2

Because the error surface contains only a single global minimum, the algorithm converges to the global minimum, regardless of the nature of the distribution of the training examples, given that the learning rate is small.

If the learning rate is large, the gradient descent might overstep the minimum error. A modification to avoid this would be to decay the learning rate as the steps of descent increase.

Stochastic Gradient Descent

Gradient descent can be applied when

1. The hypothesis space contains continuously parameterized hypotheses
2. The error can be differentiated w.r.t the hypothesis parameters

The key difficulties in applying GD are

1. Converging to a local minimum might be slow
2. If there are multiple local minima, reaching a global minima is not guaranteed.

One common variation of GD to alleviate these is to use Stochastic Gradient Descent (SGD), where SGD approximates gradient descent search by updating weights incrementally, following calculation after each example.

$$\Delta w_i = \eta(t - o) x_i$$

And the error function is similarly defined over each training example.

By making the learning rate sufficiently small, the SGD can approximate true GD closely. The key differences between standard GD and SGD are

1. In standard GD, error is summed over all examples before update. In SGD, update happens after each example.
2. Summing over individual examples in standard GD requires more computation per weight update step. Because it uses true gradients, it uses a larger step size than SGD.
3. In case of multiple local minima, SGD avoids local minima as it has different error vectors to guide its search.

This approach can be used on thresholded units as well, due to the sign of the linear output obtained. However, while this procedure will learn weights that minimize the error, these weights need not minimise the number of misclassified examples.

The main difference between the perceptron update rule and the delta rule is the convergence properties. While the perceptron rule converges after a finite number of iterations, it requires linearly separable data. Whereas, the delta rule needs unbounded time to converge, the distribution of data is voided.

Multilayer Networks

We need to represent neural networks that can model non-linear functions as well, but adding a discontinuous threshold to a perceptron makes it unsuitable for gradient descent. So we use something like a sigmoid unit, that gives a smooth, differentiable threshold.

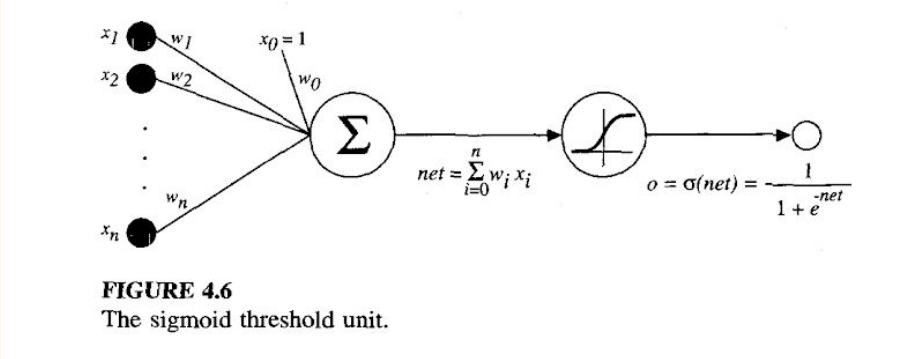


FIGURE 4.6
The sigmoid threshold unit.

More precisely, the output o is computed as

$$o = \sigma(w.x)$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

Because the sigmoid maps a very large input domain to a small range of outputs, it is called a squashing function. It has a very useful property that its derivative is easily expressed as

$$\frac{d\sigma}{dy} = \sigma(y)(1 - \sigma(y))$$

Backpropagation

The backprop algorithm learns weights for a multilayer network. It employs gradient descent to attempt to minimise the squared error between the network output values for these outputs.

Since there are multiple output units, the error E is redefined as

$$E(w) = \frac{1}{2} \sum_{d \in Dk \in outputs} \sum \left(t_{kd} - o_{kd}\right)^2$$

Where outputs is the set of output units in the network and the subscript kd is for the values associated with the kth output unit and training example d.

The idea of backpropagation is to search for a hypothesis space defined by all possible weight values for all units in the network.

The algorithm is given by

BACKPROPAGATION($training\_examples, \eta, n_{in}, n_{out}, n_{hidden}$)

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

η is the learning rate (e.g., .05). n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input from unit i into unit j is denoted x_{ji}, and the weight from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and .05).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in $training\_examples$, Do

Propagate the input forward through the network:

1. Input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network.

Propagate the errors backward through the network:

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{T4.3}$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh}\delta_k \tag{T4.4}$$

4. Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

This algorithm still has a problem of settling at a local minima, but in practice provides good results.

To use SGD for this

1. An index is assigned to each node in the network
2. Let x_{ji} denote the input from node i to node j, and w the weight associated with x.
3. If d denotes the error term associated with unit n, which is just (t-o), the d is the differential of the error by the net weight at that node.

The true gradient of E is obtained by summing all the d*x values before altering the weights.

Adding Momentum

We can make the weight update partially depend on the update on the (n-1)th iteration as

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

This alpha term is called the momentum. This is done to keep the ball rolling in the gradient descent search for global minima.

Learning in Arbitrary Cyclic Networks

The backpropagation algorithm above works for only two layer networks. To generalise this, we change the procedure to calculate the error. For a layer m and unit r,

$$\delta_r = o_r\left(1 - o_r\right) \sum_{s \in layer\ m+1} w_{sr}\delta_s$$

It is equally simple to generalise it for any DAG, regardless of network units arranged uniformly. In the case that they're not, for an internal unit

$$\delta_r = o_r\left(1 - o_r\right) \sum_{s \in Downstream(r)} w_{sr}\delta_s$$

Where Downstream(r) is the set of units immediately downstream from unit r.

Derivation of Backpropagation

The weight update rule is given by

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

where

$$E_d(w) = \frac{1}{2} \sum_{k \in outputs} \left(t_k - o_k\right)^2$$

The main goal is to find a convenient expression for dE/dw.

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$
$$= \frac{\partial E_d}{\partial net_j} x_{ji}$$

Case 1: Output Unit Weights

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

Substituting the values, we get

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 \\ &= \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned}$$

And

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned}$$

Giving the final gradient as

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j)x_{ji}$$

Case 2: For Hidden Weights

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)$$

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$, we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k \, w_{kj}$$

and

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Common heuristics to attempt to alleviate the problem of local minima include

1. Add a momentum term to the update rule
2. Use SGD over standard GD
3. Train multiple networks with different random weights.

The problem with overtraining is, when the network trains for a long time, the weight update rules make the weight plane form very complex rules over the learning surface, causing overfitting in the later iterations.

Approaches to address this

1. Weight decay - add a penalty term to E
2. Cross validations - k fold and simple

Activation Units

The tanh Activation Unit

This unit is similar to the sigmoid unit, but the output ranges from -1 to 1.

$F(x) = (e^{x}-e^{-x})/(e^{x}+e^{-x})$

And $dF/dx = 1 - \{F(x)\}^2$ and this derivative ranges only from 0 to 1.

This is a big advantage as this is more robust to the vanishing gradient problem.

The ReLU Activation Unit

This is called the Rectified Linear Unit, and is defined by

$R(z) = \max(0,z)$.

The Softmax Activation Unit

This is used when multiple outputs are expected from a network. The softmax function outputs probabilities as

$$p(a_i) = \frac{e^{a_i}}{\sum_j e^{a_j}}$$