MODULE - 3

CHAPTER 4

INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known.

Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes.

NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle driving at normal speeds on public highways. The input to the neural network is a 30 x 32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered.

APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones. ~t is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. It is appropriate for problems with the following characteristics:

- ***Instances are represented by many attribute-value pairs.*** The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- ***The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.*** For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering

command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.

- ***The training examples may contain errors.*** ANN learning methods are quite robust to noise in the training data.
- ***Long training times are acceptable.*** Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- ***Fast evaluation of the learned target function may be required.*** Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- ***The ability of humans to understand the learned target function is not important.*** The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

PERCEPTRONS

One type of ANN system is based on a unit called a perceptron, illustrated in Figure 4.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_1 through x_n, the output $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

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

where

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

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$.

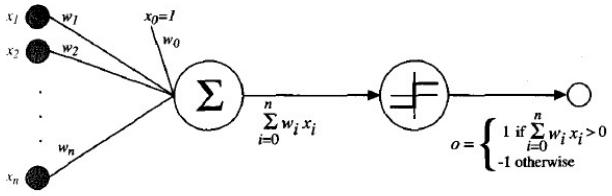

FIGURE 4.2
A perceptron.

Representational Power of Perceptrons

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights w_o = *-.8,* and $w_1 = w_2 = .5$. This perceptron can be made to represent the OR function instead by altering the threshold to w_o = -.3.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND, and NOR. Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function.

The Perceptron Training Rule

Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct output for each of the given training examples. Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example.

This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly.

Weights are modified at each step according to the ***perceptron training rule,*** which revises the weight w_i associated with input ***xi*** according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

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

Here ***t*** is the target output for the current training example, ***o*** is the output generated by the perceptron, and n is a positive constant called the ***learning rate.*** The role of the learning rate is to moderate the degrees to which weights are changed at each step. It is usually set to some small value (e.g., 0.1)

To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, ***(t - o)*** is zero, making Δw_i zero, so that no weights are updated.

Suppose the perceptron outputs a -1, when the target output is +1. To make the perceptron output a +1 instead of -1 in this case, the weights must be altered to increase the value of $\vec{w} \cdot \vec{x}$.

For example, if $x_i = .8$, $n = 0.1$, $t = 1$, and $o = -1$, then the weight update will be $\Delta w_i = n(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$. On the other hand, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be decreased rather than increased.

Gradient Descent and the Delta Rule

A second training rule, called the ***delta rule,*** is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.

The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output ***o*** is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \qquad (4.1)$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is:

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \qquad (4.2)$$

where D is the set of training examples, t_d is the target output for training example *d,* and o_d is the output of the linear unit for training example d. By this definition, $E(\vec{w})$ is simply half the squared difference between the target output t_d and the hear unit output o_d, summed over all training examples.

STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever

(l) The hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and

(2) The error can be differentiated with respect to these hypothesis parameters.

The key practical difficulties in applying gradient descent are

(1) Converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and

(2) If there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$*, where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \quad \text{(T4.1)}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \quad \text{(T4.2)}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

One common variation on gradient descent intended to alleviate these difficulties is called ***incremental gradient descent,*** or alternatively ***stochastic gradient descent.*** Whereas the gradient descent training rule presented in Equation ***(4.7)*** computes weight updates after summing over ***a22*** the training examples in D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error for ***each*** individual example. The modified training rule is like the training rule given by Equation ***(4.7)*** except that ***as*** we iterate through each training example we update the weight according to

The key differences between standard gradient descent and stochastic gradient descent are:

$$\Delta w_i = \eta(t - o)\ x_i \tag{4.10}$$

where t, o, and x_i are the target value, unit output, and ith input for the training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation (T4.2) is simply deleted and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t-o)\ x_i$. One way to view this stochastic gradient descent is to consider a distinct error function $E_d(\vec{w})$ defined for each individual training example d as follows

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{4.11}$$

where t_d and o_d are the target value and the unit output value for training example d. Stochastic gradient descent iterates over the training examples d in D, at each iteration altering the weights according to the gradient with respect to $E_d(\vec{w})$. The sequence of these weight updates, when iterated over all training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E(\vec{w})$.

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to $E(\vec{w})$, stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various $\nabla E_d(\vec{w})$ rather than $\nabla E(\vec{w})$ to guide its search.

MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

A typical multilayer network and decision surface is depicted in Figure **4.5.** Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of "h-d" (i.e., "hid," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily visualize the decision surface over the two-dimensional instance space.

A Differentiable Threshold Unit

What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the sigmoid unit-a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a continuous function of its input. More precisely, the sigmoid unit computes its output ***o*** as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{4.12}$$

σ is often called the sigmoid function or, alternatively, the logistic function. Note its output ranges between 0 and 1, increasing monotonically with its input (see the threshold function plot in Figure 4.6.). Because it maps a very large input domain to a small range of outputs, it is often referred to as the *squashing function* of the unit.

The BACKPROPAGATION Algorithm

The BACKPROPAGATaIlOgNor ithm learns the weights for a multilayer network, given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \qquad (4.13)$$

where ***outputs*** is the set of output units in the network, and t_{kd} and O_{kd} are the target and output values associated with the kth output unit and training example d.

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by **all** possible weight values for all the units in the network.

The BACKPROPAGATaIlgOoNri thm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION.

The notation used here is as follows:

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

Each training example is a pair of the form $(\vec{x}, \vec{t})$, *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 $(\vec{x}, \vec{t})$ 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

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \qquad (T4.3)$$

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

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

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \qquad (T4.5)$$

TABLE 4.2

- An index (e.g., an integer) is assigned to each node in the network, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As we shall see later, $\delta_n = -\frac{\partial E}{\partial net_n}$.

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

ADDING MOMENTUM

Because **BACKPROPAGATisIO** suNc h a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the nth iteration depend partially on the update that occurred during the (n - 1)th iteration, as follows:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \quad (4.18)$$

Here $\Delta w_{ji}(n)$ is the weight update performed during the nth iteration through the main loop of the algorithm, and $0 \leq \alpha < 1$ is a constant called the *momentum*.

Notice the first term on the right of this equation is just the weight-update rule of Equation (T4.5) in the **BACKPROPAGATION Algorithm**.

The second term on the right is new and is called the momentum term. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of α is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next.

This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum.

It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.