MATH 462 LECTURE NOTES. NEURAL NETWORKS

ADAM M. OBERMAN

1. Resources for learning neural networks

- Read [Mit97, Chapter 4], in particular, parts of section 4.4, 4.5, and 4.6. This is a long chapter, but the length makes it easier to understand, compared to more concise books.
- For the full code example on multilayer perceptrons, see code provided based on `https://towardsdatascience.com/how-neural-networks-solve-the-xor-problem-59763136bdd7`
- Use the $\vec{x}$ notation just for introduction then transition to x notation, implicitly vector or scalar.

2. Neural networks

This material adapted from [Mit97, Chapter 4]

Remark 2.1 (Notation). For now, using the $o(x)$ notation from Mitchell. TODO: make this consistent, by changing to $c(h(x)) = \mathrm{sgn}(h(x))$ notation.

2.1. **About Neural Networks.** Neural networks (NNs) provide a general, practical method for learning real-valued, as well as discrete-valued, and vector-valued functions from examples. Algorithms use stochastic gradient descent to tune network parameters to fit a large training dataset.

Remark 2.2. Some of the simpler examples (eg. decision trees, naive bayes), implemented a formula / algorithm directly to obtain the model.

For neural networks, need to "train" (optimize) the parameters, iteratively. Need a stopping condition.

Overview (details will be filled in progressively).

- In many other learning methods, the data must consist of a small number of *semantically meaningful features*.
- However, neural networks can learn from high dimensional data, for example the pixel values in images.
- A two layer neural network can be thought of as (i) learning features from data (ii) then classifying using the features.
- The training examples may contain errors. NN learning methods are quite robust to noise in the training data.
- Long training times are acceptable. Network training algorithms typically require longer training times. Pass over the entire dataset many times (epochs).
- Although DNN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.
- 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.

Date: August 2023.

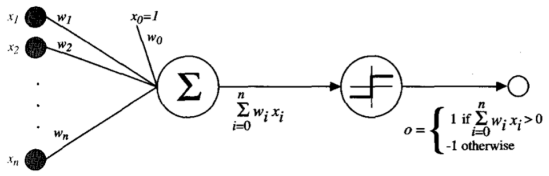


FIGURE 1. Perceptron. Inputs $x_1, \ldots, x_n$ multiplied by weights $w_1, \ldots, w_n$, then passed through a nonlinearity, in this case, the sgn function.

3. Perceptrons

First consider the case of discrete (binary) input, and discrete (binary) output.

One type of NN system is based on a unit called a perceptron, illustrated in Figure 1.

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. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + \cdots + w_n x_n$ must surpass in order for the perceptron to output a 1 .

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

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

where

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

Example 3.1. So if $\vec{x}' = (x_1, x_2, x_3)$ and $\vec{w} = (w_0, \ldots, w_x) = (2.5, .5, -1, 0.2)$, then given input $(3, 0, 1)$ we write $\vec{x} = (1, x') = (1, 3, 0, 1)$ and

$$o(\vec{x}) = \mathrm{sgn}(\vec{w} \cdot \vec{x}) = (2.5, .5, -1, 0.2) \cdot (1, 3, 0, 1) = 2.5 + 1.5 + 0 + 0.2 = 4.2$$

This leads us to the following definition. We remove the vector notation, since it is implied by the spaces.

Definition 3.2 (The perceptron hypothesis space)**.** The space of candidate hypotheses considered in perceptron learning is represented parametrically by the set of all possible real-valued weight vectors.

$$\mathcal{H} = \{h : \mathbb{R}^n \to \mathbb{R} \mid h(x, w) = w \cdot x + w_0, \ w \in \mathbb{R}^n, w \in \mathbb{R}\}$$

with output

$$c(x, w) = \mathrm{sgn}(h(x, w))$$

3.1. **Representational Power of Perceptrons for boolean functions.** We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 2. The equation for this decision hyperplane is $\vec{w} \cdot \vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.

A single perceptron can be used to represent many boolean functions.

Example 3.3. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron

$$h(x, w) = w \cdot x + w_0,$$

to implement the AND function is to set the weights $w_0 = -.8$, and $w_1 = w_2 = .5$.

$$h_{AND}(x, w) = w \cdot x + w_0, \quad w_0 = -0.8,\ w = (0.5, 0.5)$$

This perceptron can be made to represent the OR function instead by altering the threshold to $w_0 = -.3$.

$$h_{OR}(x, w) = w \cdot x + w_0, \quad w_0 = -0.3,\ w = (0.5, 0.5)$$

Remark 3.4. In fact, AND and OR can be viewed as special cases of m-of- n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to $m = 1$ and the AND function to $m = n$. Any m-of- n function is easily represented using a perceptron by setting all input weights to the same value (e.g., $1/n$) and then setting the threshold w_0 accordingly.

Perceptrons can represent all of the primitive boolean functions

- AND,
- OR,
- NAND ($\neg$ AND), and
- NOR ($\neg$ OR).

Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_1 \neq x_2$. Note the set of linearly nonseparable training examples shown in Figure 2 corresponds to this XOR function. The ability of perceptrons to represent AND, OR, NAND, and NOR is important because every boolean function can be represented by some network of interconnected units based on these primitives.

However, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight.

Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

- Link to Collab Perceptrons Code

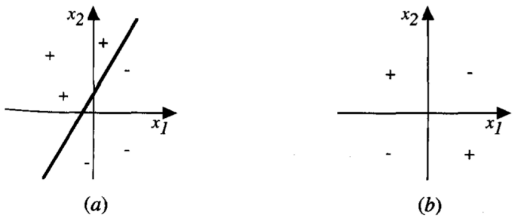


FIGURE 2. The decision surface represented by a two-input perceptron. (a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_l and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

4. The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. 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.

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 x_i according to the rule

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

where

$$\Delta w_i = \alpha(y(x) - c(x))x_i$$

Example 4.1. Consider learning the function $y(x) = \mathrm{sgn}(x)$ which can be represented as a perceptron $c(h(x, w))$ with $h(x, w) = 1x + 0$. Starting with random initialization, $w_1 = 0.2, w_0 = -0.4$. Let the dataset be $S = \{(-1, -1), (1, 1)\}$. Take $\alpha = 0.25$

When $x = -1$, $h(x, w) = -.2 - .4 = -0.6$ and $c(x) = -1$, so $\Delta w_i = 0$

When $x = 1$, we have $h(x, w) = -0.2, c(x) = -1$. So $\Delta w_i = \alpha 2 x_i = 2\alpha$. This will increase the weights. So now $\Delta w_0 = \Delta w_1 = .5$

Thus $w_1 = 0.7, w_0 = 0.1$. Now $c(1) = 1$ and $c(-1) = -1$ as desired.

This method can be shown to converge when the training exmaples are linearly separable. But it may not converge in general.

4.1. **Perceptron Loss.** This method goes as follows.

- First, we consider a loss based on $h(x, w)$, without thresholding
- We consider the loss, $L(w) = \frac{1}{m} \sum_{i=1}^{m} \ell(h(x_i, w), y_i)$ averaged over the dataset
- We adjust the weight vector by using gradient descent.
- First we can simply use a regression loss, treating y as a number, $\ell_2(h, y) = (h - y)^2$
- Later we can use a classification loss.

- Note, if $h(x, w)$ is close enough to y, then thresholding $c(h) = \text{sgn}(h)$ should also be correct.
- This method should converge to something, even if the data is not linearly separable.

TODO: 2d example, linearly separable with a couple errors

Definition 4.2 (Perceptron training loss)**.** Given the dataset $S = \{(x_1, y_1), \dots (x_m, y_m)\}$ with binary vector data $x \in [-1, 1]^n$ and binary y. Consider the perceptron hypothesis class defined above. Define the training loss

$$L(w) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2}(h(x_i, w) - y_i)^2$$

4.2. **Derivation of steepend descent.** How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of L with respect to each component of the vector $\vec{w}$. This vector derivative is called the gradient of L with respect to $\vec{w}$, written ∇L.

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

Notice $\nabla L(\vec{w})$ is itself a vector, whose components are the partial derivatives of E with respect to each of the w_i. When interpreted as a vector in weight space, the gradient specifies the direction that produces the steepest increase in E. The negative of this vector therefore gives the direction of steepest decrease.

TODO Show figure with level sets: For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

Since the gradient specifies the direction of steepest increase of E, the training rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$
$$\Delta\vec{w} = -\eta \nabla E(\vec{w})$$

Here η is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that decreases E. This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

which makes it clear that steepest descent is achieved by altering each component w_i of $\vec{w}$ in proportion to $\frac{\partial E}{\partial w_i}$.

To construct a practical algorithm for iteratively updating weights, we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of $\frac{\partial E}{\partial w_i}$ derivatives that form the gradient can be obtained by differentiating L.

First notice that, for the inner term

$$\frac{\partial}{\partial w_i}(h(x, w) - y)^2/2 = (h(x, w) - y)\frac{\partial}{\partial w_i} h(x, w)$$

and

$$\frac{\partial}{\partial w_i} h(x, w) = \frac{\partial}{\partial w_i} \sum_j w_j x_j = x_i$$

(keeping track of the fact that $x_0 = 1$ as per our convention).

Thus

$$\begin{aligned}\frac{\partial L}{\partial w_i} &= \frac{\partial}{\partial w_i}\sum_j (h(x_j, w) - y_j)^2/2 \\ &= \sum_j \frac{\partial}{\partial w_i}(h(x_j, w) - y_j)^2/2 \\ &= \sum_j (h(x_j, w) - y)(x_j)_i\end{aligned}$$

where $(x_j)_i$ denotes the single input component x_i for training example j.

We now have an equation that gives $\frac{\partial L}{\partial w_i}$ in terms of the inputs x_{ij}, outputs $h(x_j, w)$, and target values y_j associated with the training examples.

Substituting yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{j \in S} (h(x_j, w) - y)(x_j)_i$$

To summarize, the gradient descent algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, then compute Δw_i for each weight according to Equation (4.7). Update each weight w_i by adding Δw_i, then repeat this process.

Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used. If η is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of η as the number of gradient descent steps grows.

5. Vector calculus review

For more vector calc review, more details to come

Now consider a function of d variables, $L : \mathbb{R}^d \to \mathbb{R}$. The gradient of the function is a vector defined at each w,

$$g(w) = \nabla L(w) = [g_1(w), \ldots g_d(w)]^T$$

where each component is partial derivative

$$g_j(w) = \frac{\partial}{\partial w_j} L(w)$$

- The gradient vector $g(w) = \nabla L(w)$ points in the direction of greatest increase of the function L at w.
- A critical point w is a point where $g(w) = 0$.
- As in the one variable case, every local minimum is a critical point. A critical point can be a local minimum, local maximum, or saddle point.
- As in the one variable case, there is a condition for a critical point to be a local minimum: the Hessian matrix $H(w)$ is positive-definite. Here $H(w)_{ij} = \frac{\partial^2}{\partial_i \partial_j} L$. (This condition can be difficult to check).
- As in the one variable case, if the function is convex, then every critical point *global* minimum.

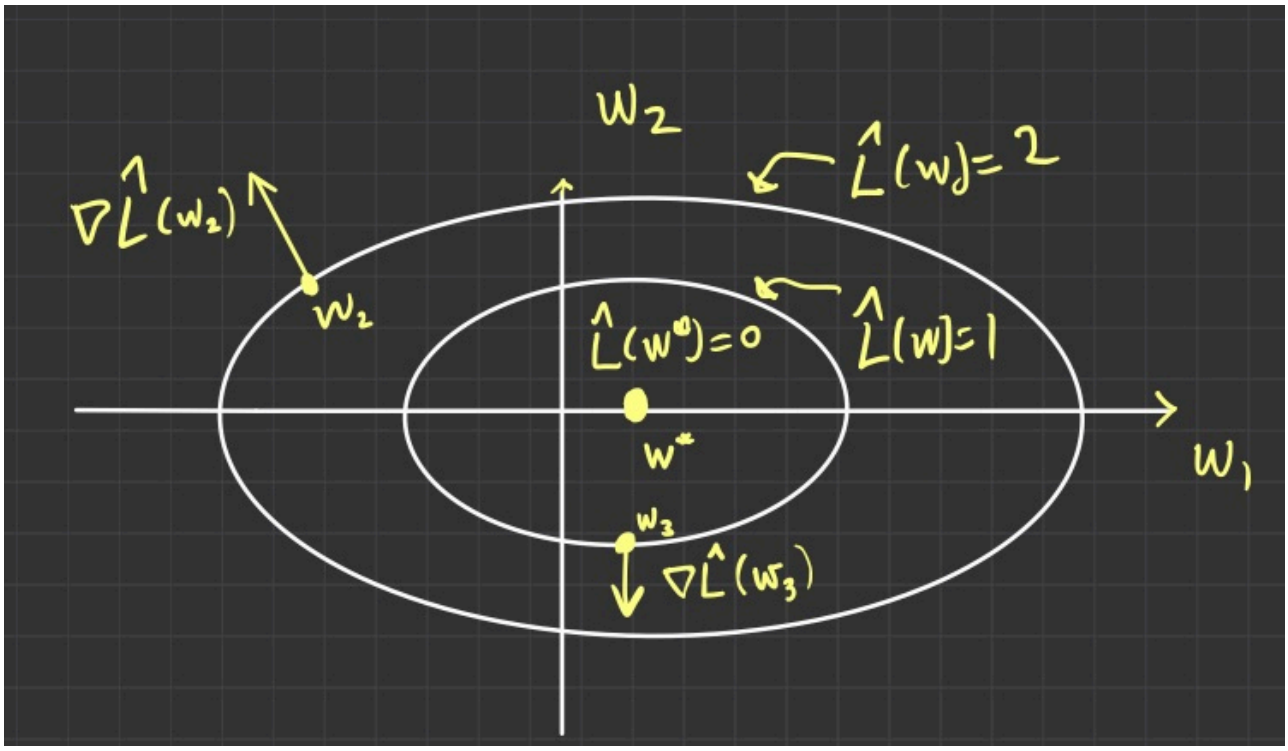


FIGURE 3. Illustration of the gradient of the loss

6. Gradient Descent and SGD

For more details: Refer to separate notes on GD and SGD
Short version for intuition:
Starting from the definition of the empirical loss.

$$L(w) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(x_i, w), y_i) \tag{EL}$$

Then define the gradient vector using the following notation

$$g(w) = \nabla_w L(w) = \frac{1}{m}\sum_{i=1}^{m} \partial_h \ell(h_w(x_i), y_i)\nabla_w h_w(x_i) \qquad \text{by the chain rule} \tag{G}$$

Note, we still need to compute $\nabla_w h_w(x_i)$, which may use chain rule more times.

Remark 6.1. Note, by the chain rule, above, the gradient of the loss is the weighted sum of the model gradients, with weights given by the loss derivative. So points where the loss is larger have a larger (in magnitude) weight.

Then the gradient descent algorithm is given by the following.

Definition 6.2. Gradient descent with learning rate $\alpha > 0$ for (EL) is given by

$$w_{t+1} = w_t - \alpha g(w_t),$$

where $g(w_t) = \nabla_w L(w_t)$ given by (G)

TO BE EDITED

7. Backpropagation: Chain rule

We are combining the ideas of

- Gradient descent to decrease the loss $w^{n+1} = w^n - \alpha \nabla_w L(w)$
- backpropagation (the chain rule) to find the gradient $g = \nabla_w h(x, w)$ of the model.

8. Multilayer networks and backpropagation algorithm

This section discusses how to learn such multilayer networks using a gradient descent algorithm similar to that discussed in the previous section.

8.1. **A Differentiable Threshold Unit.** What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the previous section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. 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.

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

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

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 sigmoid function has the useful property that its derivative is easily expressed in terms of its output

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

As we shall see, the gradient descent learning rule makes use of this derivative. Other differentiable functions with easily calculated derivatives are sometimes used in place of σ.

Example 8.1 (temperature scaling). For example, the term e^{-y} in the sigmoid function definition is sometimes replaced by $e^{-t \cdot y}$ where t is some positive constant that determines the steepness of the threshold.

Example 8.2 (other nonlinearities). The function $\tanh$ is also sometimes used in place of the sigmoid function. Also ReLU (rectified linear unit)

$$\sigma_{ReLU}(t) = \max(t, 0)$$

which is piecewise linear.

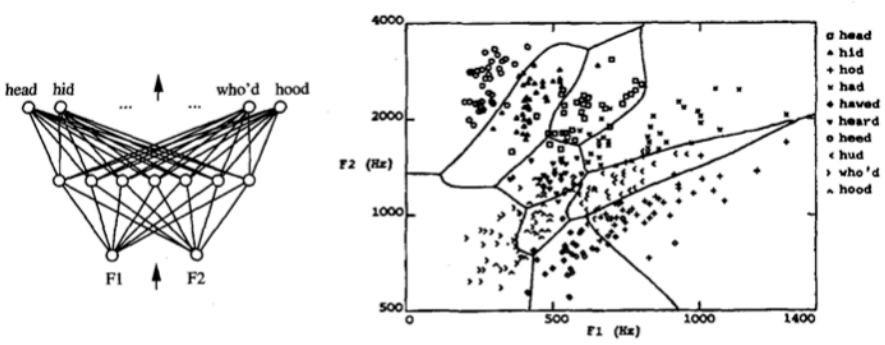


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

FIGURE 4. dd

9. (Skip for now) Representational Power of Feedforward Networks

As noted above, single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface is depicted in Figure 4.

As shown in the figure, it is possible for the multilayer network to represent highly nonlinear decision surfaces that are much more expressive than the linear decision surfaces of single units.

What set of functions can be represented by feedforward networks? Of course the answer depends on the width and depth of the networks. Although much is still unknown about which function classes can be described by which types of networks, three quite general results are known: - Boolean functions. Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of network inputs. To see how this can be done, consider the following general scheme for representing an arbitrary boolean function: For each possible input vector, create a distinct hidden unit and set its weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output unit as an OR gate that activates just for the desired input patterns. - Continuous functions. Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units (Cybenko 1989; Hornik et al. 1989). The theorem in this case applies to networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer. The number of hidden units required depends on the function to be approximated. - Arbitrary functions. Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko 1988). Again, the output layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general. The proof of this involves showing that any function can be approximated by a linear combination of many localized functions that have value 0 everywhere except for some small region, and then showing that two layers of sigmoid units are sufficient to produce good local approximations.

These results show that limited depth feedforward networks provide a very expressive hypothesis space for BACKPROPAGATION. However, it is important to keep in mind that the network weight vectors reachable by gradient descent from the initial weight values may not include all possible weight vectors. Hertz et al. (1991) provide a more detailed discussion of the above results.

Remark 9.1. Existence results just say can fit the data. But do not give a method to fit. Backpropagation algorithm gives a method to fit data. But does not show that will fit the function on unseen data (generalization).

9.1. **Hidden Layer Representations.** One intriguing property of BACKPROPAGATION is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error E. This can lead BACKPROPAGATION to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

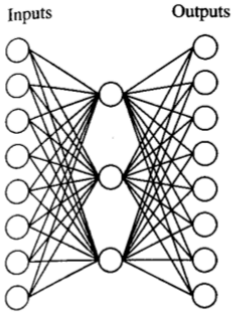
Consider, for example, the network shown in Figure 4.7. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.

Consider training the network shown in Figure 4.7 to learn the simple target function $f(\vec{x}) = \vec{x}$, where $\vec{x}$ is a vector containing seven 0 's and a single 1 . The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from all eight input units must be captured by the three learned hidden units.

When BACKPROPAGATION is applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. What hidden layer representation is created by the gradient descent BACKPROPAGATION algorithm? By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., $000, 001, 010, \ldots, 111$). The exact values of the hidden units for one typical run of BACKPROPAGATION are shown in Figure 5.

This ability of multilayer networks to automatically discover useful representations at the hidden layers is a key feature of ANN learning. In contrast to learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer. Of course these invented features must still be computable as sigmoid unit functions of the provided network inputs. Note when more layers of units are used in the network, more complex features can be invented. Another example of hidden layer features is provided in the face recognition application discussed in Section 4.7.

9.2. **Classification Loss.** Minimizing the cross entropy of the network with respect to the target values. Consider learning a probabilistic function, such as predicting whether a loan applicant will pay back a loan based on attributes such as the applicant's age and bank balance. Although the training examples exhibit only boolean target values (either a 1 or 0 , depending on whether this applicant paid back the loan), the underlying target function might be best modeled by outputting the probability that the given applicant will repay the loan, rather than attempting to output the



Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

FIGURE 4.7
Learned Hidden Layer Representation. This $8 \times 3 \times 8$ network was trained to learn the identity function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the encoded values are rounded to zero or one, the result is the standard binary encoding for eight distinct values.

FIGURE 5. Hidden Layers

actual 1 and 0 value for each input instance. Given such situations in which we wish for the network to output probability estimates, it can be shown that the best (i.e., maximum likelihood) probability estimates are given by the network that minimizes the cross entropy, defined as

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

Here o_d is the probability estimate output by the network for training example d, and t_d is the 1 or 0 target value for training example d. Chapter 6 discusses when and why the most probable network hypothesis is the one that minimizes this cross entropy and derives the corresponding gradient descent weight-tuning rule for sigmoid units. That chapter also describes other conditions under which the most probable hypothesis is the one that minimizes the sum of squared errors.

References

[Mit97] Tom M Mitchell. *Machine Learning*. McGraw Hill, 1997.