Introduction to feedforward neural networks

1. Problem statement and historical context

A. Learning framework

Figure 1 below illustrates the basic framework that we will see in *artificial neural network* learning. We assume that we want to learn a classification task G with n inputs and m outputs, where,

$$\mathbf{y} = G(\mathbf{x}), \tag{1}$$

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^T \text{ and } \mathbf{y} = \begin{bmatrix} y_1 & y_2 & \dots & y_m \end{bmatrix}^T. \tag{2}$$

In order to do this modeling, let us assume a model Γ with trainable parameter vector $\mathbf{w}$, such that,

$$\mathbf{z} = \Gamma(\mathbf{x}, \mathbf{w}) \tag{3}$$

where,

$$\mathbf{z} = \begin{bmatrix} z_1 & z_2 & \dots & z_m \end{bmatrix}^T. \tag{4}$$

Now, we want to minimize the error between the desired outputs $\mathbf{y}$ and the model outputs $\mathbf{z}$ for all possible inputs $\mathbf{x}$. That is, we want to find the parameter vector $\mathbf{w}^*$ so that,

$$E(\mathbf{w}^*) \le E(\mathbf{w}), \ \forall \mathbf{w}, \tag{5}$$

where $E(\mathbf{w})$ denotes the error between G and Γ for model parameter vector $\mathbf{w}$. Ideally, $E(\mathbf{w})$ is given by,

$$E(\mathbf{w}) = \int_{\mathbf{x}} \|\mathbf{y} - \mathbf{z}\|^2 p(\mathbf{x}) d\mathbf{x} \tag{6}$$

where $p(\mathbf{x})$ denotes the probability density function over the input space $\mathbf{x}$. Note that $E(\mathbf{w})$ in equation (6) is dependent on $\mathbf{w}$ through $\mathbf{z}$ [see equation (3)]. Now, in general, we cannot compute equation (6) directly; therefore, we typically compute $E(\mathbf{w})$ for a training data set of input/output data,

$$\{(\mathbf{x}_i, \mathbf{y}_i)\}, \ i \in \{1, 2, \dots, p\}, \tag{7}$$

where $\mathbf{x}_i$ is the n-dimensional input vector,

$$\mathbf{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \dots & x_{in} \end{bmatrix}^T \tag{8}$$

corresponding to the ith training pattern, and $\mathbf{y}_i$ is the m-dimensional output vector,

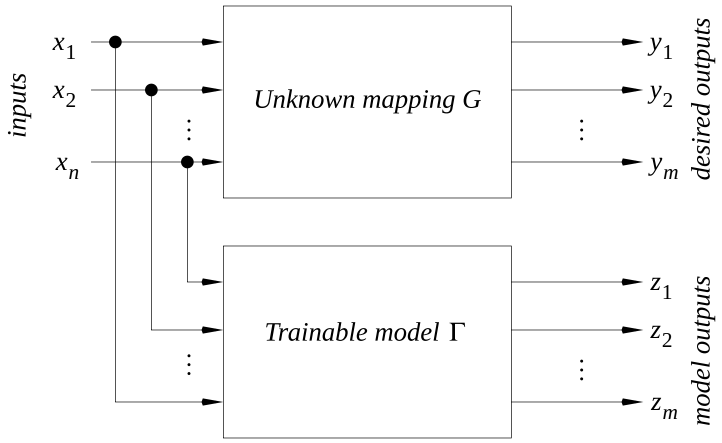


Figure 1

$$\mathbf{y}_i = \left[y_{i1}\ y_{i2}\ \dots\ y_{im}\right]^T \tag{9}$$

corresponding to the i th training pattern, $i \in \{1, 2, \dots, p\}$. For (7), we can define the *computable* error function $E(\mathbf{w})$,

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{p}\|\mathbf{y}_i - \mathbf{z}_i\|^2 = \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{m}(y_{ij} - z_{ij})^2 \tag{10}$$

where,

$$\mathbf{z}_i \equiv \Gamma(\mathbf{x}_i, \mathbf{w}). \tag{11}$$

If the data set is well distributed over possible inputs, equation (10) gives a good approximation of the error measure in (6).

As we shall see shortly, artificial neural networks are one type of parametric model Γ for which we can minimize the error measure in equation (10) over a given training data set. Simply put, artificial neural networks are nonlinear function approximators, with adjustable (i.e. trainable) parameters $\mathbf{w}$, that allow us to model functional mappings, including classification tasks, between inputs and outputs.

B. Biological inspiration

So why are artificial neural networks called artificial neural networks? These models are referred to as neural networks because their structure and function is loosely based on *biological* neural networks, such as the human brain. Our brains consist of basic cells, called *neurons*, connected together in massive and parallel fashion. An individual neuron receives electrical signals from *dentrites*, connected from other neurons, and passes on electrical signals through the neuron’s output, the *axon*, as depicted (crudely) in Figure 2 below.

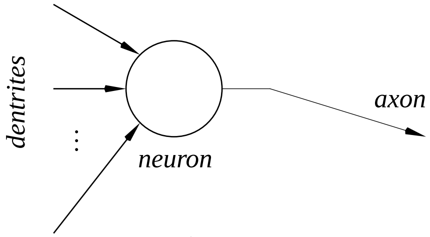


Figure 2

A neuron’s transfer function can be roughly approximated by a threshold function as illustrated in Figure 3 below. In other words, a neuron’s axon fires if the net stimulus from all the incoming dentrites is above some threshold. Learning in our brain occurs through adjustment of the strength of connection between neurons (at the axon-dentrite junction). [Note, this description is a gross simplification of what really goes on in a brain; nevertheless, this brief summary is adequate for our purposes.]

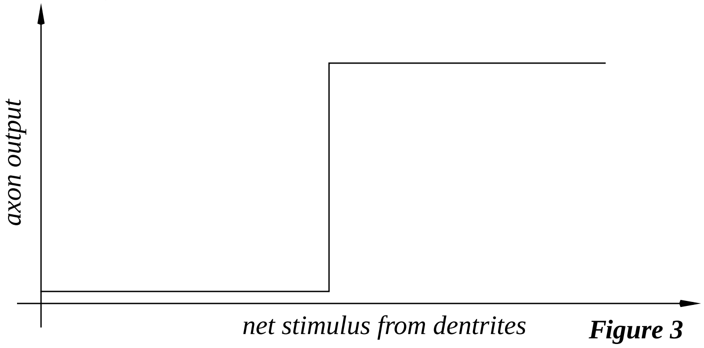


Figure 3

Now, artificial neural networks attempt to crudely emulate biological neural networks in the following important ways:

1. Simple basic units are the building blocks of artificial neural networks. It is important to note that artificial “neurons” are much, much simpler than their biological counterparts.
2. Individual units are connected massively and in parallel.
3. Individual units have threshold-type activation functions.
4. Learning in artificial neural networks occurs by adjusting the strength of connection between individual units. These parameters are known as the *weights* of the neural network.

We point out that artificial neural networks are much, much, much simpler than complex biological neural networks (like the human brain). According to the Encyclopedia Britannica, the average human brain consists of approximately 10^{10} individual neurons with approximately 10^{12} connections. Even very complicated artificial neural networks typically do not have more than 10^4 to 10^5 connections between, at most, 10^4 individual basic units.

As of September, 2001, an INSPEC database search generated over 45,000 hits with the keyword “neural network.” Considering that neural network research did not really take off until 1986, with the publication of the backpropagation training algorithm, we see that research in artificial neural networks has exploded over the past 15 years and is still quite active today. We will try to cover some of the highlights of that research. First, however, we will formalize our discussion above, clearly defining what a neural network is, and how we can train artificial neural networks to model input/output data; that is, how learning occurs in artificial neural networks.

2. What makes a neural network a neural network?

A. Basic building blocks of neural networks

Figure 4 below illustrates the basic building block of artificial neural networks; the *unit*’s basic function is intended to roughly approximate the behavior of biological neurons, although biological neurons tend to be orders-of-magnitude more complex than these artificial units.

In Figure 4,

$$\underset{\sim}{\phi} \equiv \begin{bmatrix}\phi_0 & \phi_1 & \ldots & \phi_q\end{bmatrix}^T \tag{12}$$

represents a vector of scalar inputs to the unit, where the ϕ_i variables are either neural network inputs x_j, or the outputs from previous units, including the *bias unit* ϕ_0, which is fixed at a constant value (typically 1). Also,

$$\mathbf{w} \equiv \begin{bmatrix}\omega_0 & \omega_1 & \ldots & \omega_q\end{bmatrix}^T \tag{13}$$

represents the input *weights* of the unit, indicating the strength of connection from the unit inputs ϕ_i; as we shall see later, these are the trainable parameters of the neural network. Finally, γ represents the (typically nonlinear) *activation function* of the unit, and ψ represents the scalar output of the unit where,

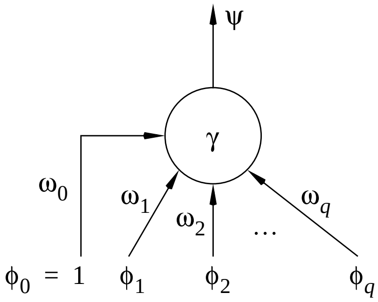


Figure 4

$$\psi \equiv \gamma(\mathbf{w} \cdot \underset{\sim}{\phi}) = \gamma\left(\sum_{i=0}^{q} \omega_i \phi_i\right) \tag{14}$$

Thus, a unit in an artificial neural network sums up its total input and passes that sum through some (in general) nonlinear activation function.

B. Perceptrons

A simple *perceptron* is the simplest possible neural network, consisting of only a single unit. As shown in Figure 6, the output unit's activation function is the *threshold* function,

$$\gamma_t(u) = \begin{cases} 1 & u \geq \theta \\ 0 & u < \theta \end{cases} \tag{15}$$

which we plot in Figure 5. The output z of the perceptron is thus given by,

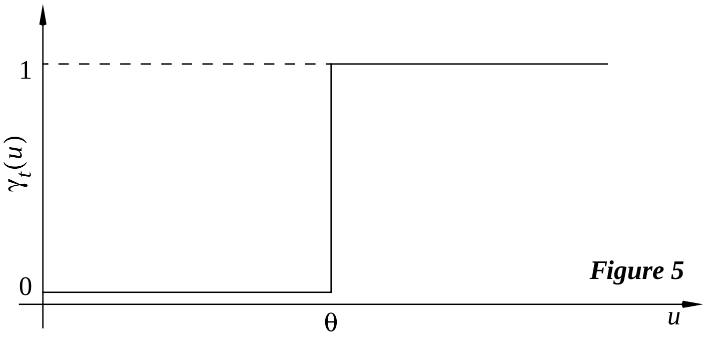


Figure 5

$$z = \begin{cases} 1 & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases} \tag{16}$$

where,

$$\mathbf{x} = \begin{bmatrix} 1 \ x_1 \ \dots \ x_n \end{bmatrix}^T \text{ and,} \tag{17}$$

$$\mathbf{w} = \begin{bmatrix} \omega_0 \ \omega_1 \ \dots \ \omega_n \end{bmatrix}^T \tag{18}$$

A perceptron like that pictured in Figure 6 is capable of learning a certain set of decision boundaries, specifically those that are *linearly separable*. The property of linear separability is best understood geometrically. Consider the two, two-input Boolean functions depicted in Figure 7 — namely, the OR and the XOR functions (filled circles represent 0, while hollow circles represent 1). The OR function *can* be represented (and learned) by a two-input perceptron, because a straight line can completely separate the two classes. In other

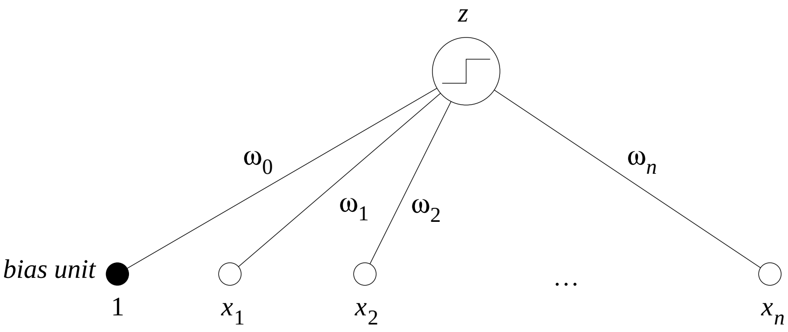


Figure 6

$$\omega_0 = -0.5 \quad \omega_1 = 1 \quad \omega_2 = 1$$

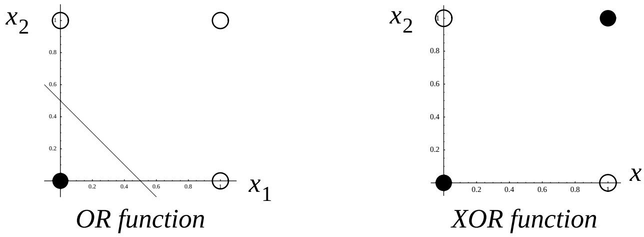


Figure 7

words, the two classes are linearly separable. On the other hand, the XOR function *cannot* be represented (or learned) by a two-input perceptron because a straight line *cannot* completely separate one class from the other. For three inputs and above, whether or not a Boolean function is representable by a simple perceptron depends on whether or not a plane (or a hyperplane) can completely separate the two classes.

The algorithm for learning a linearly separable Boolean function is known as the *perceptron learning rule*, which is guaranteed to converge for linearly separable functions. Since this training algorithm does not generalize to more complicated neural networks, discussed below, we refer the interested reader to [2] for further details.

C. Activation function

In biological neurons, the activation function can be roughly approximated as a threshold function [equation (15)], as in the case of the simple perceptron above. In artificial neural networks that are more complicated than simple perceptrons, we typically emulate this biological behavior through nonlinear functions that are similar to the threshold function, but are, at the same time, continuous and differentiable. [As we will see later, differentiability is an important and necessary property for training neural networks more complicated than simple perceptrons.] Thus, two common activation functions used in artificial neural networks are the *sigmoid* function,

$$\gamma(u) = \frac{1}{1+e^{-u}} \tag{19}$$

or the *hyperbolic tangent* function,

$$\gamma(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}} \tag{20}$$

These two functions are plotted in Figure 8 below. Note that the two functions closely resemble the threshold function in Figure 5 and differ from each other only in their respective output ranges; the sigmoid function's range is $[0, 1]$, while the hyperbolic tangent function's range is $[-1, 1]$. In some cases, when a system output does not have a predefined range, its corresponding output unit may use a linear activation function,

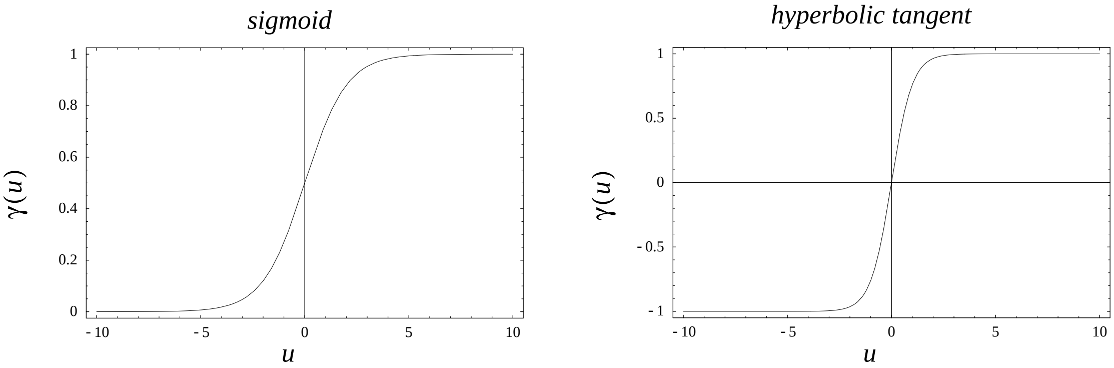


Figure 8

$$\gamma(u) = u \tag{21}$$

From Figure 8, the role of the bias unit ϕ_0 should now be a little clearer; its role is essentially equivalent to the threshold parameter θ in Figure 5, allowing the unit output ψ to be shifted along the horizontal axis.

D. Neural network architectures

Figures 9 and 10 show typical arrangements of units in artificial neural networks. In both figures, all connections are feedforward and layered; such neural networks are commonly referred to as *feedforward multilayer perceptrons (MLPs)*. Note that units that are not part of either the input or output layer of the neural network are referred to as *hidden units*, in part since their output activations cannot be directly observed from the outputs of the neural network. Note also that each unit in the neural network receives as input a connection from the bias unit.

The neural networks in Figures 9 and 10 are typical of many neural networks in use today in that they arrange the hidden units in *layers*, fully connected between consecutive layers. For example, ALVINN, a neural network that learned how to autonomously steer an automobile on real roads by mapping coarse camera images of the road ahead to corresponding steering directions [3], used a single-hidden-layer architecture to achieve its goal (see Figure 11 below).

MLPs are, however, not the only appropriate or allowable neural network architecture. For example, it is frequently advantageous to have direct input-output connections; such connections, which jump hidden-unit layers, are sometimes referred to as *shortcut* connections. Furthermore, hidden units do not necessarily have to be arranged in layers; later in the course, we will, for example, study the cascade learning architecture, an adaptive architecture that arranges hidden units in a particular, non-layered manner. We will say more about neural network architectures later within the context of specific, successful neural network applications.

Finally, we point out that there also exist neural networks that allow *cyclic* connections; that is, connections from any unit in the neural network to any other unit, including self-connections. These *recurrent* neural networks present additional challenges and will be studied later in the course; for now, however, we will confine our studies to feedforward (acyclic) neural networks only.

E. Simple example

Consider the simple, single-input, single-output neural network shown in Figure 12 below. Assuming sigmoidal hidden-unit and linear output-unit activation functions (equations (19) and (21), respectively), what values of the weights $\{\omega_1, \omega_2, \ldots, \omega_7\}$ will approximate the function $f(x)$ in Figure 12?

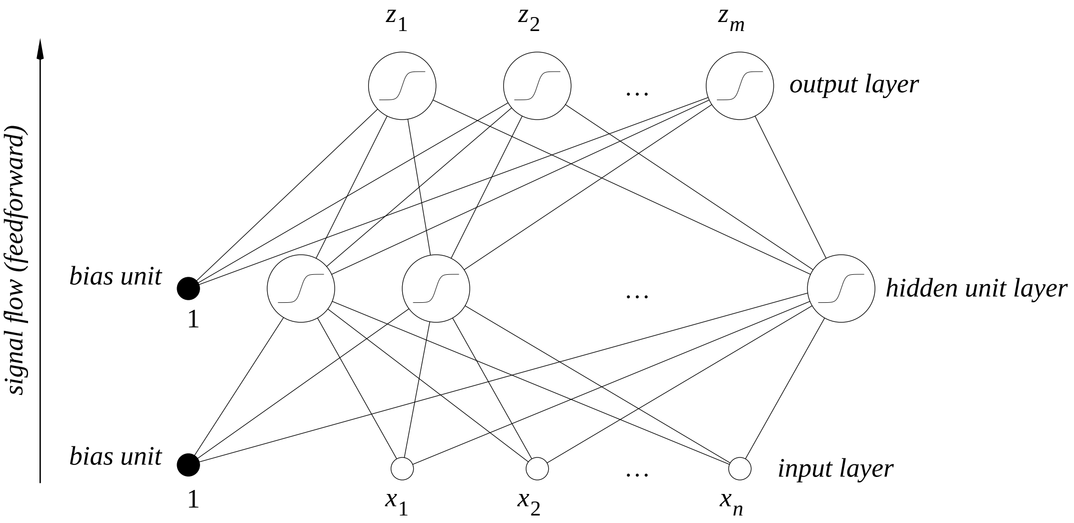


Figure 9

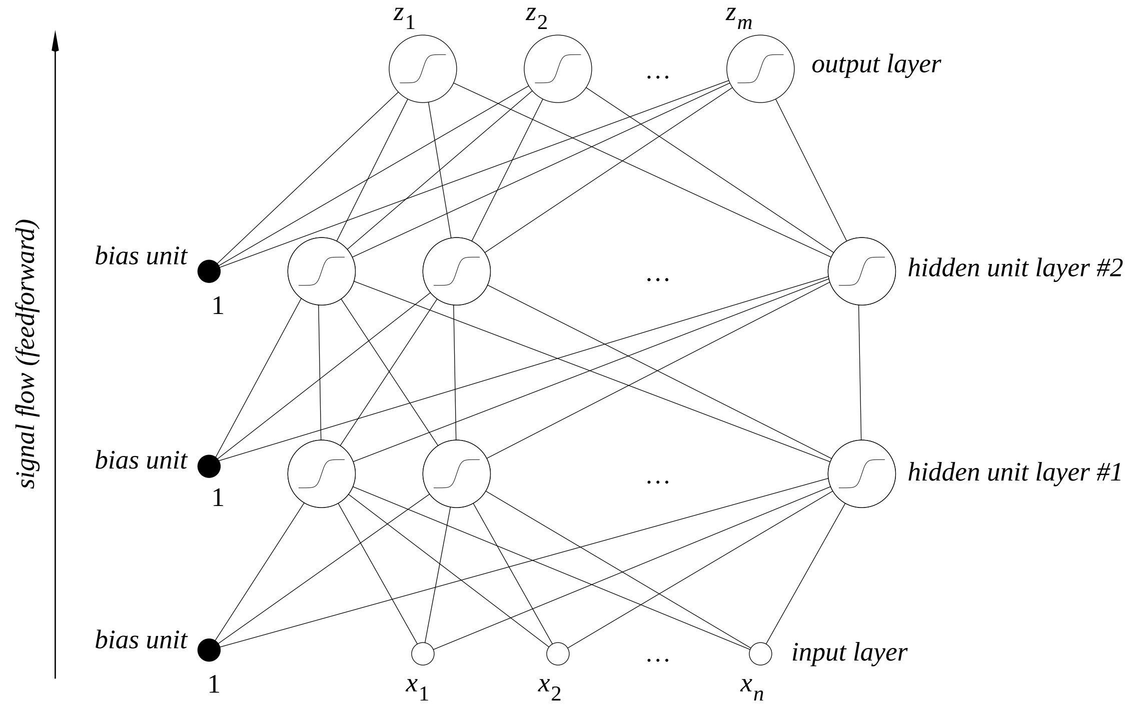


Figure 10

To answer this question, let us first express $f(x)$ in terms of threshold activation functions [equation (15)]:

$$f(x) = c[\gamma_t(x-a) - \gamma_t(x-b)] \tag{22}$$

$$f(x) = c\gamma_t(x-a) - c\gamma_t(x-b) \tag{23}$$

Recognizing that the threshold function can be approximated arbitrarily well by a sigmoid function [equation (19)],

$$\gamma_t(u) \to \gamma(ku) \text{ as } k \to \infty \tag{24}$$

we can rewrite (23) in terms of sigmoidal activation functions,

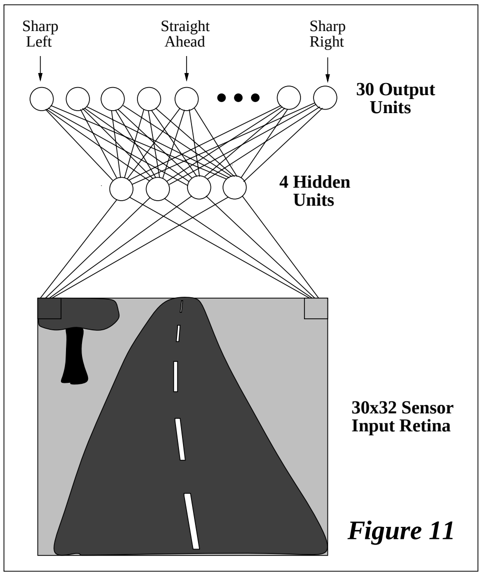


Figure 11

ALVINN: Neural Network for Autonomous Steering

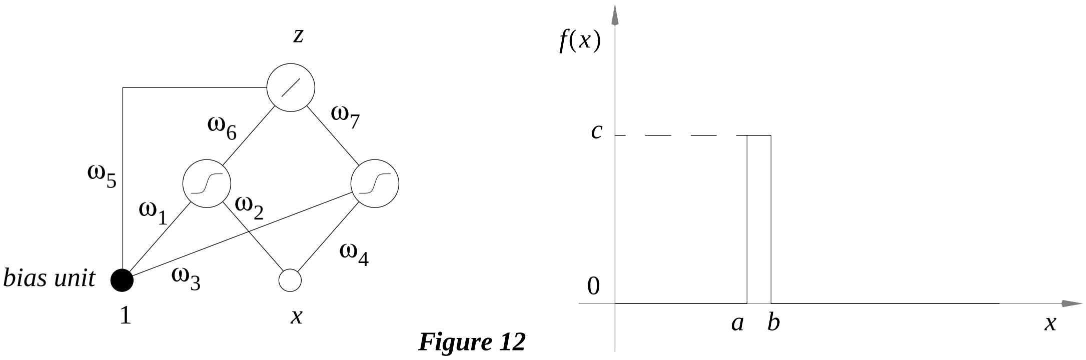


Figure 12

$$f(x) \approx c\gamma[k(x-a)] - c\gamma[k(x-b)] \quad \text{for large } k\,. \tag{25}$$

Now, let us write down an expression for z, the output of the neural network. From Figure 12,

$$z = \omega_5 + \omega_6\gamma(\omega_1 + \omega_2 x) + \omega_7\gamma(\omega_3 + \omega_4 x) \tag{26}$$

Comparing (25) and (26), we arrive at two possible sets of weight values for approximating $f(x)$ with z:

weights	ω_1	ω_2	ω_3	ω_4	ω_5	ω_6	ω_7
set #1	*–kb*	*k*	*–ka*	*k*	*0*	*–c*	*c*
set #2	*–ka*	*k*	*–kb*	*k*	*0*	*c*	*–c*

3. Some theoretical properties of neural networks

A. Single-input functions

From the example in Section 2(E), we can conclude that a single-hidden layer neural network can model *any* single-input function arbitrarily well with a sufficient number of hidden units, since any one-dimensional function can be expressed as the sum of localized "bumps." It is important to note, however, that typically, a neural network does not actually approximate functions as the sum of localized bumps. Consider, for example, Figure 13. Here, we used a three-hidden neural network to approximate a scaled sine wave. Note that even with only three hidden units, the maximum neural network error is less than 0.01.

B. Multi-input functions

Now, does this *universal function approximator* property for single-hidden layer neural networks hold for multi-dimensional functions? No, because the creation of localized peaks in multiple dimensions requires an

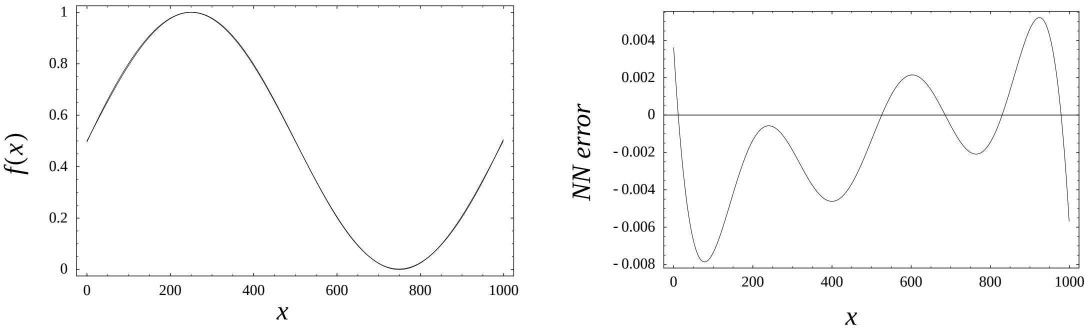
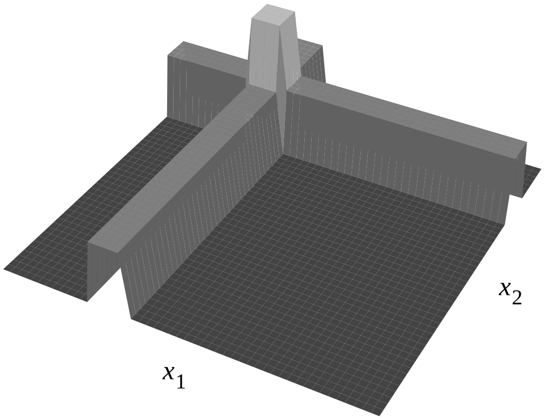


Figure 13

additional hidden layer. Consider, for example, Figure 14 below, where we used a four-hidden unit network to create a localized peak. Note, however, that unlike in the single-dimensional example, secondary ridges are also present. Thus, an additional sigmoidal hidden unit in a second layer is required to suppress the secondary ridges, but, at the same time, preserve the localized peak. This *ad hoc* “proof” indicates that *any* multi-input function can be modeled arbitrarily well by a two-hidden-layer neural network, as long as a sufficient number of hidden units are present in each layer. A formal proof of this is given by Cybenko [1].

Figure 14



4. Neural network training

There are three basic steps in applying neural networks to real problems:

1. Collect input/output training data of the form:

$$\{(\mathbf{x}_i, \mathbf{y}_i)\}\ ,\ i \in \{1, 2, \ldots, p\}\ , \tag{27}$$

where $\mathbf{x}_i$ is the n-dimensional input vector,

$$\mathbf{x}_i = \begin{bmatrix} x_{i1} & x_{i2} & \ldots & x_{in} \end{bmatrix}^T \tag{28}$$

corresponding to the i th training pattern, and $\mathbf{y}_i$ is the m-dimensional output vector,

$$\mathbf{y}_i = \begin{bmatrix} y_{i1} & y_{i2} & \ldots & y_{im} \end{bmatrix}^T \tag{29}$$

corresponding to the i th training pattern, $i \in \{1, 2, \ldots, p\}$.

2. Select an appropriate neural network architecture. Generally, this involves selecting the number of hidden layers, and the number of hidden units in each layer. For notational convenience, let,

$$\mathbf{z} = \Gamma(\mathbf{w}, \mathbf{x}) \tag{30}$$

denote the m-dimensional output vector $\mathbf{z}$ for the neural network Γ, with q-dimensional weight vector $\mathbf{w}$,

$$\mathbf{w} = \begin{bmatrix} \omega_1 & \omega_2 & \ldots & \omega_q \end{bmatrix}^T \tag{31}$$

and input vector $\mathbf{x}$. Thus,

$$\mathbf{z}_i = \Gamma(\mathbf{w}, \mathbf{x}_i) \tag{32}$$

denotes the neural network outputs $\mathbf{z}_i$ corresponding to the input vector for the i th training pattern.

3. Train the weights of the neural network to minimize the error measure,

$$E = \frac{1}{2}\sum_{i=1}^{p}\|\mathbf{y}_i - \mathbf{z}_i\|^2 = \frac{1}{2}\sum_{i=1}^{p}\sum_{j=1}^{m}(y_{ij} - z_{ij})^2 \tag{33}$$

which measures the difference between the neural network outputs $\mathbf{z}_i$ and the training data outputs $\mathbf{y}_i$. This error minimization is also frequently referred to as *learning*.

Steps 1 and 2 above are quite application specific and will be discussed a little later. Here, we will begin to investigate Step 3 — namely, the training of the neural network parameters (weights) from input/output training data.

A. Gradient descent

Note that since $\mathbf{z}_i$ (as defined in equation (32) above) is a function of the weights $\mathbf{w}$ of the neural network, E is implicitly a function of those weights as well. That is, E changes as a function of $\mathbf{w}$. Therefore, our goal is to find that set of weights $\mathbf{w}^*$ which minimizes E over a given training data set.

The first algorithm that we will study for neural network training is based on a method known as *gradient descent*. To understand the intuition behind this algorithm, consider Figure 15 below, where a simple one-dimensional error surface is drawn schematically. The basic question we must answer is: how do we find the parameter ω^* that corresponds to the minimum of that error surface (point d)?

Gradient descent offers a partial answer to this question. In gradient descent, we initialize the parameter ω to some random value and then incrementally change that value by an amount proportional to the negative derivative,

$$-\frac{dE}{d\omega} \tag{34}$$

Denoting $\omega(t)$ as parameter ω at step t of the gradient descent procedure, we can write this in equation form as,

$$\omega(t+1) = \omega(t) - \eta\frac{dE}{d\omega(t)} \tag{35}$$

where η is a small positive constant that is frequently referred to as the *learning rate*. In Figure 15, given an initial parameter value of a and a small enough learning rate, gradient descent will converge to the global minimum d as $t \to \infty$. Note, however, that the gradient descent procedure is not guaranteed to always converge to the *global* minimum for general (non-convex) error surfaces. If we start at an initial ω value of b, iteration (35) will converge to e, while for an initial ω value of c, gradient descent will converge to f as $t \to \infty$. Thus, gradient descent is only guaranteed to converge to a *local* minimum of the error surface (for sufficiently small learning rates η), not a global minimum.

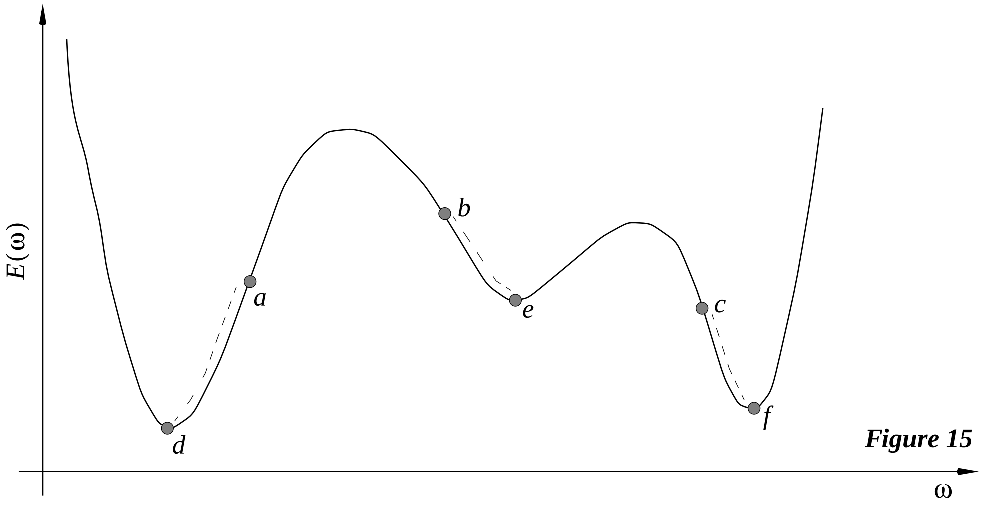


Figure 15

Iteration (35) is easily generalized to error minimization over multiple dimensions (i.e. parameter *vectors* $\mathbf{w}$),

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E[\mathbf{w}(t)] \tag{36}$$

where $\nabla E[\mathbf{w}(t)]$ denotes the gradient of E with respect to $\mathbf{w}(t)$,

$$\nabla E[\mathbf{w}(t)] = \left[\frac{\partial E}{\partial \omega_1(t)} \; \frac{\partial E}{\partial \omega_2(t)} \; \cdots \; \frac{\partial E}{\partial \omega_q(t)}\right]^T \tag{37}$$

Thus, one approach for training the weights in a neural network implements iteration (37) with the error measure defined in equation (33).

B. Simple example

Consider the simple single-input, single-output feedforward neural network in Figure 16 below, with sigmoidal hidden-unit activation functions γ, and a linear output unit. For this neural network, let us, by way of example, compute,

$$\frac{\partial E}{\partial \omega_4} \tag{38}$$

where,

$$E = \frac{1}{2}(y-z)^2 \tag{39}$$

for a single training pattern $\langle x, y \rangle$. Note that since differentiation is a linear operator, the derivative for multiple training patterns is simply the sum of the derivatives of the individual training patterns,

$$\frac{\partial E}{\partial \omega_j} = \frac{\partial}{\partial \omega_j}\left[\frac{1}{2}\sum_{i=1}^{p}(y_i - z_i)^2\right] = \sum_{i=1}^{p}\frac{\partial}{\partial \omega_j}\left[\frac{1}{2}(y_i - z_i)^2\right] . \tag{40}$$

Therefore, generalizing the example below to multiple training patterns is straightforward.

First, let us explicitly write down z as a function of the neural network weights. To do this, we define some intermediate variables,

$$net_1 \equiv \omega_1 + \omega_2 x \tag{41}$$

$$net_2 \equiv \omega_3 + \omega_4 x \tag{42}$$

which denote the net input to the two hidden units, respectively, and,

$$h_1 \equiv \gamma(net_1) \tag{43}$$

$$h_2 \equiv \gamma(net_2) \tag{44}$$

which denote the outputs of the two hidden units, respectively. Thus,

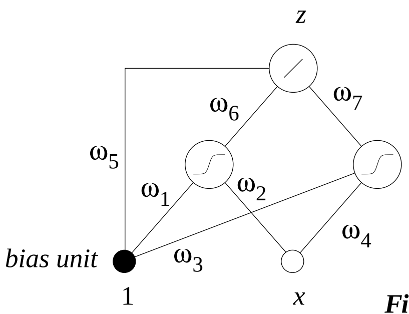


Figure 16

$$z = \omega_5 + \omega_6 h_1 + \omega_7 h_2 \text{ (linear output unit).} \tag{45}$$

Now, we can compute the derivative of E with respect to ω_4. From (39), and remembering the chain rule of differentiation,

$$\frac{\partial E}{\partial \omega_4} = -(y-z)\frac{\partial z}{\partial \omega_4} \tag{46}$$

$$\frac{\partial E}{\partial \omega_4} = (z-y)\left(\frac{\partial z}{\partial h_2}\right)\left(\frac{\partial h_2}{\partial net_2}\right)\left(\frac{\partial net_2}{\partial \omega_4}\right) \tag{47}$$

$$\frac{\partial E}{\partial \omega_4} = (z-y)\omega_7\gamma'(net_2)x \tag{48}$$

where γ' denotes the derivative of the activation function. This example shows that, in principle, computing the partial derivatives required for the gradient descent algorithm simply requires careful application of the chain rule. In general, however, we would like to be able to simulate neural networks whose architecture is not known *a priori*. In other words, rather than hard-code derivatives with explicit expressions like (48) above, we require an algorithm which allows us to compute derivatives in a more general way. Such an algorithm exists, and is known as the *backpropagation algorithm*.

C. Backpropagation algorithm

The backpropagation algorithm was first published by Rumelhart and McClelland in 1986 [4], and has since led to an explosion in previously dormant neural-network research. *Backpropagation* offers an efficient, algorithmic formulation for computing error derivatives with respect to the weights of a neural network. As such, it allows us to implement gradient descent for neural network training without explicitly hard-coding derivatives.

In order to develop the backpropagation algorithm, let us first look at an arbitrary (hidden or output) unit in a feedforward (acyclic) neural network with activation function γ. In Figure 17, that unit is labeled j. Let h_j be the output of unit j, and let net_j be the net input to unit j. By definition,

$$h_j \equiv \gamma(net_j) \tag{49}$$

$$net_j \equiv \sum_k h_k \omega_{kj} \tag{50}$$

Note that net_j is summed over all units feeding into unit j; unit i is one of those units. Let us now compute,

$$\frac{\partial E}{\partial \omega_{ij}} = \left(\frac{\partial E}{\partial net_j}\right)\left(\frac{\partial net_j}{\partial \omega_{ij}}\right) \tag{51}$$

From equation (50),

$$\frac{\partial net_j}{\partial \omega_{ij}} = h_i \tag{52}$$

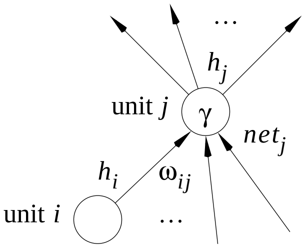


Figure 17

since all the terms in summation (50), $k \neq i$ are independent of ω_{ij}. Defining,

$$\delta_j \equiv \frac{\partial E}{\partial net_j} \tag{53}$$

we can write equation (51) as,

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i \tag{54}$$

As we will see shortly, equation (54) forms the basis of the backpropagation algorithm in that the δ_j variables can be computed recursively from the outputs of the neural network back to the inputs of the neural network. In other words, the δ_j values are *backpropagated* through the network (hence, the name of the algorithm).

D. Backpropagation example

Consider Figure 18, which plots a small part of a neural network. Below, we derive an expression for δ_k (output unit) and δ_j (hidden unit one layer removed from the outputs of the neural network). For a single training pattern, we can write,

$$E = \frac{1}{2}\sum_{l=1}^{m}(y_l - z_l)^2 \tag{55}$$

where l indexes the outputs (not the training patterns). Now,

$$\delta_k \equiv \frac{\partial E}{\partial net_k} = \left(\frac{\partial E}{\partial z_k}\right)\left(\frac{\partial z_k}{\partial net_k}\right) \tag{56}$$

Since,

$$z_k = \gamma(net_k) \tag{57}$$

we have that,

$$\frac{\partial z_k}{\partial net_k} = \gamma'(net_k) \tag{58}$$

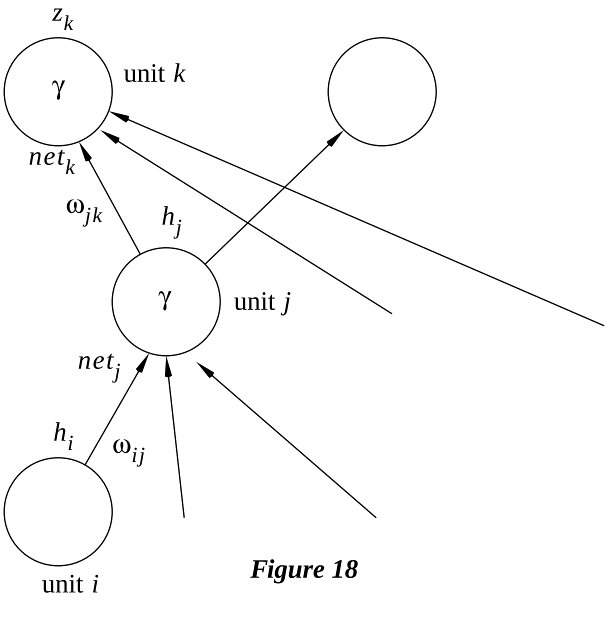


Figure 18

Furthermore, from equation (55),

$$\frac{\partial E}{\partial z_k} = (z_k - y_k) \tag{59}$$

since all the terms in summation (55), $l \neq k$ are independent of z_k. Combining equations (56), (58) and (59), and recalling equation (54),

$$\delta_k = (z_k - y_k)\gamma'(net_k) \tag{60}$$

$$\frac{\partial E}{\partial \omega_{jk}} = \delta_k h_j \tag{61}$$

Note that equations (60) and (61) are valid for *any* weight in a neural network that is connected to an output unit. Also note that h_j is the output value of units feeding into output unit k. While this may be the output of a hidden unit, it could also be the output of the bias unit (i.e. 1) or the value of a neural network input (i.e. x_j). Next, we want to compute δ_j in Figure 18 in terms of the δ values that follow unit j. Going back to definition (53),

$$\delta_j \equiv \frac{\partial E}{\partial net_j} = \sum_l \left(\frac{\partial E}{\partial net_l}\right)\left(\frac{\partial net_l}{\partial net_j}\right) \tag{62}$$

Note that the summation in equation (62) is over all the immediate successor units of unit j. Thus,

$$\delta_j = \sum_l \delta_l \left(\frac{\partial net_l}{\partial net_j}\right) \tag{63}$$

By definition,

$$net_l = \sum_s \omega_{sl}\gamma(net_s) \tag{64}$$

So, from equation (64),

$$\frac{\partial net_l}{\partial net_j} = \omega_{jl}\gamma'(net_j) \tag{65}$$

since all the terms in summation (64), $s \neq j$ are independent of net_j. Combining equations (63) and (65),

$$\delta_j = \sum_l \delta_l \omega_{jl}\gamma'(net_j) \tag{66}$$

$$\delta_j = \left(\sum_l \delta_l \omega_{jl}\right)\gamma'(net_j) \tag{67}$$

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i \tag{68}$$

Note that equation (67) computes δ_j in terms of those δ values one connection ahead of unit j. In other words, the δ values are backpropagated from the outputs back through the network. Also note that h_i is the output value of units feeding into unit j. While this may be the output of a hidden unit from an earlier hidden-unit layer, it could also be the output of a bias unit (i.e. 1) or the value of a neural network input (i.e. x_i).

It is important to note that (1) the general derivative expression in (54) is valid for *all* weights in the neural network; (2) the expression for the output δ values in (60) is valid for *all* neural network output units; and (3) the recursive relationship for δ_j in (67) is valid for *all* hidden units, where the l-indexed summation is over all immediate successors of unit j.

E. Summary of backpropagation algorithm

Below, we summarize the results of the derivation in the previous section. The partial derivative of the error,

$$E = \frac{1}{2}\sum_{l=1}^{m}(y_l - z_l)^2 \tag{69}$$

(i.e. a single training pattern) with respect to a weight ω_{jk} connected to output unit k of a neural network is given by,

$$\delta_k = (z_k - y_k)\gamma'(net_k) \tag{70}$$

$$\frac{\partial E}{\partial \omega_{jk}} = \delta_k h_j \tag{71}$$

where h_j is the output of hidden unit j (or the input j), and net_k is the net input to output unit k. The partial derivative of the error E with respect to a weight ω_{ij} connected to hidden unit j of a neural network is given by,

$$\delta_j = \left(\sum_l \delta_l \omega_{jl}\right)\gamma'(net_j) \tag{72}$$

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_j h_i \tag{73}$$

where h_i is the output of hidden unit i (or the input i), and net_j is the net input to hidden unit j. The above results are trivially extended to multiple training patterns by summing the results for individual training patterns over all training patterns.

5. Basic steps in using neural networks

So, now we know what a neural network is, and we know a basic algorithm for training neural networks (i.e. backpropagation). Here, we will extend our discussion of neural networks by discussing some practical aspects of applying neural networks to real-world problems. Below, we review the steps that need to be followed in using neural networks.

A. Collect training data

In order to apply a neural network to a problem, we must first collect input/output training data that adequately represents that problem. Often, we also need to condition, or preprocess that data so that the neural network training converges more quickly and/or to better local minima of the error surface. Data collection and preprocessing is very application-dependent and will be discussed in greater detail in the context of specific applications.

B. Select neural network architecture

Selecting a neural network architecture typically requires that we determine (1) an appropriate number of hidden layers and (2) an appropriate number of hidden units in each hidden layer for our specific application, assuming a standard multilayer feedforward architecture. Often, there will be many different neural network structures that work about equally well; which structures are most appropriate is frequently guided by experience and/or trial-and-error. Alternatively, as we will talk about later in this course, we can use neural network learning algorithms that adaptively change the structure of the neural network as part of the learning process.

C. Select learning algorithm

If we use simple backpropagation, we must select an appropriate learning rate η. Alternatively, as we will talk about later in this course, we have a choice of more sophisticated learning algorithms as well, including the conjugate gradient and extended Kalman filtering methods.

D. Weight initialization

Weights in the neural network are usually initialized to small, random values.

E. Forward pass

Apply a random input vector $\mathbf{x}_i$ from the training data set to the neural network and compute the neural network outputs (z_k), the hidden-unit outputs (h_j), and the net input to each hidden unit (net_j).

F. Backward pass

1. Evaluate δ_k at the outputs, where,

$$\delta_k = \frac{\partial E}{\partial net_k} \tag{74}$$

for each output unit.

2. Backpropagate the δ values from the outputs backwards through the neural network.
3. Using the computed δ values, calculate,

$$\frac{\partial E}{\partial \omega_i}, \tag{75}$$

the derivative of the error with respect to each weight ω_i in the neural network.

4. Update the weights based on the computed gradient,

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \nabla E[\mathbf{w}(t)]. \tag{76}$$

G. Loop

Repeat steps E and F (forward and backward passes) until training results in a satisfactory model.

6. Practical issues in neural networks

A. What should the training data be?

Some questions that need to be answered include:

1. Is your training data sufficient for the neural network to adequately learn what you want it to learn? For example, what if, in ALVINN [3], we down-sampled to 10×10 images, instead of 30×32 images? Such coarse images would probably not suffice for learning the steering of the on-road vehicle with enough accuracy. At the same time we must make sure that we don’t include training data that is too much or irrelevant for our application (e.g. for ALVINN, music played while driving). Poorly correlated or irrelevant inputs can easily slowdown convergence of, or completely sidetrack, neural network learning algorithms.
2. Is your training data biased? Suppose for ALVINN, we trained the neural network on race track oval. How would ALVINN drive on real roads? Well, it would probably not have adequately learned right turns, since the race track consists of left turns only. The distribution of your training data needs to approximately reflect the expected distribution of input data where the neural network will be used after training.
3. Is your task deterministic or stochastic? Is it stationary or nonstationary? Nonstationary problems cannot be trained from fixed data sets, since, by definition, things change over time.

We will have more on these concerns within the context of specific applications later.

B. What should your neural network architecture/structure be?

This question is largely task dependent, and often requires experience and/or trial-and-error to answer adequately. Therefore, we will have more on this question within the context of specific applications later. In general, though, it helps to look at similar problems that have previously been solved with neural networks, and apply the lessons learned there to our current application. Adaptive neural network architectures, that change the structure of the neural network as part of training, are also an alternative to manually selecting an appropriate structure.

C. Preprocessing of data

Often, it is wise to preprocess raw input/output training data, since it can make the learning (i.e. neural network training) converge much better and faster. In computer vision applications, for example, intensity normalization can remove variation in intensity — caused perhaps by sunny *vs*. overcast days — as a potential source of confusion for the neural network. We will have more on this question within the context of specific applications later.

D. Weight initialization

Since the weight parameters $\mathbf{w}$ are learned through the recursive relationship in (76), we obviously need to initialized the weights [i.e. set $\mathbf{w}(0)$]. Typically, the weights are initialized to *small*, *random* values. If we were to initialize the weights to *uniform* (i.e. identical) values instead, the significant weight symmetries in the neural network would substantially reduce the effective parameterization of the neural network since many partial error derivatives in the neural network would be identical at the beginning of training and remain so throughout. If we were to initialize the weights to *large* values, there is a high likelihood that many of the hidden unit activations in the neural network would be stuck in the flat areas of the typical sigmoidal activation functions, where the derivatives evaluate to approximately zero. As such, it could take quite a long time for the weights to converge.

E. Select a learning parameter

If using standard gradient descent, we must select an appropriate learning rate η . This can be quite tricky, as the simple example below illustrates. Consider the trivial two-dimensional, quadratic “error” function,

$$E = 20\omega_1^2 + \omega_2^2 \tag{77}$$

which we plot in Figure 19 below. [Note that equation (77) could never really be a neural network error function, since a neural network typically has many hundreds or thousands of weights.]

For this error function, note that the global minimum occurs at $(\omega_1, \omega_2) = (0, 0)$. Now, let us investigate how quickly gradient-descent converges to this global minimum for different learning rates η ; for the purposes of this example, we will say that gradient descent has converged when $\sqrt{E} < 10^{-6}$. First, we must compute the derivatives,

$$\frac{\partial E}{\partial \omega_1} = 40\omega_1 \text{, and,} \tag{78}$$

$$\frac{\partial E}{\partial \omega_2} = 2\omega_2 \text{,} \tag{79}$$

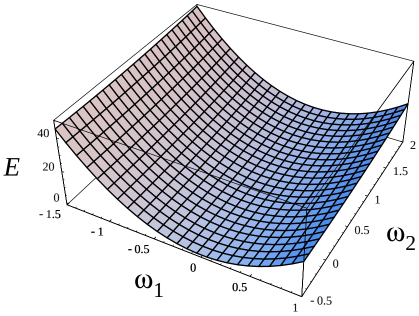


Figure 19

so that the gradient-descent weight recursion in (76) is given by,

$$\omega_1(t+1) = \omega_1(t) - \eta\frac{\partial E}{\partial\omega_1(t)} \tag{80}$$

$$\omega_1(t+1) = \omega_1(t)(1-40\eta) \tag{81}$$

and similarly,

$$\omega_2(t+1) = \omega_2(t)(1-2\eta)\,. \tag{82}$$

From an initial point $(\omega_1, \omega_2) = (1, 2)$, Figure 20 below plots the number of steps to convergence as a function of the learning parameter η. Note that the number of steps to convergence decreases as a function of the learning rate parameter η until about 0.047 (intuitive), but then shoots up sharply until 0.05, at which point the gradient-descent equations in (81) and (82) become unstable and diverge (counter-intuitive).

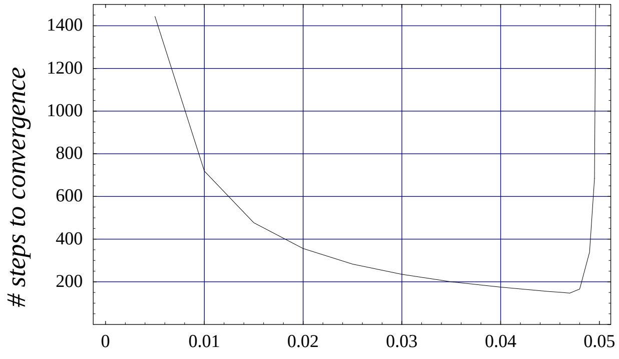


Figure 20

Figure 21 plots some actual gradient-descent trajectories for the learning rates 0.02, 0.04 and 0.05. Note that for $\eta = 0.05$, gradient descent does not converge but oscillates about $\omega_2 = 0$. To understand why this is happening, consider the fixed-point iterations in (81) and (82). Each of these is of the form,

$$\omega(t+1) = c\omega(t) \tag{83}$$

which will diverge for any nonzero $\omega(0)$ and $\|c\| > 1$, and converge for $\|c\| < 1$. Thus, equation (81) will converge for,

$$\|1-40\eta\| < 1 \tag{84}$$

$$-1 < 1-40\eta < 1 \tag{85}$$

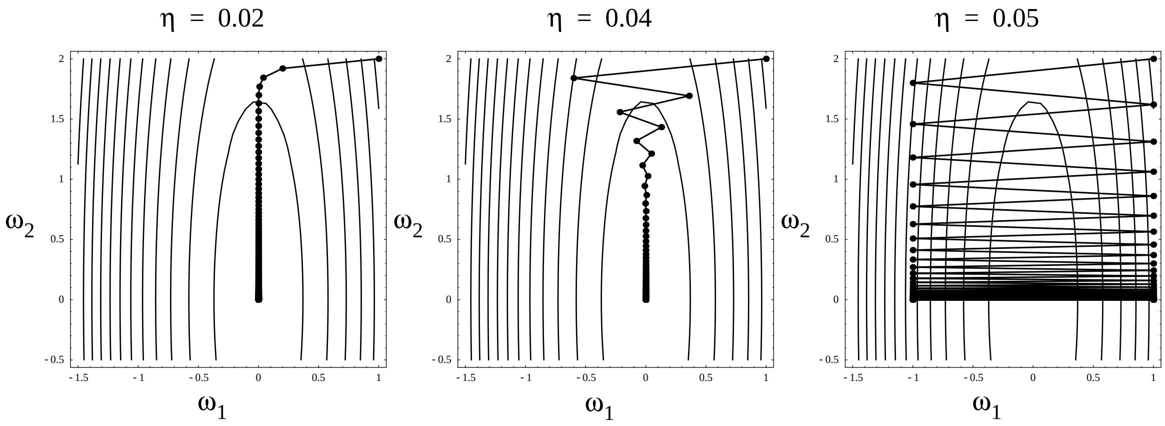


Figure 21

$$0 < \eta < 0.05 \qquad (86)$$

Since recursion (82) generates the weaker bound,

$$0 < \eta < 1 \, , \qquad (87)$$

the upper bound in (86) is controlling in that it determines the range of learning rates for which gradient descent will converge in this example.

We make a few observations from this specific example: First, “long, steep-sided valleys” in the error surface typically cause slow convergence with a single learning rate, since gradient descent will converge quickly down the steep valleys of the error surface, but will take a long time to travel along the shallow valley. Slow convergence of gradient descent is largely why we will study more sophisticated learning algorithms, with *de facto* adaptive learning rates, later in this course. In this example, convergence along the ω_2 axis is assured for larger η ; however, the upper bound in (86) prevents us from using a (fixed) learning rate greater than or equal to 0.05 . Second, Figure 20, although drawn specifically for this example, is generally reflective of gradient-descent convergence rates for more complex error surfaces as well. If the chosen learning rate is too small, convergence can take a very long time, while learning rates that are too large will cause gradient descent to diverge. This is another reason to study more sophisticated algorithms — since selecting an appropriate learning rate can be quite frustrating, algorithms that do not require such a selection have a real advantage. Finally, note that, in general, it is not possible to determine theoretical convergence bounds, such as those in (86), for real neural networks and error functions. Only the very simple error surface in (77) allowed us to do that here.

F. Pattern *vs.* batch training

In *pattern training,* we compute the error E and the gradient of the error ∇E for one input/output pattern at a time, and update weights based on that single training example (Section 5 describes pattern training). It is usually a good idea to randomize the order of training patterns in pattern training, so that the neural network does not converge to a bad local minima or forget training examples early in the training.

In *batch training*, we compute the error E and the gradient of the error ∇E for all training examples at once, and update the weights based on that aggregate error measure.

G. Good generalization

Generalization to examples not explicitly seen in the training data set is one of the most important properties of a good model, including neural network models. Consider, for example, Figure 22. Which is a better model, the left curve or the right curve? Although the right curve (i.e. model) has zero error over the specific data set, it will probably generalize more poorly to points not in the data set, since it appears to have modeled the noise properties of the specific training data set. The left model, on the other hand, appears to have abstracted the essential feature of the data, while rejecting the random noise superimposed on top.

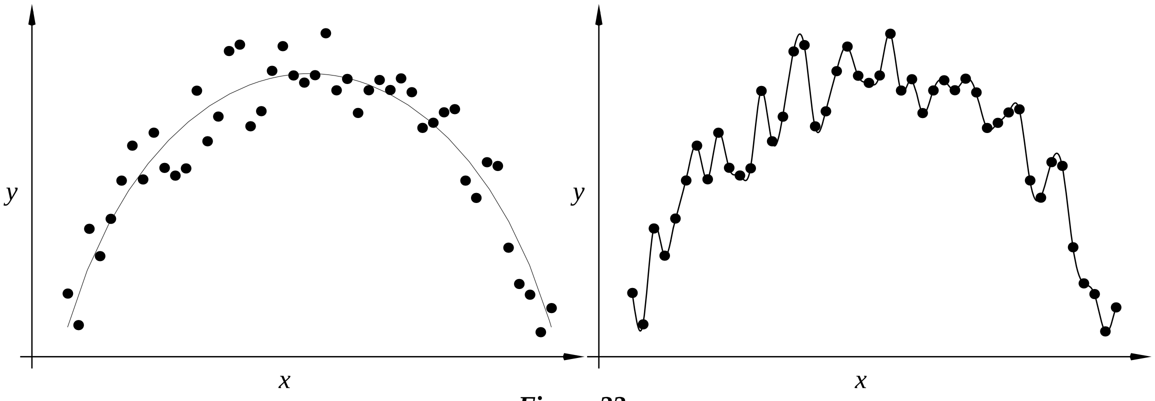


Figure 22

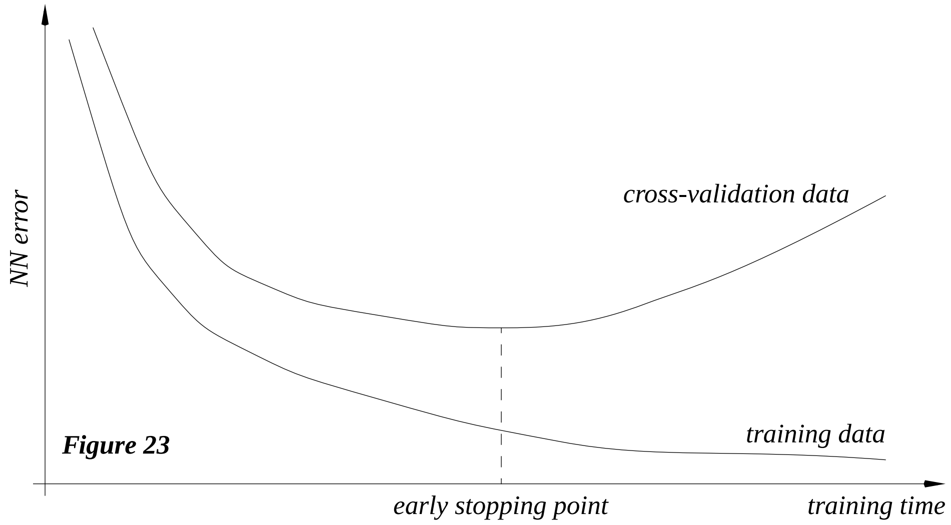


Figure 23

There are two ways that we can ensure that neural networks generalize well to data not explicitly in the training data set. First we need to pick a neural network architecture that is not over-parameterized — in other words, the smallest neural network that will perform its task well. Second, we can use a method known as *cross-validation*. In typical neural network training, we take our complete data set, and split that data set in two. The first data set is called the training data set, and is used to actually train the weights of the neural network; the second data set is called the cross-validation data set, and is not explicitly used in training the weights; rather, the cross-validation set is reserved as a check on neural network learning to prevent overtraining. While training (with the training data set), we keep track of both the training data set error and the cross-validation data set error. When the cross-validation error no longer decreases, we should stop training, since that is a good indication that further learning will adjust the weights only to fit peculiarities of the training data set. This scenario is depicted in the generic diagram of Figure 23 below, where, we plot neural network error as a function of training time. As we indicate in the figure, the training data set error will generally be lower than the cross-validation data set error; moreover, the training data set error will usually continue to decrease as a function of training time, whereas the cross-validation data set error will typically begin to increase at some point in the training.

[1] G. Cybenko, “Approximation by Superposition of a Sigmoidal Function,” *Mathematics of Control, Signals, and Systems*, vol. 2, no. 4, pp. 303-14, 1989.

[2] Richard O. Duda, Peter E. Hart and David G. Stork, *Pattern Classification, 2nd ed.*, Chapters 5 and 6, John Wiley & Sons, New York, 2001. .

[3] D. A. Pomerleau, “Neural Network Perception for Mobile Robot Guidance,” Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1992.

[4] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing: Exploration in the Microstructure of Cognition*, vols. 1 and 2, MIT Press, Cambridge, MA, 1986.