2 *Mutilayer Networks*

Objective

This chapter discusses multilayer networks and associated notation that will be needed in other parts of the book. It describes the components of the multilayer network and the basic architecture. The chapter also describes the functionality of the multilayer network and how increasing the number of layers, creating deep networks, efficiently increases the power of the network.

Theory and Examples

Architecture

Let's begin by reviewing some of the basic components of a multilayer neural network. Much of this was also covered in Chapter 2 of NND2. The architecture of this network, as well as the general training concepts (discussed in Chapter 11 of NND2), have been known since the 1980's. Until about 2010, most people used neural networks with one or two hidden layers, but the theory for an unlimited number of layers was available 30 years before that. Since 2010 a few small additions have been made to facilitate the training of networks with many hidden layers, but the basic concepts have not changed.

The fundamental building block is the single neuron, shown in Figure 2.1. As in NND2, we use p to represent external inputs into the network. In the neuron, p is first multiplied by the weight w. The result is added to the bias b to form the net input n, which is then passed through the activation (or transfer) function f to produce the neuron output a. In future chapters, it will be convenient to consider the result of the weight operation wp as a separate variable, for which we will use the letter z.

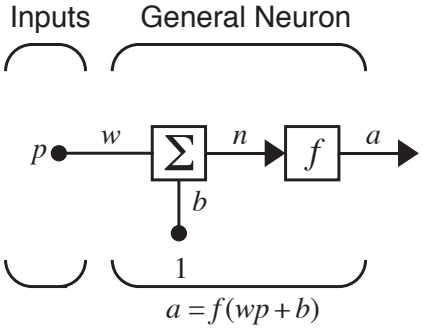


Figure 2.1: Single Neuron

There are four activation functions that we we will use for deep networks. The first is the linear activation function , illustrated in Figure 2.2. This is really equivalent to having no activation function, but we include it for consistency. The graph on the right of the figure illustrates the effect of the bias on the neuron response. A positive bias shifts the response to the left, and a negative bias shifts

the response to the right. If the bias is zero, the activation will pass through the origin. Linear activation functions are generally used in the final layer of networks that are used for function approximation, or nonlinear regression.

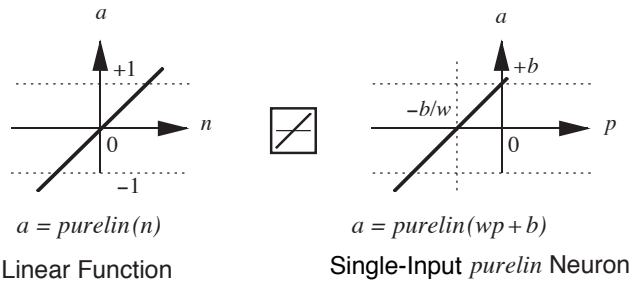


Figure 2.2: Linear Activation Function

The second commonly used activation function is the hyperbolic tangent function. This is a class of function called a sigmoid function (see also the *logsig* activation function in Chapter 2 of NND2), because of the S-shape. It is illustrated in Figure 2.3 . It has been shown that two-layer networks with sigmoid activation functions are universal approximators. This nonlinear function has the advantage that it is differentiable, which is important for network training.

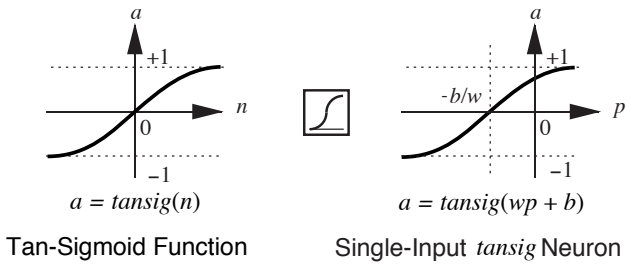


Figure 2.3: Hyperbolic Tangent Activation Function

The *tansig* function was commonly used in the hidden layers of neural networks until larger numbers of hidden layers began to be used (deep networks). When the number of hidden layers becomes larger, sigmoid activation functions lead to a problem in network training called the *vanishing gradient*, which we will discuss later. It occurs because the sigmoid function is limited in magnitude, and therefore its derivative becomes vanishingly small as the net input becomes larger in magnitude. In 2011 a different activation function was suggested for the hidden layers of deep networks (see [Glorot et al., 2011]). It was called the rectified linear unit, or ReLU . It is another name for the *poslin* (positive linear) activation

function that was described in 1995 in the first edition of NND2. The function is shown in Figure 2.4. The output is zero when the net input is negative, and it is equal to the net input when the net input is positive.

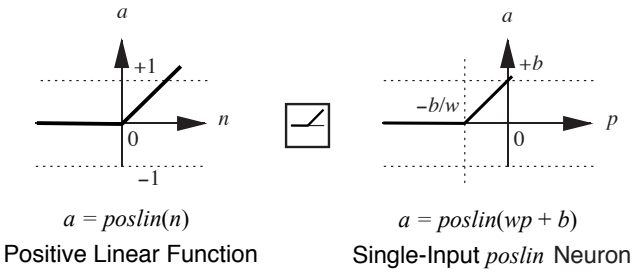


Figure 2.4: Positive Linear (or ReLU) Activation Function

The fourth activation function that we frequently use for pattern classification problems is the *softmax* function, which was described in Chapters 22 and 24 of NND2. This function is used in the final layer of classification networks, and is defined by Eq. 2.1.

$$a_i = f_i(\mathbf{n}) = \frac{e^{n_i}}{\sum_{j=1}^{S} e^{n_j}} \tag{2.1}$$

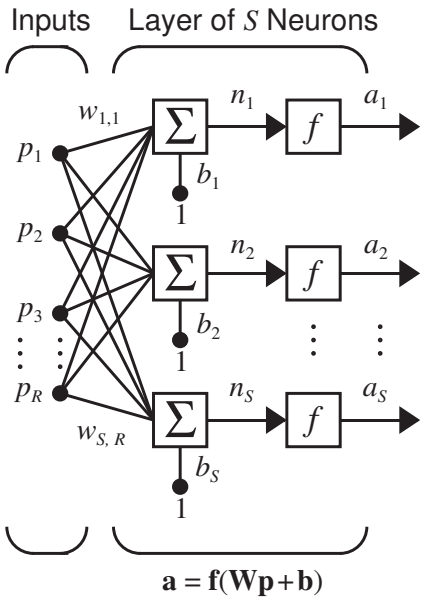


Figure 2.5: Layer of Neurons

This activation function is not defined for single neurons, but only when neurons are stacked together to create a layer, as shown in Figure 2.5. The sum of the layer outputs will be equal to 1 when the softmax activation function is used, and the output of neuron i can be interpreted as the probability that the input belongs to class i.

As in NND2, we prefer to use matrix notation to represent layers of neurons, as shown in Figure 2.6. This provides simpler diagrams in which all operations are clearly visible and emphasizes that matrix operations are taking place.

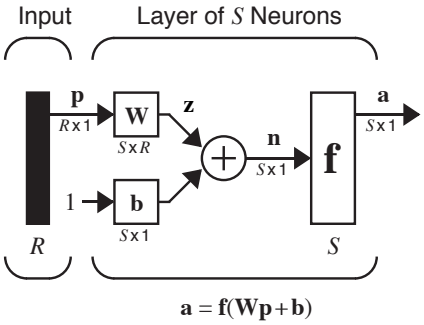


Figure 2.6: Layer of Neurons in Matrix Notation

Overall, there are three components to a layer: 1) a weight operation ($\mathbf{z} = \mathbf{Wp}$ in this case), 2) a net input operation ($\mathbf{n} = \mathbf{z} + \mathbf{b}$ in this case), and 3) an activation function operation ($\mathbf{f}(\mathbf{n})$).

Layers of neurons can be cascaded to obtain multilayer networks, as shown in Figure 2.7. By extending the network to more than one layer, the power of the network is increased dramatically. As described in Chapter 4 of NND2, a single layer network can only implement a linear decision boundary or a linear network response, while, as described in Chapter 11 of NND2, a network with two layers is a universal approximator.

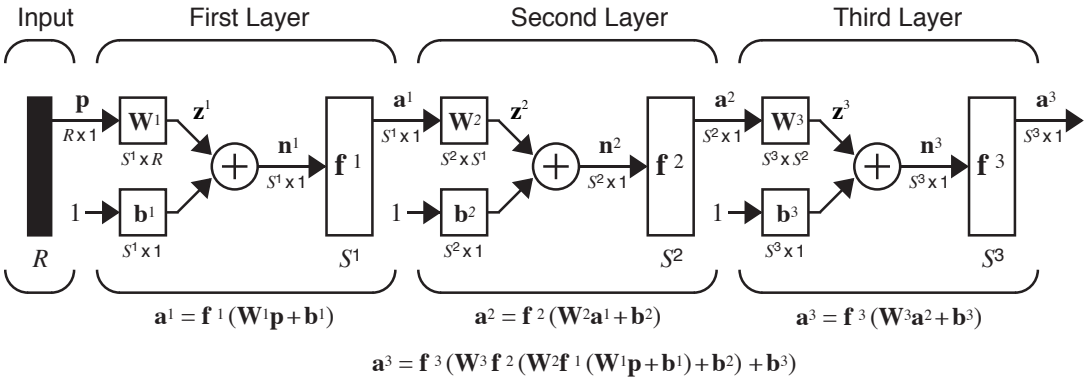


Figure 2.7: Three Layer Network

Operation

To illustrate the operation of a single layer network, consider the network in Figure 2.8. The activation function is the *hardlims*, which outputs -1 for net inputs less than zero, and $+1$ otherwise. This produces a simple decision making network.

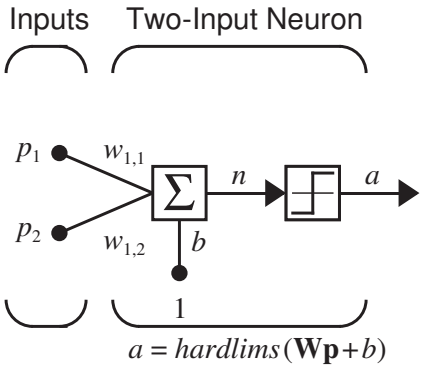


Figure 2.8: Example One Layer Network

If the weight matrix is chosen to be $\mathbf{W} = \begin{bmatrix} -1 & 1 \end{bmatrix}$, and the bias is chosen to be $b = [-1]$, then the network response would be

$$a = f(n) = f(\mathbf{W}\mathbf{p} + b) = hardlims(-1p_1 + 1p_2 - 1) \qquad (2.2)$$

The network output will be 1 whenever the net input is greater than or equal to 0. This defines the decision boundary, which is shown in Figure 2.9. All single layer networks have linear decision boundaries. (See Chapter 4 of NND2.) The function created by the net input is linear, and the locations where that function is equal to a fixed value defines a line (or hyperplane in higher dimensions). If the classes we are trying to recognize cannot be separated by a linear decision boundary, then a multilayer network must be used.

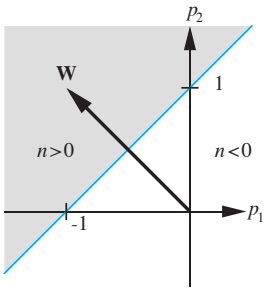


Figure 2.9: Linear Decision Boundary

To understand how multilayer networks allow us to create very general nonlinear functions, consider the network in Figure 2.10.

This is like the example in Chapter 11 of NND2, except that the *poslin* activation function is used in the hidden layer. Since the *poslin* (*ReLU*) function was one of the important innovations for deep network training, it is useful to see how it changes the form of the function that is created by the network. Suppose that the weights and biases of the network are set to the following values:

$$\mathbf{W}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} -1 & 1 \end{bmatrix}, \mathbf{b}^2 = [0] \qquad (2.3)$$

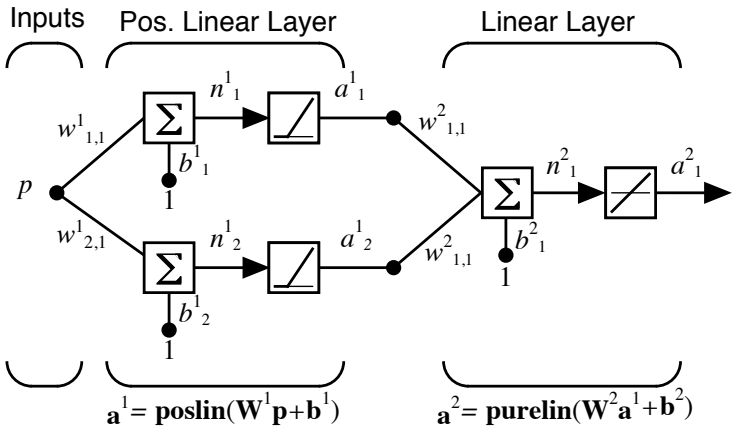


Figure 2.10: Two Layer *poslin* Network

The function created by this network is shown in Figure 2.11. We can see that with the *poslin* activation function in the hidden layer the network creates piece-wise linear functions.

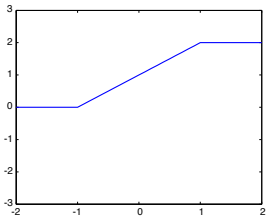


Figure 2.11: Function Created by the *poslin* Network

By changing the weights in the network, we can change the slopes of the linear segments of the network function. By changing the biases in the hidden layer, we can move the break points between the line segments, and by changing the bias in the final, linear layer, we can shift the function up or down. This is illustrated in Figure 2.12.

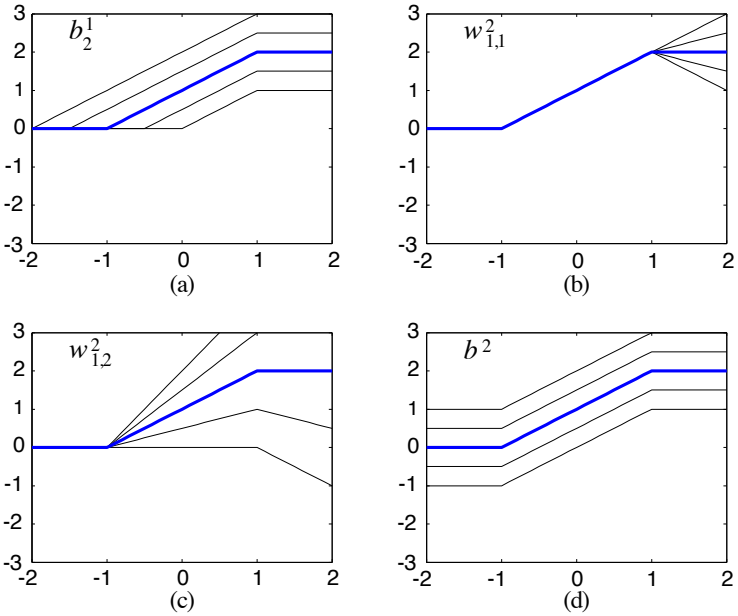


Figure 2.12: Variations in Network Function

The set of functions that can be created by a two-layer network, with *poslin* activation functions in the hidden layer, and a linear output layer, is the set of all piece-wise linear functions. This makes the two-layer *poslin* network a universal approximator.

To experiment with this poslin network, use the Deep Learning Demonstration Poslin Network Function (**dl2pnf**).

In addition to creating functions, we can also use the two layer network to perform pattern classification. Consider the network in Figure 2.13. Let the weights be set as follows:

$$\mathbf{W}^1 = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}^T, \mathbf{b}^1 = \begin{bmatrix} -1 & 3 & 1 & 1 \end{bmatrix}^T \tag{2.4}$$

$$\mathbf{W}^2 = \begin{bmatrix} -1 & -1 & -1 & -1 \end{bmatrix}, \mathbf{b}^2 = [5] \tag{2.5}$$

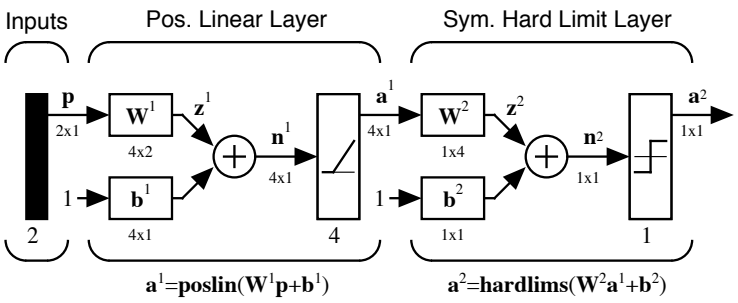
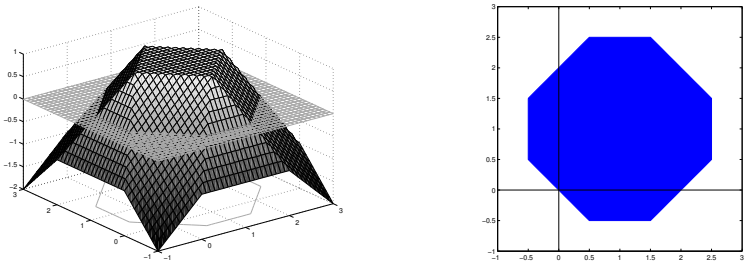


Figure 2.13: Pattern Classification Network

Since this network has two inputs, each row of the first weight matrix points in the direction of a sloping plane, whose starting location is determined by the corresponding bias. Each row of the weight creates a separate plane, and they are all added together at the output. The resulting composite function that is created at the net input of the second layer ($\mathbf{n}^2$) is shown in Figure 2.14(a). When this is passed through the *hardlims* activation function, we obtain the decision region shown in Figure 2.14(b). The solid region shows the area where $\mathbf{n}^2$ is greater than or equal to zero. By including more neurons in the second layer, it is possible to create essentially arbitrary decision regions.



(a) Network Function

(b) Decision Boundary

Figure 2.14: Network Response (a) and Decision Boundary (b)

To experiment with these two dimensional decision regions, use the Deep Learning Demonstration Poslin Decision Regions (**dl2pdr**).

Deep Networks

With two-layer networks it is possible to create almost arbitrary functions and decision regions. Then why are deep networks needed? At the current time, this question has not been fully answered to everyone's satisfaction. In practice, it has been shown that networks with many layers have been able to solve problems that have not, until now, been solved with two layer networks, but the theoretical explanation of this is not complete. Some initial theoretical work has been done, and we will describe a couple of these explanations here. The main point of both explanations is that deeper networks can produce more efficient realizations of some complex functions than two layer networks can.

One approach to understanding the benefits of deep networks was taken by Telgarsky. [Telgarsky, 2016] He compared the effect of increasing the number of neurons in a single hidden layer with increasing the number of layers. This involved describing the operation of creating composite functions by cascading functions together. A full description of his theory is beyond the scope of this book, but a simple example can illustrate the concept.

Suppose that we use the network of Figure 2.10, but we change the weights to the following:

$$\mathbf{W}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0 \\ -0.5 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} 2 & -4 \end{bmatrix}, \mathbf{b}^2 = [0] \qquad (2.6)$$

This creates the triangle function shown in Figure 2.15. Now, create a composite function by cascading this function with itself: $f \circ f(p) = f(f(p))$. This process is illustrated in Figure 2.16. The blue line represents the original triangle function $f(p)$. The green line at 45° represents p. The red line represents the composite function $f \circ f(p)$. In Figure 2.16(a) we are evaluating the composite function at $p = 0.2$. We first find $f(0.2)$, which is 0.4. This is then input to $f()$ again, which produces 0.8, which means that $f \circ f(0.2) = 0.8$, which falls on the red line. In Figure 2.16(b) the process is repeated for $p = 0.8$.

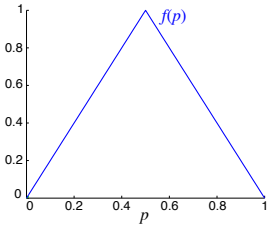
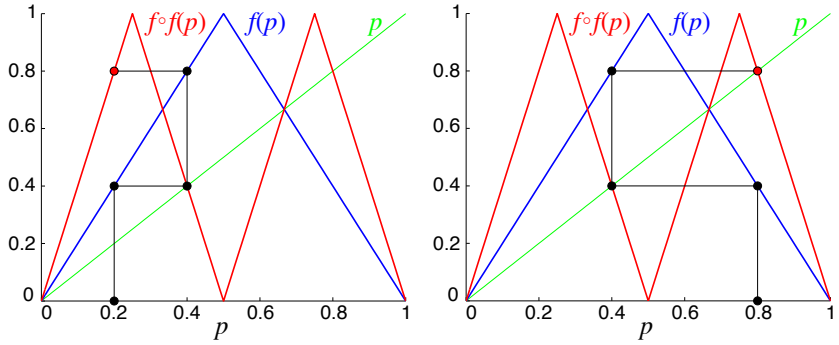


Figure 2.15: Linear Decision Boundary



(a) p=0.2 (b) p=0.8

Figure 2.16: Composite function

By cascading the single triangle function, we have created a function with two triangles. It may seem that this requires a four layer network, since we have cascaded two networks, each with two layers. However, since the second layer is linear, it can be merged with the *poslin* layer that follows it, so the resulting network only requires three layers. (The weights of the corresponding three layer network are shown in Eq. 2.7.) Therefore, by adding a single layer to the network we have doubled the complexity of the function we have created.

$$\mathbf{W}^1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 0 \\ -0.5 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} 2 & -4 \\ 2 & -4 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} 0 \\ -0.5 \end{bmatrix}, \mathbf{W}^3 = \begin{bmatrix} 2 & -4 \end{bmatrix}, \mathbf{b}^3 = [0] \tag{2.7}$$

The function shown in Figure 2.16 could be created with a two-layer network, but it would require doubling the number of neurons in the hidden layer (since each *poslin* neuron creates one linear segment). This difference in complexity becomes more pronounced

as we increase the number of layers in the network. Each time we add a layer (which corresponds with a linear increase in the number of neurons), we double the number of neurons that would be required for a two layer network to implement the same function.

To experiment with increasing the number of layers in a network, use the Deep Learning Demonstration Cascaded Function (**dl2cf**).

The concept demonstrated in the above example is also used by [Montufar et al., 2014], who show that in deep networks with multiple inputs the number of distinct linear regions in the input space is geometrically larger than for two-layer networks with a similar number of neurons. The take-away is that two-layer networks can create any function that can be created with a deep network. However, the number of neurons required in the two-layer network increase geometrically, while the number of neurons increase linearly in the equivalent deep network.

Another approach for demonstrating the efficiency of deep networks was used by [Eldan and Shamir, 2016]. They also showed that to achieve the equivalent power of approximation of a deep network requires that the number of neurons in the hidden layer of a two-layer network must grow geometrically. They demonstrate this by considering radial functions – functions whose response depends only on the distance of the input from the origin. Even simple radial functions cannot be efficiently approximated by two-layer networks, but can be easily approximated by three-layer networks.

The basic idea is that a radial function depends only on the norm of the input. Therefore, it can be approximated using a three-layer network by first using the first two layers to approximate the squared norm function, and then using the last layer to approximate the function acting on the norm. (To be more precise, the second layer of the three layer network combines parts of both approximations.)

To demonstrate this concept, consider the radial function

$$g(\mathbf{p}) = \cos(\pi \|\mathbf{p}\| / 2), \tag{2.8}$$

which is shown in Figure 2.17. To approximate the squared norm function $\|\mathbf{p}\|^2 = p_1^2 + p_2^2$, we can start with the simple square function p^2. We can use a two-layer network with *logsig* neurons in the first layer and a linear neuron in the second layer. To approximate the square function over the range $-2 < p < 2$, the weights for this network could be

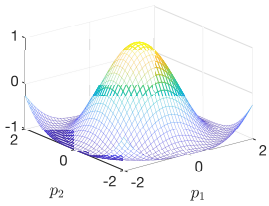


Figure 2.17: Radial Function $\cos(\pi \|\mathbf{p}\| /2)$

$$\mathbf{W}^1 = \begin{bmatrix} -0.2 \\ -0.2 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 2 \\ -2 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} -320 & 320 \end{bmatrix}, \mathbf{b}^2 = [243.7] \quad (2.9)$$

The network response and the square function are shown in Figure 8.4a. Even with only two hidden layer neurons, the approximation is almost exact over the range $-2 < p < 2$. We can now stack two of these networks, with p_1 as the input to the top network and p_2 as the input to the bottom network, and sum the outputs to create the function $p_1^2 + p_2^2$. This would be input to another network that would approximate the function $\cos(\pi\sqrt{p}/2)$, so that the total network response approximates $\cos(\pi \|\mathbf{p}\| /2)$.

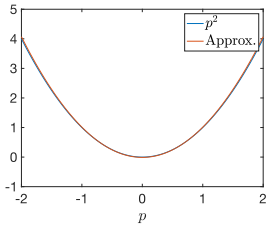


Figure 2.18: Square Function and Network Approximation

For the second network, we need to approximate $\cos(\pi\sqrt{p}/2)$ over the range $-8 < p < 8$, since the input to this network is $p_1^2 + p_2^2$. The weights that could approximate this function are given by Eq. 2.10, and the approximation is shown in Figure 2.19.

$$\mathbf{W}^1 = \begin{bmatrix} -0.2489 \\ -0.3541 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 1.2931 \\ -2.6491 \end{bmatrix}, \mathbf{W}^2 = \begin{bmatrix} -6.6898 & 69.3431 \end{bmatrix}, \mathbf{b}^2 = [1.6693] \quad (2.10)$$

Putting the smaller networks together, and combining the linear output layers of the first (square) networks with the input layers of the $\cos(\pi\sqrt{p}/2)$ network, produces a three-layer network with architecture 2-4-2-1. The weights are given by Eq. 2.11.

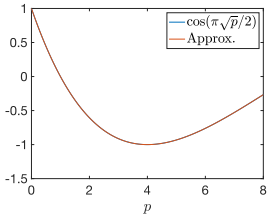


Figure 2.19: $\cos(\pi\sqrt{p}/2)$ and Network Approximation

$$\begin{aligned} \mathbf{W}^1 &= \begin{bmatrix} -0.2 & 0.0 \\ -0.2 & 0.0 \\ 0.0 & -0.2 \\ 0.0 & -0.2 \end{bmatrix}, \mathbf{b}^1 = \begin{bmatrix} 2 \\ -2 \\ 2 \\ -2 \end{bmatrix} \\ \mathbf{W}^2 &= \begin{bmatrix} 79.648 & -79.648 & 79.648 & -79.648 \\ 113.312 & -113.312 & 113.312 & -113.312 \end{bmatrix}, \mathbf{b}^2 = \begin{bmatrix} -120.0208 \\ -175.2374 \end{bmatrix} \\ \mathbf{W}^3 &= \begin{bmatrix} -6.6898 & 69.3431 \end{bmatrix}, \mathbf{b}^3 = [1.6693] \end{aligned} \quad (2.11)$$

The resulting three-layer network approximation of the radial function is shown in Figure 2.20. With three layers, even a simple network is able to capture the radial form. However, doing this with a two-layer network is much more difficult, especially if the radial form is more complex.

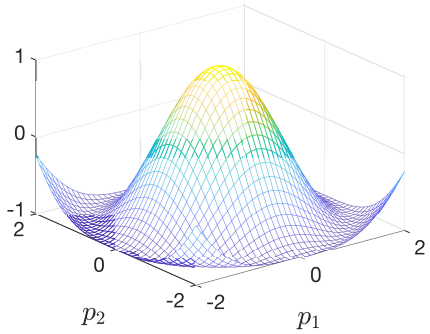


Figure 2.20: Three-Layer Network Approximation of $\cos(\pi \|\mathbf{p}\| /2)$

The reason that approximating radial functions is more difficult with two-layer networks can be seen if we consider the operation of the simple two-layer network in Figure 2.10. The individual neuron responses from the first layer are combined in the second layer. The total network response is a weighted sum of single neuron responses. With a single input, each neuron response takes form of the activation function. With multiple inputs, the neuron response also takes the form of the activation function, but along the direction of the corresponding row of the weight matrix. In directions orthogonal to the weight direction, the neuron response is constant, forming a ridge.

The response of a single *logsig* neuron with two inputs is illustrated in Figure 2.21. This neuron has a weight $\mathbf{W} = \begin{bmatrix} -5 & 5 \end{bmatrix}$ and a zero bias. The arrow in the figure points in the direction of the weight. The key idea is that the response does not change orthogonal to the weight direction, which creates a ridge function. It is difficult to create radial functions by superimposing ridge functions, since they have fundamentally different characteristics. Radial functions are constant along contours that are equal distance from some origin, and ridge functions are constant along contours that are orthogonal to the weight vector. Approximating a radial function with a combination of ridge functions is a little like approximating a circle with a series of line segments. It can be done, but it is not very efficient.

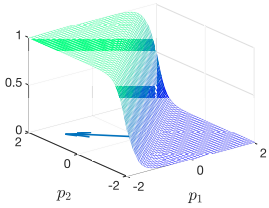


Figure 2.21: Single *logsig* Neuron Response

This is illustrated in the following figures. Figure 2.22 shows the function $\cos(\pi \|\mathbf{p}\| /2)$, a three-layer approximation of the function, and the corresponding error. The network used here is even simpler than the one described earlier, since there are only three neurons in the first layer. All weights were adjusted together to produce the approximation.

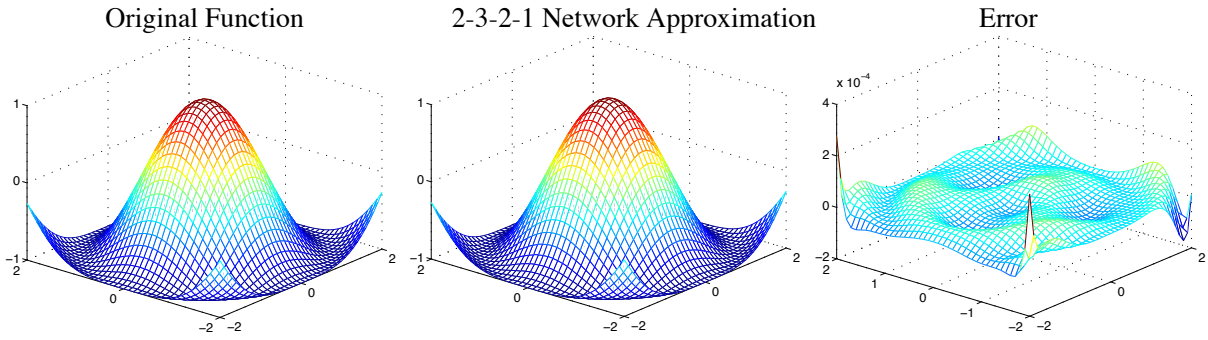


Figure 2.22: Three-layer network approximation of $\cos(\pi \|\mathbf{p}\| /2)$

Figure 2.23 shows approximations obtained with two-layer networks, and their corresponding errors. The far left subfigure shows the approximation obtained by a 2-5-1 network. This network has 21 parameters (weights and biases), which compares with the 20 parameters of the 2-3-2-1 network demonstrated in Figure 2.22. Even though it has the same number of parameters, the two-layer network approximation is much worse. Even the 2-7-1 network in the right subfigure, which has 29 parameters, does not achieve as accurate an approximation as the three-layer network with 20 parameters.

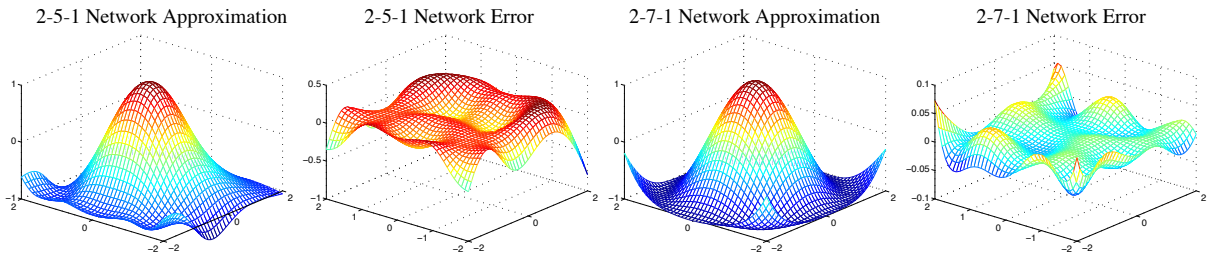


Figure 2.23: Two-layer network approximation of $\cos(\pi \|\mathbf{p}\| /2)$

To experiment with two- and three-layer network approximations, use the Deep Learning Demonstration Function Approximation (**`dl2fa`**).

Epilogue

This chapter has reviewed the basic structure and operation of the multilayer network. The basic concepts used in deep networks are the same as those for shallow networks. This why much of this chapter has reviewed material from Chapters 2 and 11 of NND2.

One important modification to multilayer networks when used for deep learning is the use of the *poslin*, or ReLU activation function in the hidden layers. This function was described in NND2, but was used principally for the Hamming network. The *poslin* function has become important for deep networks, because of issues with the sigmoid activation function when the number of layers in the multilayer network becomes large. This is because the sigmoid output is limited in magnitude, and therefore its derivative becomes vanishingly small as the net input gets larger. We will have more to say about this in the next chapter.

This chapter also demonstrated how deep networks can be more efficient in approximating certain types of functions than two-layer networks. Essentially, the power of approximation of a multilayer network increases linearly as you increase the number of neurons in a single hidden layer, while the power increases geometrically as you increase the number of layers.

In the next chapter we will discuss the training of the deep multilayer network. We will review some of the concepts from Chapter 11 and 12 of NND2, since the basic training concepts are the same, no matter how many layers are involved. However, we will highlight some recent modifications that are well-suited to deep networks.

Further Reading

[Glorot et al., 2011] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011

This paper describes the use of the ReLU (*poslin*) activation function in the hidden layers of deep networks. The output of this function is equal to zero for negative inputs and equal to the input for positive inputs. This is useful when training networks with many layers, because it prevents the vanishing gradient problem that occurs when using sigmoid activation functions.

[Telgarsky, 2016] Matus Telgarsky. Benefits of depth in neural networks. In *Proceedings of the 29th Annual Conference on Learning Theory*, pages 1517–1539, 2016

Telgarsky proves that it takes a geometric increase in the number of neurons in the hidden layer of a two-layer network to approximate a function that a deep network can approximate with a linear increase in the number of layers.

[Montufar et al., 2014] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pages 2924–2932, 2014

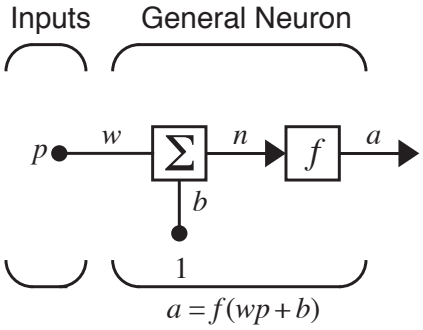
This work is related to Telgarsky's. They show that in deep networks with multiple inputs the number of distinct linear regions in the input space is exponentially larger than for two-layer networks with a similar number of neurons.

[Eldan and Shamir, 2016] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016

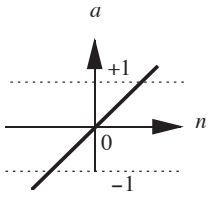
This paper also shows that to achieve the equivalent power of approximation of a deep network requires that the number of neurons in the hidden layer of a two-layer network must grow geometrically. They demonstrate this by considering radial functions – functions whose response depends only on the distance of the input from the origin. Even simple radial functions cannot be efficiently approximated by two-layer networks, but can be easily approximated by three-layer networks.

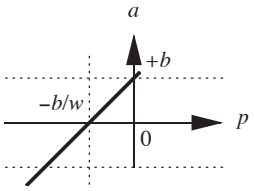
Summary of Results

Single Input Neuron



Linear Activation Function





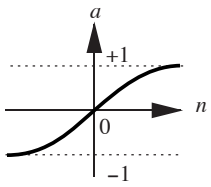
$a = purelin(n)$

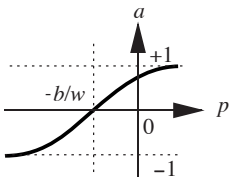
Linear Function

$a = purelin(wp + b)$

Single-Input *purelin* Neuron

Hyperbolic Tangent Activation Function





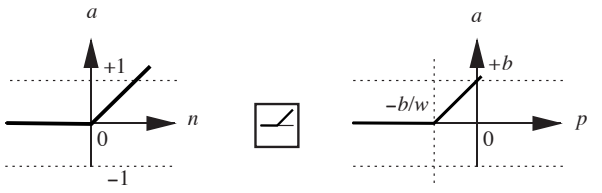
$a = tansig(n)$

Tan-Sigmoid Function

$a = tansig(wp + b)$

Single-Input *tansig* Neuron

Positive Linear (or ReLU) Activation Function



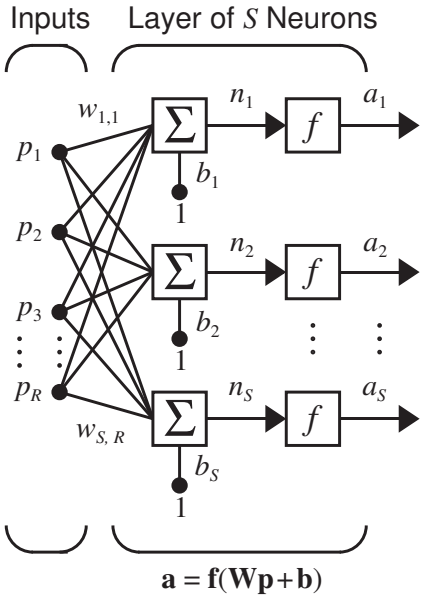
$a = poslin(n)$
Positive Linear Function

$a = poslin(wp + b)$
Single-Input *poslin* Neuron

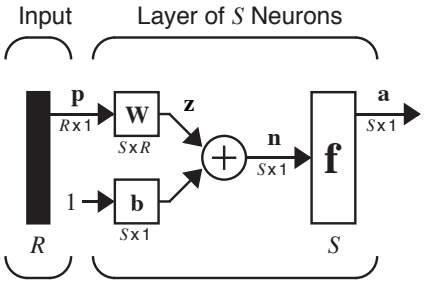
Softmax Activation Function

$$a_i = f_i(\mathbf{n}) = \frac{e^{n_i}}{\sum_{j=1}^{S} e^{n_j}}$$

Layer of Neurons

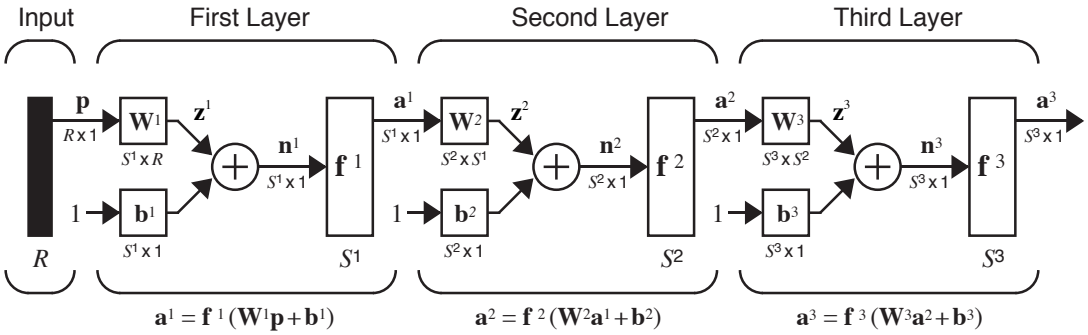


Matrix Notation of Neuron Layer



$$\mathbf{a} = \mathbf{f}(\mathbf{W}\mathbf{p}+\mathbf{b})$$

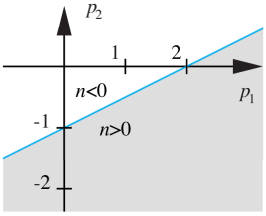
Three Layer Network



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

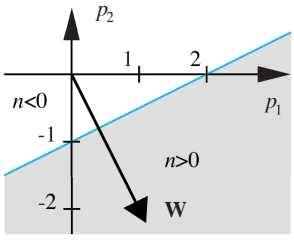
$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Solved Problems



P2.1 **Recall the network shown in Figure 2.8. Find the weights and bias needed for that network to create the decision boundary shown in the adjacent figure.**

As described in Chapter 4 of NND2, the weight should be orthogonal to the decision boundary. In this case, the weight $\mathbf{W} = \begin{bmatrix} 1 & -2 \end{bmatrix}$ would be orthogonal to the boundary, as shown in the figure below.

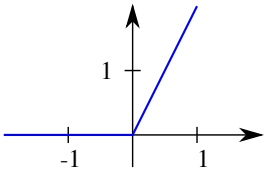


To find the bias, we can solve the equation $n = \mathbf{W}\mathbf{p} + b = 0$ or $b = -\mathbf{W}\mathbf{p}$ at a point on the boundary. If we pick the point $\mathbf{p} = \begin{bmatrix} 0 & -1 \end{bmatrix}^T$, we find that $b = -2$.

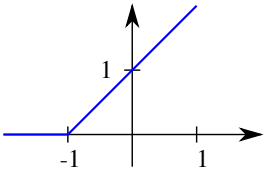
P2.2 **Consider again the two-layer *poslin* network of Figure 2.10. Assume the weights and biases are $\mathbf{W}^1 = \begin{bmatrix} 2 & 1 \end{bmatrix}^T, \mathbf{b}^1 = \begin{bmatrix} 0 & 1 \end{bmatrix}^T, \mathbf{W}^2 = \begin{bmatrix} 1 & -1 \end{bmatrix}, \mathbf{b}^2 = [0]$.**

i. **Plot the response of each neuron in the first layer as the input varies in the range $-1 < p < 1$.**

ii. **Plot the total network reponse a^2 as the input varies in the range $-1 < p < 1$.**



(a) Neuron 1

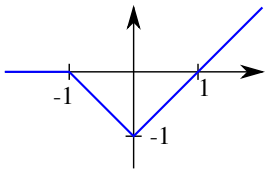


(b) Neuron 2

i. For neuron 1 in the first layer, the neuron net input is $n_1^1 = 2p + 0$. This is a line with slope of 2 that passes through the origin. To get the neuron output, we pass this through the *poslin* activation function, which sets all negative values to zero. The resulting function is zero for $p < 0$, and is equal to $2p$ for $p \geq 0$. For neuron 2 in the first layer, the neuron net input is $n_1^1 = p + 1$. This is a line with slope 1 that is equal to zero when $p = -1$. If we pass this through

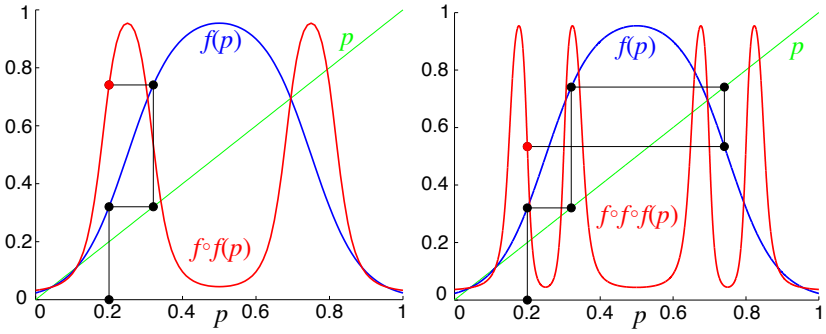
the *poslin* function, it will be zero for $p < -1$, and then it will have a slope of $+1$ starting at $p = -1$. (Note that the breakpoint in the *poslin* function is at $p = -b/w$.) The two functions are shown in the margin.

ii. The second layer weights are 1 and -1, which means that the second neuron output is subtracted from the first neuron output. Since the bias in the second layer is zero, the output is not shifted up or down. Since the first neuron output is zero until zero, the total response will be the negative of the second neuron response until $p = 0$. For $p > 0$, the negative of the neuron two response will have a slope of -1, and the first neuron response will have a slope of $+2$, so the total response will have a slope of +1. The total response is shown in the margin.



P2.3 **The example illustrated in Figure2.16 showed how adding layers of *poslin* neurons created functions of geometrically increasing complexity. How would these functions change if *sigmoid* activation functions replaced the *poslin* functions? What would happen to the total network response as the number of layers became very large? Explain any benefits of using the *poslin* function when the number of layers is large.**

Using the demonstration (**`dl2cf`**), we can substitute the *logsig* activation function for the *poslin* activation function. The result is shown in the following figure for three-layer (left figure) and four-layer (right figure) networks. The key difference between this response and Figure2.16 is that when a new *logig* layer is added, not only is the two-layer function repeated, but the pulses become narrower. This is caused by the shape of the *sigmoid* function. If we were to continue to add layers (assuming the same weights were used), we would get more pulses, but the pulses would become narrower. This means that there will be more regions where the slope of the function will be small. This is related to the vanishing gradient problem, which is discussed in the next chapter.

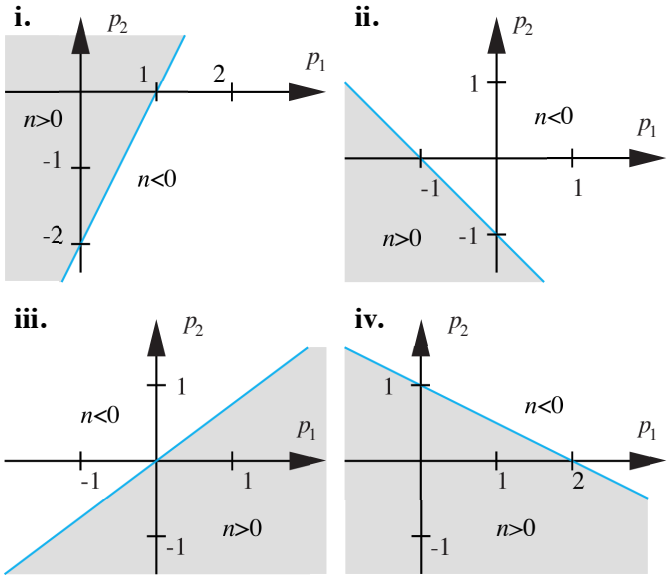


(a) Three-Layer Reponse

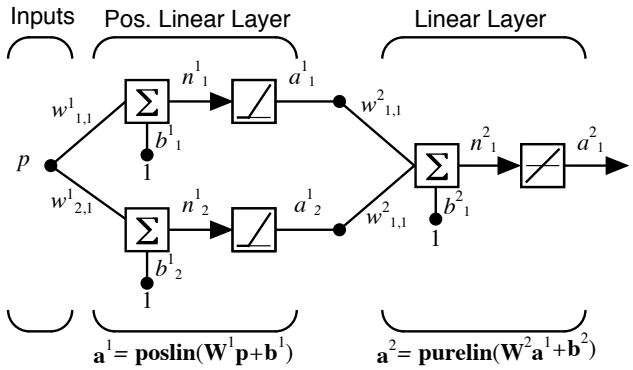
(b) Four-Layer Response

Exercises

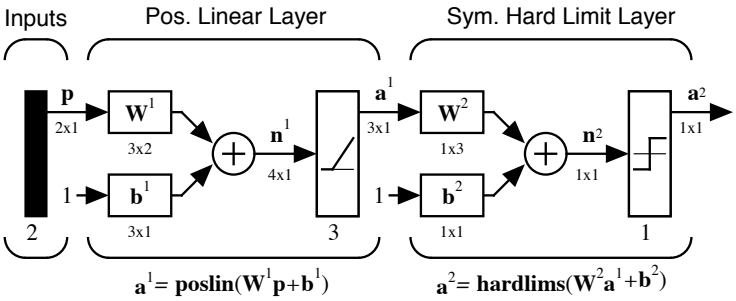
E2.1 For the network shown in Figure 2.8, find the weights and bias needed to create the decision boundaries shown in the following figure.

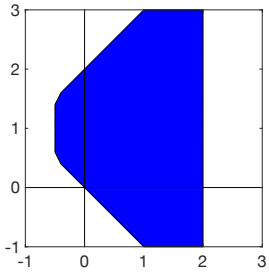


E2.2 Sketch the response of the following network when the weights and biases are $\mathbf{W}^1 = \begin{bmatrix} -1 & 1 \end{bmatrix}^T, \mathbf{b}^1 = \begin{bmatrix} 0.5 & 1 \end{bmatrix}^T, \mathbf{W}^2 = \begin{bmatrix} 1 & 1 \end{bmatrix}, \mathbf{b}^2 = [-1]$.



E2.3 For the following network, find the weights and biases to create the adjacent decision boundaries.





E2.4 Consider the following training data (XOR).

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = 1\right\}, \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-1\end{bmatrix}, t_2 = 1\right\}, \left\{\mathbf{p}_3 = \begin{bmatrix}1\\1\end{bmatrix}, t_3 = -1\right\}, \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\-1\end{bmatrix}, t_4 = -1\right\}$$

i. You want to use the two-layer poslin network shown below, with two neurons in the hidden layer, to classify these input vectors correctly. Describe what kind of decision regions you can create with this network. Give some examples.

ii. Draw a decision region that this network can create, and which can correctly classify the training vectors.

iii. Find the weights and biases that would create that decision region.

iv. Test your network by applying the four input vectors and demonstrating that the outputs match the targets.