EE 610, Selected Topics: Machine Learning Fundamentals

Neural Networks

Dr. W. D. Pan

Dept. of ECE
Univ. of Alabama in Huntsville

Topics

- Background and History
- Perceptron
- Fully Connected Neural Network
- Backpropagation Methods
- Convolutional Neural Network
- Deep Learning

Background

- **Neural networks** are models that use a multitude of elemental nonlinear computing elements (called artificial neurons), organized as networks whose interconnections are similar in some respects to the way in which neurons are interconnected in the visual cortex of mammals.
- These models are referred to by various names, including neural networks, neurocomputers, parallel distributed processing models, neuromorphic systems, layered self-adaptive networks, and connectionist models.
- Here, we use the name **neural networks**, or **neural nets** for short.
- We use these networks as vehicles for adaptively learning the parameters of decision functions via successive presentations of training patterns.

Brief History

- Interest in neural networks dates back to the early 1940s, as exemplified by the work of McCulloch and Pitts, who proposed neuron models in the form of binary thresholding devices, and stochastic algorithms involving sudden 0–1 and 1–0 changes of states, as the basis for modeling neural systems.
- Subsequent work by Hebb in 1949 was based on mathematical models that attempted to capture the concept of learning by reinforcement or association.
- During the mid-1950s and early 1960s, a class of so-called learning machines originated by Frank Rosenblatt caused a great deal of excitement among researchers and practitioners of pattern recognition.
- The reason for the interest in these machines, called **perceptrons**, was the development of mathematical proofs showing that perceptrons, when trained with linearly separable training sets (i.e., training sets separable by a hyperplane), would converge to a solution in a finite number of iterative steps.
- The solution took the form of parameters (coefficients) of hyperplanes that were capable of correctly separating the classes represented by patterns of the training set.

Rise of Deep Learning

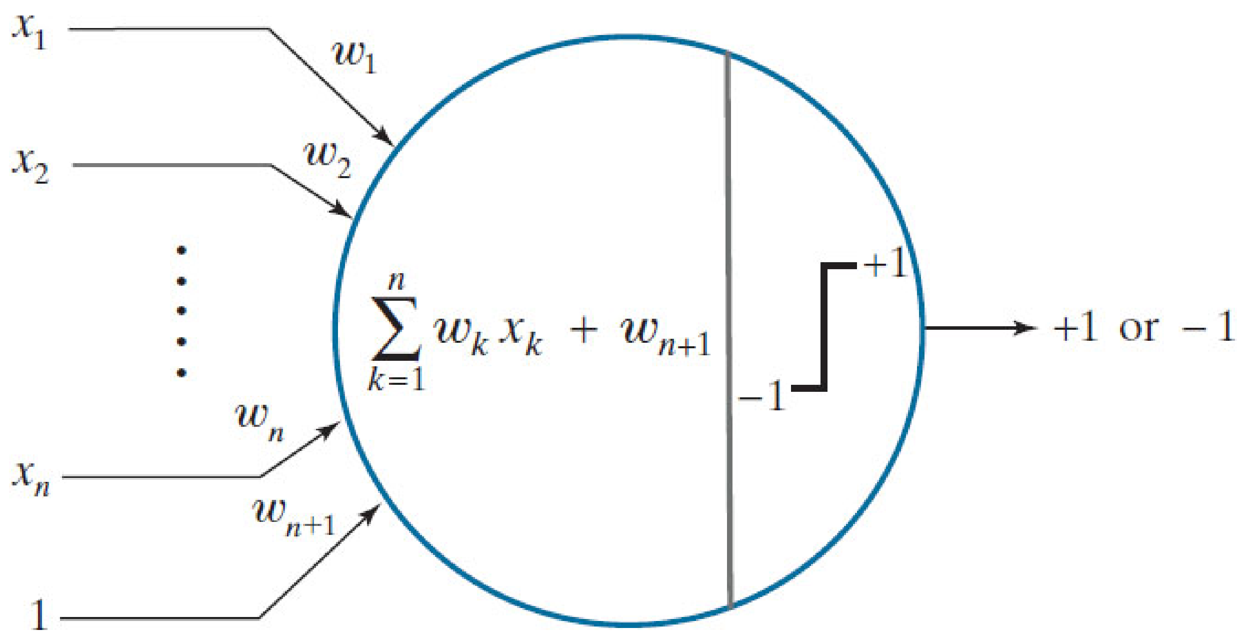
- Unfortunately, the basic perceptron, and some of its generalizations, were found to be inadequate for most pattern recognition tasks of practical significance.
- Subsequent attempts to consider multiple layers of perceptrons lacked effective training algorithms.
- In 1986, Rumelhart, Hinton, and Williams proposed an effective training method via backpropagation for multilayer networks. Although this training algorithm cannot be shown to converge to a solution in the sense of the proof for the single-layer perceptron, backpropagation is capable of generating results that have revolutionized the field of pattern recognition.
- Neural networks can now use backpropagation to automatically learn representations suitable for recognition, starting with raw data. Each layer in the network "refines" the representation into more abstract levels. This type of multilayered learning is commonly referred to as **deep learning**.

Limitations of Deep Learning

- Deep learning has been shown to be highly successful in many practical applications generally associated with large data sets, due to its capability to learn features automatically.
- However, deep learning models are not “magical” systems that assemble themselves. Human intervention is still required for specifying parameters, e.g., the number of layers, the number of artificial neurons per layer, and various coefficients that are problem dependent.
- Teaching proper recognition to a complex multilayer “deep” neural network is less a science than an art, which requires considerable knowledge, experience and experimentation on the part of the designer.
- A great many applications of pattern recognition, especially in constrained environments, are best handled by more “traditional” methods.
- Deep learning, as a huge **black-box** model, remains difficult to diagnose as to **explain** what aspects of the model drive the decisions. In many real-world domains, from legislation and law enforcement to healthcare, such diagnosis is essential to ensure that AI system decisions are driven by aspects appropriate in the context of its use.

Learning Machines

- The vast body of work for neural network is rapidly evolving.
  - For example, the development of methods and studies enabling the explanation of an deep learning based AI system is an active, broad area of research.
- The focus of this course is on fundamentals of theory (mathematical underpinning) and algorithms.
- We will illustrate the foundation of how neural nets are trained, and how they operate after training.
- We will begin by discussing perceptrons, which are simple learning machines.
- Although perceptrons are not used *per se* in state-of-the-art neural network architectures, the operations they perform are almost identical to artificial neurons, which are the basic computing units of neural nets.



Schematic of a perceptron, showing the operations it performs.

Preliminaries

- Inputs
  - An input vector is the data given as one input to the algorithm. Written as $\boldsymbol{x}$, with elements x_i, where i runs from 1 to the number of input dimensions, m.
- Weights
  - w_{ij} , are the weighted connections between nodes i and j. For neural networks, these weights are analogous to the synapses in the brain. They are arranged into a matrix $\boldsymbol{W}$.
- Outputs
  - The output vector is y, with elements y_j , where j runs from 1 to the number of output dimensions, n. We can write $y(x, W)$ to show that the output depends on the inputs to the algorithm and the current set of weights of the network.

- Activation Function
  - For neural networks, $h(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs, such as the threshold function.
- Error
  - E, a function that computes the inaccuracies of the network as a function of the outputs y and targets t.

Linear Decision Boundary

- A single perceptron unit learns a linear boundary between two linearly separable pattern classes.
- A linear boundary in 2-D is a straight line with equation $y = ax + b$, where the y-intercept parameter b is to displace the line from the origin without affecting its slope. For this reason, this “floating” coefficient that is not multiplied by a coordinate is often referred to as the **bias**, the bias coefficient, or the bias weight.
- Generally, we work with patterns in much higher dimensions than two. For a point in n dimensions, the test would be against a hyperplane, whose equation is

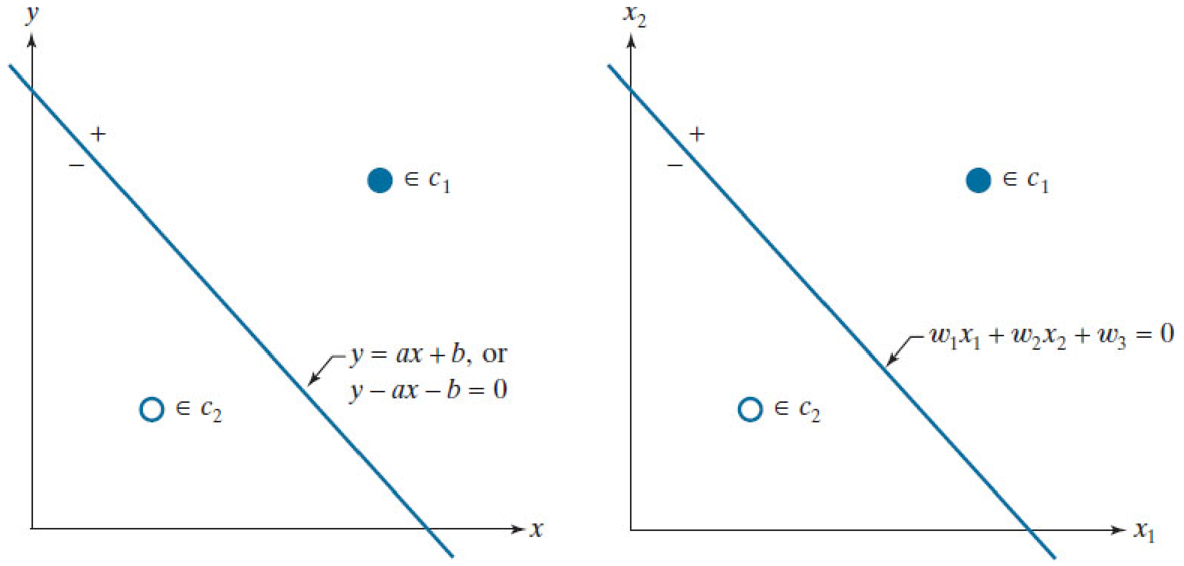
$$w_1x_1 + w_2x_2 + \cdots + w_nx_n + w_{n+1} = 0$$

or in the vector form:

$$\boldsymbol{w}^T\mathbf{x} + w_{n+1} = 0$$

Perceptron

- A single perceptron learns a linear boundary between two linearly separable pattern classes.



(a) The simplest two-class example in 2-D, showing one possible decision boundary out of an infinite number of such boundaries. (b) Same as (a), but with the decision boundary expressed using more general notation.

Test for Decision

- Given any pattern vector **x** from a vector population, we want to find a set of weights with the property

$$\boldsymbol{w}^T\mathbf{x} + w_{n+1} = \begin{cases} > 0 & \text{if } \mathbf{x} \in c_1 \\ < 0 & \text{if } \mathbf{x} \in c_2 \end{cases}$$

We can simplify the equation if we add a 1 at the end of every pattern vector and include the bias in the weight vector.

$$\boldsymbol{w}^T\mathbf{x} = \begin{cases} > 0 & \text{if } \mathbf{x} \in c_1 \\ < 0 & \text{if } \mathbf{x} \in c_2 \end{cases}$$

where

$$\mathbf{x} \triangleq [x_1, x_2, \ldots, x_n, 1]^T$$

$$\boldsymbol{w} \triangleq [w_1, w_2, \ldots, w_n, w_{n+1}]^T$$

Perceptron Training Algorithm

- Let $\alpha > 0$ denote a correction increment (also called the learning increment or the **learning rate**)
- Let the initial weight vector $\boldsymbol{w}(1)$ take arbitrary values. Then, repeat the following steps for $k = 2, 3, \dots,$:

  For an augmented pattern vector, $\mathbf{x}(k)$, at step k,

If $\mathbf{x}(k) \in c_1$ and $\boldsymbol{w}^T(k)\mathbf{x}(k) \leq 0$, let

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \alpha\mathbf{x}(k)$$

If $\mathbf{x}(k) \in c_2$ and $\boldsymbol{w}^T(k)\mathbf{x}(k) \geq 0$, let

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) - \alpha\mathbf{x}(k)$$

Otherwise, let

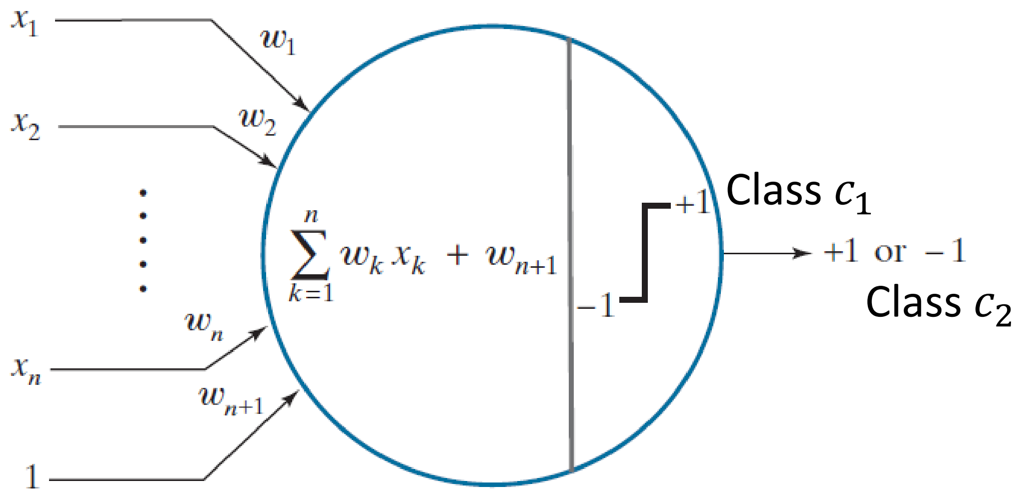
$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k)$$

$$\boxed{\boldsymbol{w}^T\mathbf{x} = \begin{cases} > 0 & \text{if } \mathbf{x} \in c_1 \\ < 0 & \text{if } \mathbf{x} \in c_2 \end{cases}}$$

Schematic of a Perceptron

- The perceptron performs a sum of products of an input pattern using the weights and bias found during training.
- The output of this operation is a scalar value that is then passed through an **activation function** (called a hard-limit transfer function in Matlab) to produce the unit's output.
- For the perceptron, the activation function is a thresholding function.

- Values 1 and 0 sometimes are used to denote the two possible states of the output (e.g., in Matlab perceptron() function)



Convergence

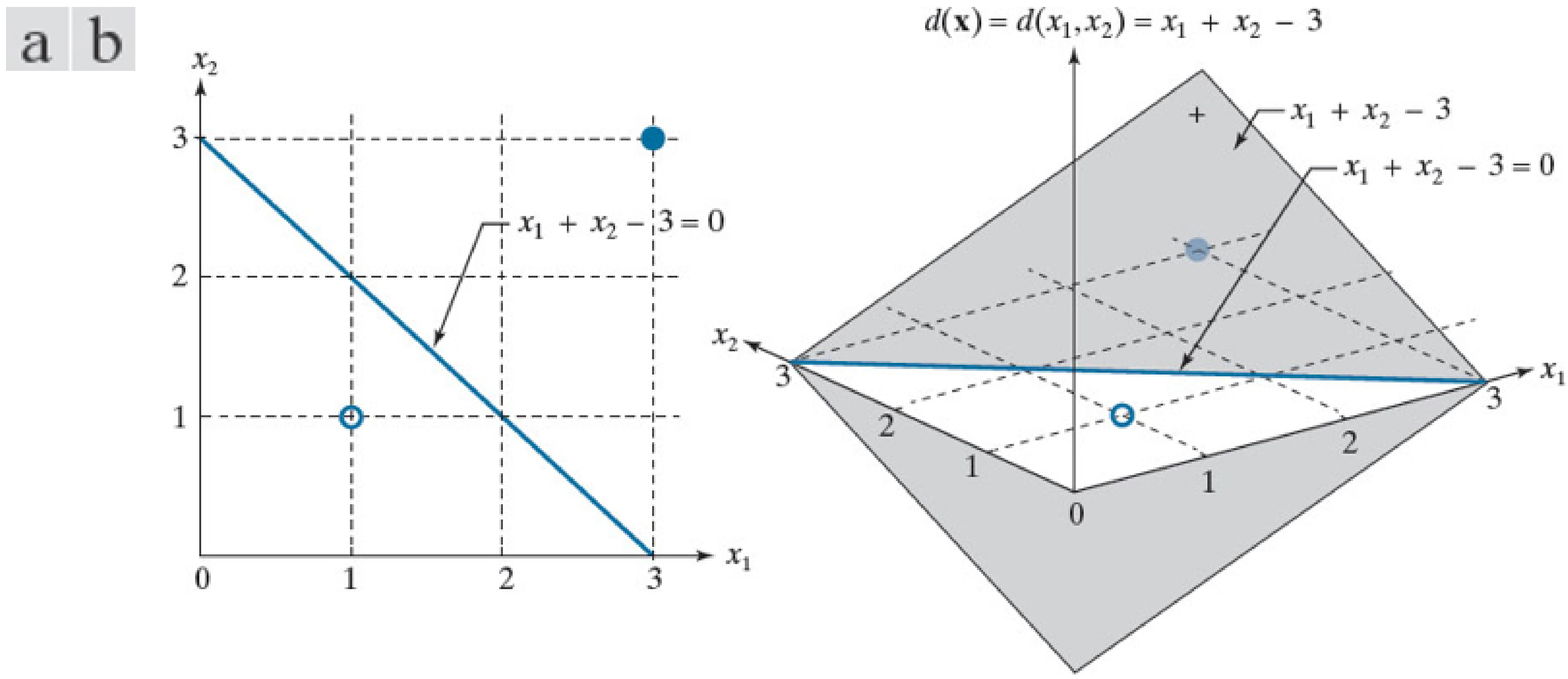
- the perceptron convergence theorem states that if the training data set is linearly separable, then the perceptron learning algorithm is guaranteed to find an exact solution in a finite number of steps.
- However, the number of steps required to achieve convergence could still be substantial, and in practice, until convergence is achieved, we will not be able to distinguish between a non-separable problem and one that is simply too slow to converge.
- Even when the data set is linearly separable, there may be many solutions, and the solutions eventually found will depend on the initialization of the parameters and on the order of presentation of the data points.
- For data sets that are not linearly separable, the perceptron learning algorithm will never converge.

perceptron_demo.m

```
% Class 1: [3 3 1]
% Class 2: [1 1 1]

% learning rate
a = 1;

x1 = [3 3 1];
x2 = [1 1 1];

% initial weight vector
w = [0 0 0];
```

```
for iter = 1: 20
    w_prev = w;
    x = x1;
    y = dot(w, x);

    if (y <= 0)
        w = w + a*x;
    end

    x = x2;
    y = dot(w, x);

    if (y >= 0)
        w = w - a*x;
    end

    if (w == w_prev)
        break;
    end
end

iter
w

dot(w, x1)
dot(w, x2)
```

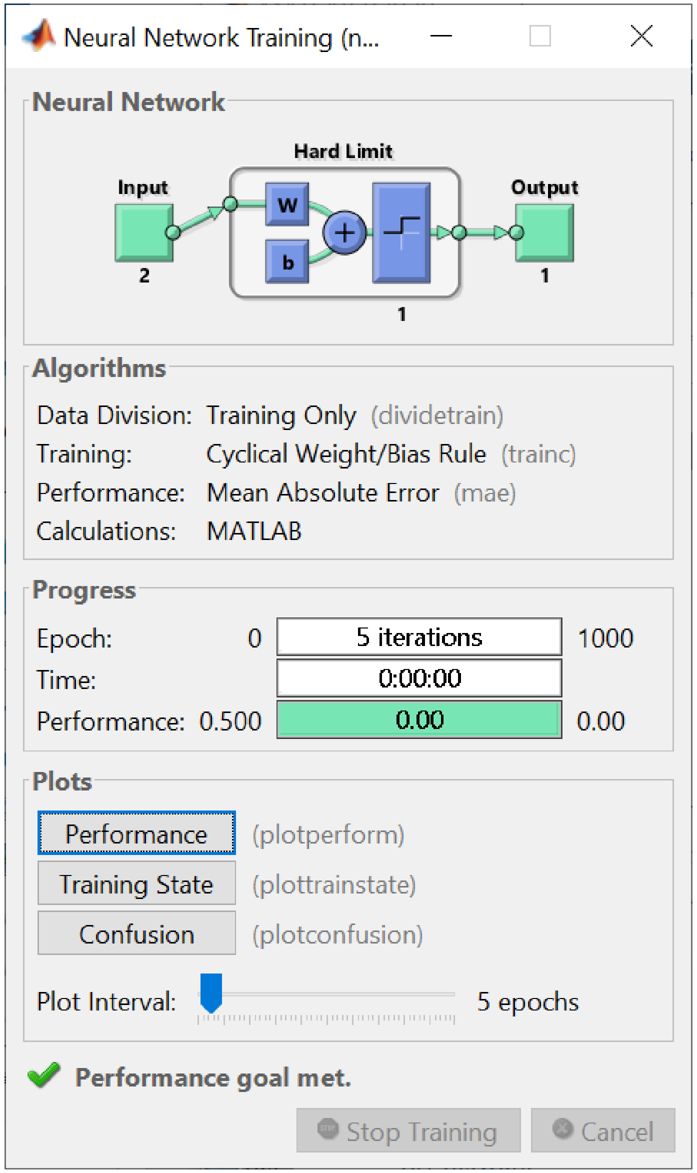
```
iter =
   6
w =
   1   1  -3
ans =
   3
ans =
  -1
```



(a) Segment of the decision boundary learned by the perceptron algorithm. (b) Section of the decision surface. The decision boundary is the intersection of the decision surface with the with the x_1x_2-plane.

perceptron()

```
% Do not use the augmented input vector
x1 = [3 3];
x2 = [1 1];

% The input matrix:
%(vert: features, horz: samples)
x = [x1' x2'];

% Target has to be 0/1 values for binary
classification
target = [0 1];
method = perceptron;
net = train(method, x, target);

% View the weights for the connection from
the first input to the first layer
net.iw{1,1}
% View the bias values for the first layer
net.b{1}

y = net(x);
error = y - target
```


Neural Network Training (n...
Neural Network
Hard Limit
Input
2
W
b
1
Output
1
Performance (plotperform)
Training State (plottrainstate)
Confusion (plotconfusion)
Stop Training
Cancel

Algorithms

Data Division:	Training Only (dividetrain)
Training:	Cyclical Weight/Bias Rule (trainc)
Performance:	Mean Absolute Error (mae)
Calculations:	MATLAB

Progress

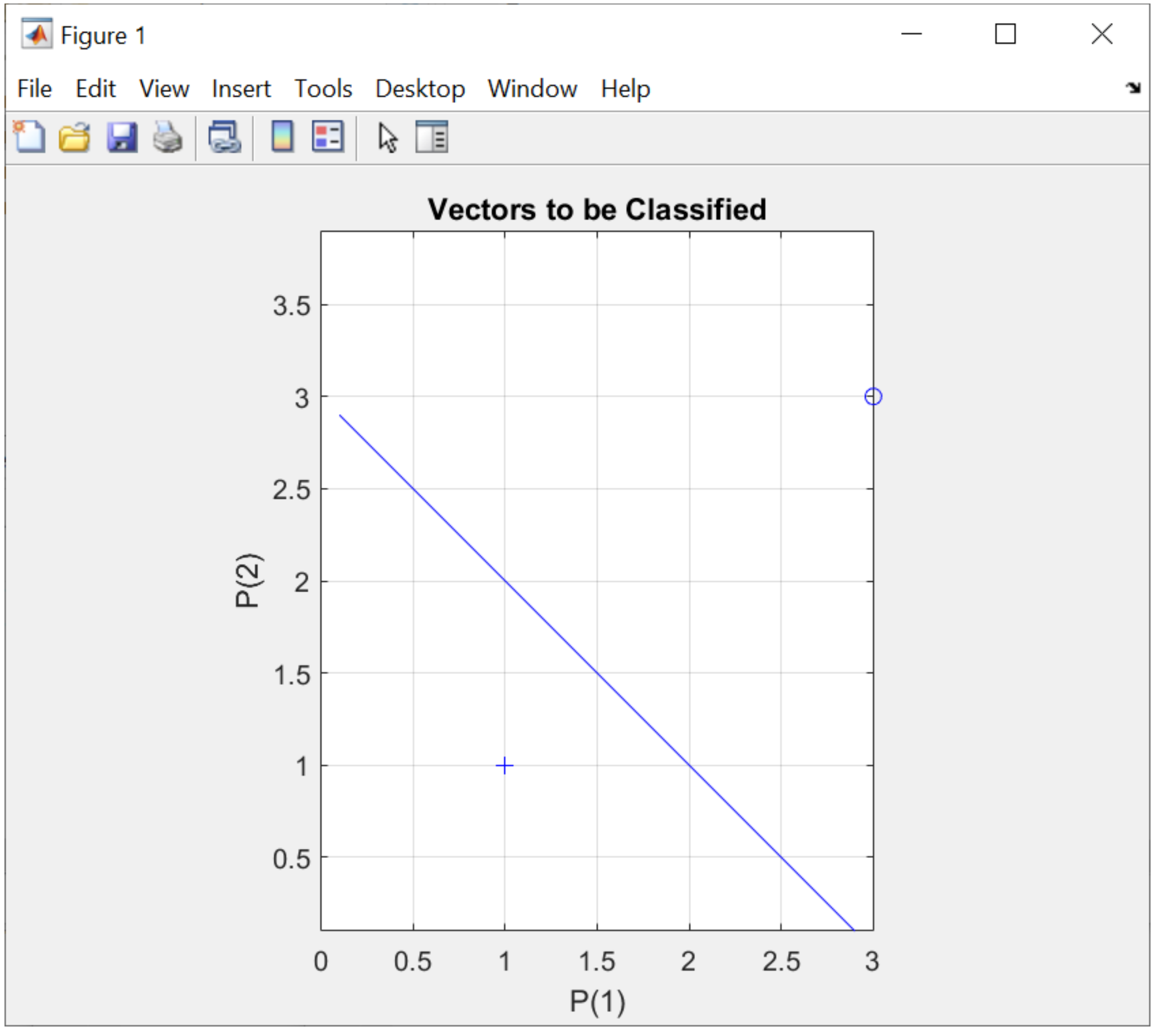
Epoch:	0	5 iterations	1000
Time:		0:00:00	
Performance:	0.500	0.00	0.00

Plot Interval: 5 epochs

✓ Performance goal met.

plotpv and plotpc functions

```
figure;
hold on;
plotpv(x,target);
plotpc(net.iw{1},net.b{1});
axis equal
grid
```

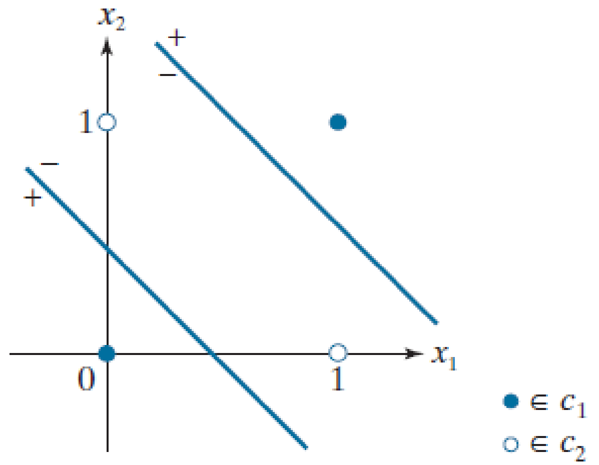


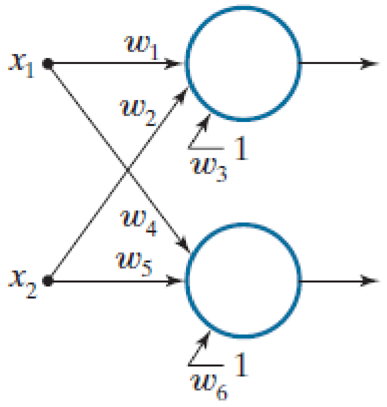
sklearn

```
import numpy as np
from sklearn.linear_model import Perceptron

x1 = np.array([3, 3])
x2 = np.array([1, 1])

X = np.vstack((x1, x2)) # Features are along the row
y = np.array([1,2])

clf = Perceptron()
clf.fit(X, y)
clf.coef_
clf.intercept_
clf.score(X, y)
```

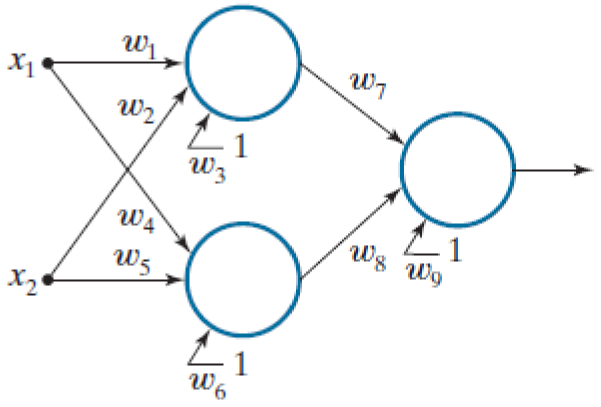
Need for Multilayer Neural Network

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

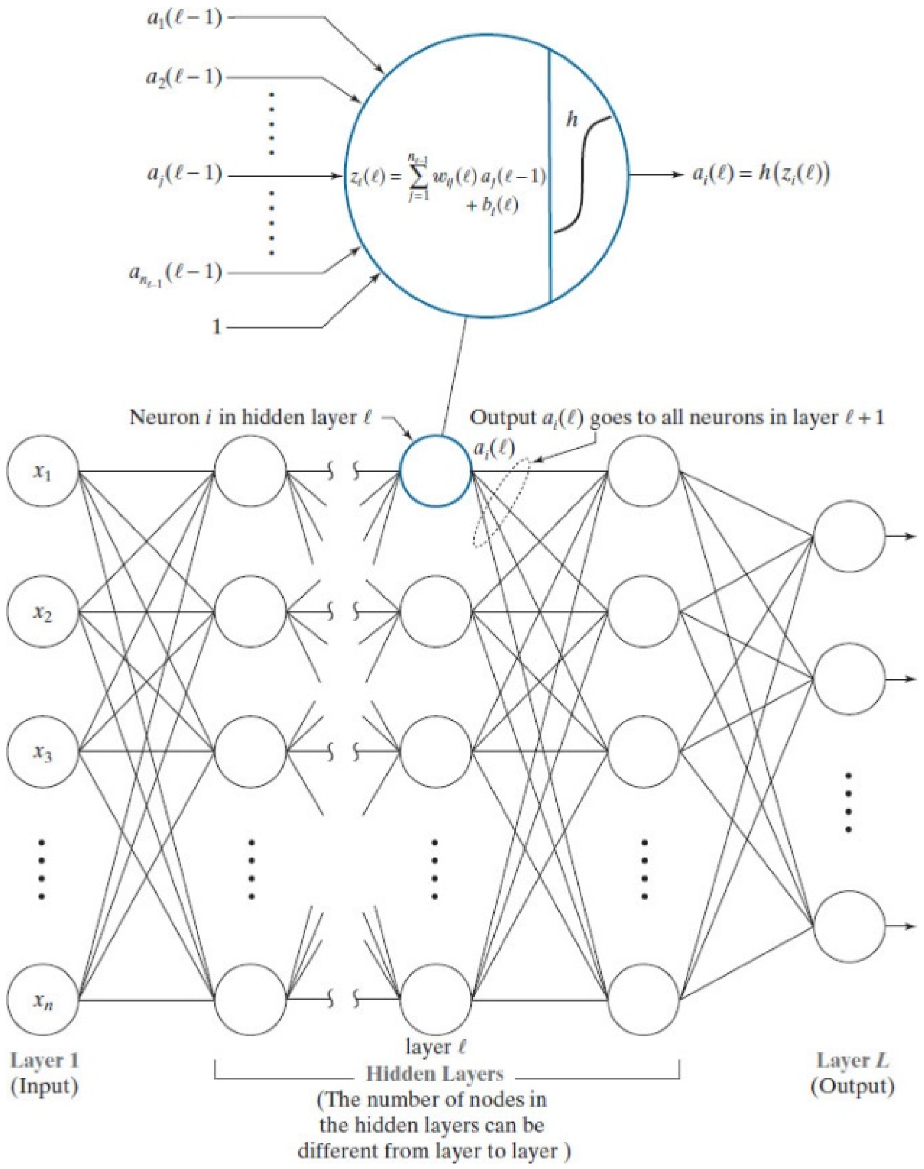




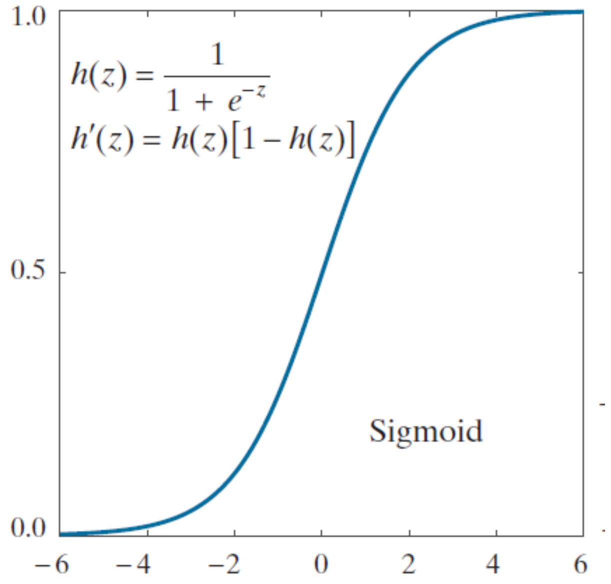
- one perceptron in the first layer maps any input from one class into a 1, and the other perceptron maps a pattern from the other class into a 0.
- This reduces the four possible inputs into two outputs -- a two-point problem that can be solved by a single perceptron.
- Therefore, we need three perceptrons to implement the XOR table.



Model of a Feedforward, Fully Connected Neural Network


$a_1(\ell-1)$
$a_2(\ell-1)$
$a_j(\ell-1)$
$a_{n_{\ell-1}}(\ell-1)$
1
$z_i(\ell) = \sum_{j=1}^{n_{\ell-1}} w_{ij}(\ell)\, a_j(\ell-1) + b_i(\ell)$
h
$a_i(\ell) = h(z_i(\ell))$
Neuron i in hidden layer ℓ
Output $a_i(\ell)$ goes to all neurons in layer $\ell+1$
$a_i(\ell)$
x_1
x_2
x_3
x_n
Layer 1 (Input)
layer ℓ
Hidden Layers
(The number of nodes in the hidden layers can be different from layer to layer)
Layer L (Output)

An example activation function

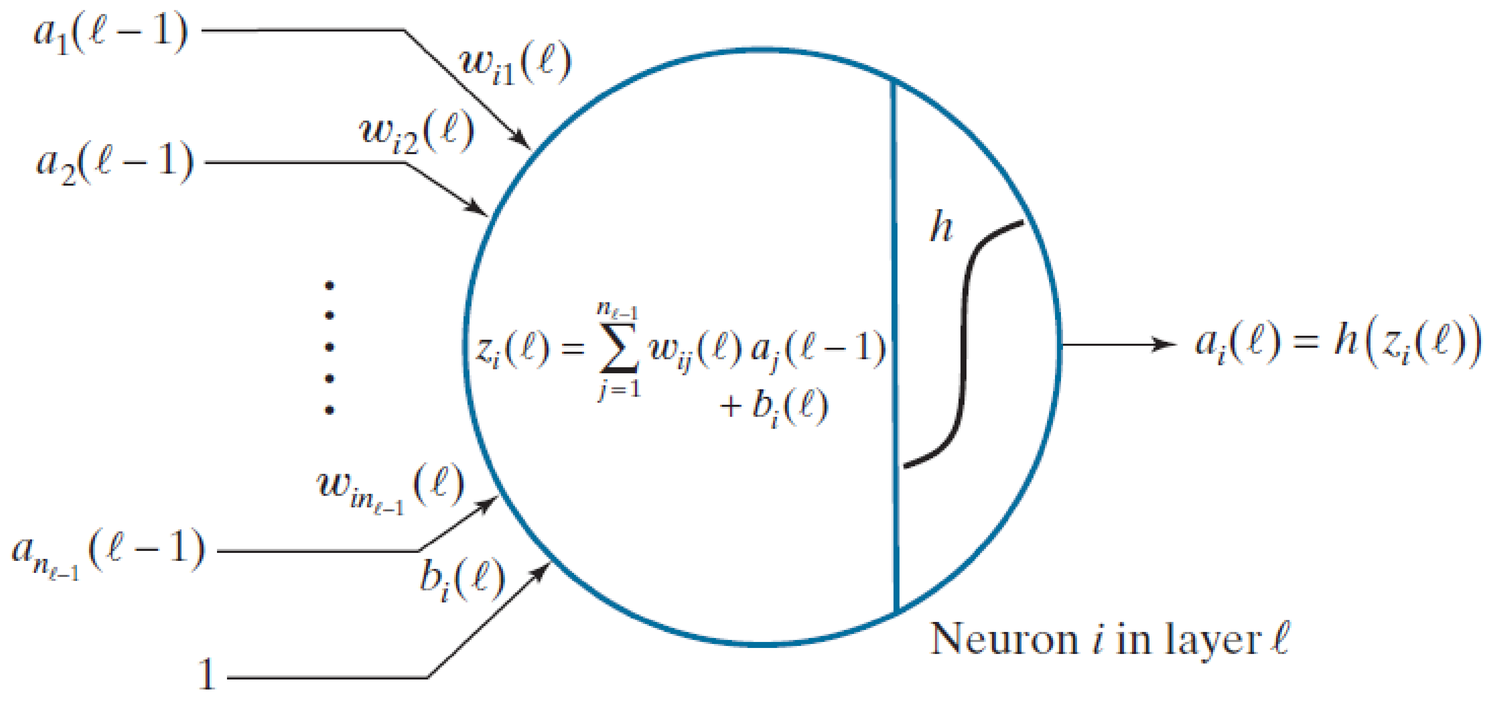


The output of each neuron goes to the input of all neurons in the following layer, hence the name:
“fully connected” for this type of architecture.

Shallow and Deep Neural Network

- The input layer is special -- its nodes are the components of an input pattern vector. Therefore, the outputs (activation values) of the first layer are the values of the elements of x.
- The outputs of all other nodes are the activation values of neurons in a particular layer.
- Each layer in the network can have a different number of nodes, but each node has a single output.
- We also require that there be no loops in the network. Such networks are called **feedforward** networks.
- We know the values of the nodes in the first layer, and we can observe the values of the output neurons. However, all others are **hidden** neurons, and the layers that contain them are called **hidden layers**.
- Generally, we call a neural net with a single hidden layer a **shallow neural network**, and refer to network with two or more hidden layers as a **deep** neural network. However, this terminology is not universal, and can be used subjectively.

Activation (Transfer) Function h( )



$a_1(\ell-1)$
$w_{i1}(\ell)$
$a_2(\ell-1)$
$w_{i2}(\ell)$
$w_{in_{\ell-1}}(\ell)$
$a_{n_{\ell-1}}(\ell-1)$
$b_i(\ell)$
1
$z_i(\ell) = \sum_{j=1}^{n_{\ell-1}} w_{ij}(\ell)\, a_j(\ell-1) + b_i(\ell)$
h
$a_i(\ell) = h\big(z_i(\ell)\big)$
Neuron i in layer ℓ

Simplest linear identity function: $a_i(l) = h\big(z_i(l)\big) = z_i(l)$

- Unable to model non-linear systems, however;
- Useful to explain the ***backpropagation*** method, instrumental to training multilayer neural network.

Error/Loss Function

- Neural network is trained in order to minimize the error (loss) between the actual and desired response.
- The mean squared error is a commonly measured, where we want to find the augmented weight vector that minimizes the mean squared error (**MSE**) between the desired and actual responses.
- In a single-layer neural network, we use the loss function

$$E(\boldsymbol{w}) = \frac{1}{2}\left(r - \boldsymbol{w}^T\mathbf{x}\right)^2$$

- The function is differentiable and have a unique minimum due to its quadratic form.

Iterative Gradient Descent Algorithm

- We find the minimum of the loses function using an **iterative gradient descent algorithm**, whose form is

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) - \alpha \left[\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} \right]_{\boldsymbol{w}=\boldsymbol{w}(k)}$$

- The value of α determines the relative magnitude of the correction in weight value.
  - If α is too small, the step changes will be correspondingly small and the weight would move slowly toward convergence.
  - On the other hand, choosing a too large α could cause large oscillations on either side of the minimum, or even become unstable.
- There is no general rule for choosing α. We can start with a small value and experiment by increasing α to determine its influence on the training data.

Convergence

$$E(\boldsymbol{w}) = \frac{1}{2}\left(r - \boldsymbol{w}^T\mathbf{x}\right)^2 \Rightarrow \frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}} = -\left(r - \boldsymbol{w}^T\mathbf{x}\right)\mathbf{x}$$

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) - \alpha\left[\frac{\partial E(\boldsymbol{w})}{\partial \boldsymbol{w}}\right]_{w=w(k)} \Rightarrow \boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \alpha\left[r(k) - \boldsymbol{w}^T(k)\mathbf{x}(k)\right]\mathbf{x}(k)$$

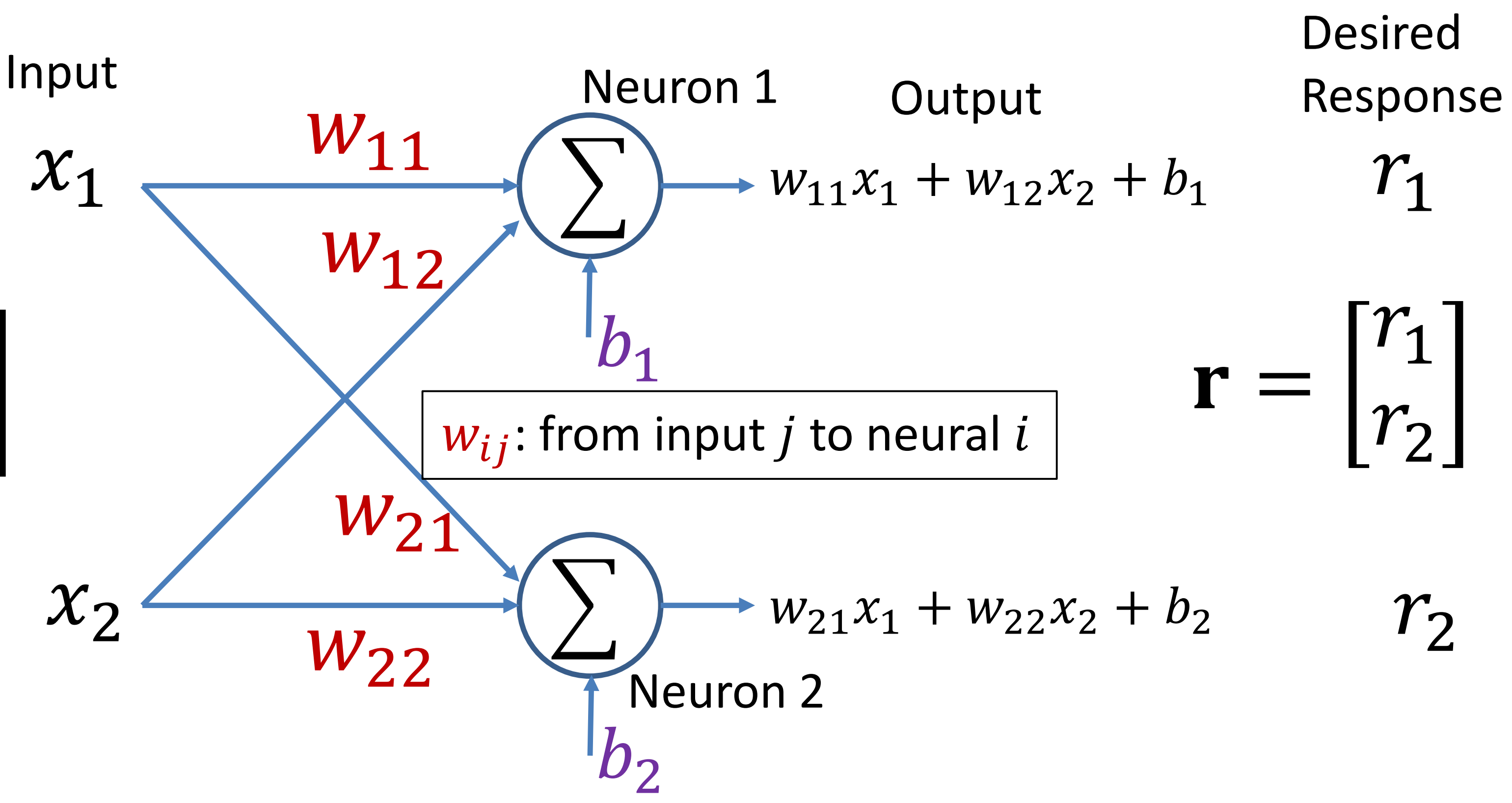
- We do not need to compute the gradient explicitly at every step, since the error function is given analytically and it is differentiable.
- In theory, this *least-mean-squared-error* (LMSE) algorithm will converge to a solution that minimizes the mean squared error over the patterns of the training set.
- In practice, we declare the algorithm has converged when the error decreases below a specified threshold.

Training Neural Network

- To illustrate the principle of neural network training, we start with a single layer network, without any hidden layer.
- To provide an insight into the backpropagation method, we then investigate a neural network with only one hidden layer, where the activation function for the hidden layer and the output layers is the identity linear function.
- We then look at the same three-layer network, where the activation functions are now changed to sigmoid function, and see how the derivatives of the activation function are integrated into the backpropagation processing flow.
- Next, we use the **Softmax** activation function for the final output layer, and compare the sigmoid function and softmax function in terms of the weights and biases learned.
- We then discuss implementations of training multilayer neural network in Matlab and sklearn.

Single Layer Network without Hidden Layer

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



$$\mathbf{r} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$$

Actual Output: $\boldsymbol{W}\mathbf{x} + \mathbf{b} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix}$

$$E(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{2}\|\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})\|^2$$

$$= \frac{1}{2}\{[r_1 - (w_{11}x_1 + w_{12}x_2 + b_1)]^2 + [r_2 - (w_{21}x_1 + w_{22}x_2 + b_2)]^2\}$$

$\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}}$ and $\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}}$

$$E(\boldsymbol{W},\boldsymbol{b}) = \frac{1}{2}\{[r_1 - (w_{11}x_1 + w_{12}x_2 + b_1)]^2 + [r_2 - (w_{21}x_1 + w_{22}x_2 + b_2)]^2\}$$

$$\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial w_{11}} & \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial w_{12}} \\ \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial w_{21}} & \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial w_{22}} \end{bmatrix}$$

$$= \begin{bmatrix} -[r_1 - (w_{11}x_1 + w_{12}x_2 + b_1)]x_1 & -[r_1 - (w_{11}x_1 + w_{12}x_2 + b_1)]x_2 \\ -[r_2 - (w_{21}x_1 + w_{22}x_2 + b_2)]x_1 & -[r_2 - (w_{21}x_1 + w_{22}x_2 + b_2)]x_2 \end{bmatrix}$$

$$= -\begin{bmatrix} r_1 - (w_{11}x_1 + w_{12}x_2 + b_1) \\ r_2 - (w_{21}x_1 + w_{22}x_2 + b_2) \end{bmatrix} \begin{bmatrix} x_1 & x_2 \end{bmatrix}$$

$$= -[\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})]\mathbf{x}^{\mathrm{T}}$$

$$\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}} = \begin{bmatrix} \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial b_1} \\ \frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial b_2} \end{bmatrix} = -\begin{bmatrix} r_1 - (w_{11}x_1 + w_{12}x_2 + b_1) \\ r_2 - (w_{21}x_1 + w_{22}x_2 + b_2) \end{bmatrix} = -[\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})]$$

Updating the Weights and Biases

$$E(\boldsymbol{W},\boldsymbol{b}) = \frac{1}{2}\|\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})\|^2$$

$$\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}} = -[\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})]\mathbf{x}^{\mathrm{T}}$$

$$\boldsymbol{W}(k+1) = \boldsymbol{W}(k) - \alpha\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}}$$

$$\boldsymbol{W}(k+1) = \boldsymbol{W}(k) - \alpha[(\boldsymbol{W}\mathbf{x} + \boldsymbol{b}) - \mathbf{r}]\mathbf{x}^{\mathrm{T}}$$

$$\frac{\partial E(\boldsymbol{w},\boldsymbol{b})}{\partial \boldsymbol{b}} = -[\mathbf{r} - (\boldsymbol{W}\mathbf{x} + \boldsymbol{b})]$$

$$\boldsymbol{b}(k+1) = \boldsymbol{b}(k) - \alpha\frac{\partial E(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{b}} = \boldsymbol{b}(k) - \alpha[(\boldsymbol{W}\mathbf{x} + \boldsymbol{b}) - \mathbf{r}]$$

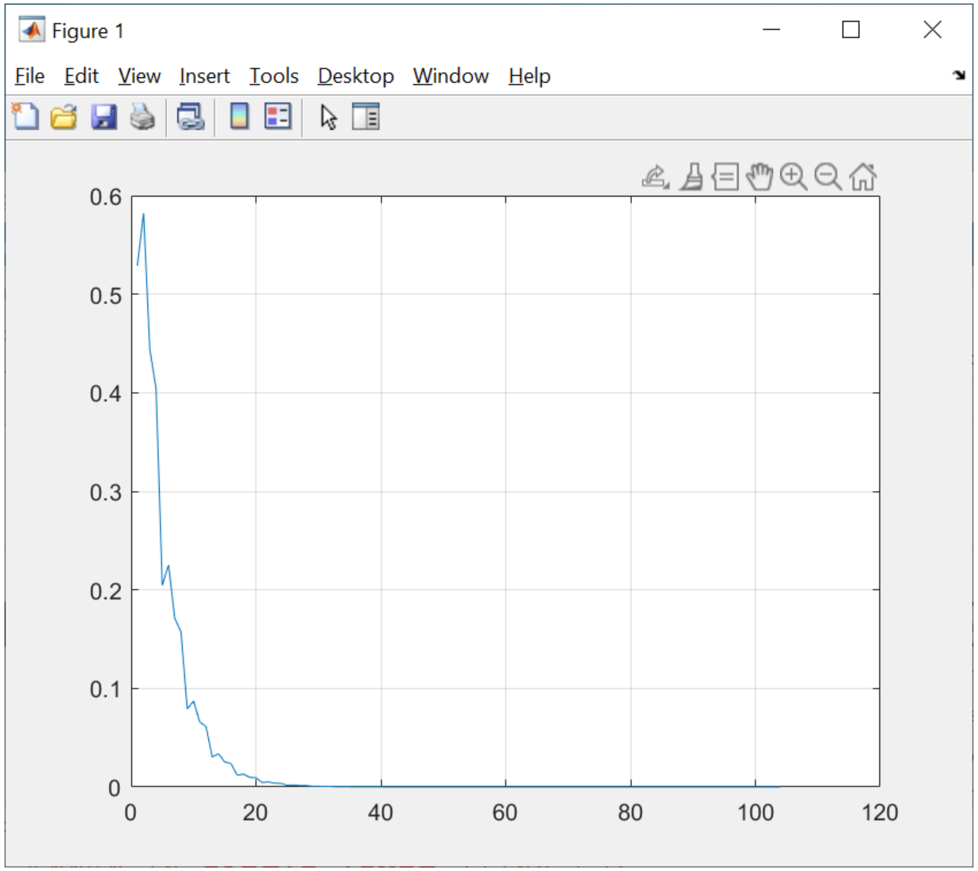
single_layer.m

```
alpha = 0.1;  % learning rate

X = [1 -1 -1 1; 1 -1 1 -1];
% Response
R = [1  0  1 0; 0  1 0  1];

rng('default');
Std = 0.02;

% Initial weights and biases
W2 = Std*randn(2,2);
b2 = Std*randn(2,1);

max_iter = 100;
mse = zeros(1, max_iter);

epoch = 0;
```

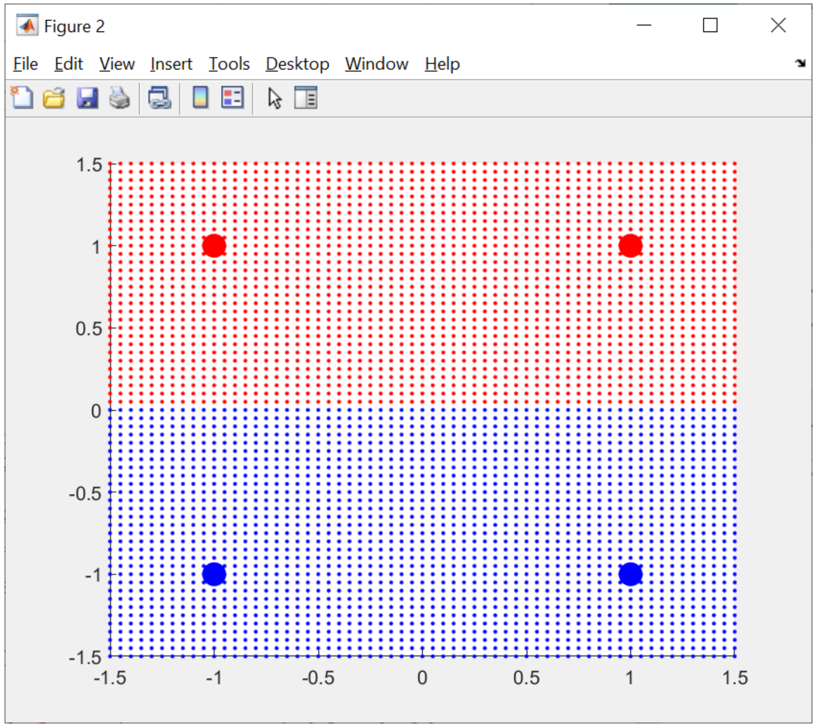
```
while (epoch <= max_iter)
    for i = 1: 4
        epoch = epoch + 1;
        A1 = X(:,i);

        A2 = W2*A1 + b2;

        D2 = A2 - R(:,i);

        mse(epoch) = 0.5*norm(D2)^2;

        % Update the weights and biases
        W2 = W2 - alpha*D2*A1';
        b2 = b2 - alpha*D2;

    end
end
```

$$\boldsymbol{W}(k+1) = \boldsymbol{W}(k) - \alpha[(\boldsymbol{W}\mathbf{x} + \boldsymbol{b}) - \mathbf{r}]\mathbf{x}^{\mathrm{T}}$$
$$\boldsymbol{b}(k+1) = \boldsymbol{b}(k) - \alpha[(\boldsymbol{W}\mathbf{x} + \boldsymbol{b}) - \mathbf{r}]$$

Convergence



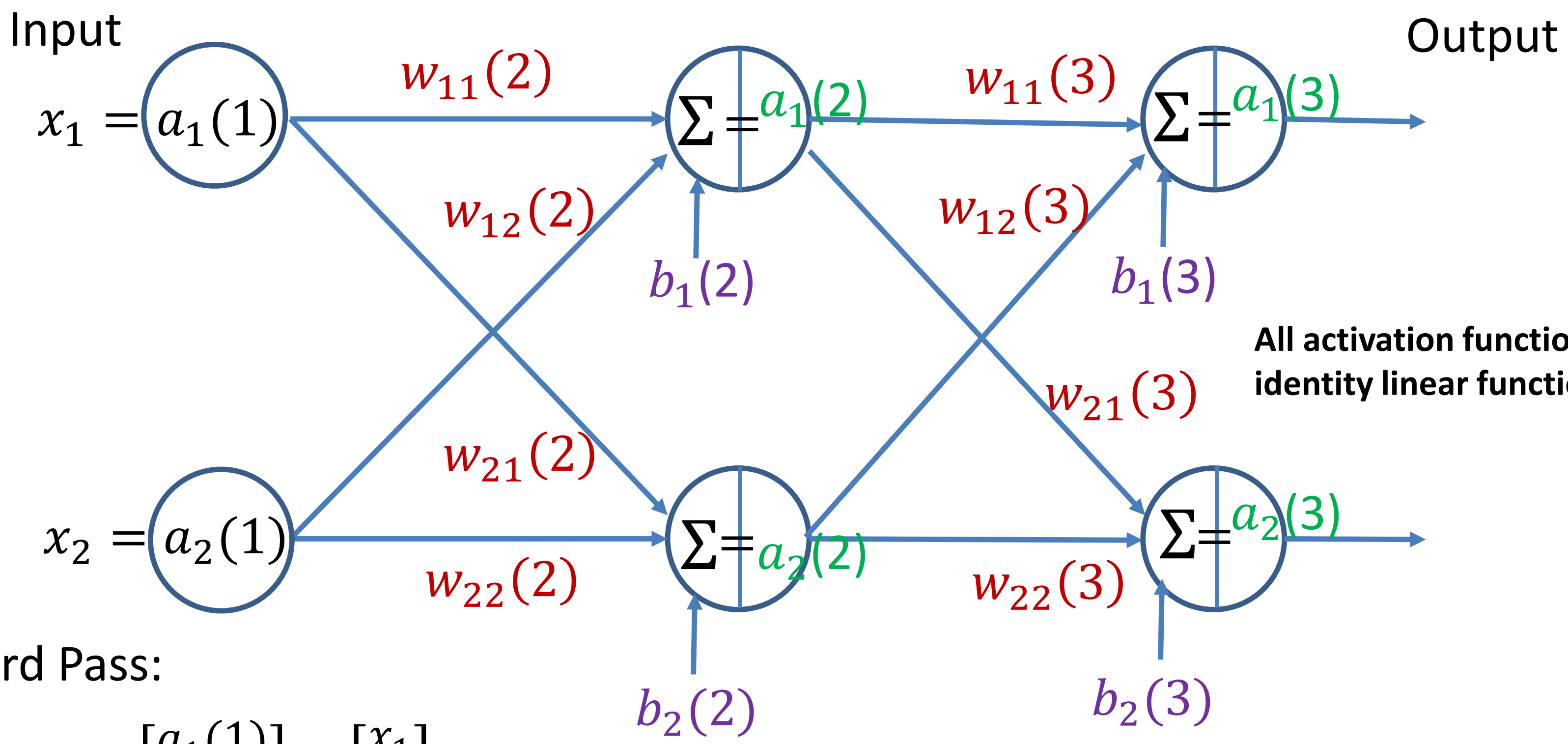
Stochastic Gradient Descent



```
>> W2
W2 =
   -0.0000    0.5000
   -0.0000   -0.5000
```

```
>> b2
b2 =
    0.5000
    0.5000
```

A Network with one Hidden Layer



All activation functions are identity linear function.

Forward Pass:

$$\boldsymbol{A}(1) = \begin{bmatrix} a_1(1) \\ a_2(1) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\boldsymbol{A}(2) = \begin{bmatrix} a_1(2) \\ a_2(2) \end{bmatrix} = \boldsymbol{W}(2)\boldsymbol{A}(1) + \boldsymbol{b}(2) = \begin{bmatrix} w_{11}(2) & w_{12}(2) \\ w_{21}(2) & w_{22}(2) \end{bmatrix} \begin{bmatrix} a_1(1) \\ a_2(1) \end{bmatrix} + \begin{bmatrix} b_1(2) \\ b_2(2) \end{bmatrix}$$

$$\boldsymbol{A}(3) = \begin{bmatrix} a_1(3) \\ a_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3) = \begin{bmatrix} w_{11}(3) & w_{12}(3) \\ w_{21}(3) & w_{22}(3) \end{bmatrix} \begin{bmatrix} a_1(2) \\ a_2(2) \end{bmatrix} + \begin{bmatrix} b_1(3) \\ b_2(3) \end{bmatrix}$$

Loss Function for a Multilayer Neural Network

- Given a set of training patterns and a multilayer feedforward neural network architecture, we want to find the network parameters. that minimize an error (also called cost or objective) function.
- Our interest is in classification performance, so we define the error function for a neural network as the average of the differences between desired and actual responses.
- The activation values of neuron j in the output layer is $a_j(L)$. We define the error of that neuron as $E_j = \frac{1}{2}\left(r_j - a_j(L)\right)^2$, for $j = 1,2,\ldots,n_L$.
- The output error with respect to a single $\mathbf{x}$ is the sum of the errors of all output neurons with respect to that vector (using the Euclidean vector norm):

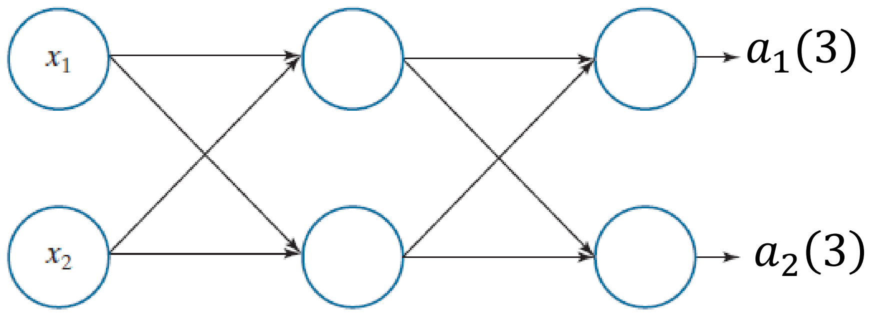
$$E = \sum_{j=1}^{n_L} E_j = \frac{1}{2}\sum_{j=1}^{n_L}\left(r_j - a_j(L)\right)^2 = \frac{1}{2}\|\mathbf{r} - \mathbf{a}(L)\|^2$$

- The *total network output error* over all training patterns is defined as the sum of the errors of the individual patterns.

Difficulty with Training a Multilayer Network

- We want to find the weights that minimize this total error. As we did for the LMSE method on a single-layer network, we find the solution using the iterative gradient descent.
- Thus we need a scheme to adjust all weights in a network using training patterns. In order to do this, we need to know how the total error changes with respect to all the weights in the network.
- However, the challenge arises as to how we can compute the gradients of the weights in the hidden nodes.
- The solution is the **backpropagation** method based on the **chain rule** in calculus, which allows the following quantity $\delta_j(L) = \frac{\partial E}{\partial z_j(L)}$ to propagate from output back into each of the hidden layers in the network, where $z_j(L)$ is the output of the last layer, before we apply the activation function $h()$ on $z_j(L)$ to obtain $a_j(L) = h\left(z_j(L)\right)$.
- We will use a three-layer ($L = 3$) network to illustrate the principle of backpropagation, where the activation function output is simply the same as its input: $a_j(L) = h\left(z_j(L)\right) = z_j(L)$.
- Next, we will extend the result and consider the general case where the activation function is a non-linear function such as the sigmoid function.

The output error with respect to a single $\mathbf{x}$



Desired response: $\mathbf{r} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$

$$\boldsymbol{A}(3) = \begin{bmatrix} a_1(3) \\ a_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$$

Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(3)\|^2 = \frac{1}{2}\{[r_1 - a_1(3)]^2 + [r_2 - a_2(3)]^2\}$

Let the derivatives of the output error with respect to the final output be:

$$\boldsymbol{D}(3) = \frac{\partial E}{\partial \boldsymbol{A}(3)} = \begin{bmatrix} \dfrac{\partial E}{\partial a_1(3)} \\ \dfrac{\partial E}{\partial a_2(3)} \end{bmatrix} = -\begin{bmatrix} r_1 - a_1(3) \\ r_2 - a_2(3) \end{bmatrix} = \boldsymbol{A}(3) - \mathbf{r}$$

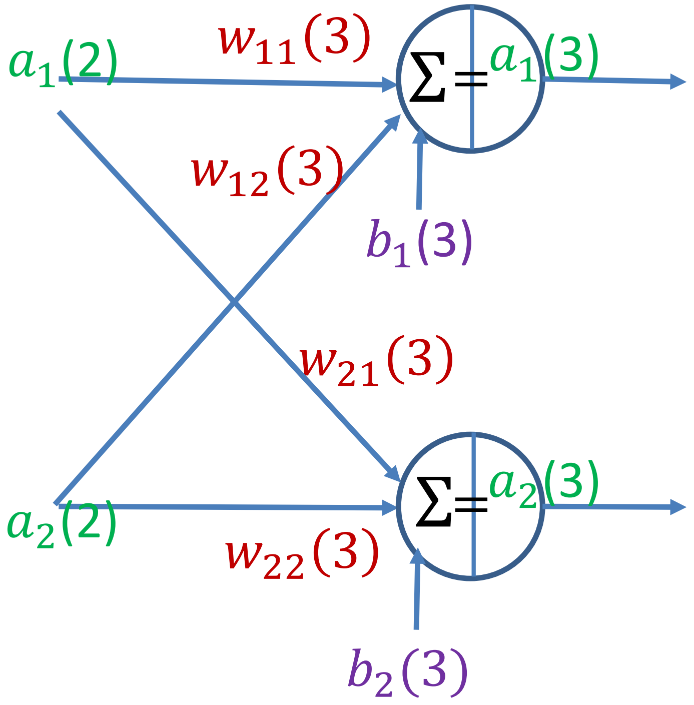
Gradient of the Error with respect to Weights

$$\frac{\partial E}{\partial w_{11}(3)} = \frac{\partial E}{\partial a_1(3)}\frac{\partial a_1(3)}{\partial w_{11}(3)} = \frac{\partial E}{\partial a_1(3)} a_1(2)$$

$$\frac{\partial E}{\partial w_{12}(3)} = \frac{\partial E}{\partial a_1(3)}\frac{\partial a_1(3)}{\partial w_{12}(3)} = \frac{\partial E}{\partial a_1(3)} a_2(2)$$

$$\frac{\partial E}{\partial w_{21}(3)} = \frac{\partial E}{\partial a_2(3)}\frac{\partial a_2(3)}{\partial w_{21}(3)} = \frac{\partial E}{\partial a_2(3)} a_1(2)$$

$$\frac{\partial E}{\partial w_{22}(3)} = \frac{\partial E}{\partial a_2(3)}\frac{\partial a_2(3)}{\partial w_{22}(3)} = \frac{\partial E}{\partial a_2(3)} a_2(2)$$



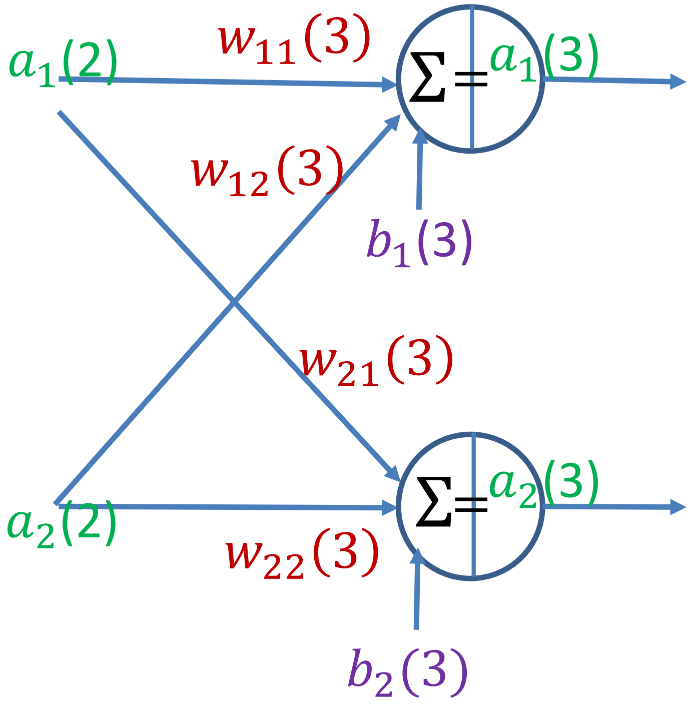
$$\frac{\partial E}{\partial \boldsymbol{W}(3)} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}(3)} & \frac{\partial E}{\partial w_{12}(3)} \\ \frac{\partial E}{\partial w_{21}(3)} & \frac{\partial E}{\partial w_{22}(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix} [a_1(2) \quad a_2(2)] = \frac{\partial E}{\partial \boldsymbol{A}(3)} \boldsymbol{A}(2)^{\mathrm{T}} = \boldsymbol{D}(3)\boldsymbol{A}(2)^{\mathrm{T}}$$

where $\boldsymbol{A}(3) = \begin{bmatrix} a_1(3) \\ a_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$

Gradient of the Error with respect to Biases

$$\frac{\partial E}{\partial b_1(3)} = \frac{\partial E}{\partial a_1(3)} \frac{\partial a_1(3)}{\partial b_1(3)} = \frac{\partial E}{\partial a_1(3)}$$

$$\frac{\partial E}{\partial b_2(3)} = \frac{\partial E}{\partial a_2(3)} \frac{\partial a_2(3)}{\partial b_2(3)} = \frac{\partial E}{\partial a_2(3)}$$



$$\frac{\partial E}{\partial \boldsymbol{b}(3)} = \begin{bmatrix} \frac{\partial E}{\partial b_1(3)} \\ \frac{\partial E}{\partial b_2(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix} = \frac{\partial E}{\partial \boldsymbol{A}(3)} = \boldsymbol{D}(3)$$

Relation between $\boldsymbol{D}(2)$ and $\boldsymbol{D}(3)$

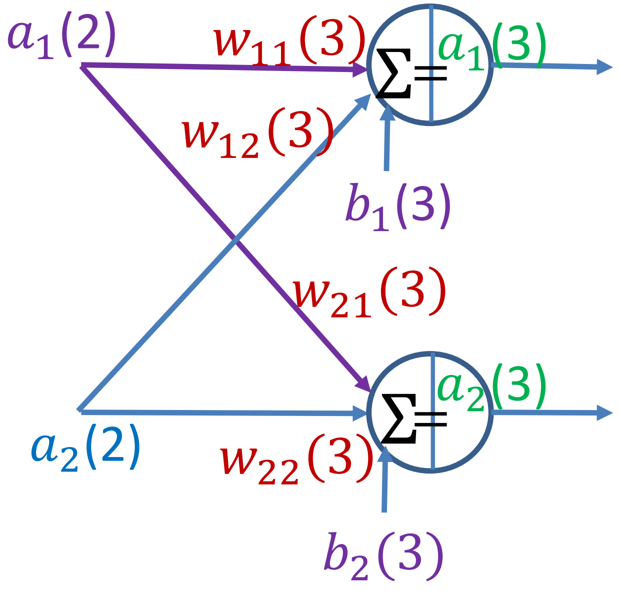
$$\boldsymbol{D}(2) = \frac{\partial E}{\partial \boldsymbol{A}(2)} = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix}$$

$$\frac{\partial E}{\partial a_1(2)} = \frac{\partial E}{\partial a_1(3)}\frac{\partial a_1(3)}{\partial a_1(2)} + \frac{\partial E}{\partial a_2(3)}\frac{\partial a_2(3)}{\partial a_1(2)}$$

$$= \frac{\partial E}{\partial a_1(3)} w_{11}(3) + \frac{\partial E}{\partial a_2(3)} w_{21}(3)$$

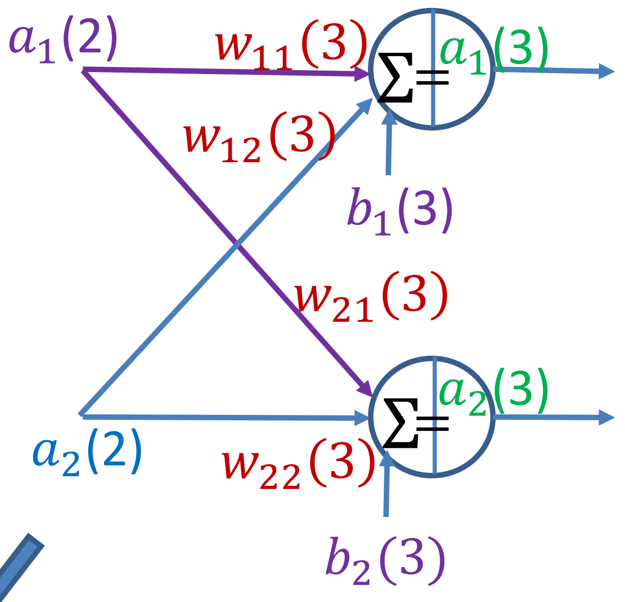
$$\frac{\partial E}{\partial a_2(2)} = \frac{\partial E}{\partial a_1(3)}\frac{\partial a_1(3)}{\partial a_2(2)} + \frac{\partial E}{\partial a_2(3)}\frac{\partial a_2(3)}{\partial a_2(2)}$$

$$= \frac{\partial E}{\partial a_1(3)} w_{12}(3) + \frac{\partial E}{\partial a_2(3)} w_{22}(3)$$



Thus $\boldsymbol{D}(2) = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix} = \begin{bmatrix} w_{11}(3) & w_{12}(3) \\ w_{21}(3) & w_{22}(3) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix} = \boldsymbol{W}(3)^{\mathrm{T}}\boldsymbol{D}(3)$

Backpropagation of $\boldsymbol{D}(3)$

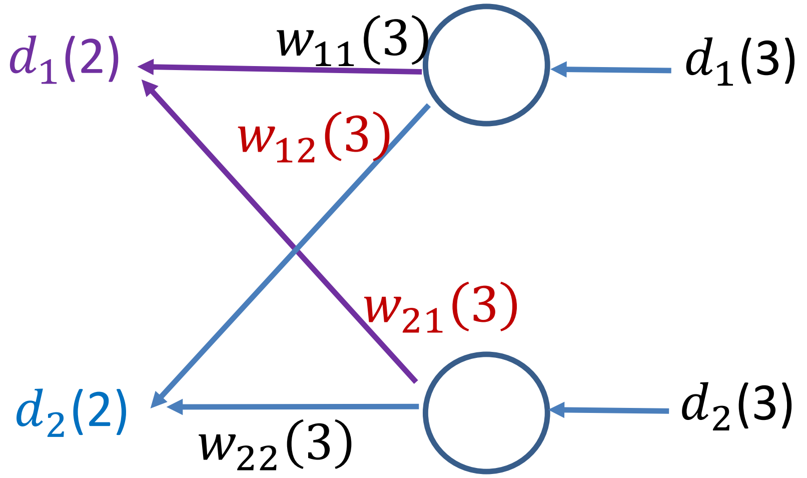


To calculate $\boldsymbol{D}(2)$, we back propagate $\boldsymbol{D}(3)$ as:

$$\boldsymbol{D}(2) = \begin{bmatrix} d_1(2) \\ d_2(2) \end{bmatrix} = \begin{bmatrix} w_{11}(3) & w_{21}(3) \\ w_{12}(3) & w_{22}(3) \end{bmatrix} \begin{bmatrix} d_1(3) \\ d_2(3) \end{bmatrix}$$

$$= \boldsymbol{W}(3)^{\mathrm{T}} \boldsymbol{D}(3)$$

Note the reversed directions of the arrows, thus the transpose of the weight matrix $\boldsymbol{W}(3)^{\mathrm{T}}$.



$$\boldsymbol{D}(2) = \begin{bmatrix} d_1(2) \\ d_2(2) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix}$$

$$\boldsymbol{D}(3) = \begin{bmatrix} d_1(3) \\ d_2(3) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix}$$

Gradient of the Error with respect to Weights (Level Two)

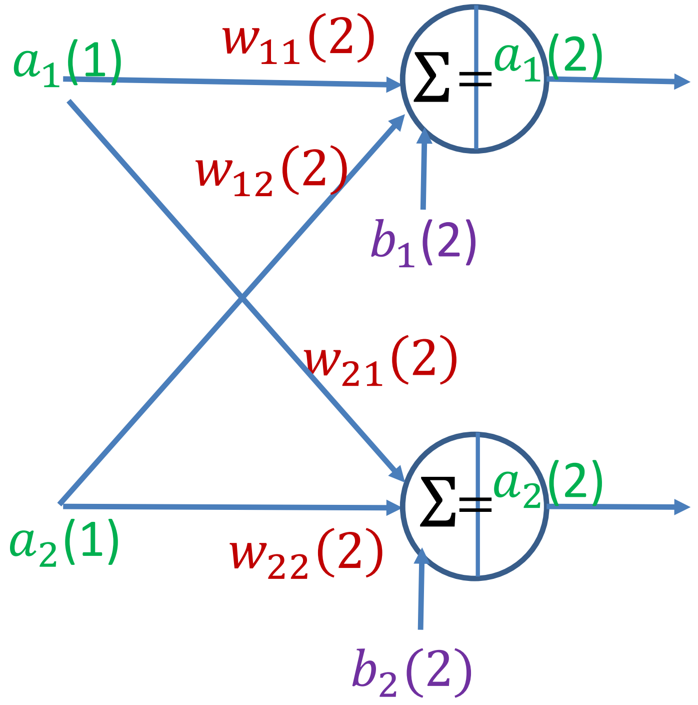
Similar to the previous derivations, for the 2nd layer:

$$\frac{\partial E}{\partial w_{11}(2)} = \frac{\partial E}{\partial a_1(2)}\frac{\partial a_1(2)}{\partial w_{11}(2)} = \frac{\partial E}{\partial a_1(2)} a_1(1)$$

$$\frac{\partial E}{\partial w_{12}(2)} = \frac{\partial E}{\partial a_1(2)}\frac{\partial a_1(2)}{\partial w_{12}(2)} = \frac{\partial E}{\partial a_1(2)} a_2(1)$$

$$\frac{\partial E}{\partial w_{21}(2)} = \frac{\partial E}{\partial a_2(2)}\frac{\partial a_2(2)}{\partial w_{21}(2)} = \frac{\partial E}{\partial a_2(2)} a_1(1)$$

$$\frac{\partial E}{\partial w_{22}(2)} = \frac{\partial E}{\partial a_2(2)}\frac{\partial a_2(2)}{\partial w_{22}(2)} = \frac{\partial E}{\partial a_2(2)} a_2(1)$$



$$\frac{\partial E}{\partial \boldsymbol{W}(2)} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}(2)} & \frac{\partial E}{\partial w_{12}(2)} \\ \frac{\partial E}{\partial w_{21}(2)} & \frac{\partial E}{\partial w_{22}(2)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix} [a_1(1) \quad a_2(1)] = \frac{\partial E}{\partial \boldsymbol{A}(2)} \boldsymbol{A}(1)^{\mathrm{T}} = \boldsymbol{D}(2)\boldsymbol{A}(1)^{\mathrm{T}}$$

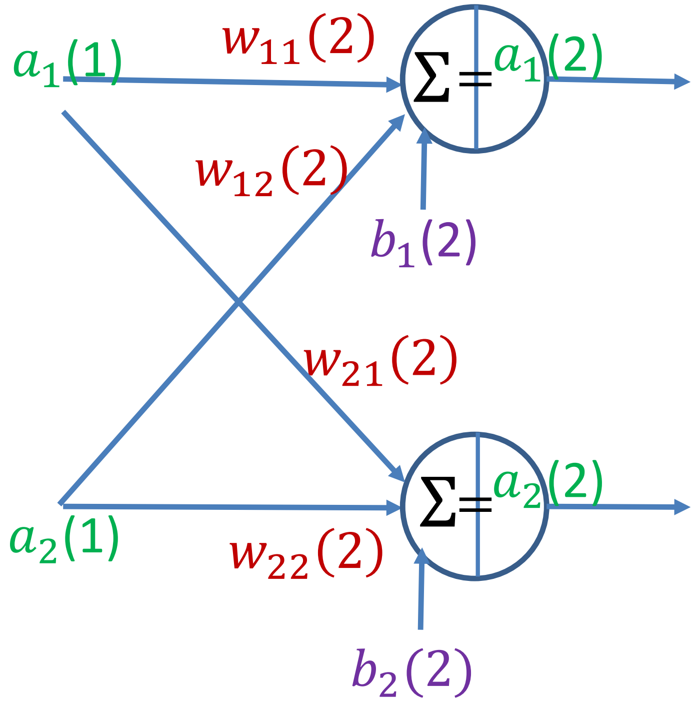
Where $\boldsymbol{D}(2)$ is obtained by back propagating $\boldsymbol{D}(3)$, and $\boldsymbol{A}(1) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the input vector.

Gradient with respect to Biases (2nd Layer)

$$\frac{\partial E}{\partial b_1(2)} = \frac{\partial E}{\partial a_1(2)} \frac{\partial a_1(2)}{\partial b_1(2)} = \frac{\partial E}{\partial a_1(2)}$$

$$\frac{\partial E}{\partial b_2(2)} = \frac{\partial E}{\partial a_2(2)} \frac{\partial a_2(2)}{\partial b_2(2)} = \frac{\partial E}{\partial a_2(2)}$$

$$\frac{\partial E}{\partial \boldsymbol{b}(2)} = \begin{bmatrix} \frac{\partial E}{\partial b_1(2)} \\ \frac{\partial E}{\partial b_2(2)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix} = \frac{\partial E}{\partial \boldsymbol{A}(2)} = \boldsymbol{D}(2)$$



Summary of the Results

$$\boldsymbol{A}(1) = \begin{bmatrix} a_1(1) \\ a_2(1) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\boldsymbol{A}(2) = \begin{bmatrix} a_1(2) \\ a_2(2) \end{bmatrix} = \boldsymbol{W}(2)\boldsymbol{A}(1) + \boldsymbol{b}(2)$$

$$\boldsymbol{A}(3) = \begin{bmatrix} a_1(3) \\ a_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$$

Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(3)\|^2$

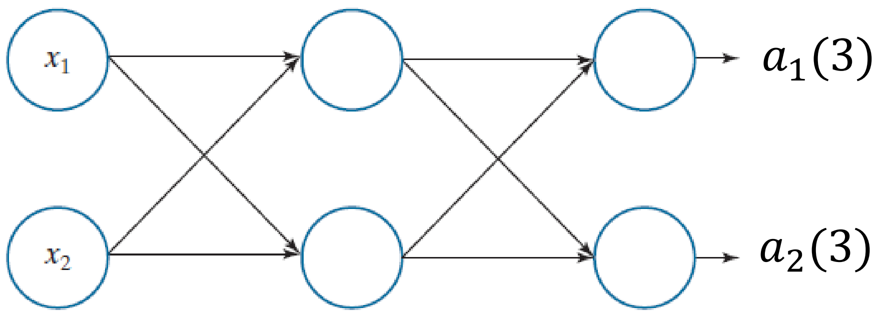
$\boldsymbol{D}(3) = \boldsymbol{A}(3) - \mathbf{r}$, where $\mathbf{r}$ is the desired response.

Backpropagation: $\boldsymbol{D}(\mathbf{2}) = \boldsymbol{W}(3)^{\mathrm{T}}\boldsymbol{D}(3)$

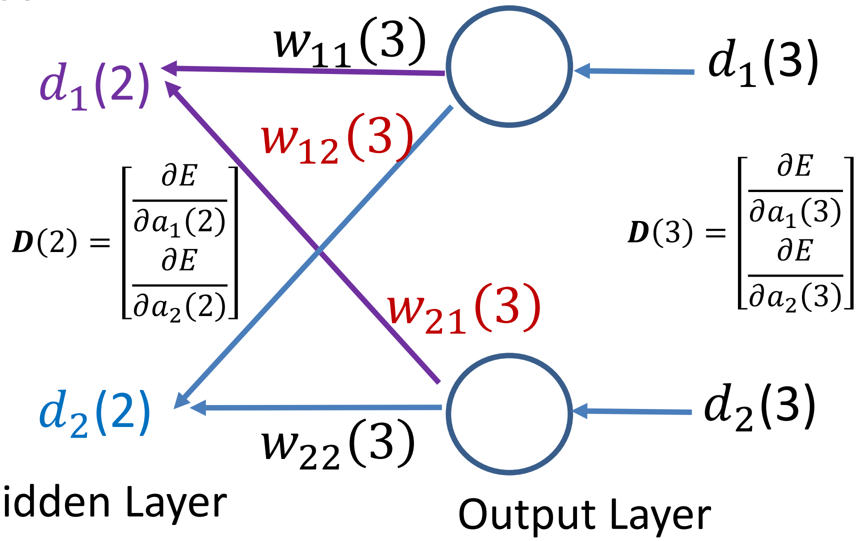
$$\frac{\partial E}{\partial \boldsymbol{W}(3)} = \boldsymbol{D}(3)\boldsymbol{A}(2)^{\mathrm{T}}, \quad \frac{\partial E}{\partial \boldsymbol{b}(3)} = \boldsymbol{D}(3)$$

$$\frac{\partial E}{\partial \boldsymbol{W}(2)} = \boldsymbol{D}(2)\boldsymbol{A}(1)^{\mathrm{T}}, \quad \frac{\partial E}{\partial \boldsymbol{b}(2)} = \boldsymbol{D}(2)$$

Forward Pass



Backpropagation of error gradient from output to hidden layer:



$d_1(2)$
$d_2(2)$
$w_{11}(3)$
$w_{12}(3)$
$w_{21}(3)$
$w_{22}(3)$
$d_1(3)$
$d_2(3)$
$\boldsymbol{D}(2) = \begin{bmatrix} \frac{\partial E}{\partial a_1(2)} \\ \frac{\partial E}{\partial a_2(2)} \end{bmatrix}$
$\boldsymbol{D}(3) = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix}$
Hidden Layer
Output Layer

Training Procedure using Iterative Gradient Descent

Initialize the weights and biases, and repeat the following until a convergence criterion is met (α is the *learning rate*):

- Forward pass $\boldsymbol{A}(l) = \boldsymbol{W}(l)\boldsymbol{A}(l-1) + \boldsymbol{b}(l)$, where the layer index $l = 2, \ldots, L$. In the illustrative example, $L = 3$.
- Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(L)\|^2$, and its gradient at the final output layer: $\boldsymbol{D}(L) = \boldsymbol{A}(L) - \mathbf{r}$.
- Backpropagation: $\boldsymbol{D}(l) = \boldsymbol{W}(l+1)^{\mathrm{T}}\boldsymbol{D}(l+1)$, for $l = L - 1, \ldots, 2$.
- Update weights and biases for $l = 2, \ldots, L$:

$$\boldsymbol{W}(l) = \boldsymbol{W}(l) - \alpha\frac{\partial E}{\partial \boldsymbol{W}(l)} = \boldsymbol{W}(l) - \alpha\boldsymbol{D}(l)\boldsymbol{A}^{\mathrm{T}}(l-1);$$

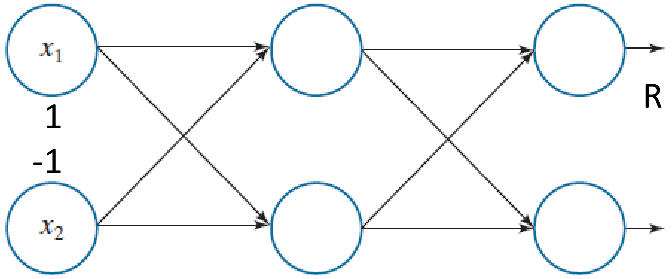
$$\boldsymbol{b}(l) = \boldsymbol{b}(l) - \alpha\frac{\partial E}{\partial \boldsymbol{b}(l)} = \boldsymbol{b}(l) - \alpha\boldsymbol{D}(l).$$

Linearly Separable Case

```
% backprop.m
% Explain the backpropagation algorithm using a fully connected neural
% network with one hidden layer.
% However, the activation of each neuron is a linear function, thus the
% network output is a linear combination of the input. Therefore, this
% network cannot handle linearly non-separable cases.
% The weights and biases are updated for each input sample

alpha = 0.1;  % learning rate

% Linearly separable example
% Input data pattern
X = [1 -1 -1 1; 1 -1 1 -1];
% Response
R = [1  0  1 0; 0  1 0  1];

rng('default');
Std = 0.02;

% Initial weights and biases
W2 = Std*randn(2,2);
b2 = Std*randn(2,1);
W3 = Std*randn(2,2);
b3 = Std*randn(2,1);

max_iter = 100;
mse = zeros(1, max_iter);
```

X =

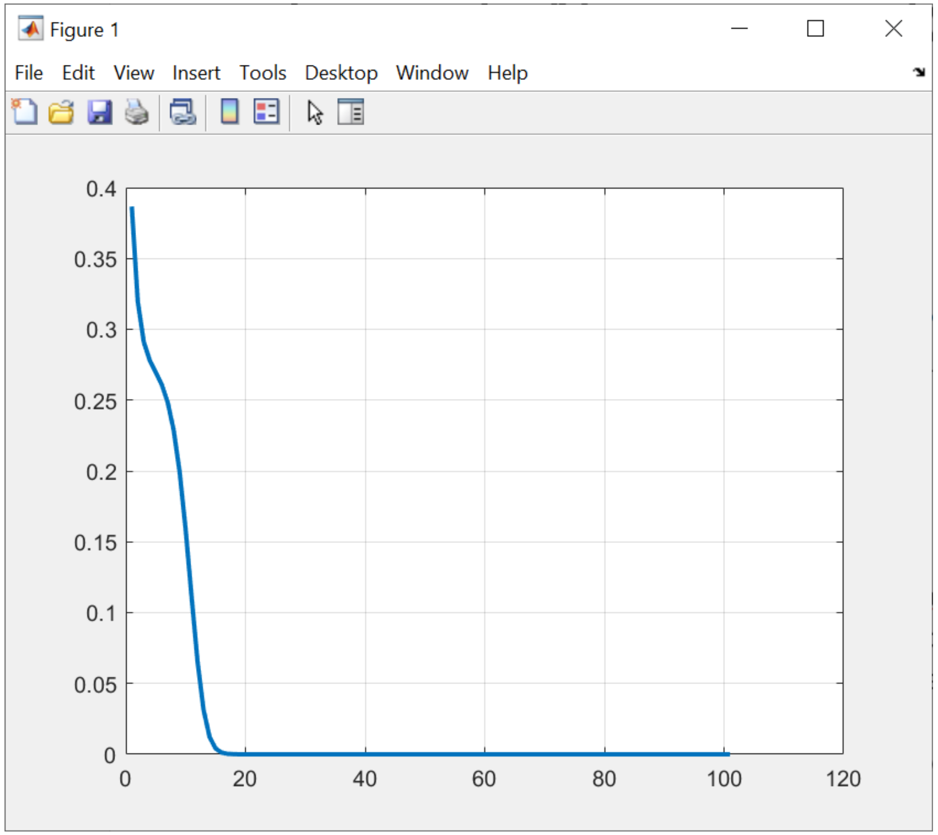
$$\begin{array}{cccc} 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{array}$$



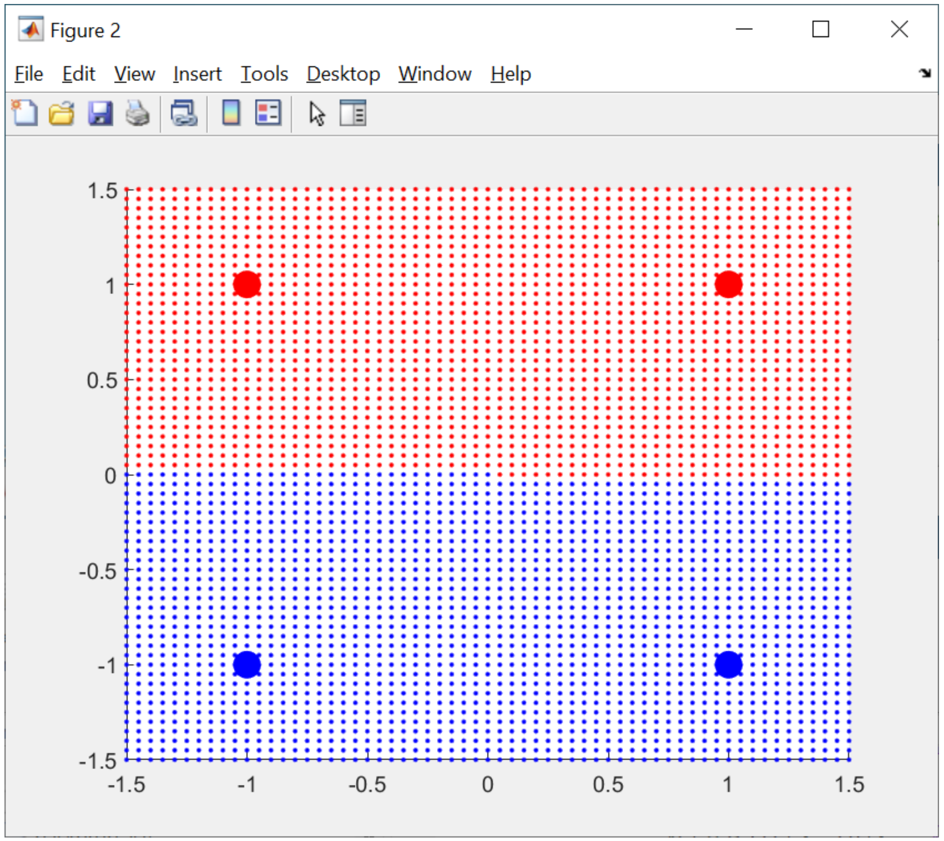
R =

$$\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array}$$

```
epoch = 0;
while (epoch <= max_iter)
    epoch = epoch + 1;

    for i = 1: 4
        A1 = X(:,i);
        A2 = W2*A1 + b2;
        A3 = W3*A2 + b3;

        D3 = A3 - R(:,i);

        mse(epoch) = 0.5*norm(D3)^2;

        % backpropagation
        D2 = W3'*D3;

        % Update the weights and biases
        W3 = W3 - alpha*D3*A2';
        W2 = W2 - alpha*D2*A1';

        b3 = b3 - alpha*D3;
        b2 = b2 - alpha*D2;
    end

end
mse(epoch)
plot(mse); grid
```



```
figure;
hold on;
for x1 = -1.5:0.05:1.5
    for x2 = -1.5:0.05:1.5
        X_test = [x1; x2];
        A1 = X_test;
        Z2 = W2*A1 + b2;
        A2 = 1./(1+exp(-Z2));

        Z3 = W3*A2 + b3;
        A3 = 1./(1+exp(-Z3));

        if (A3(1)>=0.5)
            plot(x1, x2, 'r.');
        else
            plot(x1, x2, 'b.');
        end
    end
end
```

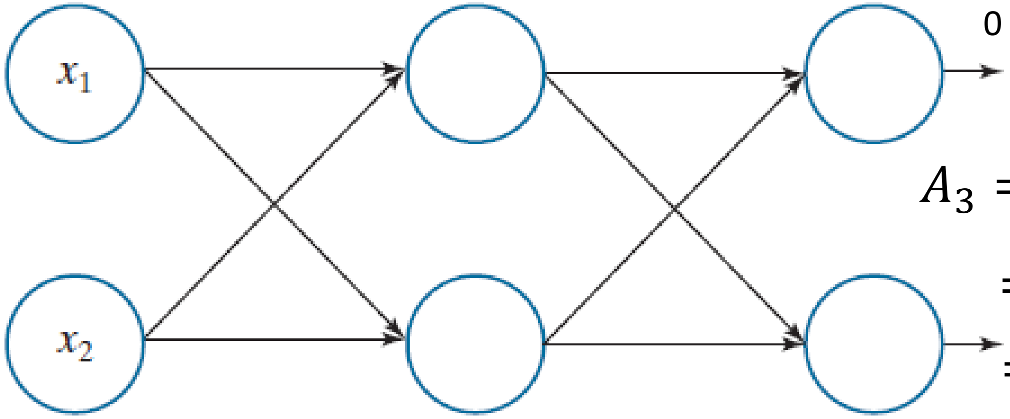


```
plot(X(1,1),X(2,1),'ro','MarkerSize',12, 'MarkerFaceColor','r');
plot(X(1,2),X(2,2),'ro','MarkerSize',12, 'MarkerFaceColor','r');
plot(X(1,3),X(2,3),'bo','MarkerSize',12, 'MarkerFaceColor','b');
plot(X(1,4),X(2,4),'bo','MarkerSize',12, 'MarkerFaceColor','b');
```

Weights and Biases Learned

```
X =
   1   -1   -1    1
   1   -1    1   -1
```

```
R =
   1    0    1    0
   0    1    0    1
```



$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{aligned} A_3 &= W_3(W_2X + b_2) + b_3 \\ &= (W_3W_2)X + (W_3b_2 + b_3) \\ &= WX + B \end{aligned}$$

```
W2 =
   0.0134   -0.7274
   0.0220    0.4375

b2 =
   0.0079
   0.0296
```

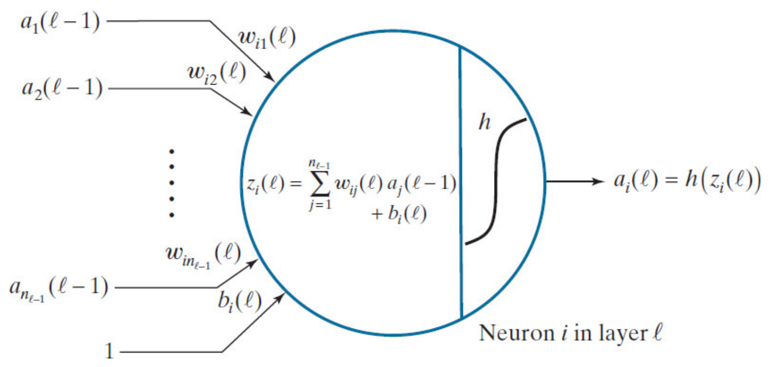
```
W3 =
  -0.4763    0.3512
   0.5303   -0.2608

b3 =
   0.4934
   0.5035
```

```
>> W3*W2
ans =
   0.0014    0.5001
   0.0014   -0.4999

>> W3*b2 + b3
ans =
   0.5000
   0.5000
```

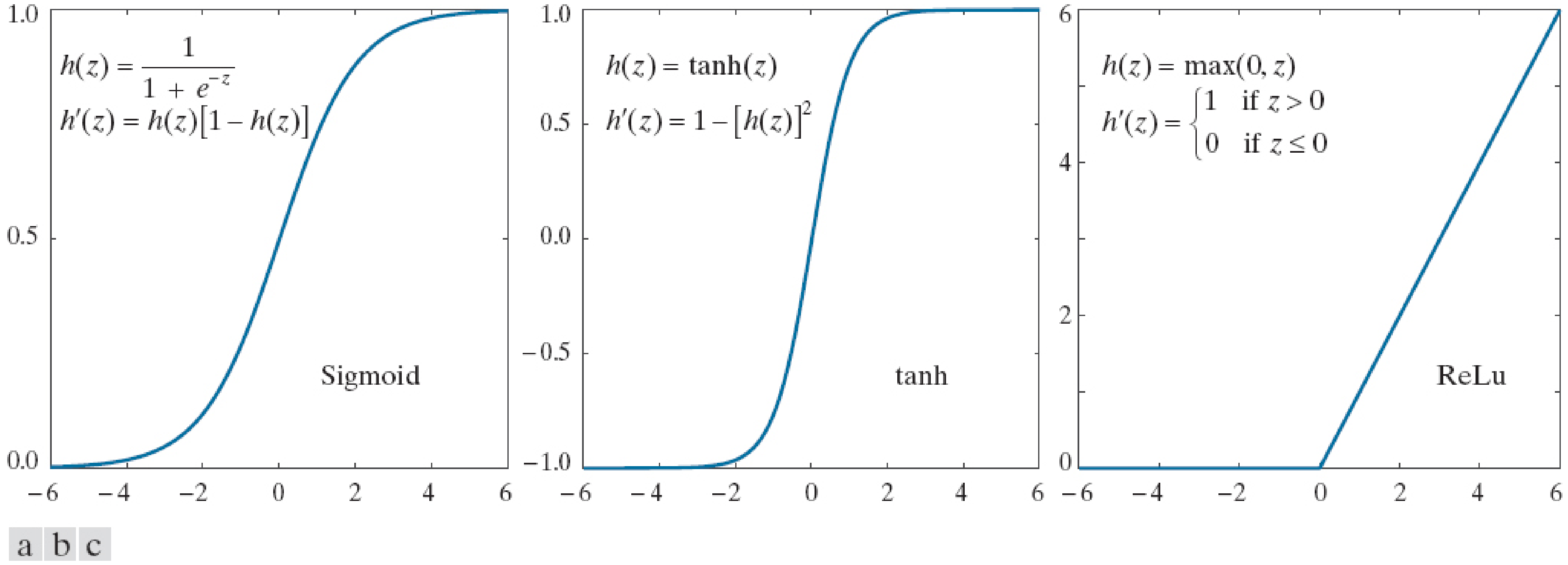
The output of the entire network is a linear combination of the input.

Various Activation Functions

Model of An Artificial Neuron



$a_1(\ell-1)$
$a_2(\ell-1)$
$a_{n_{\ell-1}}(\ell-1)$
1
$w_{i1}(\ell)$
$w_{i2}(\ell)$
$w_{in_{\ell-1}}(\ell)$
$b_i(\ell)$
$z_i(\ell) = \sum_{j=1}^{n_{\ell-1}} w_{ij}(\ell)\, a_j(\ell-1) + b_i(\ell)$
h
$a_i(\ell) = h(z_i(\ell))$
Neuron i in layer ℓ



(a) Sigmoid. (b) Hyperbolic tangent (also has a sigmoid shape, but it is centered about 0 in both dimensions). (c) Rectifier linear unit (ReLU).

Softmax in the Final Output Layer

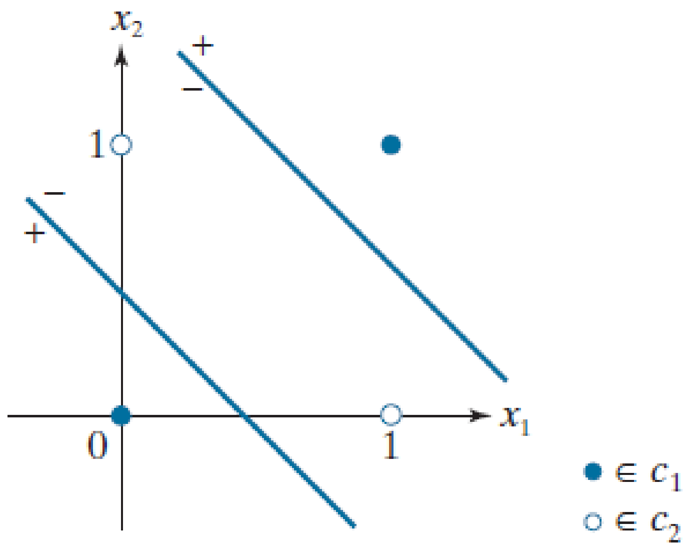
- Instead of a sigmoid or similar function in the final output layer, sometimes a ***softmax*** *function* used instead in multilayer neural network for multiclass classification problems.
- The activation values in a softmax implementation are given by

$$a_i(L) = \frac{e^{z_i(L)}}{\sum_{k=1}^{N_L} e^{z_k(L)}},$$

  where the summation is over all N_L outputs.
- In this formulation, the sum of all activations is 1, thus giving the outputs a probabilistic interpretation.

Linearly Non-separable Case

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

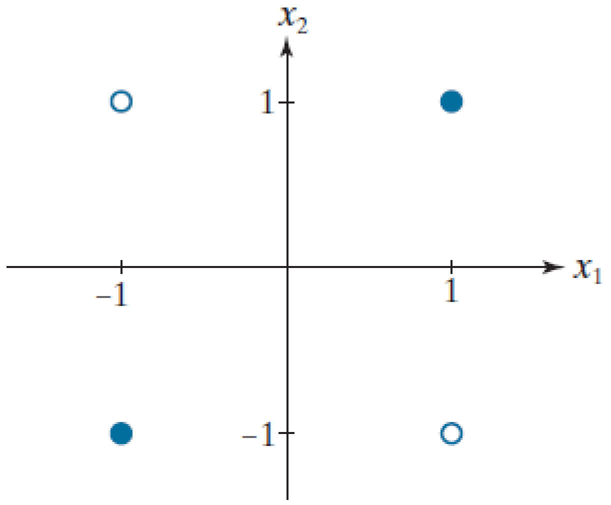


- Multilayer neural networks are needed to solve the linearly non-separable problems.
- Due to their use of “hard” thresholding functions, perceptrons’ sensitivity to the sign of small signals can cause serious stability problems in a multilayer interconnected system, making perceptrons unsuitable for layered architectures.
- The solution is to change the characteristic of the activation function from a hardlimiter to a smooth function for activation.

XOR Data Pattern Classification

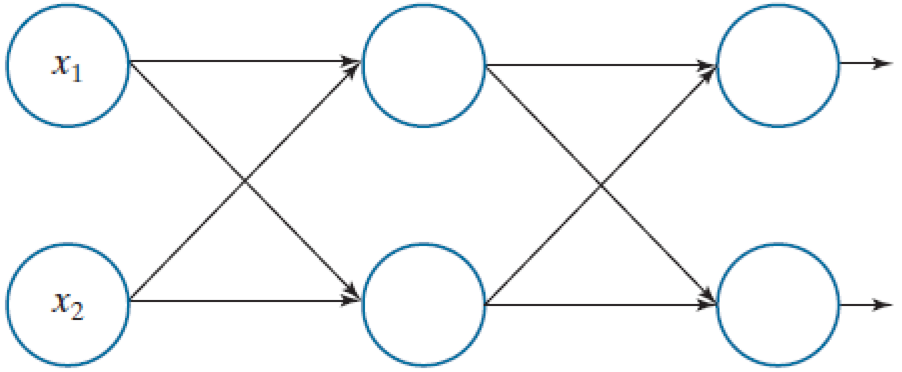
Train a three-layer fully connected neural network to classify the input data **X**, with the desired membership response **R**:

- One input layer (with two components/features)
- One hidden layer (with two neurons)
- One output layer (with two neurons)
- Activation function for the hidden layer and output layer is the **sigmoid function**



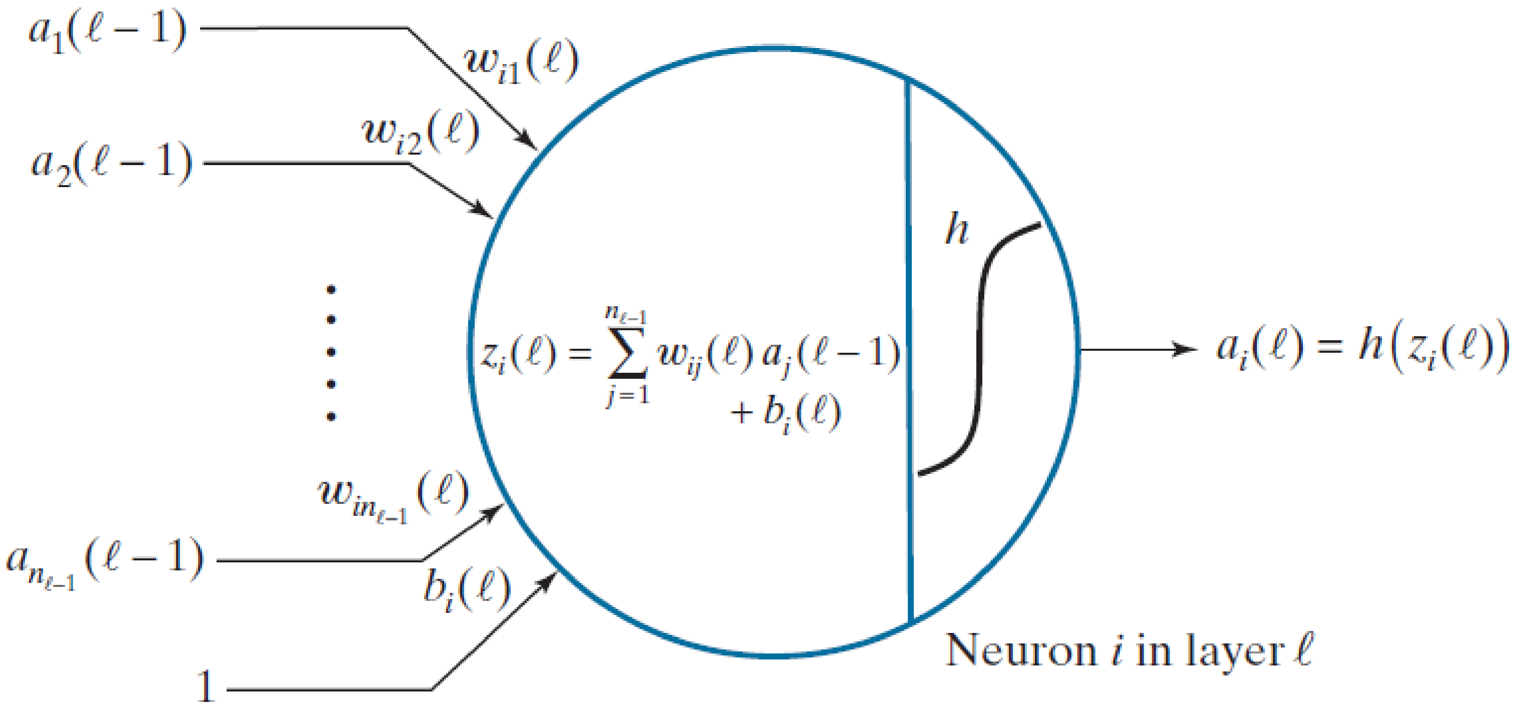
Pattern matrix **X** and class membership matrix **R** are:

$$\mathbf{X} = \begin{bmatrix} 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{bmatrix}$$



$$\mathbf{R} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Backpropagation of Error Gradient

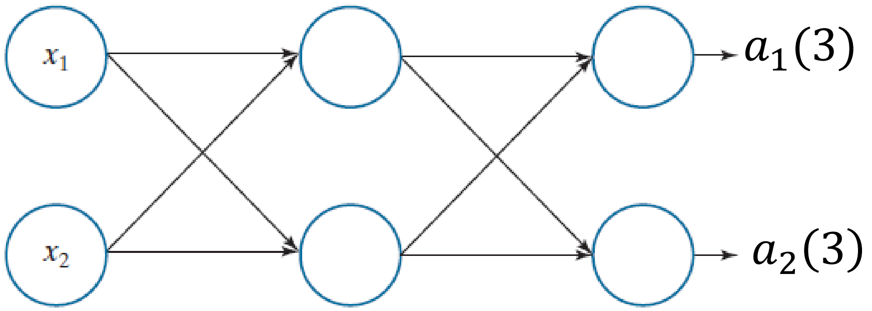


$a_1(\ell-1)$
$w_{i1}(\ell)$
$a_2(\ell-1)$
$w_{i2}(\ell)$
$w_{in_{\ell-1}}(\ell)$
$a_{n_{\ell-1}}(\ell-1)$
$b_i(\ell)$
1
$z_i(\ell) = \sum_{j=1}^{n_{\ell-1}} w_{ij}(\ell)\, a_j(\ell-1) + b_i(\ell)$
h
$a_i(\ell) = h(z_i(\ell))$
Neuron i in layer ℓ

- Previously, the activation function is a linear (identify) function, where $\frac{\partial E}{\partial a_i(l)} = \frac{\partial E}{\partial z_i(l)}$, since $a_i(l)$=$h(z_i(l)) = z_i(l)$.
- In general, $\frac{\partial E}{\partial z_i(l)} = \frac{\partial E}{\partial a_i(l)} \frac{\partial a_i(l)}{\partial z_i(l)} = \frac{\partial E}{\partial a_i} \frac{d(z_i(l))}{dz_i(l)} = \frac{\partial E}{\partial a_i} h'(z_i(l))$.
- Therefore, we need to integrate $h'\big(z_i(l)\big)$ in the backpropagation formulation derived earlier.

The output error with respect to a single $\mathbf{x}$

The activation function is $h(\,)$ for both the hidden layer and output layer



Desired response: $\mathbf{r} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}$

$$\boldsymbol{A}(3) = \begin{bmatrix} a_1(3) \\ a_2(3) \end{bmatrix} = \begin{bmatrix} h[z_1(3)] \\ h[z_2(3)] \end{bmatrix}, \text{ where } \boldsymbol{Z}(3) = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$$

Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(3)\|^2 = \frac{1}{2}\{[r_1 - a_1(3)]^2 + [r_2 - a_2(3)]^2\}$

The gradient of the output error with respect to the final output $\boldsymbol{A}(3)$ is:

$$\frac{\partial E}{\partial \boldsymbol{A}(3)} = \begin{bmatrix} \dfrac{\partial E}{\partial a_1(3)} \\ \dfrac{\partial E}{\partial a_2(3)} \end{bmatrix} = -\begin{bmatrix} r_1 - a_1(3) \\ r_2 - a_2(3) \end{bmatrix} = \boldsymbol{A}(3) - \mathbf{r}$$

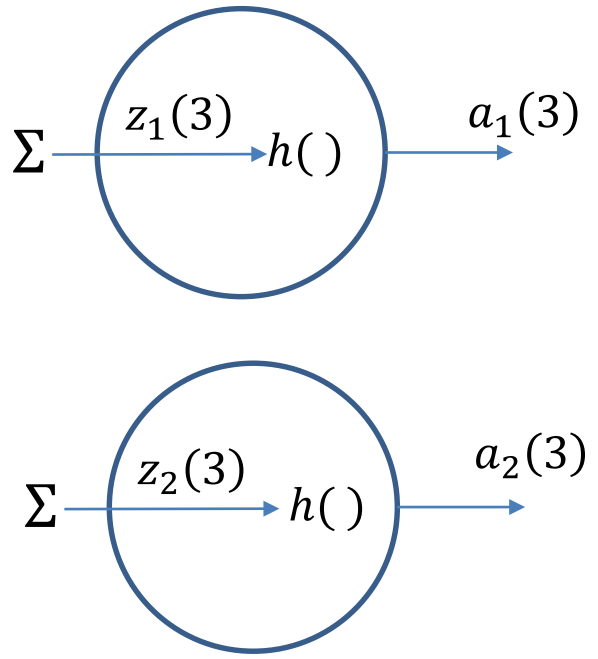
The newly defined $\boldsymbol{D}(3)$

The gradient of the output error with respect to the input of the final layer $\boldsymbol{Z}(3)$ is:

$$\boldsymbol{D}(3) = \frac{\partial E}{\partial \boldsymbol{Z}(3)} = \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_2(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \frac{\partial a_1(3)}{\partial z_1(3)} \\ \frac{\partial E}{\partial a_1(3)} \frac{\partial a_1(3)}{\partial z_1(3)} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} h'(z_1(3)) \\ \frac{\partial E}{\partial a_1(3)} h'(z_2(3)) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix} \odot \begin{bmatrix} h'(z_1(3)) \\ h'(z_2(3)) \end{bmatrix}$$

Elementwise Multiplication



Since $\frac{\partial E}{\partial \boldsymbol{A}(3)} = \begin{bmatrix} \frac{\partial E}{\partial a_1(3)} \\ \frac{\partial E}{\partial a_2(3)} \end{bmatrix} = \boldsymbol{A}(3) - \mathbf{r}$

$$\boldsymbol{D}(3) = [\boldsymbol{A}(3) - \mathbf{r}] \odot \begin{bmatrix} h'(z_1(3)) \\ h'(z_2(3)) \end{bmatrix}$$

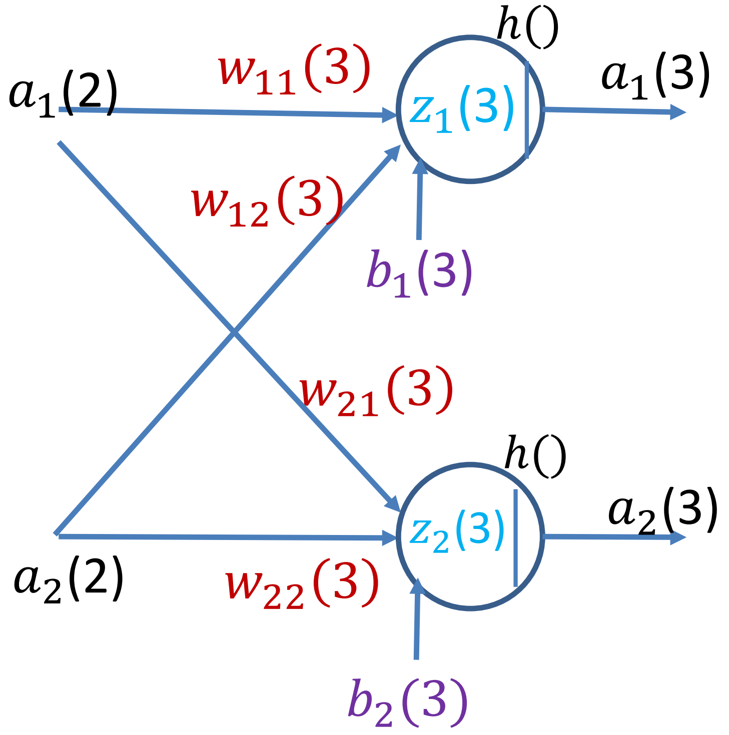
Modified Gradient of the Error wrt. Weights

$$\frac{\partial E}{\partial w_{11}(3)} = \frac{\partial E}{\partial z_1(3)} \frac{\partial z_1(3)}{\partial w_{11}(3)} = \frac{\partial E}{\partial z_1(3)} a_1(2)$$

$$\frac{\partial E}{\partial w_{12}(3)} = \frac{\partial E}{\partial z_1(3)} \frac{\partial z_1(3)}{\partial w_{12}(3)} = \frac{\partial E}{\partial z_1(3)} a_2(2)$$

$$\frac{\partial E}{\partial w_{21}(3)} = \frac{\partial E}{\partial z_2(3)} \frac{\partial z_2(3)}{\partial w_{21}(3)} = \frac{\partial E}{\partial z_2(3)} a_1(2)$$

$$\frac{\partial E}{\partial w_{22}(3)} = \frac{\partial E}{\partial z_2(3)} \frac{\partial z_2(3)}{\partial w_{22}(3)} = \frac{\partial E}{\partial z_2(3)} a_2(2)$$



$$\frac{\partial E}{\partial \boldsymbol{W}(3)} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}(3)} & \frac{\partial E}{\partial w_{12}(3)} \\ \frac{\partial E}{\partial w_{21}(3)} & \frac{\partial E}{\partial w_{22}(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_2(3)} \end{bmatrix} [a_1(2) \quad a_2(2)] = \frac{\partial E}{\partial \boldsymbol{Z}(3)} \boldsymbol{A}(2)^{\mathrm{T}} = \boldsymbol{D}(3)\boldsymbol{A}(2)^{\mathrm{T}}$$

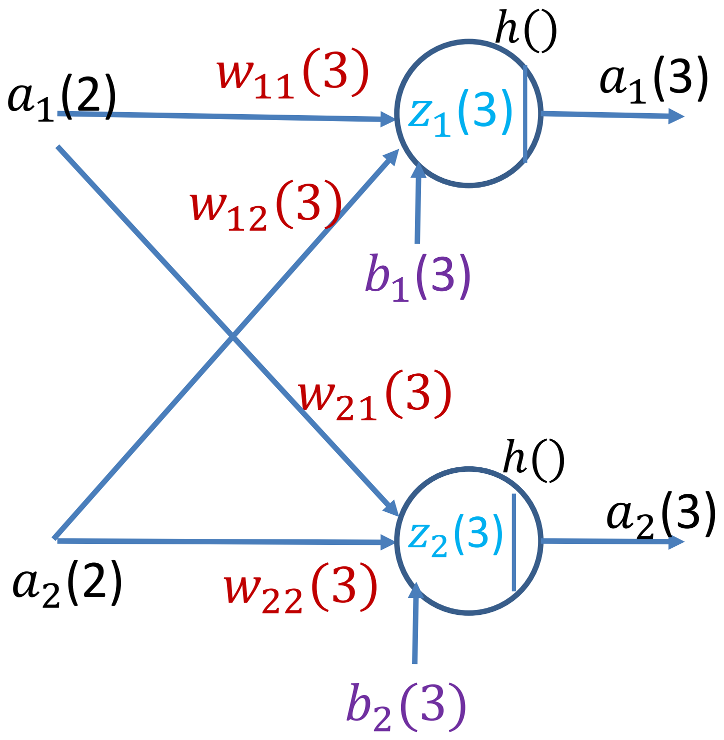
where $\boldsymbol{Z}(3) = \begin{bmatrix} z_1(3) \\ z_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$

Modified Gradient of the Error wrt. Biases

$$\frac{\partial E}{\partial b_1(3)} = \frac{\partial E}{\partial z_1(3)}\frac{\partial z_1(3)}{\partial b_1(3)} = \frac{\partial E}{\partial z_1(3)}$$

$$\frac{\partial E}{\partial b_2(3)} = \frac{\partial E}{\partial z_2(3)}\frac{\partial z_2(3)}{\partial b_2(3)} = \frac{\partial E}{\partial z_2(3)}$$

$$\frac{\partial E}{\partial \boldsymbol{b}(3)} = \begin{bmatrix} \frac{\partial E}{\partial b_1(3)} \\ \frac{\partial E}{\partial b_2(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_2(3)} \end{bmatrix} = \frac{\partial E}{\partial \boldsymbol{Z}(3)} = \boldsymbol{D}(3)$$



Modified Relation between $\boldsymbol{D}(2)$ and $\boldsymbol{D}(3)$

$$\boldsymbol{D}(2) = \frac{\partial E}{\partial \boldsymbol{Z}(2)} = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix}$$

$$\frac{\partial E}{\partial z_1(2)} = \frac{\partial E}{\partial z_1(3)} \frac{\partial z_1(3)}{\partial z_1(2)} + \frac{\partial E}{\partial z_2(3)} \frac{\partial z_2(3)}{\partial z_1(2)}$$

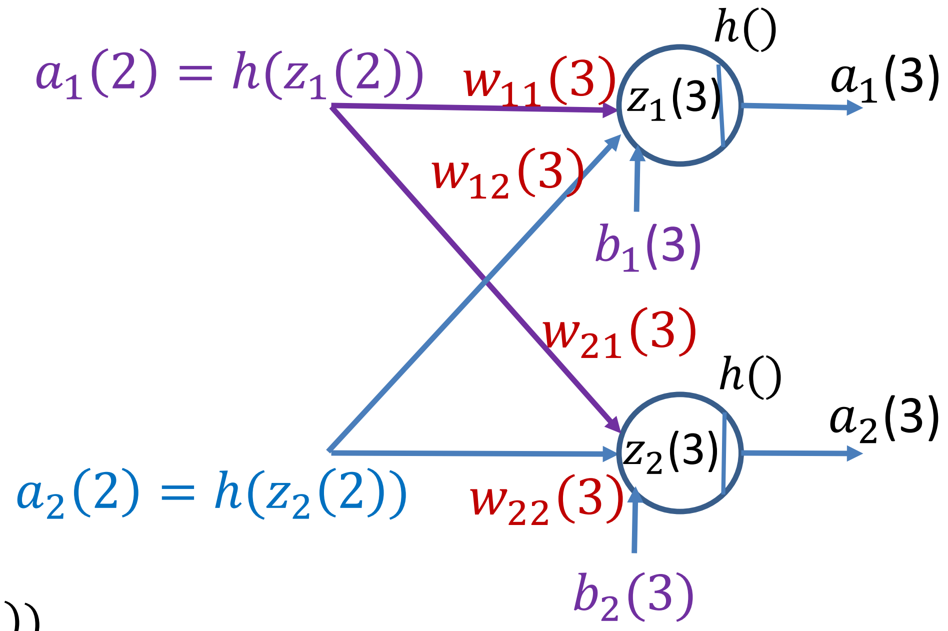
where

$$\frac{\partial z_1(3)}{\partial z_1(2)} = \frac{\partial z_1(3)}{\partial a_1(2)} \frac{\partial a_1(2)}{\partial z_1(2)} = w_{11}(3) h'(z_1(2))$$

$$\frac{\partial z_2(3)}{\partial z_1(2)} = \frac{\partial z_2(3)}{\partial a_1(2)} \frac{\partial a_1(2)}{\partial z_1(2)} = w_{21}(3) h'(z_1(2))$$

Thus

$$\frac{\partial E}{\partial z_1(2)} = \frac{\partial E}{\partial z_1(3)} w_{11}(3) h'(z_1(2)) + \frac{\partial E}{\partial z_2(3)} w_{21}(3) h'(z_1(2))$$



$a_1(2) = h(z_1(2))$

Modified Backpropagation Rule

$$\frac{\partial E}{\partial z_1(2)} = \frac{\partial E}{\partial z_1(3)} w_{11}(3) h'(z_1(2)) + \frac{\partial E}{\partial z_2(3)} w_{21}(3) h'(z_1(2))$$

Similarly,

$$\frac{\partial E}{\partial z_2(2)} = \frac{\partial E}{\partial z_1(3)} w_{12}(3) h'(z_2(2)) + \frac{\partial E}{\partial z_2(3)} w_{22}(3) h'(z_2(2))$$

Thus $\boldsymbol{D}(2) = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix} = \left\{ \begin{bmatrix} w_{11}(3) & w_{12}(3) \\ w_{21}(3) & w_{22}(3) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_1(3)} \end{bmatrix} \right\} \odot \begin{bmatrix} h'(z_1(2)) \\ h'(z_2(2)) \end{bmatrix}$

$$\boldsymbol{D}(2) = [\boldsymbol{W}(3)^{\mathrm{T}} \boldsymbol{D}(3)] \odot h'(\boldsymbol{Z}(2))$$

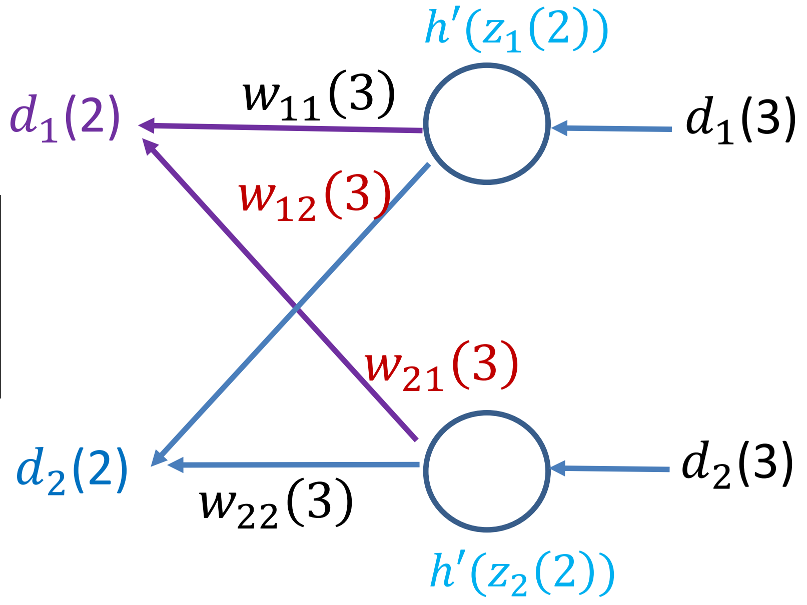
Backpropagation of $\boldsymbol{D}(3)$

To calculate $\boldsymbol{D}(2)$, we back propagate $\boldsymbol{D}(3)$ as:

$$\boldsymbol{D}(2) = \begin{bmatrix} d_1(3) \\ d_2(2) \end{bmatrix} = \boldsymbol{W}(3)^{\mathrm{T}}\boldsymbol{D}(3) \odot h'(\boldsymbol{Z}(2))$$

Note the reversed directions of the arrows, thus the transpose of the weight matrix $\boldsymbol{W}(3)^{\mathrm{T}}$.

$$\boldsymbol{D}(2) = \begin{bmatrix} d_1(2) \\ d_2(2) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix}$$



$$\boldsymbol{D}(3) = \begin{bmatrix} d_1(3) \\ d_2(3) \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_2(3)} \end{bmatrix}$$

Modified Gradient of the Error wrt. Weights (Level Two)

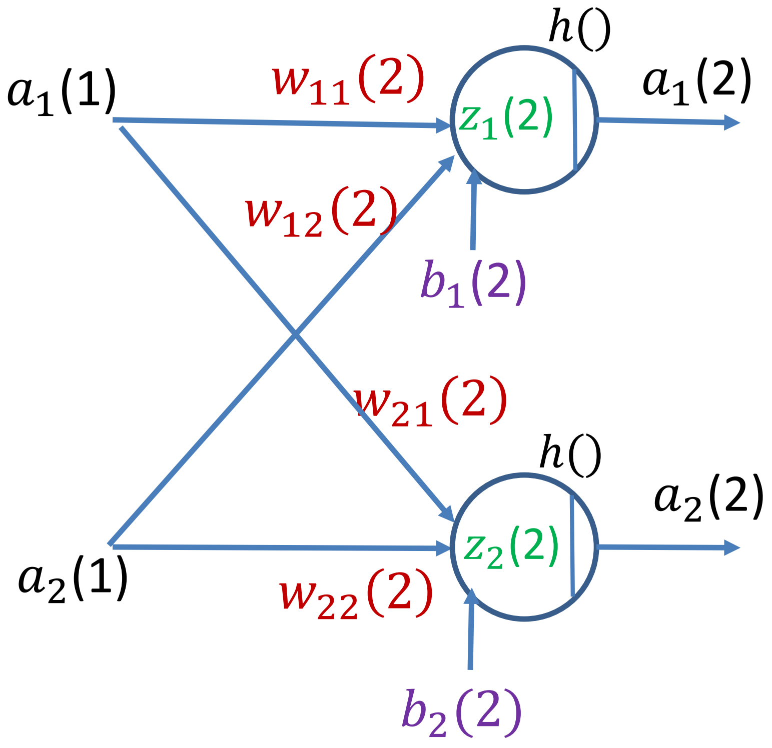
Similar to the previous derivations, for the 2nd layer:

$$\frac{\partial E}{\partial w_{11}(2)} = \frac{\partial E}{\partial z_1(2)} \frac{\partial z_1(2)}{\partial w_{11}(2)} = \frac{\partial E}{\partial z_1(2)} a_1(1)$$

$$\frac{\partial E}{\partial w_{12}(2)} = \frac{\partial E}{\partial z_1(2)} \frac{\partial z_1(2)}{\partial w_{12}(2)} = \frac{\partial E}{\partial z_1(2)} a_2(1)$$

$$\frac{\partial E}{\partial w_{21}(2)} = \frac{\partial E}{\partial z_2(2)} \frac{\partial z_2(2)}{\partial w_{21}(2)} = \frac{\partial E}{\partial z_2(2)} a_1(1)$$

$$\frac{\partial E}{\partial w_{22}(2)} = \frac{\partial E}{\partial z_2(2)} \frac{\partial z_2(2)}{\partial w_{22}(2)} = \frac{\partial E}{\partial z_2(2)} a_2(1)$$



$$\frac{\partial E}{\partial \boldsymbol{W}(2)} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}(2)} & \frac{\partial E}{\partial w_{12}(2)} \\ \frac{\partial E}{\partial w_{21}(2)} & \frac{\partial E}{\partial w_{22}(2)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix} [a_1(1) \quad a_2(1)] = \frac{\partial E}{\partial \boldsymbol{Z}(2)} \boldsymbol{A}(1)^{\mathrm{T}} = \boldsymbol{D}(2)\boldsymbol{A}(1)^{\mathrm{T}}$$

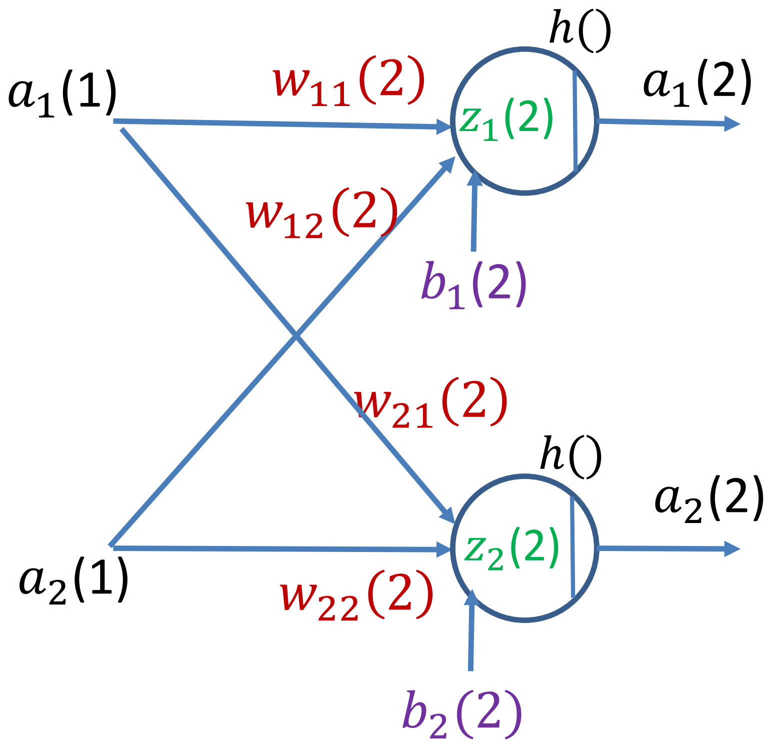
Where $\boldsymbol{D}(2)$ is obtained by back propagating $\boldsymbol{D}(3)$, and $\boldsymbol{A}(1) = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ is the input vector.

Modified Gradient wrt. Biases (2nd Layer)

$$\frac{\partial E}{\partial b_1(2)} = \frac{\partial E}{\partial z_1(2)} \frac{\partial z_1(2)}{\partial b_1(2)} = \frac{\partial E}{\partial z_1(2)}$$

$$\frac{\partial E}{\partial b_2(2)} = \frac{\partial E}{\partial z_2(2)} \frac{\partial z_2(2)}{\partial b_2(2)} = \frac{\partial E}{\partial z_2(2)}$$

$$\frac{\partial E}{\partial \boldsymbol{b}(2)} = \begin{bmatrix} \frac{\partial E}{\partial b_1(2)} \\ \frac{\partial E}{\partial b_2(2)} \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix} = \frac{\partial E}{\partial \boldsymbol{Z}(2)} = \boldsymbol{D}(2)$$



Summary of the Results

$$\boldsymbol{A}(1) = \begin{bmatrix} a_1(1) \\ a_2(1) \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\boldsymbol{Z}(2) = \begin{bmatrix} z_1(2) \\ z_2(2) \end{bmatrix} = \boldsymbol{W}(2)\boldsymbol{A}(1) + \boldsymbol{b}(2)$$

$$\boldsymbol{A}(2) = h(\boldsymbol{Z}(2)) = \begin{bmatrix} h(z_1(2)) \\ h(z_2(2)) \end{bmatrix}$$

$$\boldsymbol{Z}(3) = \begin{bmatrix} z_1(3) \\ z_2(3) \end{bmatrix} = \boldsymbol{W}(3)\boldsymbol{A}(2) + \boldsymbol{b}(3)$$

$$\boldsymbol{A}(3) = h(\boldsymbol{Z}(3)) = \begin{bmatrix} h(z_1(3)) \\ h(z_2(3)) \end{bmatrix}$$

Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(3)\|^2$

$$\boldsymbol{D}(3) = [\boldsymbol{A}(3) - \mathbf{r}] \odot \begin{bmatrix} h'(z_1(3)) \\ h'(z_2(3)) \end{bmatrix}$$

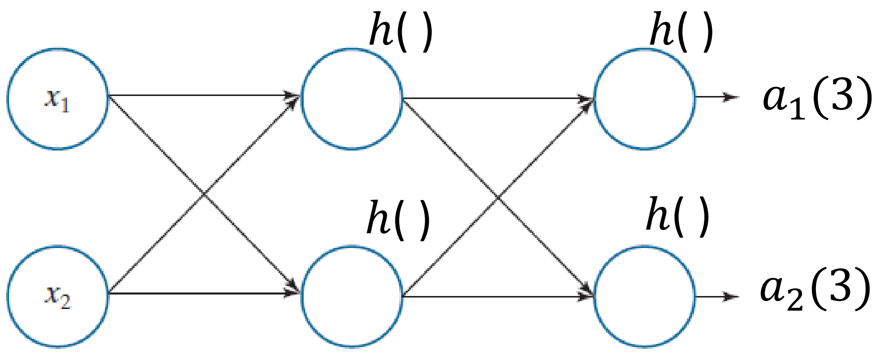
Backpropagation:

$$\boldsymbol{D}(2) = \begin{bmatrix} d_1(2) \\ d_2(2) \end{bmatrix} = \boldsymbol{W}(3)^{\mathrm{T}}\boldsymbol{D}(3) \odot h'(\boldsymbol{Z}(2))$$

$$\frac{\partial E}{\partial \boldsymbol{W}(3)} = \boldsymbol{D}(3)\boldsymbol{A}(2)^{\mathrm{T}}, \qquad \frac{\partial E}{\partial \boldsymbol{b}(3)} = \boldsymbol{D}(3)$$

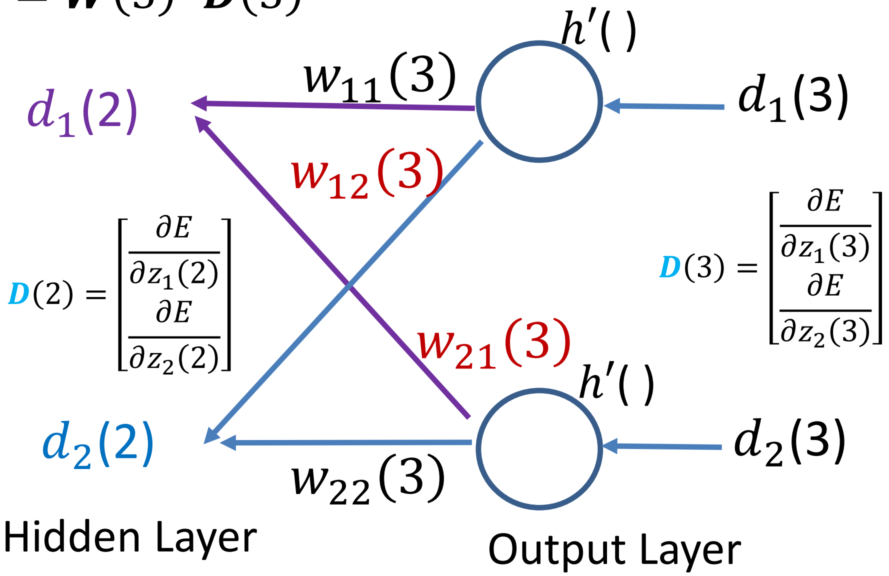
$$\frac{\partial E}{\partial \boldsymbol{W}(2)} = \boldsymbol{D}(2)\boldsymbol{A}(1)^{\mathrm{T}}, \qquad \frac{\partial E}{\partial \boldsymbol{b}(2)} = \boldsymbol{D}(2)$$

Forward Pass



Backpropagation of error gradient from output to hidden layer:

$$\boldsymbol{D}(2) = \boldsymbol{W}(3)^{\mathrm{T}}\boldsymbol{D}(3)$$



$h'(\,)$
$w_{11}(3)$
$d_1(2)$
$d_1(3)$
$w_{12}(3)$
$\boldsymbol{D}(3) = \begin{bmatrix} \frac{\partial E}{\partial z_1(3)} \\ \frac{\partial E}{\partial z_2(3)} \end{bmatrix}$
$\boldsymbol{D}(2) = \begin{bmatrix} \frac{\partial E}{\partial z_1(2)} \\ \frac{\partial E}{\partial z_2(2)} \end{bmatrix}$
$w_{21}(3)$
$h'(\,)$
$d_2(2)$
$d_2(3)$
$w_{22}(3)$
Hidden Layer
Output Layer

Training Procedure using Batch Gradient Descent

Initialize the weights and biases, and repeat the following until a convergence criterion is met (α is the *learning rate*):

- Forward pass
  $\boldsymbol{Z}(l) = \boldsymbol{W}(l)\boldsymbol{A}(l-1) + \boldsymbol{b}(l)$, and $\boldsymbol{A}(l) = h(\boldsymbol{Z}(l))$, where the layer index $l = 2, \ldots, L$. In the illustrative example, $L = 3$.
- Error: $E = \frac{1}{2}\|\mathbf{r} - \boldsymbol{A}(L)\|^2$, and its gradient at the final output layer:
  $\boldsymbol{D}(L) = [\boldsymbol{A}(L) - \mathbf{r}] \odot h'(\boldsymbol{Z}(l))$.
- Backpropagation: $\boldsymbol{D}(l) = [\boldsymbol{W}(l+1)^{\mathrm{T}}\boldsymbol{D}(l+1)] \odot h'(\boldsymbol{Z}(l))$, for $l = L - 1, \ldots, 2$.
- Update weights and biases for $l = 2, \ldots, L$:

$$\boldsymbol{W}(l) = \boldsymbol{W}(l) - \alpha \frac{\partial E}{\partial \boldsymbol{W}(l)} = \boldsymbol{W}(l) - \alpha \boldsymbol{D}(l)\boldsymbol{A}^{\mathrm{T}}(l-1);$$

$$\boldsymbol{b}(l) = \boldsymbol{b}(l) - \alpha \frac{\partial E}{\partial \boldsymbol{b}(l)} = \boldsymbol{b}(l) - \alpha \boldsymbol{D}(l).$$

- We can train the network by using the **batch mode**, where the weights and biases are updated only once after we process all the input patterns.
- The *total network output error* over all training patterns is defined as the sum of the errors of the individual patterns.

‘backprop_sigmoid_xor.m’

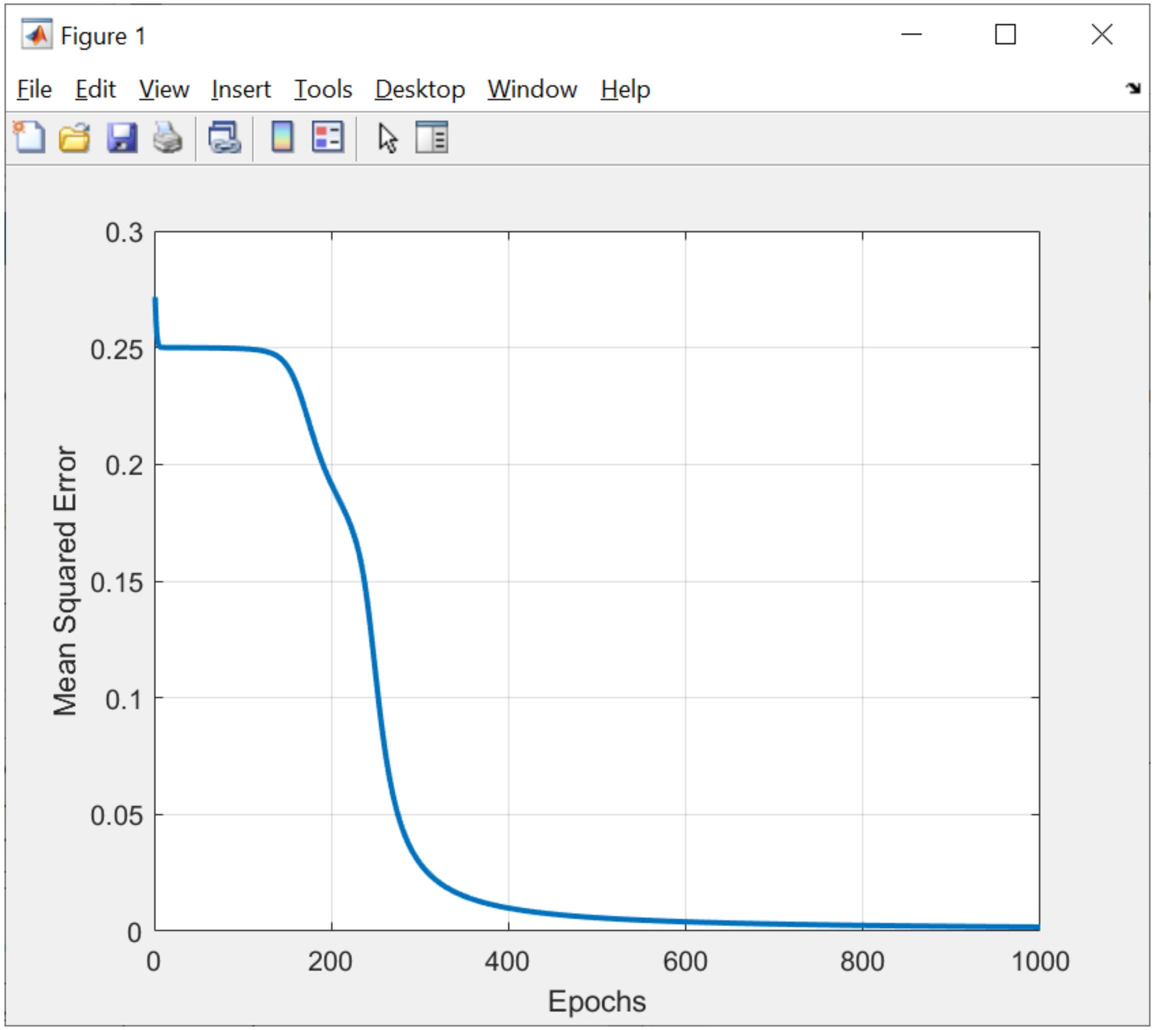
```
alpha = 1;  % learning rate
max_iter = 1000;

% Linearly separable example (1,000
epochs are enough)
% Input data pattern
%X = [1 -1 -1 1; 1 -1 1 -1];
% Response
%R = [1  0  1 0; 0  1 0  1];
```

```
% Linearly non-separable example (with
XOR pattern)
% Input data pattern
X = [1 -1 -1 1; 1 -1 1 -1];
% Response
R = [1  1  0 0; 0 0  1  1];

rng('default');
Std = 0.2;

% Initial weights and biases
W2 = Std*randn(2,2);
b2 = Std*randn(2,1);

W3 = Std*randn(2,2);
b3 = Std*randn(2,1);


mse = zeros(1, max_iter);
```

```
for epoch = 1: max_iter
    E = 0;
    W3_update = zeros(2,2);
    W2_update = zeros(2,2);
    b3_update = zeros(2,1);
    b2_update = zeros(2,1);

    for i = 1: 4
        A1 = X(:,i);
        % Z2 to replace A2 in 'backprop.m'
        Z2 = W2*A1 + b2;   % Z2 is the net input to the neuron
        % A2 is the output of the activation function on the input
        A2 = 1./(1+exp(-Z2));

        Z3 = W3*A2 + b3;
        A3 = 1./(1+exp(-Z3));

        Deriv_A3 = A3 - R(:,i);
        E = E + 0.5*norm(Deriv_A3)^2;

        % Derivative of the activation function
        hd_Z3 = A3.*(1-A3);
        D3 = Deriv_A3 .* hd_Z3;

        % Backpropagation (now with sigmoid activation functions)
        hd_Z2 = A2.*(1-A2);
        D2 = W3'*D3.*hd_Z2;

        % Update the weights and biases
        W3_update = W3_update + alpha*D3*A2';
        W2_update = W2_update + alpha*D2*A1';

        b3_update  = b3_update + alpha*D3;
        b2_update  = b2_update + alpha*D2;
    end
```

```
% Update once after the whole
batch of 4 pattern are processed
    W3 = W3 - W3_update;
    W2 = W2 - W2_update;
    b3 = b3 - b3_update;
    b2 = b2 - b2_update;

    mse(epoch) = E/4;

end
```

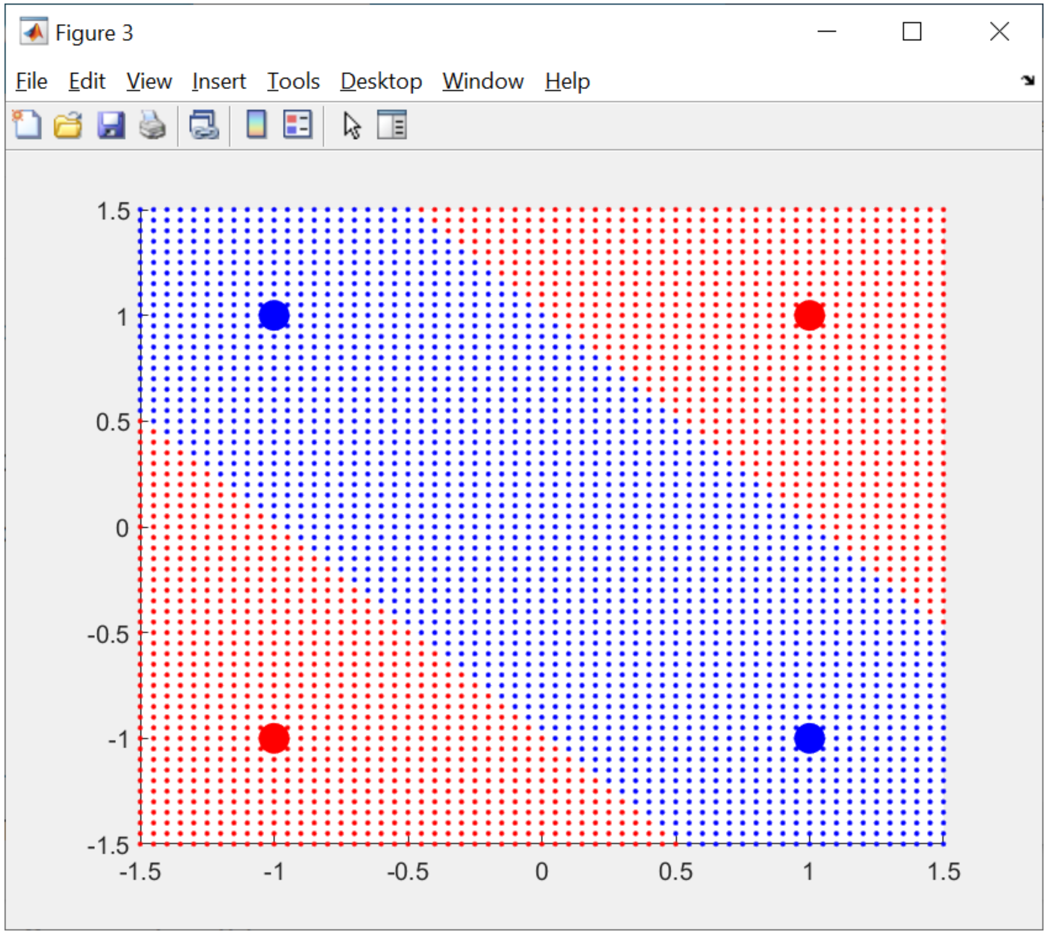
```
>> X                        >> R
X =                         R =
    1   -1   -1    1            1    1    0    0
    1   -1    1   -1            0    0    1    1

>> W2
W2 =
  -3.8633   -3.8637
   4.2082    4.2095
>> b2
b2 =
  -3.9536
  -4.3593
>> W3
W3 =
   6.6118    6.5891
  -6.6073   -6.5845
>> b3
b3 =
  -3.2829
   3.2806
```



```
final_output =
    0.9606    0.9601    0.0441    0.0441
    0.0395    0.0400    0.9558    0.9558
```

Test the Trained Network

```
figure;
hold on;
for x1 = -1.5:0.05:1.5
    for x2 = -1.5:0.05:1.5
        X_test = [x1; x2];
        A1 = X_test;
        Z2 = W2*A1 + b2;
        A2 = 1./(1+exp(-Z2));

        Z3 = W3*A2 + b3;
        A3 = 1./(1+exp(-Z3));

        if (A3(1)>=0.5)
            plot(x1, x2, 'r.');
        else
            plot(x1, x2, 'b.');
        end
    end
end
```



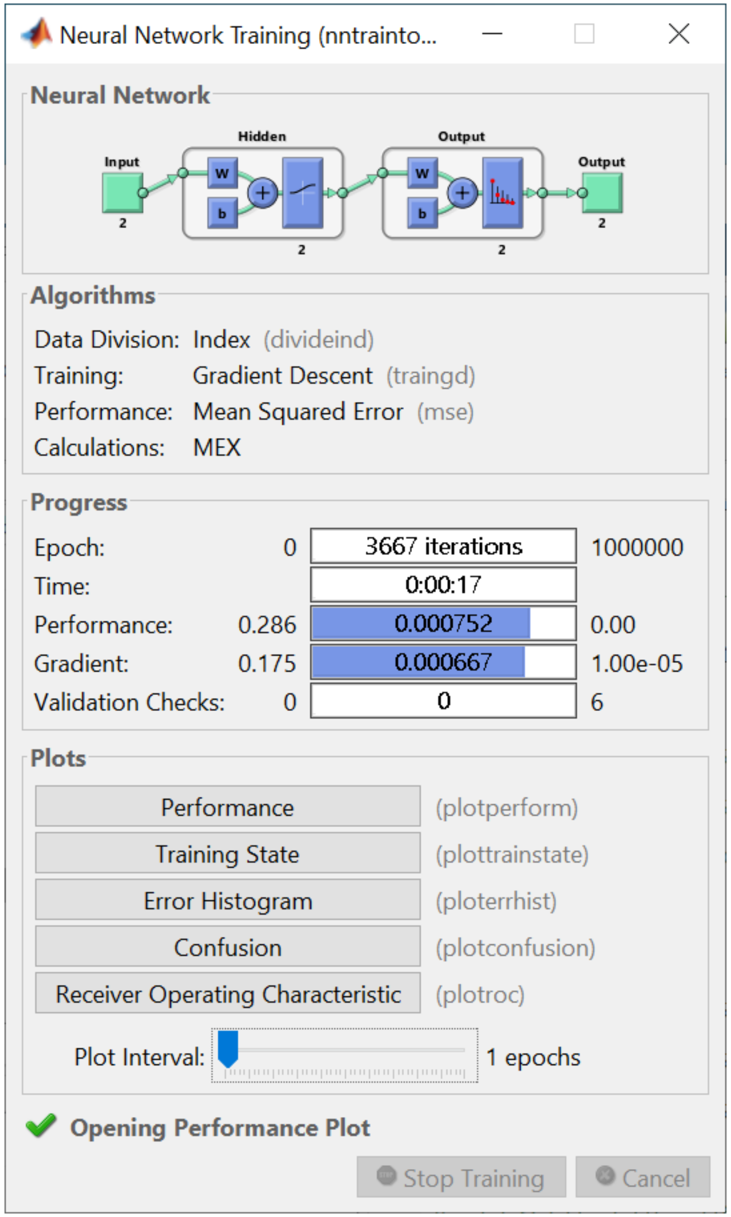
```
plot(X(1,1),X(2,1),'ro','MarkerSize',12, 'MarkerFaceColor','r');
plot(X(1,2),X(2,2),'ro','MarkerSize',12, 'MarkerFaceColor','r');
plot(X(1,3),X(2,3),'bo','MarkerSize',12, 'MarkerFaceColor','b');
plot(X(1,4),X(2,4),'bo','MarkerSize',12, 'MarkerFaceColor','b');
```

‘patternnet_demo.m’

```
% Input data pattern
X = [1 -1 -1 1; 1 -1 1 -1];
% Repeat X for training, validation and
testing
X_repeat = repmat(X, 1, 3);
% Response
R = [1  1  0 0; 0 0  1  1];
R_repeat = repmat(R, 1, 3);

rng('default');

net = patternnet(2);

% Gradient descent backpropagation
net.trainFcn = 'traingd';

net.performFcn = 'mse';

net.layers{1}.transferFcn = 'logsig';

% Maximum number of epochs to train
net.trainParam.epochs = 1000000;
```

```
% Learning rate (default = 0.01)
net.trainParam.lr  = 0.5;




% Separates targets into three sets:
% training, validation, and testing,
% according to indices provided

net.divideFcn = 'divideind';
net.divideParam.trainInd = 1:4;
net.divideParam.valInd = 5:8;
net.divideParam.testInd= 9:12;

% Train the network
net = train(net, X_repeat, R_repeat);
```

Training



Neural Network Training (nntrainto...

Neural Network

Algorithms

Data Division:	Index (divideind)
Training:	Gradient Descent (traingd)
Performance:	Mean Squared Error (mse)
Calculations:	MEX

Progress

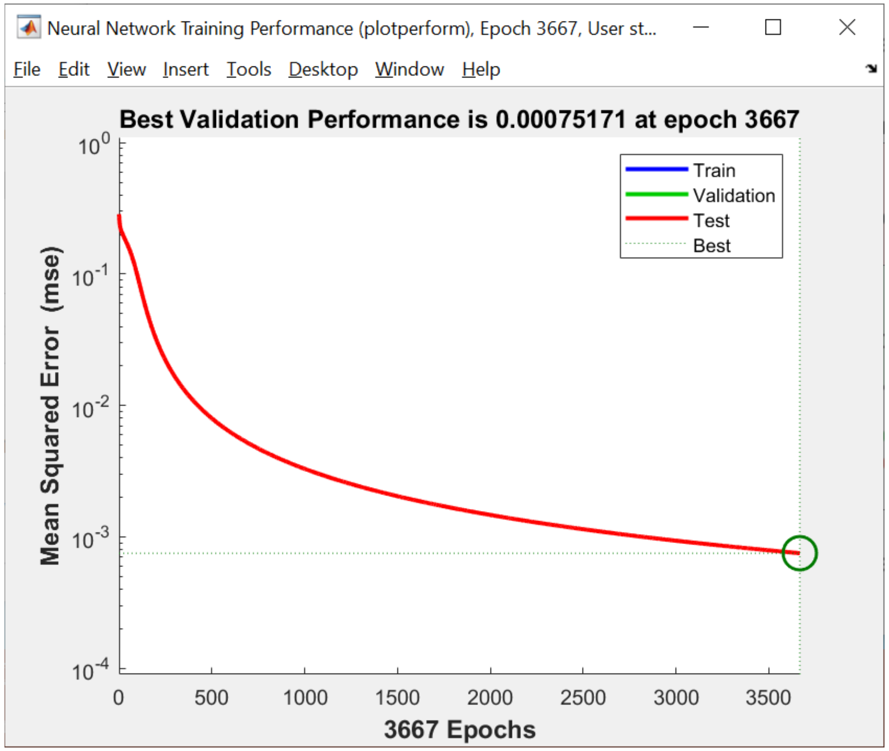
Epoch:	0	3667 iterations	1000000
Time:		0:00:17	
Performance:	0.286	0.000752	0.00
Gradient:	0.175	0.000667	1.00e-05
Validation Checks:	0	0	6

Plots

Performance	(plotperform)
Training State	(plottrainstate)
Error Histogram	(ploterrhist)
Confusion	(plotconfusion)
Receiver Operating Characteristic	(plotroc)

Plot Interval: 1 epochs

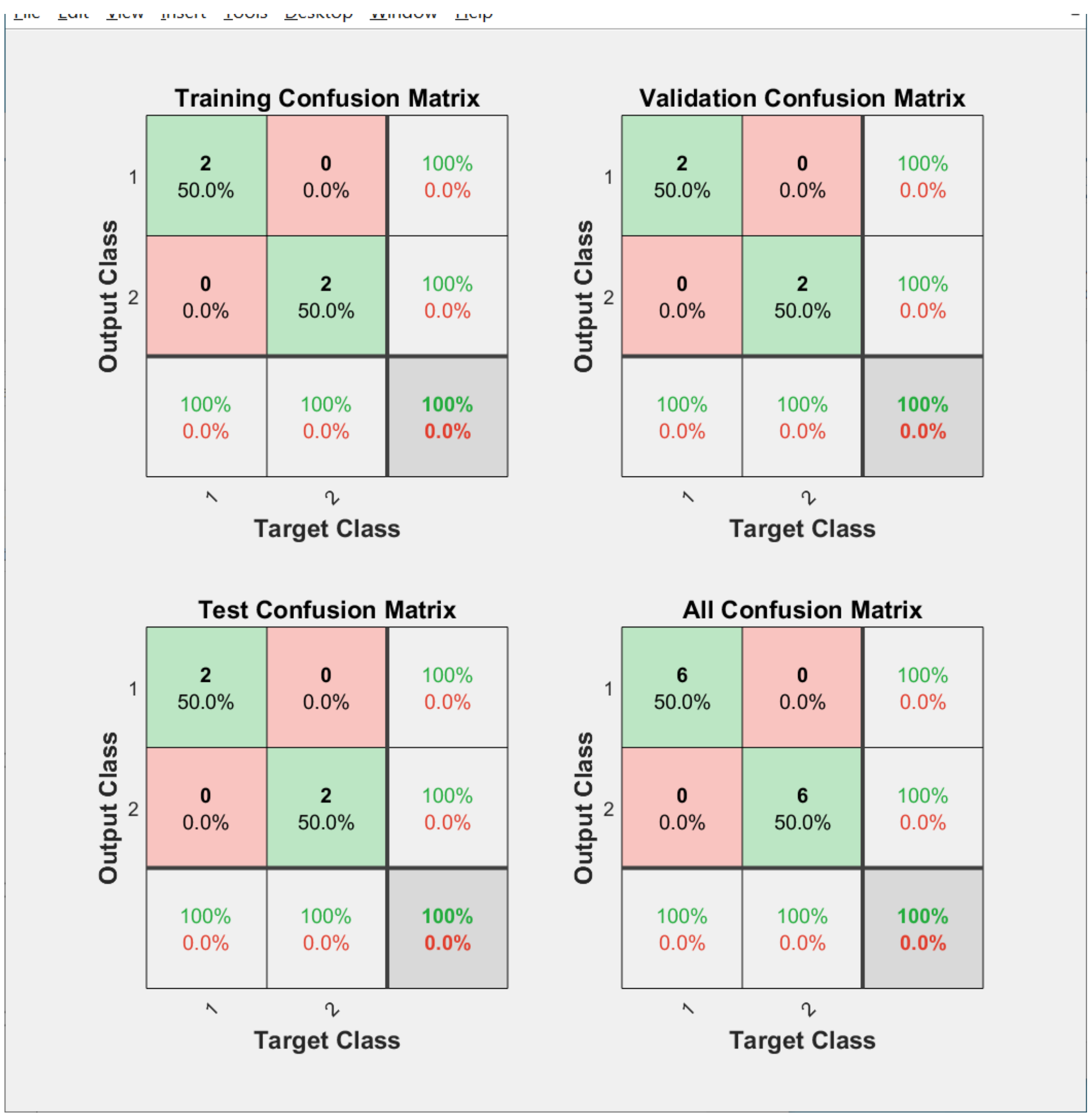
✓ **Opening Performance Plot**



Neural Network Training Performance (plotperform), Epoch 3667, User st...

Best Validation Performance is 0.00075171 at epoch 3667

Confusion Matrix



Training Confusion Matrix

Output Class \ Target Class	1	2	
1	2 50.0%	0 0.0%	100% 0.0%
2	0 0.0%	2 50.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

Validation Confusion Matrix

Output Class \ Target Class	1	2	
1	2 50.0%	0 0.0%	100% 0.0%
2	0 0.0%	2 50.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

Test Confusion Matrix

Output Class \ Target Class	1	2	
1	2 50.0%	0 0.0%	100% 0.0%
2	0 0.0%	2 50.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

All Confusion Matrix

Output Class \ Target Class	1	2	
1	6 50.0%	0 0.0%	100% 0.0%
2	0 0.0%	6 50.0%	100% 0.0%
	100% 0.0%	100% 0.0%	100% 0.0%

Weights and Biases Learned

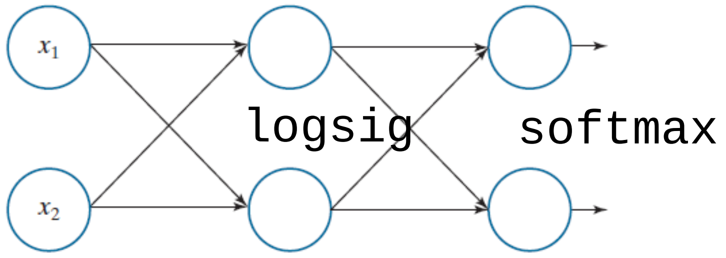
```
% Verify the output using the weights
% and biases learned by net
final_output = zeros(2, 4);

for i = 1: 4
    A1 = X(:,i);
    W2 = cell2mat(net.IW);
    b2 = net.b{1};
    Z2 = W2*A1 + b2;

% Hidden layer uses sigmoid transfer
function
    %A2 = 1./(1+exp(-Z2));
    A2 = logsig(Z2);

    W3 = cell2mat(net.LW(2));
    b3 = net.b{2};

    Z3 = W3*A2 + b3;
```

```
% The output layer uses Softmax transfer
A3 = softmax(Z3);
    final_output(:,i) = A3;
end

final_output =
    0.9679    0.9629    0.0421    0.0345
    0.0321    0.0371    0.9579    0.9655
```

$$\mathbf{R} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

W2 =
3.6689 -3.6128
4.0819 -3.4690

W3 =
-3.5056 3.9379
3.6894 -3.0098

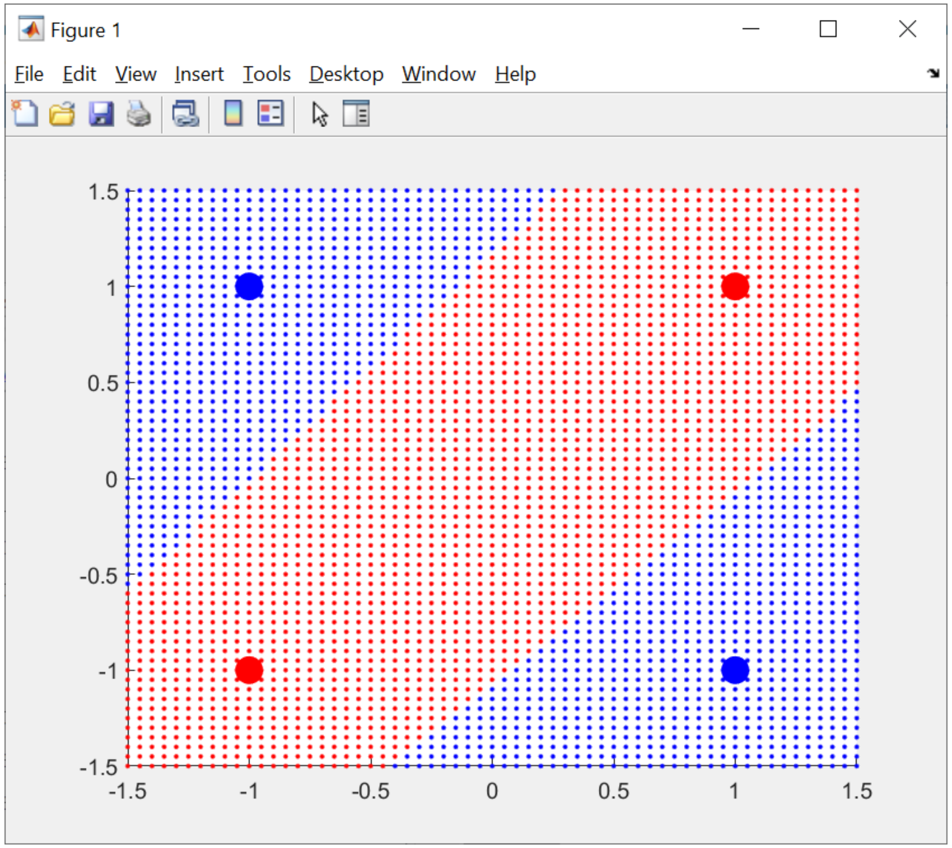


b2 =
-3.8523
3.9549

b3 =
-1.6999
1.6113

Test the Trained Network using **sim**

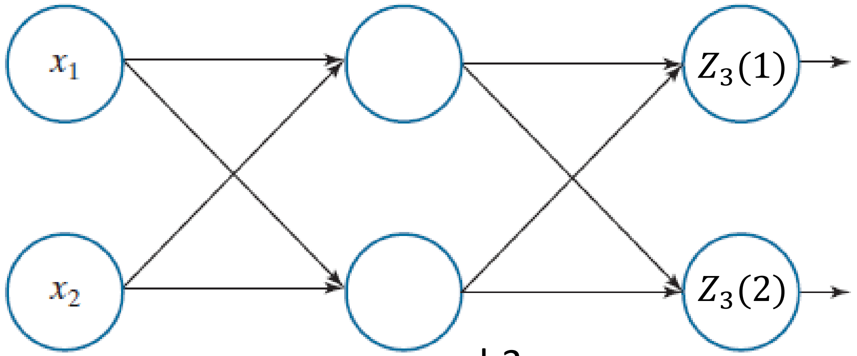
```
figure;
hold on;
for x1 = -1.5:0.05:1.5
    for x2 = -1.5:0.05:1.5
        X_test = [x1; x2];
        y = sim(net, X_test);

        if (y(1)>=0.5)
            plot(x1, x2, 'r.');
        else
            plot(x1, x2, 'b.');
        end
    end
end
```



Sigmoid Output Range: (0,1)

W3 =

6.6118	6.5891
-6.6073	-6.5845



$$A_3(1) = \frac{1}{1+e^{-Z_3(1)}} = \frac{1}{1+e^{-u}}$$

$$A_3(2) = \frac{1}{1+e^{-Z_3(2)}} \approx \frac{1}{1+e^{-(-u)}}$$

b3 =

-3.2829
3.2806

```
Z3 = W3*A2 + b3
```

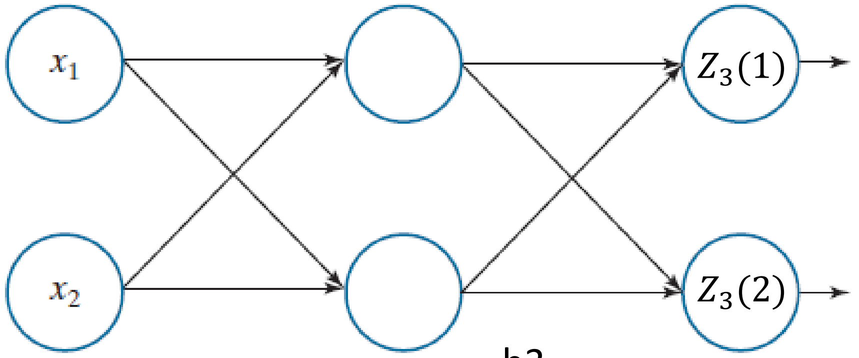
$$Z_3(1) \approx -Z_3(2) = u$$

$$A_3(1) + A_3(2) \approx \frac{1}{1+e^{-u}} + \frac{1}{1+e^{u}} = \frac{1+e^{-u}+1+e^{u}}{(1+e^{-u})(1+e^{u})} = 1$$

Softmax Output Range: [0,1]

Compared to:
W3 (sigmoid) =
6.6118 6.5891
-6.6073 -6.5845

W3 =
-3.5056 3.9379
3.6894 -3.0098



$$A_3(1) = \frac{e^{Z_3(1)}}{e^{Z_3(1)} + e^{Z_3(2)}}$$

$$A_3(2) = \frac{e^{Z_3(2)}}{e^{Z_3(1)} + e^{Z_3(2)}}$$

Compared to:
b3 (sigmoid)=
-3.2829
3.2806

b3 =
-1.6999
1.6113

$$A_3(1) + A_3(2) = 1$$

If $Z_3(1) \approx -Z_3(2) = v = \frac{u}{2}$, then Softmax is equivalent to Sigmoid for two-class case:

$A_3(1) = \frac{e^v}{e^v + e^{-v}} = \frac{1}{1+e^{-2v}} = \frac{1}{1+e^{-u}}$: sigmoid function on u;

$A_3(2) = \frac{e^{-v}}{e^v + e^{-v}} = \frac{1}{1+e^{2v}} = \frac{1}{1+e^{-(-u)}}$: sigmoid function on $(-u)$.

'mlp_demo.py'

```
import numpy as np

x1 = np.array([1, 1])
x2 = np.array([-1, -1])
x3 = np.array([-1, 1])
x4 = np.array([1, -1])
X = np.vstack([x1, x2, x3, x4])
# Features are along the row

y1 = np.array([1, 0])
y2 = np.array([1, 0])
y3 = np.array([0, 1])
y4 = np.array([0, 1])
y = np.vstack([y1, y2, y3, y4])
```

```
from sklearn.neural_network import
MLPClassifier
clf = MLPClassifier(
   # number of neurons in the hidden layer
   hidden_layer_sizes = (2),
activation='logistic',
   random_state= 100,
   alpha = 0.001,
   solver='lbfgs',
   max_iter=10000000
   )

clf.fit(X, y)
```

Weights and Biases Learned

```
clf.n_layers_
clf.loss_
clf.predict(X)

# Weight matrix for the hidden layer
clf.coefs_[0]
# Bias vector for the hidden layer
clf.intercepts_[0]

# Output layer weights and biases
clf.coefs_[1]
clf.intercepts_[1]
```

```
# Verification of the final output

Z2 = np.matmul(X,clf.coefs_[0]) +
clf.intercepts_[0]
A2 = 1/(1 + np.exp(-Z2))

Z3 = np.matmul(A2,clf.coefs_[1]) +
clf.intercepts_[1]
A3 = 1/(1 + np.exp(-Z3))
```

A3

```
Out[ ]:
array([[0.99004116, 0.00990482],
      [0.99006107, 0.00988508],
      [0.00984579, 0.99001524],
      [0.01009781, 0.99004976]])
```

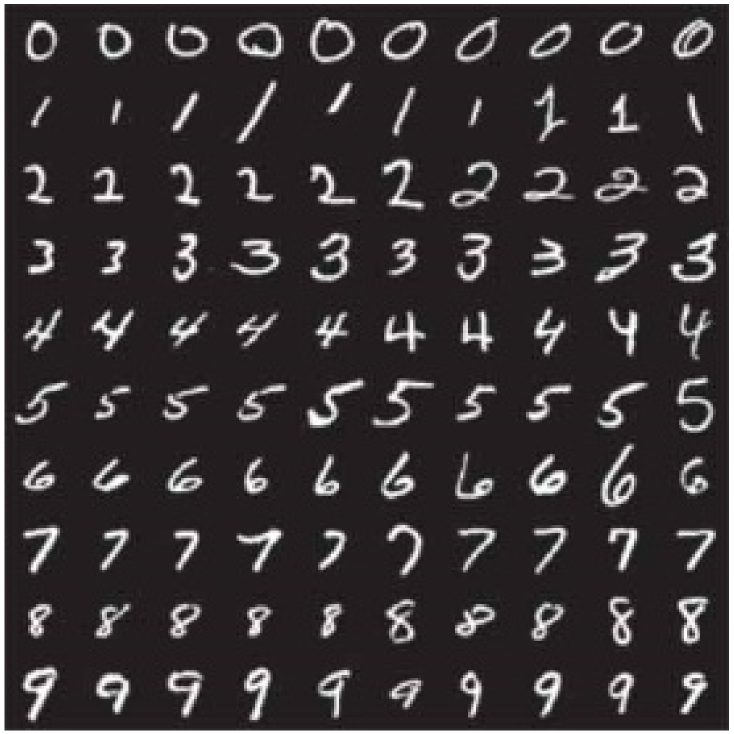
clf.predict_proba(X)

```
Out[]:
array([[0.99004116, 0.00990482],
      [0.99006107, 0.00988508],
      [0.00984579, 0.99001524],
      [0.01009781, 0.99004976]])
```

clf.predict(X)

```
Out[95]:
array([[1, 0],
      [1, 0],
      [0, 1],
      [0, 1]])
```

MNIST Image Dataset



Convolutional Neural Network (CNN)

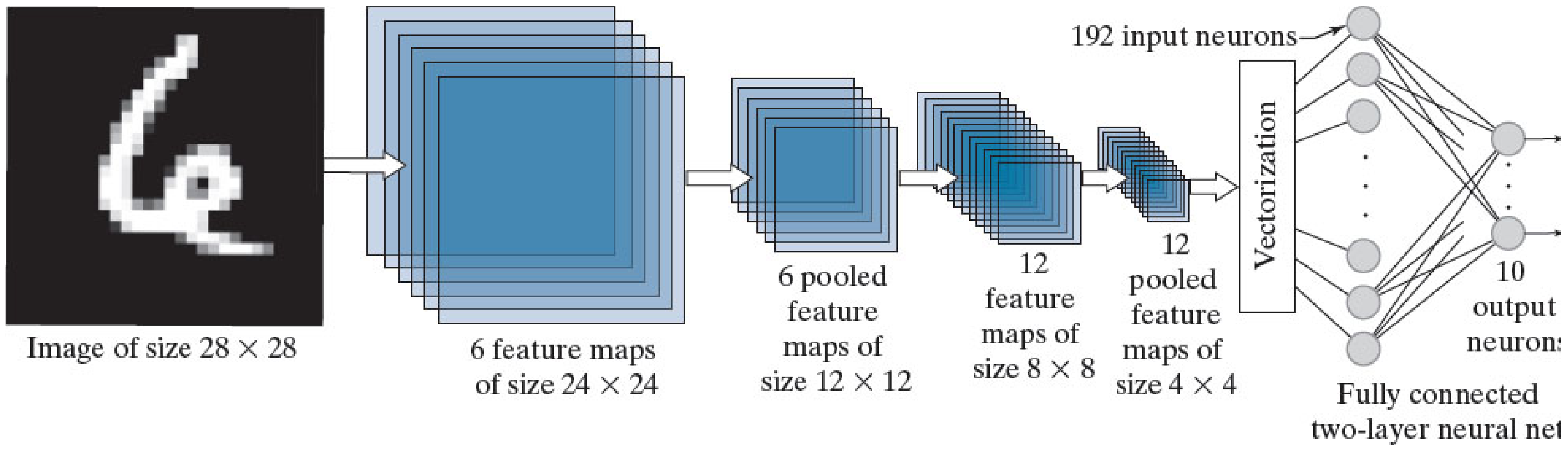
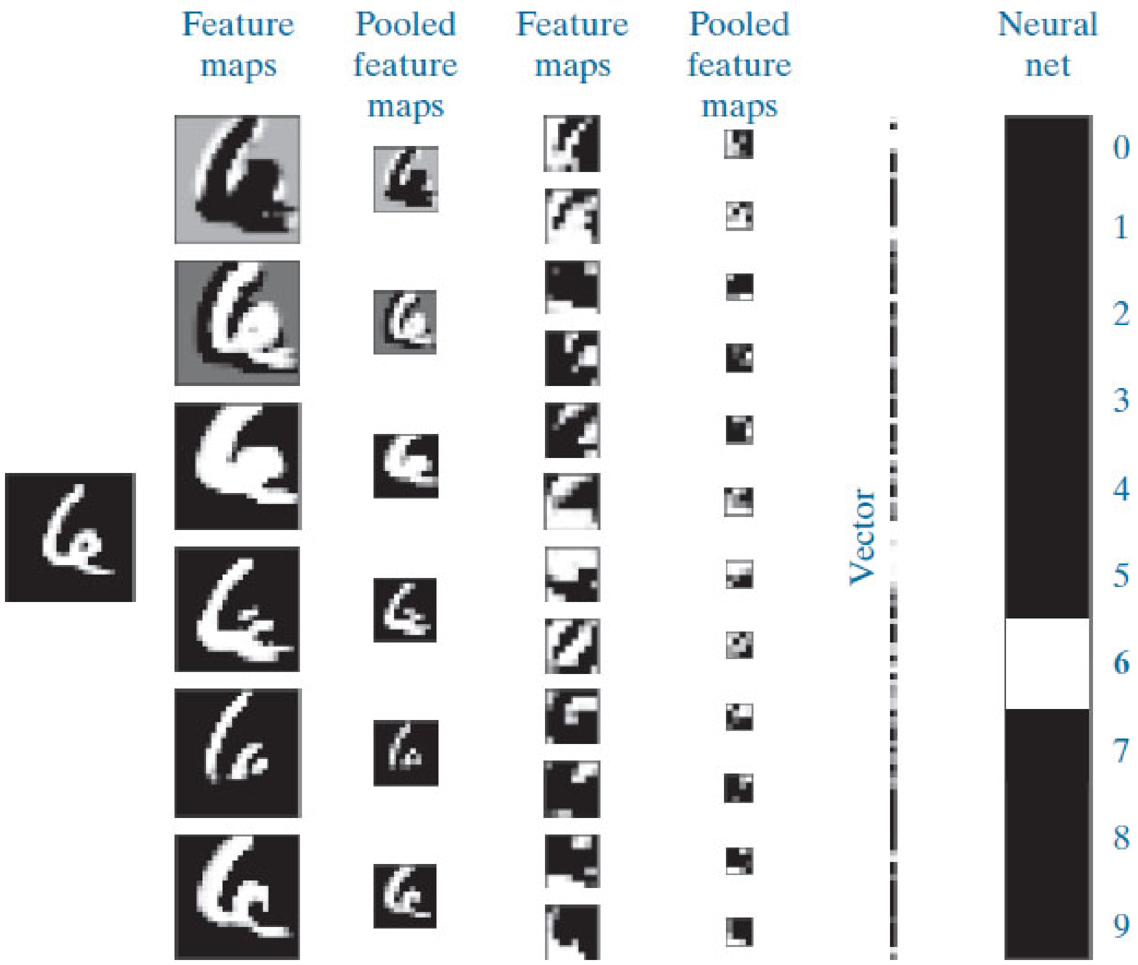


Image of size 28 × 28
6 feature maps of size 24 × 24
6 pooled feature maps of size 12 × 12
12 feature maps of size 8 × 8
12 pooled feature maps of size 4 × 4
Vectorization
192 input neurons
10 output neurons
Fully connected two-layer neural net

CNN used to recognize the ten digits in the MNIST database. The system was trained with 60,000 numerical character images of the same size as the image shown on the left.

Feature Maps



The output high value (in white) indicates that the CNN recognized the input properly.