Multilayer Perceptrons (MLPs)

Chapter 5

Kiran Bagale

St. Xavier's College

July, 2025

5.1 Introduction

What is a Multilayer Perceptron (MLP)?

- Extension of Rosenblatt's perceptron using multiple layers.
- Overcomes limitations of single-layer networks (linear separability).
- Core elements:
  - Nonlinear differentiable activation functions
  - One or more hidden layers
  - Fully connected neurons
- Enables learning complex decision boundaries.

MLP Architecture Signal Flow

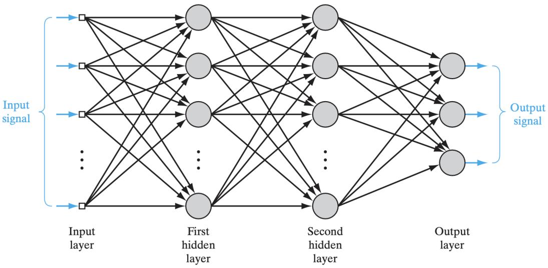


Figure a: Architectural graph of a multilayer perceptron with two hidden layers

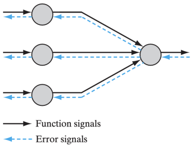


Figure b: Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back propagation of error signals.

MLP Architecture Signal Flow Cont...

- Each neuron connects to all neurons in the previous layer.
- **Function signal**:
  - Propagates input forward through the network
  - Performs useful computations via activations
- **Error signal**:
  - Propagates backward from output
  - Used to adjust weights
- Signal flow is layer-by-layer: input $\rightarrow$ hidden $\rightarrow$ output

Role of Hidden Neurons

- Hidden layers are not visible from input/output.
- Act as **feature detectors**.
- Perform nonlinear transformation to a **feature space**.
- Enable better class separation in pattern classification.
- Compute:
  1. Nonlinear output based on inputs and weights
  2. Gradient estimates for learning (used in backprop)

Learning Process in MLPs

Forward Phase

- Input signal is propagated forward.
- Outputs are computed layer by layer.

Backward Phase (Backpropagation)

- Error is computed at the output.
- Error signal propagates backward.
- Synaptic weights are adjusted using gradients.

Credit-Assignment Problem & Backpropagation

- How to assign responsibility to **hidden neurons**?
- Output errors are visible, but internal decisions are not.
- This is the **credit-assignment problem**.
- **Backpropagation** solves it:
  - Uses chain rule to distribute error to all weights.
  - Enables hidden neurons to learn indirectly from total error.
- **Result:** Effective supervised learning in deep networks.

5.2 BATCH LEARNING AND ON-LINE LEARNING

- We train an MLP using labeled examples:

$$\mathcal{T} = \{x(n), d(n)\}_{n=1}^{N}$$

- For each training input $x(n)$, the network produces output $y_j(n)$.
- The error signal at output neuron j is:

$$e_j(n) = d_j(n) - y_j(n) \tag{1}$$

- The total instantaneous error energy is:

$$e(n) = \sum_{j \in C} \frac{1}{2} e_j^2(n) \tag{2}$$

- Learning goal: minimize the error energy by adjusting synaptic weights.

Batch Learning

Definition

Adjustments to weights are made **after all training examples** have been presented (one full epoch).

- Uses average error energy:

$$e_{av}(N) = \frac{1}{N}\sum_{n=1}^{N} e(n) = \frac{1}{2N}\sum_{n=1}^{N}\sum_{j\in C}\frac{1}{2}e_j^2(n) \tag{3}$$

- Suitable for parallel computation.
- Provides accurate gradient estimation.
- Demands high memory/storage.
- Effective in **nonlinear regression** tasks.

Online (Stochastic) Learning

Definition

Adjustments are made **after each training example**, one-by-one within the epoch.

- Minimizes instantaneous error $e(n)$.
- Updates are fast and memory-efficient.
- Introduces randomness (stochastic gradient descent).
- Reduces risk of local minima.
- Adapts well to redundant or nonstationary data.

Batch vs. Online Learning

Batch Learning

- Slower updates
- High memory usage
- Accurate gradients
- Parallelizable

Online Learning

- Fast, one-by-one updates
- Low memory
- Stochastic behavior
- Better for large datasets

Conclusion

Online learning is **simple, efficient, and scalable** — ideal for large pattern classification tasks.

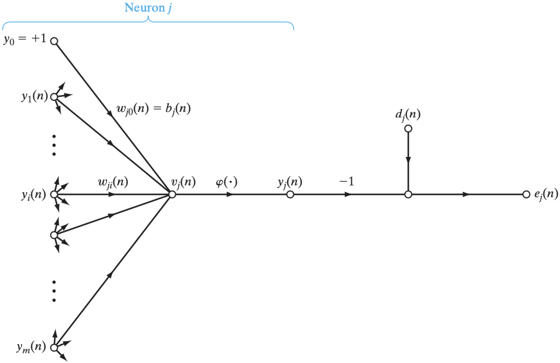
Key Takeaways

- Both methods use gradient descent but differ in update frequency.
- Batch learning is more accurate but resource-heavy.
- Online learning is lighter, more adaptive, and widely used.
- Most practical MLP implementations prefer online/stochastic learning.

Online Learning = Real-time Learning

- A supervised learning algorithm for training multilayer perceptrons.
- Adjusts weights by propagating the output error backward through the network.
- Based on gradient descent to minimize error.

Figure: Signal-flow graph highlighting the details of output neuron j.



Weight Update Mechanism

- Induced Local field $v_j(n)$ at input of the activation function with neuron j:

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{4}$$

m - total number of inputs (excluding bias)

- Output:

$$y_j(n) = \varphi(v_j(n)) \tag{5}$$

- Error:

$$e_j(n) = d_j(n) - y_j(n) \tag{6}$$

- Weight correction (Delta rule):

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{7}$$

Gradient Derivation via Chain Rule

- The back- propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\frac{\partial e(n)}{\partial w_{ji}(n)}$.

$$\frac{\partial e(n)}{\partial w_{ji}(n)} = \frac{\partial e(n)}{\partial e_j(n)} \cdot \frac{\partial e_j(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{8}$$

$$\Rightarrow \Delta w_{ji}(n) = -\eta e_j(n) \varphi'(v_j(n)) y_i(n) \tag{9}$$

- Shows dependency of weight updates on error and activation derivatives.

Sensitivity factor

- The partial derivative $\frac{\partial e(n)}{\partial w_{ji}(n)}$ represents a sensitivity factor, determining the direction of search in weight space for the synaptic weight wji.
- From equation (2)

$$\frac{\partial e(n)}{\partial e_j(n)} = e_j(n) \tag{i}$$

- From equation (1 or 6)

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{ii}$$

- From equation (5)

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi'(v_j(n)) \tag{iii}$$

- From equation (4)

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{iv}$$

Computing Local Gradient $\delta_j(n)$

Case 1: Output Neuron

- When neuron j is located in the output layer of the network, it is supplied with a desired response of its own.

$$\delta_j(n) = e_j(n)\varphi'(v_j(n)) \tag{10}$$

Case 2: Hidden Neuron

- For a hidden neuron, the error signal is calculated by recursively using the error signals of the neurons it connects to in the next layer.

$$\delta_j(n) = \varphi'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{11}$$

- $\delta_k(n)$: Local gradients of next layer
- $w_{kj}(n)$: Weights from neuron j to k

Backpropagation: Hidden Neuron Case

Consider the situation where neuron j is a hidden node. According to Eq. (10), we redefine the local gradient $\delta_j(n)$ as:

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)} \tag{12}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \cdot \varphi_j'(v_j(n)), \quad \text{neuron } j \text{ is hidden}$$

To calculate the partial derivative $\partial \mathcal{E}(n) / \partial y_j(n)$, consider:

$$\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n), \quad \text{neuron } k \text{ is an output node} \tag{13}$$

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \cdot \frac{\partial e_k(n)}{\partial y_j(n)} \tag{14}$$

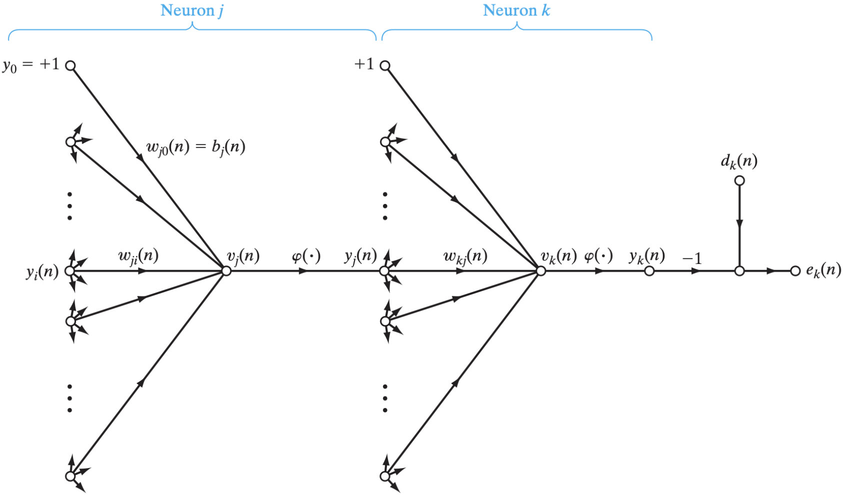


FIGURE: Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j.

Partial Derivative Using Chain Rule

Using the chain rule for $\frac{\partial e_k(n)}{\partial y_j(n)}$, we rewrite Eq. (14) as:

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{15}$$

From above figure, for output neuron k, the error is:

$$\begin{aligned} e_k(n) &= d_k(n) - y_k(n) && (1) \\ &= d_k(n) - \varphi_k(v_k(n)) && (16) \end{aligned}$$

Partial Derivatives of Output Node

- The derivative of the error w.r.t. local field $v_k(n)$ is:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'_k(v_k(n)) \tag{17}$$

- The induced local field of neuron k:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{18}$$

- Differentiating w.r.t. $y_j(n)$:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{19}$$

Combining the Derivatives

Using Eqs. (17) and (19) in Eq. (15), we get:

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n) \\ &= -\sum_k \delta_k(n) w_{kj}(n) \end{aligned} \tag{20}$$

where $\delta_k(n) = e_k(n)\varphi_k'(v_k(n))$ is the local gradient for output neuron k.

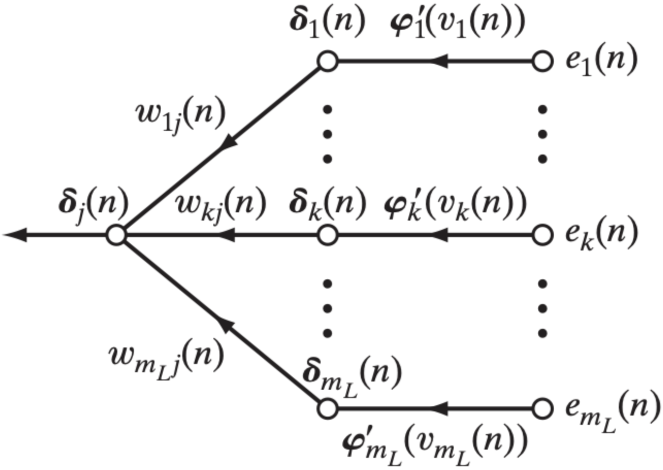


Figure: Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals

Final Backpropagation Formula (Hidden Neuron)

Substituting Eq. (20) into Eq. (12), we get the backpropagation formula for hidden neuron j:

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \quad \text{neuron } j \text{ is hidden} \tag{21}$$

Interpretation:

- The local gradient $\delta_j(n)$ is influenced by:
  - Activation derivative $\varphi_j'(v_j(n))$
  - Weighted sum of gradients from next layer

Delta Rule: Weight Correction The correction $\Delta w_{ji}(n)$ applied to the synaptic weight from neuron i to neuron j is defined by the delta rule:

$$\boxed{\Delta w_{ji}(n) = \eta \cdot \delta_j(n) \cdot y_i(n)} \tag{22}$$

- η: learning-rate parameter
- $\delta_j(n)$: local gradient of neuron j
- $y_i(n)$: input signal from neuron i

How to Compute the Local Gradient $\delta_j(n)$

The value of the local gradient $\delta_j(n)$ depends on the location of neuron j:

1. **If neuron j is an output node:**

$$\delta_j(n) = \varphi_j'(v_j(n)) \cdot e_j(n)$$

   where $e_j(n)$ is the error signal for output neuron j; see Eq. (10).

2. **If neuron j is a hidden node:**

$$\delta_j(n) = \varphi_j'(v_j(n)) \cdot \sum_k \delta_k(n) w_{kj}(n)$$

   This involves the sum of gradients from the next layer; see Eq. (21).

A. Activation Functions

- The computation of δ for each neuron in multilayer perceptron requires knowledge of the derivative of the activation function $\varphi(\cdot)$
- For the derivative to exist, the function $\varphi(\cdot)$ must be continuous
- **Differentiability** is the only requirement that an activation function has to satisfy
- Common example: **sigmoidal nonlinearity**

Logistic Function: Definition

Logistic Function (General Form)

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-a v_j(n))}, \quad a > 0$$

- $v_j(n)$ is the induced local field of neuron j
- a is an adjustable positive parameter
- Output amplitude lies inside the range $0 \leq y_j \leq 1$

Logistic Function: Derivative

Derivative of Logistic Function

$$\varphi_j'(v_j(n)) = \frac{a \exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2}$$

Simplified Form

With $y_j(n) = \varphi_j(v_j(n))$:

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)]$$

Local Gradient: Output Layer

For Output Neuron j

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$
$$= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)]$$

Where:

- $o_j(n)$ is the function signal at the output of neuron j
- $d_j(n)$ is the desired response

Local Gradient: Hidden Layer

For Hidden Neuron j

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$

$$= a y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n)$$

- The summation is over all neurons k in the next layer
- $w_{kj}(n)$ represents the weight from neuron j to neuron k

Hyperbolic Tangent Function: Definition

Hyperbolic Tangent Function (General Form)

$$\varphi_j(v_j(n)) = a \tanh(b v_j(n))$$

- a and b are positive constants
- The hyperbolic tangent function is just the logistic function rescaled and biased

Hyperbolic Tangent: Derivative

Derivative Forms

$$\begin{aligned}\varphi_j'(v_j(n)) &= ab\,\mathrm{sech}^2(b v_j(n)) \\ &= ab(1 - \tanh^2(b v_j(n))) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)]\end{aligned}$$

Local Gradient: Hyperbolic Tangent (Output)

For Output Neuron j

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n))$$

$$= \frac{b}{a}[d_j(n) - o_j(n)][a - o_j(n)][a + o_j(n)]$$

Local Gradient: Hyperbolic Tangent (Hidden)

For Hidden Neuron j

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)$$

$$= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n)$$

Important Properties of Sigmoid Functions

- The derivative $\varphi_j'(v_j(n))$ attains its maximum value at $y_j(n) = 0.5$ for logistic function
- Minimum value (zero) occurs at $y_j(n) = 0$ or $y_j(n) = 1.0$
- Weight changes are proportional to the derivative $\varphi_j'(v_j(n))$
- Synaptic weights change most for neurons where function signals are in their midrange
- This feature contributes to the stability of backpropagation learning algorithm

Computational Advantage

Key Benefit

Using equations for logistic function and hyperbolic tangent function, we may calculate the local gradient δ_j without requiring explicit knowledge of the activation function.

- Simplifies implementation
- Reduces computational complexity
- Enables efficient backpropagation

B. Learning Rate Trade-offs

- **Small learning rate η:**
  - Smaller changes to synaptic weights
  - Smoother trajectory in weight space
  - Slower rate of learning
- **Large learning rate η:**
  - Faster learning convergence
  - Risk of network instability (oscillatory behavior)
  - Large weight changes may cause instability

Solution: Incorporate momentum to balance speed and stability

The Generalized Delta Rule with Momentum

The momentum-enhanced weight update rule:

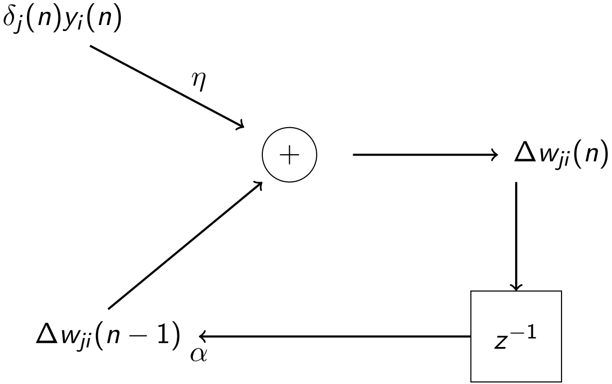
$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{2}$$

where:

- α is the **momentum constant** (usually positive)
- η is the learning rate
- $\delta_j(n)$ is the local gradient
- $y_i(n)$ is the input signal

Note: This includes the standard delta rule as a special case when $\alpha = 0$

Momentum Feedback Loop



The momentum constant α controls the feedback loop around $\Delta w_{ji}(n)$

Momentum as Exponentially Weighted Sum

Solving the difference equation yields:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) \tag{3}$$

Equivalently:

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)} \tag{4}$$

Key insight: Current weight adjustment is an exponentially weighted sum of all past gradient information

Convergence Requirements

For the exponentially weighted time series to be **convergent**:

Momentum Constraint

The momentum constant must satisfy: $0 \leq |\alpha| < 1$

- When $\alpha = 0$: Standard backpropagation (no momentum)
- $\alpha > 0$: Positive momentum (typical case)
- $\alpha < 0$: Negative momentum (rarely used in practice)

How Momentum Affects Learning

Acceleration Effect

When $\frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)}$ has the **same sign** on consecutive iterations:

- Exponentially weighted sum grows in magnitude
- Weight is adjusted by a large amount
- Accelerates descent in steady downhill directions

Stabilization Effect

When $\frac{\partial \mathcal{E}(t)}{\partial w_{ji}(t)}$ has **opposite signs** on consecutive iterations:

- Exponentially weighted sum shrinks in magnitude
- Weight is adjusted by a small amount
- Stabilizes oscillating directions

Advantages of Momentum in Learning

1. **Speed vs. Stability Balance**
   - Enables use of larger learning rates
   - Maintains stability through momentum damping
2. **Escape Local Minima**
   - Prevents termination in shallow local minima
   - Momentum carries optimization past small barriers
3. **Adaptive Behavior**
   - Accelerates in consistent gradient directions
   - Dampens oscillations in inconsistent directions

Practical Aspects

Connection-Dependent Learning Rates

In practice, learning rate should be connection-dependent: η_{ji}

Selective Weight Updates

- Can choose to make all synaptic weights adjustable
- Or constrain some weights to remain fixed during adaptation
- Fixed weights: set $\eta_{ji} = 0$ for specific connections

Error Propagation

Error signals back-propagate through the network normally, but fixed weights remain unaltered

C. Stopping Criteria

Fundamental Challenge

The backpropagation algorithm **cannot be shown to converge** in general, and there are **no well-defined criteria** for stopping its operation.

- No guaranteed convergence to global minimum
- Need practical termination conditions
- Each criterion has its own merits and drawbacks
- Must balance training time vs. performance

Goal: Develop reasonable criteria to terminate weight adjustments based on properties of local/global minima

Minimum Conditions

For a weight vector $\mathbf{w}^*$ to be a minimum (local or global):

Necessary Condition

The gradient vector $\mathbf{g}(\mathbf{w})$ (first-order partial derivatives) of the error surface must be zero:

$$\mathbf{g}(\mathbf{w}) = \mathbf{0} \quad \text{at} \quad \mathbf{w} = \mathbf{w}^* \tag{5}$$

Stationarity Property

The cost function $\mathcal{E}_{av}(\mathbf{w})$ is stationary at $\mathbf{w} = \mathbf{w}^*$

These mathematical properties form the basis for practical stopping criteria.

Euclidean Norm of Gradient Vector

Gradient-Based Convergence Criterion

The backpropagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.

Mathematical formulation:

$$\|\mathbf{g}(\mathbf{w})\| \leq \epsilon_g \tag{6}$$

where ϵ_g is a small positive threshold.

Drawbacks

- Learning times may be long for successful trials
- Requires computation of the gradient vector $\mathbf{g}(\mathbf{w})$
- Additional computational overhead

Average Squared Error Monitoring

Error-Based Convergence Criterion

The backpropagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.

Mathematical formulation:

$$\left| \frac{\Delta \mathcal{E}_{av}}{\Delta \text{epoch}} \right| \leq \epsilon_e \tag{7}$$

Typical Threshold Values

- Range: 0.1 to 1 percent per epoch
- Sometimes as small as 0.01 percent per epoch

Risk

May result in **premature termination** of the learning process

The Most Practical Approach

Generalization-Based Criterion

After each learning iteration, test the network's **generalization performance**. Stop learning when:

- Generalization performance is adequate, OR
- Generalization performance has peaked

Advantages

- **Theoretically supported**
- Directly addresses the ultimate goal of learning
- Prevents overfitting
- Most practical for real applications

Implementation: Use a separate validation/test dataset to monitor performance during training

Summary of Approaches

Criterion	Advantages	Disadvantages
Gradient Magnitude	• Theoretically sound • Direct measure of optimality	• Computational overhead • Long training times
Error Change Rate	• Simple to implement • Low computational cost	• Risk of premature stopping • Arbitrary thresholds
Generalization	• Most practical • Prevents overfitting • Goal-oriented	• Requires validation data • More complex setup

Recommended Approach

Best Practice Strategy

Combine multiple criteria for robust stopping:

1. **Primary:** Monitor generalization performance
   - Use validation set after each epoch
   - Track validation error trend
2. **Secondary:** Set maximum training epochs
   - Prevents infinite training
   - Computational budget control
3. **Optional:** Monitor training error change rate
   - Additional safety check
   - Early detection of convergence issues

Stop when: Validation error increases consistently OR maximum epochs reached OR training error change becomes negligible

Implementation Details

Validation-Based Early Stopping

1. Split data: Training / Validation / Test
2. Train on training set
3. Evaluate on validation set after each epoch
4. Track best validation performance
5. Stop if validation error increases for k consecutive epochs

Key Parameters

- **Patience:** Number of epochs to wait ($k = 5 - 20$)
- **Validation frequency:** Every epoch vs. every n epochs
- **Improvement threshold:** Minimum improvement to reset patience

This approach provides the best balance between training effectiveness and generalization performance.

Summary of Back-Propagation

- Weight updates aim to minimize error via gradient descent.
- Key components: local field, activation function, local gradient, error signal.
- Different formulas for hidden vs. output neurons.
- Credit assignment for hidden layers via recursive error propagation.

5.4 The XOR Problem

- XOR = Exclusive-OR logic gate
- Output is 1 when inputs differ; otherwise, output is 0.
- Input-output pairs:

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

- These input pairs form the four corners of a unit square.

Why a Single-Layer Perceptron Fails ?

- A perceptron creates a linear decision boundary:

$$y = \text{sign}(w^T x + b)$$

- XOR classes are not linearly separable.
- No single straight line can separate classes 0 and 1.

Solving XOR with a Hidden Layer

- Use a multilayer perceptron with one hidden layer.
- Hidden layer has 2 neurons; output layer has 1 neuron.
- Each neuron is a McCulloch–Pitts model (threshold activation).
- Inputs: 0 and 1 are represented by logic levels 0 and +1.

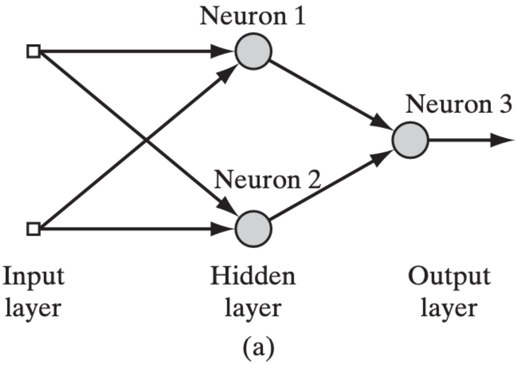


FIGURE 5.4 (a) Architectural graph of network for solving the XOR problem.

Neuron Weights and Biases

Hidden Neuron 1: $w_{11} = w_{12} = +1, \quad b_1 = -\frac{3}{2}$
Hidden Neuron 2: $w_{21} = w_{22} = +1, \quad b_2 = -\frac{1}{2}$
Output Neuron: $w_{31} = -2, \quad w_{32} = +1, \quad b_3 = -\frac{1}{2}$

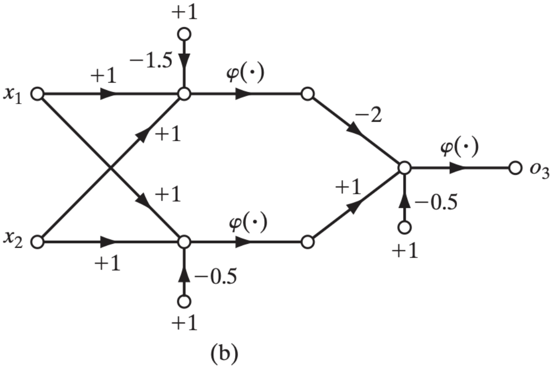


FIGURE 5.4 (b) Signal-flow graph of the network.

How the Network Solves XOR

- (0,0): Both hidden neurons off → output off (0)
- (1,1): Both hidden neurons on → output off (0)
- (0,1) or (1,0): Only bottom hidden neuron on → output on (1)
- Top hidden neuron is inhibitory ($w_{31} = -2$), bottom is excitatory ($w_{32} = +1$)

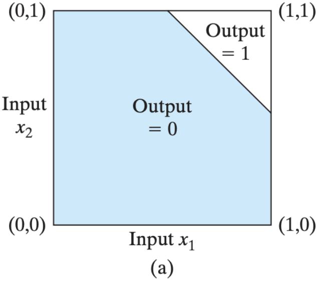


FIGURE 5.5 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 5.4.

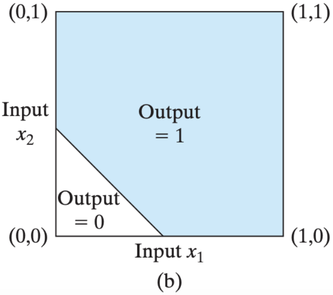


FIGURE 5.5(b) Decision boundary constructed by hidden neuron 2 of the network.

Frame Title

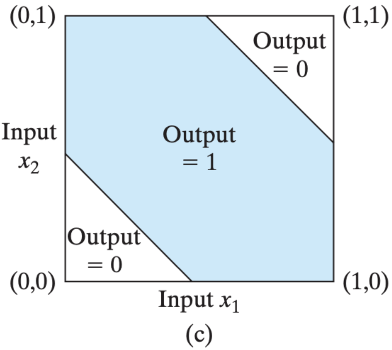


FIGURE 5.5(c) (c) Decision boundaries constructed by the complete network.

Conclusion: XOR is solved using a non-linear mapping via hidden neurons.

Thank You!