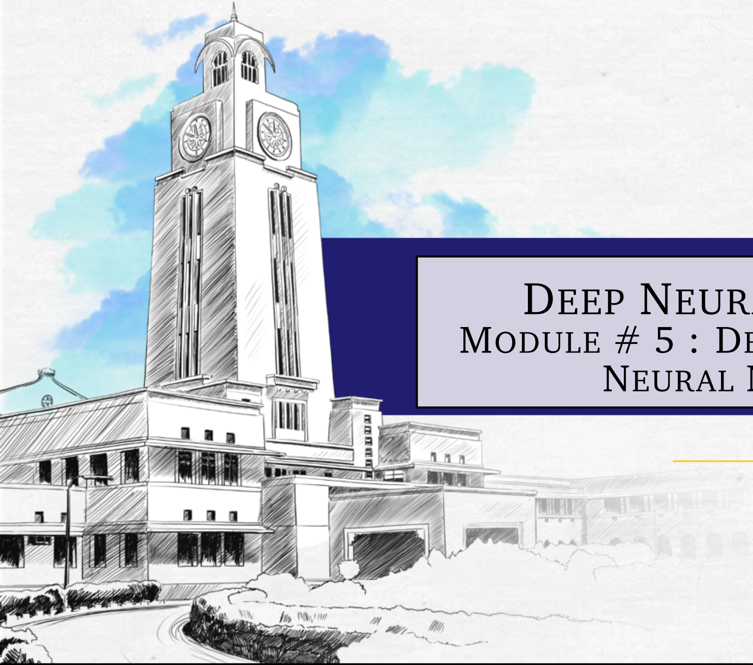


Deep Neural Network
Module # 5 : Deep Feed-Forward Neural Networks

Seetha Parameswaran
BITS Pilani WILP

The instructor is gratefully acknowledging
the authors who made their course
materials freely available online.
This deck is prepared by Seetha Parameswaran.

Table of Contents

Linear Models Have Limitations

Recall: Linear models uses a single neuron with activation function $f(\cdot)$

$$\hat{y} = f(\mathbf{w}^T \mathbf{x} + b) \tag{1}$$

Problem: Many real-world relationships are non-linear

- Image recognition (pixel intensities $\rightarrow$ object classes)
- Natural language understanding (words $\rightarrow$ sentiment)
- Medical diagnosis (symptoms $\rightarrow$ disease probability)
- Game playing (board state $\rightarrow$ optimal move)

Linear models cannot solve non-linearly separable problems!

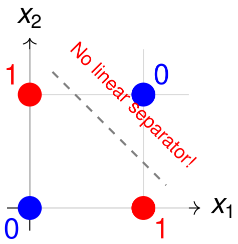
The XOR Problem

XOR (Exclusive OR): A classic non-linearly separable problem

Truth Table:

x_1	x_2	y (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

Visual Representation:



Challenge: No single straight line can separate the two classes!

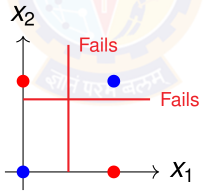
Why Single Neuron Fails on XOR

Single neuron model:

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b) \tag{2}$$

where σ is any activation function.

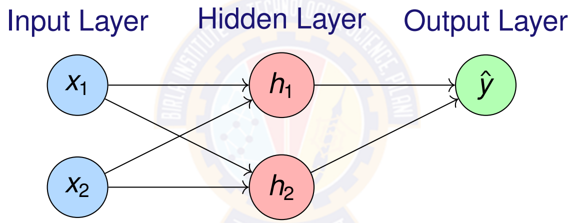
Decision boundary: $w_1 x_1 + w_2 x_2 + b = 0$ (a line/hyperplane)



Conclusion: We need models that can learn non-linear decision boundaries!

Solution: Multi-Layer Perceptron (MLP)

Key Idea: Stack multiple layers of neurons with non-linear activations



Non-linear activation functions
enable non-linear boundaries

MLP Solves XOR

Architecture: 2 inputs → 2 hidden units → 1 output

How it works:

- Hidden layer creates new feature representations
- Each hidden unit learns a linear separator
- Output layer combines these separators
- Non-linear activation enables complex boundaries

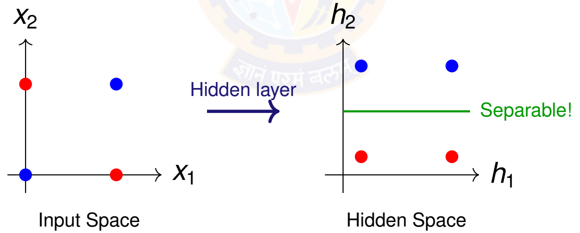


Table of Contents

What is Non-Linearity?

Definition

A function f is non-linear if it does not satisfy the linearity property:

$$f(ax_1 + bx_2) \neq a\,f(x_1) + b\,f(x_2) \tag{3}$$

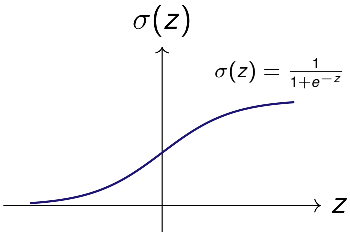
for all scalars a, b and inputs x_1, x_2.

Why non-linearity matters:

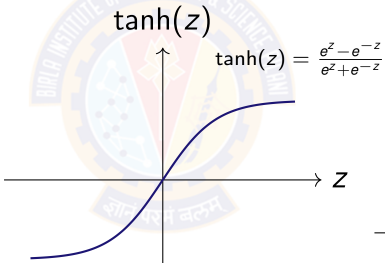
- Without non-linearity, deep networks collapse to single-layer models
- Non-linear activations enable learning complex patterns

Common Activation Functions

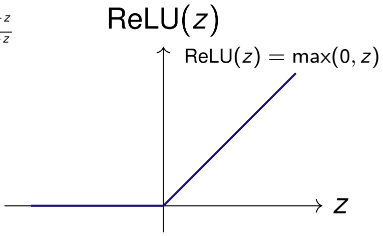
Activation functions introduce non-linearity



Sigmoid



Tanh



ReLU

Activation Function Properties

Property	Sigmoid	Tanh	ReLU	Leaky ReLU
Range	$(0, 1)$	$(-1, 1)$	$[0, \infty)$	$(-\infty, \infty)$
Zero-centered	✗	✓	✗	✓
Differentiable	✓	✓	✗ at 0	✗ at 0
Vanishing gradient	High	Moderate	No	No
Computational cost	High	High	Low	Low
Typical use	Output layer	Hidden layer	Hidden layer	Hidden layer

Modern preference: ReLU and variants for hidden layers

- Fast computation (simple thresholding)
- No vanishing gradient for positive values
- Sparse activation (biological plausibility)

Why ReLU Works Well?

ReLU: $f(z) = \max(0, z)$

Advantages:

1. Computational efficiency
   - Simple comparison and multiplication
   - Much faster than sigmoid/tanh (no exponentials)
2. Avoids vanishing gradients
   - Gradient is 1 for $z > 0$, flows unchanged through network
   - Enables training of very deep networks
3. Sparse activation
   - Many neurons output exactly 0
   - More efficient representations
   - Closer to biological neurons

Caveat: "Dying ReLU" problem when neurons always output 0

- Solution: Use Leaky ReLU, PReLU, or ELU variants

Table of Contents

Deep Feed-Forward Neural Network (DFN

Definition

A DFNN is a neural network with one or more hidden layers between input and output layers, where information flows forward through non-linear transformations.

Complete system requires four components:

1. Data: Input features and target outputs
2. Model: Multi-layer architecture with non-linear activations
3. Objective Function: Loss function to minimize
4. Learning Algorithm: Backpropagation with gradient descent

This is the same framework as linear models, but with added depth and non-linearity!

Data Representation

$$\text{Dataset:} \quad \mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N} \tag{4}$$

$$\text{Input Design Matrix:} \quad \mathbf{X} \in \mathbb{R}^{N \times d} = \begin{bmatrix} \mathbf{x}^{(1)T} \\ \mathbf{x}^{(2)T} \\ \vdots \\ \mathbf{x}^{(N)T} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_d^{(N)} \end{bmatrix} \tag{5}$$

$$\text{Target Matrix:} \quad \mathbf{Y} \in \mathbb{R}^{N \times K} = \begin{bmatrix} \mathbf{y}^{(1)T} \\ \mathbf{y}^{(2)T} \\ \vdots \\ \mathbf{y}^{(N)T} \end{bmatrix} \tag{6}$$

where K is the output dimension (1 for regression, C for classification with C classes).

Mini-Batch Representation

For computational efficiency, we process data in mini-batches

$$\text{Mini-batch of size } B: \quad \mathcal{B} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i \in \text{batch}} \tag{7}$$

$$\text{Mini-batch input matrix:} \quad \mathbf{X}_{\mathcal{B}} \in \mathbb{R}^{B \times d} \tag{8}$$

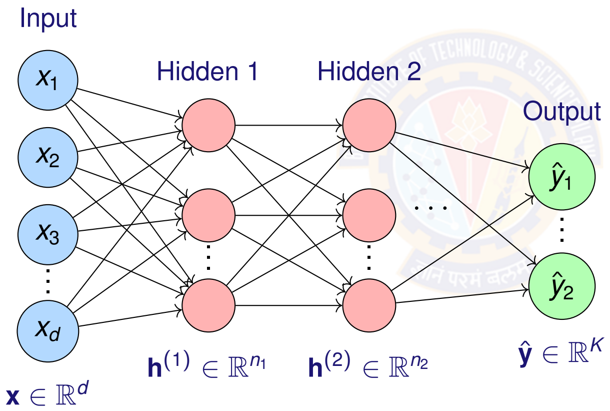
$$\text{Mini-batch target matrix:} \quad \mathbf{Y}_{\mathcal{B}} \in \mathbb{R}^{B \times K} \tag{9}$$

Common batch sizes: $B \in \{32, 64, 128, 256\}$

- Balances computational efficiency with gradient stability
- Enables parallelization on GPUs

DFNN Architecture

Network with L layers:



Input
Hidden 1
Hidden 2
Output
x_1
x_2
x_3
x_d
$\hat{y}_1$
$\hat{y}_2$
$\cdots$
$\mathbf{x} \in \mathbb{R}^d$
$\mathbf{h}^{(1)} \in \mathbb{R}^{n_1}$
$\mathbf{h}^{(2)} \in \mathbb{R}^{n_2}$
$\hat{\mathbf{y}} \in \mathbb{R}^K$

Notation:

- L: Number of layers (including output, excluding input)
- n_ℓ: Number of units in layer ℓ
- $d = n_0$: Input dimension
- $K = n_L$: Output dimension

Layer-wise Computation

For each layer $\ell = 1, 2, \ldots, L$:

$$\text{Parameters:} \quad \mathbf{W}^{(\ell)} \in \mathbb{R}^{n_{\ell-1} \times n_\ell} \quad \text{(weight matrix)} \tag{10}$$

$$\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell} \quad \text{(bias vector)} \tag{11}$$

$$\text{Initial condition:} \quad \mathbf{h}^{(0)} = \mathbf{x} \quad \text{(input)} \tag{12}$$

$$\text{Pre-activation (linear transformation):} \quad \mathbf{z}^{(\ell)} = \mathbf{h}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell} \tag{13}$$

$$\text{Activation (non-linear transformation):} \quad \mathbf{h}^{(\ell)} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}) \in \mathbb{R}^{n_\ell} \tag{14}$$

$$\text{Final output:} \quad \hat{\mathbf{y}} = \mathbf{h}^{(L)} \tag{15}$$

where $\sigma^{(\ell)}$ is the activation function for layer ℓ (applied element-wise).

Vectorized Computation for Mini-Batch

For mini-batch $\mathbf{X}_{\mathcal{B}} \in \mathbb{R}^{B\times d}$:

$$\text{Initial:} \quad \mathbf{H}^{(0)} = \mathbf{X}_{\mathcal{B}} \tag{16}$$

$$\text{Layer } \ell \text{ computation:} \quad \mathbf{Z}^{(\ell)} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_B\mathbf{b}^{(\ell)T} \in \mathbb{R}^{B\times n_\ell} \tag{17}$$

$$\mathbf{H}^{(\ell)} = \sigma^{(\ell)}(\mathbf{Z}^{(\ell)}) \in \mathbb{R}^{B\times n_\ell} \tag{18}$$

$$\text{Final:} \quad \hat{\mathbf{Y}}_{\mathcal{B}} = \mathbf{H}^{(L)} \in \mathbb{R}^{B\times K} \tag{19}$$

where:

- $\mathbf{H}^{(\ell)} \in \mathbb{R}^{B\times n_\ell}$ contains activations for all B examples
- $\mathbf{1}_B \in \mathbb{R}^{B\times 1}$ is a vector of ones (for broadcasting bias)
- Activation $\sigma^{(\ell)}$ applied element-wise to entire matrix

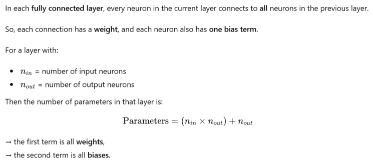
Total Parameters in Network

Parameter count for layer ℓ:

$$\text{Parameters}^{(\ell)} = n_{\ell-1} \times n_\ell + n_\ell = n_\ell(n_{\ell-1} + 1) \tag{20}$$

Total parameters in network:

$$\text{Total} = \sum_{\ell=1}^{L} n_\ell(n_{\ell-1} + 1)$$



In each **fully connected layer**, every neuron in the current layer connects to **all** neurons in the previous layer.

So, each connection has a **weight**, and each neuron also has **one bias term**.

For a layer with:

- n_{in} = number of input neurons
- n_{out} = number of output neurons

Then the number of parameters in that layer is:

$$\text{Parameters} = (n_{in} \times n_{out}) + n_{out}$$

→ the first term is all **weights**,
→ the second term is all **biases**.

Example: Network with 3-layer architecture $784 \to 256 \to 128 \to 10$ neurons

$$\begin{aligned}
\text{Layer 1:} &\quad 784 \times 256 + 256 = 200{,}960 \\
\text{Layer 2:} &\quad 256 \times 128 + 128 = 32{,}896 \\
\text{Layer 3:} &\quad 128 \times 10 + 10 = 1{,}290 \\
\text{Total:} &\quad 235{,}146 \text{ parameters}
\end{aligned}$$

Deep networks can have millions to billions of parameters!

Loss Functions for Different Tasks

The loss function depends on the task:

1. Regression (continuous output):

$$\text{MSE:} \quad \ell(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \frac{1}{2}\sum_{k=1}^{K}(\hat{y}_k - y_k)^2 \tag{22}$$

2. Binary Classification ($K = 1$, output in $(0, 1)$):

$$\text{BCE:} \quad \ell(\hat{y}, y) = -[y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})] \tag{23}$$

3. Multi-class Classification ($K = C$ classes):

$$\text{CCE:} \quad \ell(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{C} y_k \log(\hat{y}_k) \tag{24}$$

where $\mathbf{y}$ is one-hot encoded and $\hat{\mathbf{y}}$ from softmax activation.

Total Loss Over Mini-Batch

Loss for single example:

$$\ell^{(i)} = \ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \tag{25}$$

Average loss over mini-batch $\mathcal{B}$ of size B:

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{B} \sum_{i \in \mathcal{B}} \ell^{(i)} = \frac{1}{B} \sum_{i \in \mathcal{B}} \ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \tag{26}$$

where $\mathbf{W} = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L)}\}$ and $\mathbf{b} = \{\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \ldots, \mathbf{b}^{(L)}\}$ represent all network parameters.

Goal: Find parameters that minimize J:

$$\mathbf{W}^*, \mathbf{b}^* = \arg\min_{\mathbf{W}, \mathbf{b}} J(\mathbf{W}, \mathbf{b}) \tag{27}$$

Mini-Batch Stochastic Gradient Descent

Core idea: Update parameters using gradients from mini-batches

Update rule for layer ℓ:

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \frac{\partial J}{\partial \mathbf{W}^{(\ell)}} \tag{28}$$

$$\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \eta \frac{\partial J}{\partial \mathbf{b}^{(\ell)}} \tag{29}$$

where $\eta > 0$ is the learning rate.

Challenge: How to compute $\frac{\partial J}{\partial \mathbf{W}^{(\ell)}}$ and $\frac{\partial J}{\partial \mathbf{b}^{(\ell)}}$ efficiently?

Solution: **Backpropagation** algorithm!

- Efficiently computes gradients using chain rule
- Key innovation that enabled deep learning

Training Algorithm: Mini-Batch SGD

Algorithm 1: DFNN Training with Mini-Batch SGD (Part 1: Forward Pass)

Input: Dataset $\mathcal{D}$, architecture $(n_0, n_1, \ldots, n_L)$, learning rate η, batch size B, epochs T

Output: Trained parameters $\{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$

```
1  for $\ell = 1$ to $L$ do
2      Initialize $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_{\ell-1} \times n_\ell}$ randomly;
3      Initialize $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$ to zeros;
4  for epoch $t = 1$ to $T$ do
5      Shuffle dataset $\mathcal{D}$;
6      for each mini-batch $\mathcal{B}$ of size $B$ do
           // Forward Pass
7          $\mathbf{H}^{(0)} \leftarrow \mathbf{X}_{\mathcal{B}}$;
8          for $\ell = 1$ to $L$ do
9              $\mathbf{Z}^{(\ell)} \leftarrow \mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_B \mathbf{b}^{(\ell)T}$;
10             $\mathbf{H}^{(\ell)} \leftarrow \sigma^{(\ell)}(\mathbf{Z}^{(\ell)})$;
11         $\hat{\mathbf{Y}}_{\mathcal{B}} \leftarrow \mathbf{H}^{(L)}$;
12 $J \leftarrow \frac{1}{B} \sum_{i \in \mathcal{B}} \ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)})$ // Compute Loss
```

Training Algorithm: Mini-Batch SGD

Algorithm 2: DFNN Training with Mini-Batch SGD (Part 2: BackProp & Update)

```
   // Backward Pass (Backpropagation)
13 $\delta^{(L)} \leftarrow \frac{\partial J}{\partial \mathbf{Z}^{(L)}}$ // Output layer gradient Depends on loss & activation
14 for $\ell = L-1$ to $1$ do
15     $\delta^{(\ell)} \leftarrow (\delta^{(\ell+1)}\mathbf{W}^{(\ell+1)T}) \odot \sigma'^{(\ell)}(\mathbf{Z}^{(\ell)})$ // Backpropagate through hidden layers
16 for $\ell = 1$ to $L$ do
17     $\frac{\partial J}{\partial \mathbf{W}^{(\ell)}} \leftarrow \frac{1}{B}\mathbf{H}^{(\ell-1)T}\delta^{(\ell)}$ // Compute parameter gradients
18     $\frac{\partial J}{\partial \mathbf{b}^{(\ell)}} \leftarrow \frac{1}{B}\mathbf{1}_B^T\delta^{(\ell)}$;
   // Update parameters
19 for $\ell = 1$ to $L$ do
20     $\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta\frac{\partial J}{\partial \mathbf{W}^{(\ell)}}$;
21     $\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \eta\frac{\partial J}{\partial \mathbf{b}^{(\ell)}}$;
```

Table of Contents

Forward Pass: Computing Predictions

Given input $\mathbf{x} \in \mathbb{R}^d$, compute output $\hat{\mathbf{y}} \in \mathbb{R}^K$

$$\begin{tikzcd}
\boxed{\mathbf{h}^{(0)} = \mathbf{x}} \arrow[r, "{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}}"] & \boxed{\mathbf{h}^{(1)}} \arrow[r, "{\mathbf{W}^{(2)}, \mathbf{b}^{(2)}}"] & \boxed{\mathbf{h}^{(2)}} \arrow[r] & \cdots \arrow[r, "{\mathbf{W}^{(L)}, \mathbf{b}^{(L)}}"] & \boxed{\mathbf{h}^{(L)} = \hat{\mathbf{y}}}
\end{tikzcd}$$

$\mathbf{z}^{(1)} = \mathbf{h}^{(0)}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$ $\quad$ $\mathbf{h}^{(1)} = \sigma^{(1)}(\mathbf{z}^{(1)})$

Forward Pass Algorithm

Algorithm 3: Forward Propagation

Input: Input $\mathbf{x} \in \mathbb{R}^d$, Parameters $\{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$
Output: Prediction $\hat{\mathbf{y}} \in \mathbb{R}^K$, All activations $\{\mathbf{h}^{(\ell)}\}_{\ell=0}^{L}$

```
  // Initialize with input
1 $\mathbf{h}^{(0)} \leftarrow \mathbf{x}$;
  // Forward through all layers
2 for $\ell = 1$ to $L$ do
      // Linear transformation
3     $\mathbf{z}^{(\ell)} \leftarrow \mathbf{h}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)}$;
      // Non-linear activation
4     $\mathbf{h}^{(\ell)} \leftarrow \sigma^{(\ell)}(\mathbf{z}^{(\ell)})$;
      // Store activations for backpropagation
  // Output is final layer activation
5 $\hat{\mathbf{y}} \leftarrow \mathbf{h}^{(L)}$;
6 return $\hat{\mathbf{y}}$, $\{\mathbf{h}^{(\ell)}, \mathbf{z}^{(\ell)}\}_{\ell=1}^{L}$;
```

Forward Pass: Vectorized for Mini-Batch

For mini-batch $\mathbf{X}_{\mathcal{B}} \in \mathbb{R}^{B\times d}$:

$$\text{Initialize:} \quad \mathbf{H}^{(0)} = \mathbf{X}_{\mathcal{B}} \in \mathbb{R}^{B\times d} \tag{30}$$

For each layer $\ell = 1, \ldots, L$:

$$\mathbf{Z}^{(\ell)} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_B \mathbf{b}^{(\ell)T} \in \mathbb{R}^{B\times n_\ell} \tag{31}$$

$$\mathbf{H}^{(\ell)} = \sigma^{(\ell)}(\mathbf{Z}^{(\ell)}) \in \mathbb{R}^{B\times n_\ell} \tag{32}$$

Output predictions :

$$\hat{\mathbf{Y}}_{\mathcal{B}} = \mathbf{H}^{(L)} \in \mathbb{R}^{B\times K} \tag{33}$$

$$\text{Computational complexity} \quad : O(B\sum_{\ell=1}^{L} n_{\ell-1}n_\ell) \quad \text{per forward pass} \tag{34}$$

Loss Computation

After forward pass, compute loss for the mini-batch:

$$\text{Per-example loss:} \quad \ell^{(i)} = \ell(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}) \tag{35}$$

$$\text{Average loss over mini-batch:} \quad J = \frac{1}{B} \sum_{i=1}^{B} \ell^{(i)} \tag{36}$$

Common loss functions:

$$\text{Regression:} \quad MSE = \frac{1}{2} \|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|^2$$

$$\text{Binary Classification:} \quad BCE = -[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

$$\text{Multi-class:} \quad CCE = -\sum_{k=1}^{C} y_k^{(i)} \log(\hat{y}_k^{(i)})$$

This loss quantifies how wrong our predictions are.

Output Layer Gradient

First step of backpropagation: compute gradient w.r.t. output layer pre-activation

Output layer gradient:

$$\delta^{(L)} = \frac{\partial J}{\partial \mathbf{Z}^{(L)}} \in \mathbb{R}^{B \times K} \tag{37}$$

For common loss-activation pairs: (for all 3, the output gradient is same.)

1. MSE + Identity: (Regression)
2. BCE + Sigmoid: (Binary classification)
3. CCE + Softmax: (Multi-class classification)

$$\delta^{(L)} = \frac{1}{B}(\hat{\mathbf{Y}} - \mathbf{Y}) \tag{38}$$

Notice the elegant form when loss and activation are paired correctly!

Backpropagation: The Key Algorithm

Purpose

Backpropagation efficiently computes gradients of the loss w.r.t. all parameters by applying the chain rule backward through the network.

Key insight: Use chain rule recursively

Notation: Define error signal for layer ℓ:

$$\delta^{(\ell)} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell)}} \in \mathbb{R}^{B \times n_\ell} \tag{39}$$

Two main steps:

1. Backpropagate error signals: $\delta^{(L)} \to \delta^{(L-1)} \to \cdots \to \delta^{(1)}$
2. Compute parameter gradients using error signals

Backpropagating Error Signals

Given: $\delta^{(\ell+1)} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell+1)}}$ Compute: $\delta^{(\ell)} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell)}}$

Chain rule: $$\delta^{(\ell)} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell)}} = \frac{\partial J}{\partial \mathbf{H}^{(\ell)}} \frac{\partial \mathbf{H}^{(\ell)}}{\partial \mathbf{Z}^{(\ell)}} \tag{40}$$

Derivation: $$\frac{\partial J}{\partial \mathbf{H}^{(\ell)}} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell+1)}} \frac{\partial \mathbf{Z}^{(\ell+1)}}{\partial \mathbf{H}^{(\ell)}} = \delta^{(\ell+1)} \mathbf{W}^{(\ell+1)T} \tag{41}$$

$$\frac{\partial \mathbf{H}^{(\ell)}}{\partial \mathbf{Z}^{(\ell)}} = \sigma'^{(\ell)}(\mathbf{Z}^{(\ell)}) \quad \text{(element-wise derivative)} \tag{42}$$

Backpropagation equation:

$$\boxed{\delta^{(\ell)} = (\delta^{(\ell+1)} \mathbf{W}^{(\ell+1)T}) \odot \sigma'^{(\ell)}(\mathbf{Z}^{(\ell)})} \tag{43}$$

where $\odot$ denotes element-wise multiplication (Hadamard product).

Computing Parameter Gradients

Given error signal $\delta^{(\ell)}$, compute gradients w.r.t. parameters:

$$\text{Weight gradient:} \quad \frac{\partial J}{\partial \mathbf{W}^{(\ell)}} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell)}} \frac{\partial \mathbf{Z}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{B} \mathbf{H}^{(\ell-1)T} \delta^{(\ell)} \in \mathbb{R}^{n_{\ell-1} \times n_\ell} \tag{44}$$

$$\text{Bias gradient:} \quad \frac{\partial J}{\partial \mathbf{b}^{(\ell)}} = \frac{\partial J}{\partial \mathbf{Z}^{(\ell)}} \frac{\partial \mathbf{Z}^{(\ell)}}{\partial \mathbf{b}^{(\ell)}} = \frac{1}{B} \mathbf{1}_B^T \delta^{(\ell)} \in \mathbb{R}^{n_\ell} \tag{45}$$

where $\mathbf{1}_B \in \mathbb{R}^{B \times 1}$ is a vector of ones.

Key: Error signal $\delta^{(\ell)}$ is shared for both weight and bias gradients!

Backpropagation Algorithm

Algorithm 4: Backpropagation

Input: Loss J, Activations $\{\mathbf{H}^{(\ell)}, \mathbf{Z}^{(\ell)}\}_{\ell=1}^{L}$, Weights $\{\mathbf{W}^{(\ell)}\}_{\ell=1}^{L}$

Output: Gradients $\{\frac{\partial J}{\partial \mathbf{W}^{(\ell)}}, \frac{\partial J}{\partial \mathbf{b}^{(\ell)}}\}_{\ell=1}^{L}$

```
  // Compute output layer error signal
1 $\delta^{(L)} \leftarrow \frac{\partial J}{\partial \mathbf{Z}^{(L)}}$ // Depends on loss function
  // Backpropagate error signals
2 for $\ell = L - 1$ to 1 do
3     $\delta^{(\ell)} \leftarrow (\delta^{(\ell+1)} \mathbf{W}^{(\ell+1)T}) \odot \sigma'^{(\ell)}(\mathbf{Z}^{(\ell)})$;
  // Compute parameter gradients
4 for $\ell = 1$ to $L$ do
5     $\frac{\partial J}{\partial \mathbf{W}^{(\ell)}} \leftarrow \frac{1}{B} \mathbf{H}^{(\ell-1)T} \delta^{(\ell)}$;
6     $\frac{\partial J}{\partial \mathbf{b}^{(\ell)}} \leftarrow \frac{1}{B} \mathbf{1}_B^T \delta^{(\ell)}$;
7 return $\{\frac{\partial J}{\partial \mathbf{W}^{(\ell)}}, \frac{\partial J}{\partial \mathbf{b}^{(\ell)}}\}_{\ell=1}^{L}$;
```

Activation Function Derivatives

Common activation derivatives (for backpropagation):

1. Sigmoid:

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \sigma'(z) = \sigma(z)(1-\sigma(z)) \tag{46}$$

2. Tanh:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \tanh'(z) = 1 - \tanh^2(z) \tag{47}$$

3. ReLU:

$$\text{ReLU}(z) = \max(0, z) \qquad \text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{48}$$

Computational Efficiency of Backpropagat

Why backpropagation is efficient:

Naive approach: Compute each $\frac{\partial J}{\partial W_{ij}^{(\ell)}}$ independently

- Requires one forward pass per parameter
- Complexity: $O(P \times C_{\text{forward}})$ where P = total parameters
- For millions of parameters, this is intractable!

Backpropagation: Reuses intermediate computations

- One forward pass + one backward pass
- Complexity: $O(C_{\text{forward}} + C_{\text{backward}}) \approx 2 \times O(C_{\text{forward}})$
- Backward pass has same complexity as forward pass

Speedup: $\frac{P}{2}$ (e.g., $\sim 100{,}000\times$ for $P = 200{,}000$)

Key insight: Dynamic programming via chain rule!

Parameter Update Step

After computing gradients, update all parameters:

For each layer $\ell = 1, \dots, L$: (Vector form for single example)

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \mathbf{h}^{(\ell-1)T} \delta^{(\ell)} \tag{49}$$

$$\mathbf{b}^{(\ell)} \leftarrow \mathbf{b}^{(\ell)} - \eta \delta^{(\ell)} \tag{50}$$

where $\eta > 0$ is the learning rate.
This moves parameters in the direction that reduces the loss!

Table of Contents

Using Trained DFNN for Prediction

After training, prediction is simply forward propagation:

Input: New example $\mathbf{x}_{\text{new}} \in \mathbb{R}^d$

Process:

1. Initialize: $\mathbf{h}^{(0)} = \mathbf{x}_{\text{new}}$
2. For $\ell = 1$ to L:

$$\mathbf{z}^{(\ell)} = \mathbf{h}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{b}^{(\ell)} \quad (51)$$

$$\mathbf{h}^{(\ell)} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}) \quad (52)$$

3. Output: $\hat{\mathbf{y}}_{\text{new}} = \mathbf{h}^{(L)}$

Interpretation depends on task:

- Regression: $\hat{\mathbf{y}}_{\text{new}}$ is the predicted continuous value
- Binary classification: $\hat{y}_{\text{new}} > 0.5 \to$ class 1, else class 0
- Multi-class: $\arg\max_k \hat{y}_{k,\text{new}}$ is the predicted class

Batch Prediction

Efficiently predict for multiple examples:

Input: Test set $\mathbf{X}_{\text{test}} \in \mathbb{R}^{N_{\text{test}} \times d}$

Vectorized forward pass:

$$\mathbf{H}^{(0)} = \mathbf{X}_{\text{test}} \tag{53}$$

$$\mathbf{Z}^{(\ell)} = \mathbf{H}^{(\ell-1)}\mathbf{W}^{(\ell)} + \mathbf{1}_{N_{\text{test}}}\mathbf{b}^{(\ell)T} \tag{54}$$

$$\mathbf{H}^{(\ell)} = \sigma^{(\ell)}(\mathbf{Z}^{(\ell)}) \tag{55}$$

$$\hat{\mathbf{Y}}_{\text{test}} = \mathbf{H}^{(L)} \in \mathbb{R}^{N_{\text{test}} \times K} \tag{56}$$

Advantages:

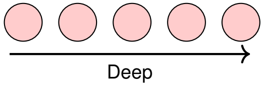
- Efficient matrix operations (GPU acceleration)
- All predictions in one forward pass
- Essential for large-scale deployment

Table of Contents

Network Depth vs Width

Two ways to increase model capacity:

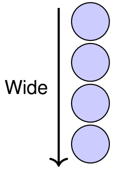
Depth (more layers):



Properties:

- Hierarchical features
- More expressive
- Better generalization

Width (more units per layer):

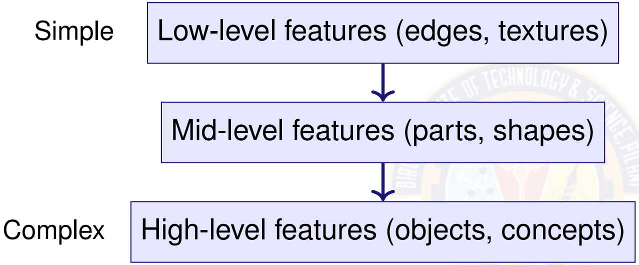


Properties:

- More parallel features
- Higher capacity
- Risk of overfitting

Why Depth Matters?

Deep networks learn hierarchical representations:


Simple
Low-level features (edges, textures)
Mid-level features (parts, shapes)
Complex
High-level features (objects, concepts)

Deep networks can represent exponentially more functions than shallow ones with the same number of parameters!

Example: Image classification

- Layer 1: Detect edges and colors
- Layer 2: Combine into textures and simple shapes
- Layer 3: Recognize object parts (eyes, wheels)
- Layer 4+: Identify complete objects (faces, cars)

Universal Approximation Theorem

Theorem (Cybenko, 1989; Hornik et al., 1989)

A feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given an appropriate activation function.

What this means:

- Theoretically, even shallow networks are powerful
- BUT: May require exponentially many hidden units
- Deep networks are much more parameter-efficient

Practice:

- Deep networks learn better representations
- Achieve better performance with fewer parameters
- Generalize better to new data

Depth enables efficient learning of hierarchical structure!

Challenges with Very Deep Networks

Problems that arise with increasing depth:

1. Vanishing Gradients:
   - Gradients become exponentially small in early layers
   - Network fails to learn useful features
   - Solution: ReLU activations, careful initialization, normalization
2. Exploding Gradients:
   - Gradients become exponentially large
   - Causes numerical instability (NaN/Inf values)
   - Solution: Gradient clipping, proper initialization
3. Degradation Problem:
   - Very deep networks harder to optimize
   - Training error may increase with depth
   - Solution: Residual connections (ResNets), skip connections

Table of Contents

Design Considerations

Key decisions when designing a DFNN:

1. Number of layers L
   - How deep should the network be?
2. Units per layer $n_1, n_2, \ldots, n_L$
   - How wide should each layer be?
3. Activation functions $\sigma^{(\ell)}$
   - Which non-linearity for each layer?
4. Output layer design
   - Depends on task (regression vs classification)
5. Regularization
   - How to prevent overfitting?

No universal answer - depends on problem, data, and computational resources!

Output Layer Design

Output layer depends on the task:

1. Regression (continuous values):
   - Units: K = dimension of output
   - Activation: Identity (linear)
   - Loss: Mean Squared Error (MSE)
2. Binary Classification (2 classes):
   - Units: $K = 1$
   - Activation: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$
   - Loss: Binary Cross-Entropy (BCE)
3. Multi-class Classification (C classes):
   - Units: $K = C$
   - Activation: Softmax $\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{C} e^{z_j}}$
   - Loss: Categorical Cross-Entropy (CCE)

General Architecture Guidelines

1. Start Simple:
    - Begin with 2-3 hidden layers
    - 10-100 units per layer (depending on problem scale)
    - Gradually increase complexity if needed
2. Common Patterns:
    - Decreasing width: $n_1 > n_2 > \cdots > n_L$
        - Funnel architecture (compression)
    - Constant width: $n_1 = n_2 = \cdots = n_{L-1}$
        - Uniform representation
    - Hourglass: Wide $\rightarrow$ Narrow $\rightarrow$ Wide
        - Learns compressed representations
3. Scale with Data:
    - More data $\rightarrow$ can support larger networks
    - Less data $\rightarrow$ keep networks smaller to avoid overfitting

Hidden Layer Activation Functions

Activation	When to Use	Avoid When
ReLU	Default choice Fast, no vanishing gradient	Very deep networks (dying ReLU)
Leaky ReLU	Deep networks Fixes dying ReLU	Never (safe choice)
Tanh	Small networks Zero-centered	Very deep networks (vanishing gradient)
Sigmoid	Avoid for hidden layers (only for binary output)	Hidden layers (strong saturation)

Modern recommendation:

- Default: ReLU for hidden layers
- Alternative: Leaky ReLU, ELU for very deep networks
- Avoid: Sigmoid/Tanh in hidden layers (unless specific reason)

Regularization Techniques

Prevent overfitting with:

1. L2 Regularization (Weight Decay):
2. Dropout:
3. Early Stopping:
4. Batch Normalization:

Note: You will learn about this in Module 10 and 11

Table of Contents

Parameter Initialization

Proper initialization is critical for training success!

✗ Bad: Zero initialization

✓ Good: Random initialization

- Xavier/Glorot Initialization for Sigmoid / Tanh
- He Initialization for ReLU
- Biases: Initialize to zero: $\mathbf{b}^{(\ell)} = \mathbf{0}$

Note: You will learn about this in Module 10 and 11

Training Hyperparameters to Tune

1. Learning Rate η:
   - Start with: $\eta \in \{0.001, 0.01, 0.1\}$
   - Consider adaptive methods (Adam with $\eta = 0.001$)
2. Batch Size B:
   - Common: $B \in \{32, 64, 128, 256\}$
   - Larger batch $\rightarrow$ faster but may hurt generalization
   - Smaller batch $\rightarrow$ better generalization but slower
   - Limited by GPU memory
3. Number of Epochs T:
   - Use early stopping based on validation loss
   - Typically 50-200 epochs for moderate datasets
   - Monitor for overfitting

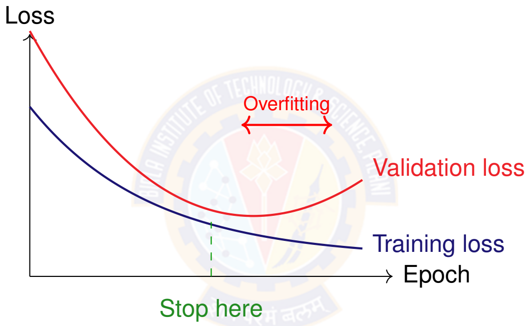
Feature Preprocessing

Preprocess features before training:

1. Normalization/Standardization:
   - Standardization (recommended)
   - Min-Max Scaling
   - Apply same transformation to validation/test sets
2. Handle Missing Values:
   - Impute with mean/median/mode
   - Or use indicator variables
3. Encode Categorical Variables:
   - One-hot encoding for nominal categories
   - Ordinal encoding for ordered categories

Monitoring Training Progress

Track metrics during training:



What to monitor:

- Training loss (should decrease steadily)
- Validation loss (should decrease then plateau/increase)
- Training/validation accuracy (for classification)
- Gradient norms (detect vanishing/exploding gradients)

Diagnosing Training Issues

Symptom	Diagnosis	Solution
Loss = NaN/Inf	Numerical instability	Decrease η, check gradients, normalize inputs
Loss not decreasing	Learning rate too small OR bug	Increase η, verify gradient computation
Loss oscillating	Learning rate too large	Decrease η, use learning rate schedule
Training loss $\downarrow$, validation loss $\uparrow$	Overfitting	Add regularization, dropout, more data
Both losses high	Underfitting	Increase capacity (wider/deeper), train longer
Slow convergence	Poor initialization OR bad features	Use proper init (He/Xavier), normalize features
Gradients $\to 0$	Vanishing gradients	Use ReLU, better init, batch normalization
Gradients $\to \infty$	Exploding gradients	Gradient clipping, decrease η

Debugging Checklist

Before training:

- Features normalized/standardized
- Data split into train/val/test
- Output layer matches task (sigmoid for binary, softmax for multi-class)
- Loss function matches output activation
- Weights initialized properly (not all zeros!)

During training:

- Loss decreasing on training set
- Validation loss tracked separately
- Gradients neither vanishing nor exploding
- No NaN/Inf values in loss or activations
- Training accuracy improving

Sanity checks:

- Try overfitting a small batch (should reach 0 loss)
- Verify shapes of all matrices/tensors
- Test forward pass before adding backprop

Implementation Best Practices

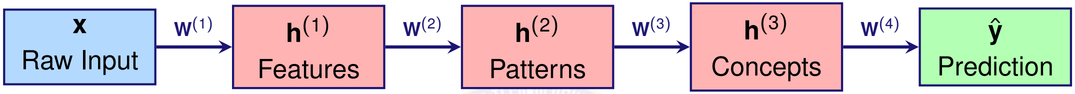
1. Vectorization:
   - Always use matrix operations, never loops over examples
   - Enables GPU acceleration
2. Modular Code:
   - Separate forward pass, loss, backward pass functions
   - Test each component independently
3. Numerical Stability:
   - Use stable implementations (e.g., log-sum-exp for softmax)
   - Clip gradients if needed: $\|\nabla\| >$ threshold
   - Monitor for NaN/Inf values
4. Reproducibility:
   - Set random seeds
   - Save hyperparameters
   - Track experiments systematically

Table of Contents

Key Takeaways

- Non-linearity is essential for learning complex patterns; without it, deep networks collapse to linear models.
- DFNN = stacked layers with non-linear activations, enabling hierarchical feature learning.
- Four components: Data, Model (architecture), Objective (loss), Learning (backpropagation + SGD).
- Backpropagation efficiently computes gradients using chain rule; enables training deep networks.
- Depth enables learning hierarchical features and improves expressiveness.
- Width increases capacity within each layer but requires more data.
- Proper initialization, normalization, and regularization are critical for successful training.
- ReLU is the default activation for hidden layers; use task-specific activations for output.

The Power of Deep Learning



Learns hierarchical representations automatically from data through gradient-based optimization

Deep learning has revolutionized:

- Computer vision (object detection, segmentation)
- Natural language processing (translation, generation)
- Speech recognition and synthesis
- Game playing (Go, Chess, StarCraft)
- Scientific discovery (protein folding, drug discovery)

What Next?

Specialized Architectures:

- Convolutional Neural Networks (CNNs): For images and spatial data
- Recurrent Neural Networks (RNNs): For sequences and time series
- Transformers: For attention-based models (GPT, BERT)
- Graph Neural Networks: For graph-structured data

Advanced Topics:

- Optimization algorithms (Adam, RMSprop, AdaGrad)
- Advanced regularization (Dropout, Batch Normalization)
- Transfer learning and pre-training
- Model interpretability and explainability
- Efficient inference and model compression

DFNNs are the foundation - master them first!

To become Proficient

1. Implement from scratch:
   - Write forward pass, backpropagation manually
   - Verify with gradient checking
   - Train on simple datasets (XOR, MNIST)
2. Use frameworks effectively:
   - PyTorch, TensorFlow, JAX handle backprop automatically
   - But understanding the math makes debugging easier
   - Focus on architecture design and hyperparameter tuning
3. Experiment and iterate:
   - Start simple, gradually increase complexity
   - Track experiments systematically
   - Learn from failures (most time spent debugging!)
4. Study successful architectures:
   - Read papers, examine code implementations
   - Understand design choices and motivations

References

- Goodfellow et al. ”Deep Learning” - Chapter 6 `http://www.deeplearningbook.org/`
- Zhang et al. ”Dive into Deep Learning” - Chapters 4, 5 `https://d2l.ai/`
- Nielsen, M. ”Neural Networks and Deep Learning” `http://neuralnetworksanddeeplearning.com/`

Thank You!