Deep Learning

Logistic Regression vs. Perceptron

Geometric Objective: Both algorithms aim to find a hyperplane that separates two classes in the feature space.

Logistic Regression:

Prediction Function: $\hat{y}_i = \sigma(W^T x + b)$ where $\sigma(z) = \frac{1}{1 + e^{-z}}$

Learning: Adjusts weights (W) and bias (b) using maximum likelihood estimation.

Output: Probabilistic (values between 0 and 1).

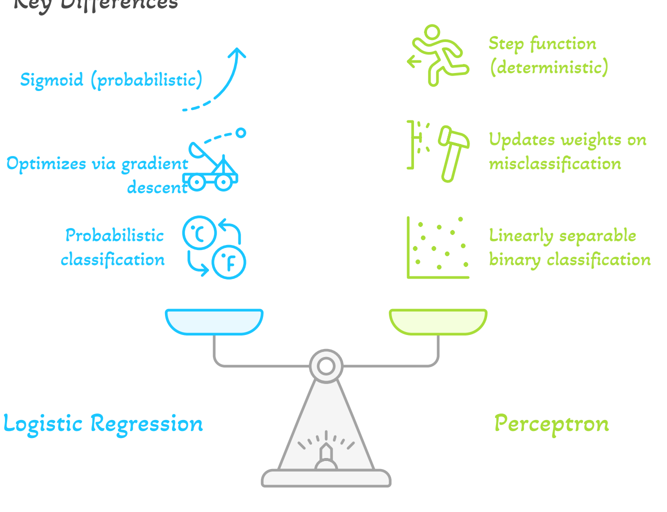
Perceptron:

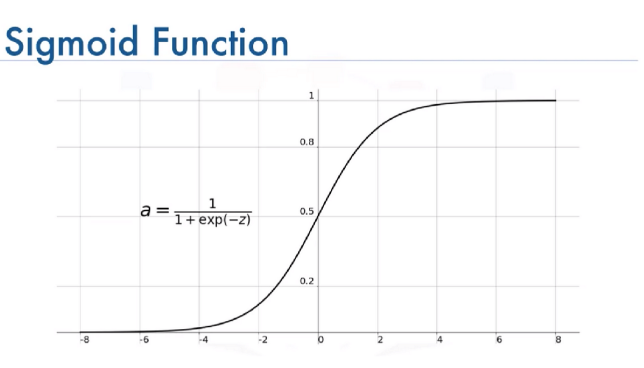
Prediction Function: $f(x_i) = \sum_{j=0}^{d} w_j x_{ij} + b$; Activation: Step function (e.g., 1 if $f(x)>0$, else 0)

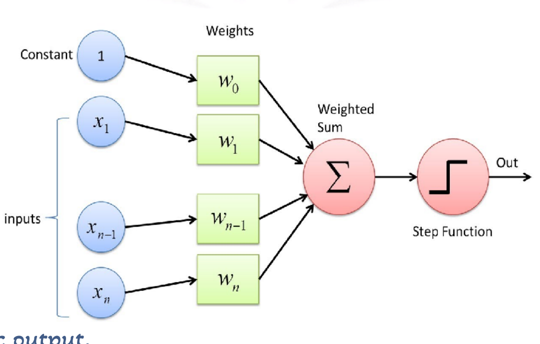
Learning: If misclassified, update weights: $w_j \leftarrow w_j + \eta\,(y_i - \hat{y}_i)\,x_{ij}$

Output: Binary (0 or 1). η: Learning rate; y_i: True label; $\hat{y}_i$: Predicted label.

Key Differences







Limitations:

Logistic Regression: Requires probabilistic interpretation; not ideal for non-linear data.

Perceptron: Only works for linearly separable data; no probabilistic output.

Multi-Layer Perceptron (MLP):

A Multi-Layer Perceptron (MLP) is a neural network with interconnected layers of neurons, including hidden layers between the input and output layers. Each neuron processes inputs through a function, and its output is passed to the next layer, mimicking the hierarchical processing of the human brain.

Function Composition:

Neurons perform operations like addition, subtraction, multiplication, or non-linear functions (e.g., ReLU, sigmoid).

Example: log(4x) can be represented as f1(f2(4,x))
where: f2(x) = a.x (linear function)
f1(x)= log(x) (non-linear function)

Hierarchical Processing:

Input → Hidden Layer 1 → Hidden Layer 2 → ... → Output.
Each layer transforms data to learn increasingly abstract features.

Mathematical Formulation:

For a neuron in layer l, the output a^l is computed as:

$a^l = \sigma(W^l a^{(l-1)} + b^l)$ Where:
σ: Activation function (e.g., sigmoid, ReLU)
W^l: Weight matrix for layer l
b^l: Bias vector for layer l

Applications:

Classification tasks (e.g., image recognition)
Regression analysis
Feature learning

Training a Single-Neuron Model:

1. Model Structure:

A single-neuron model consists of:

Inputs $(x_1,x_2,\dots,x_n)$: Features from the data.
Weights $(w_1,w_2,\dots,w_n)$: Learnable parameters.
Bias (b): Shifts the decision boundary.
Activation Function (σ): Converts the weighted sum into an output (e.g., sigmoid, ReLU).
Output ($\hat{y}_i$): Prediction based on inputs and weights.

2. Forward Propagation:

Compute the output $\hat{y}_i$: $z = \sum_{i=1}^{n} w_i x_i + b$ (Weighted sum)

$\hat{y} = \sigma(z)$ (Activation)

Example:
For binary classification, use the sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$

3. Loss Function:

Measure prediction error using a loss function:

Binary Cross-Entropy Loss (classification):

$$L = -(1/m) \sum_{i=1}^{m} [\, y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \,]$$

Mean Squared Error (regression):

$$L = (1/m) \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

4. Backpropagation:

Update weights and bias using gradient descent:

Compute gradients:

$$\partial L/\partial w_j = (1/m) \sum_{i=1}^{m} (\hat{y}_i - y_i)\, x_j^{(i)}$$

Update parameters:

$w_j \leftarrow w_j - \eta\, \partial L/\partial w_j$
$b \leftarrow b - \eta\, \partial L/\partial b$ Where η is the learning rate.

5. Training Steps:

Initialize weights (w) and bias (b) randomly:
Iterate for multiple epochs:
Compute $\hat{y}_i$ (forward pass).
Calculate loss L.
Compute gradients (backward pass).
Update w and b.
Stop when loss converges or after a fixed number of epochs.

6. Key Considerations:

Learning Rate (η): Controls step size during weight updates (too high → divergence; too low → slow training).

Activation Choice: Sigmoid for classification, ReLU/linear for regression.

Linearity: Single neurons can only model linear decision boundaries.

Training an MLP: Chain Rule for Backpropagation:

1. Overview:

Training a Multi-Layer Perceptron (MLP) involves adjusting weights and biases using the chain rule to compute gradients during backpropagation. This allows the network to minimize the loss function by propagating errors backward from the output layer to the input layer.:

2. Mathematical Framework:

Consider an MLP with:

Input: $x \in \mathbb{R}^d$
Hidden: $h = \sigma(W_1 x + b_1)$ where σ is the activation function (e.g., ReLU).
Output: $\hat{y} = \sigma(W_2 h + b_2)$
Loss: $L = \frac{1}{2}(y - \hat{y})^2$ (Mean Squared Error).

3. Chain Rule in Backpropagation:

To update weights (W_1,W_2) and biases (b_1,b_2), compute gradients using the chain rule:

Output Layer: $\partial L/\partial W_2 = (y - \hat{y}) \cdot \sigma'(W_2 h + b_2) \cdot h^T$

Hidden Layer: $\partial L/\partial W_1 = [W_2^T \cdot (y - \hat{y}) \cdot \sigma'(W_2 h + b_2)] \cdot \sigma'(W_1 x + b_1) \cdot x^T$

4. Step-by-Step Gradient Calculation:

For a network with ReLU activation ($\sigma(z)=\max(0,z)$):

Forward Pass: Compute hidden layer: $h = \mathrm{ReLU}(W_1 \cdot x + b_1)$
Compute output: $\hat{y} = \mathrm{ReLU}(W_2 \cdot h + b_2)$
Backward Pass:
Output error: $\delta_{out} = (y - \hat{y}) \odot \mathrm{ReLU}'(z_2)$
Hidden error: $\delta_{hidden} = (W_2^T \cdot \delta_{out}) \odot \mathrm{ReLU}'(z_1)$
Update weights: $W_2 \leftarrow W_2 + \eta \cdot \delta_{out} \cdot h^T$
$W_1 \leftarrow W_1 + \eta \cdot \delta_{hidden} \cdot x^T$

5. Key Equations:

Output Layer Weights: $\partial L/\partial W_2 = \delta_{out} \cdot h^T$
Hidden Layer Weights: $\partial L/\partial W_1 = \delta_{hidden} \cdot x^T$
Output Layer Bias: $\partial L/\partial b_2 = \delta_{out}$
Hidden Layer Bias: $\partial L/\partial b_1 = \delta_{hidden}$

6. Why the Chain Rule Matters:

- Efficiently computes gradients for deep networks by breaking derivatives into local components.
- Enables optimization algorithms like SGD to update weights iteratively.

Memoization in MLP Training

1. What is Memoization?

Memoization is an optimization technique that stores the results of computationally expensive operations during the forward pass (e.g., layer activations, intermediate derivatives) to reuse them in the backward pass. This avoids redundant recalculations, significantly speeding up training.

2. Memoization in MLPs

During training, an MLP performs:

Forward Pass:
Computes layer outputs (e.g., $z_1=W_1x+b_1$, $a_1= \sigma(z_1)$).
Stores these values in memory (memoization).

Backward Pass:
Uses cached values to compute gradients via the chain rule.
Example: Reuses z_1 and a_1 to calculate $\partial L/\partial W_1$.

3. Code Example: Memoization in PyTorch

```
import torch

# Forward pass (memoizes z1, a1, z2)

x = torch.randn(10, 5)              # Input
z1 = torch.matmul(x, W1) + b1       # Memoized
a1 = torch.relu(z1)                 # Memoized
z2 = torch.matmul(a1, W2) + b2      # Memoized

# Backward pass (uses cached values)

loss = ((z2 - y)**2).mean()
loss.backward()  # Gradients reuse z1, a1, z2
```

4. Trade-offs

Speed: Reduces computation time by ~50%.
Memory: Requires storing intermediate values (challenge for large models).

5. Why It Matters

Memoization is foundational for efficient deep learning. Frameworks like PyTorch and TensorFlow automate this process through computation graphs, enabling scalable training of complex models like GPT-3 and ResNet.

Back Propagation

Backpropagation is a combination of memoization & chain rule. The procedure is as follows:

Initialize w_{ij}^k
For each x_i in D
Forward Propagation: Pass x_i forward through the network
Compute Loss function $L(y_i, \hat{y})$
Compute all derivatives using chain rule & memoization
Update the weights again from end to the start such that loss will be minimized
Repeat step 2 till convergence

Note: Backpropagation works only if activation functions are differentiable. If these functions are easily or fast differentiable, it speeds up the training & Neural Network.

Activation Functions

Sigmoid (σ): The sigmoid function is a non-linear activation function that squashes input values between 0 and 1. It is particularly useful for binary classification tasks.

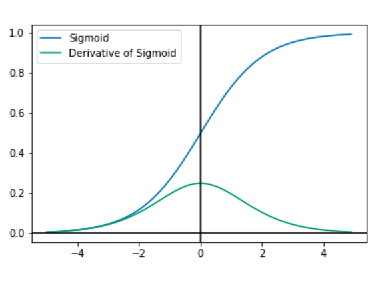
Mathematical Definition:
The sigmoid function is defined as:

$\sigma(z) = 1 / (1 + e^{-z})$
Where: $z = W \cdot x + b$ (weighted sum of inputs + bias)

Key Properties:
. Output range: (0, 1)
. Derivative range: (0,0.25)
. Smooth, S-shaped curve

Derivation of the Derivative:

The derivative of the sigmoid function is critical for backpropagation:
. Start with $\sigma(z) = 1 / (1 + e^{-z})$
. Let $g(z) = (1 + e^{-z})$, so $\sigma(z) = g(z)^{-1}$
. Apply the chain rule: $d\sigma/dz = (d\sigma/dg) \cdot (dg/dz)$
. Calculate individual terms: $d\sigma/dg = -g^{-2}$; $dg/dz = -e^{-z}$
. Combine results: $d\sigma/dz = e^{-z} / (1 + e^{-z})^2 = \sigma(z) \cdot (1 - \sigma(z))$



Limitations
Vanishing gradients for extreme inputs (saturates at 0/1)
Not zero-centered (can slow down learning)

HyperbolicTangent(tanh): The tanh function is a zero-centered activation function that squashes input values between -1 and 1. It is preferred over sigmoid for hidden layers in many architectures due to its symmetry.

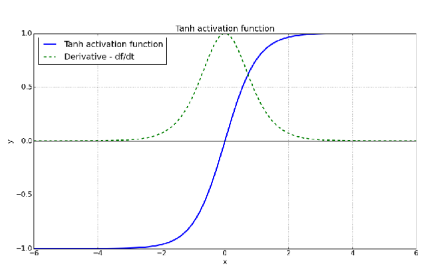
Mathematical Definition:
The tanh function is defined as:

$\tanh(z) = (e^{z} - e^{-z}) / (e^{z} + e^{-z})$

Key Properties:
. Output range: (−1, 1)
. Derivative range: (0, 1)
. Zero-centered (mitigates learning slowdown)

Derivation of the Derivative:

The derivative of the tanh function is critical for backpropagation:
. Start with $\tanh(z) = (e^{z} - e^{-z}) / (e^{z} + e^{-z})$
. Differentiate using quotient rule:
$d/dz \tanh(z) = 1 - \tanh^2(z)$



Limitations
Still suffers from vanishing gradients for extreme values
Computationally more expensive than ReLU

Rectified Linear Unit (ReLU): ReLU is the most widely used activation function for deep networks. It outputs the input directly if positive; otherwise, it outputs zero. Computationally efficient and mitigates vanishing gradients.

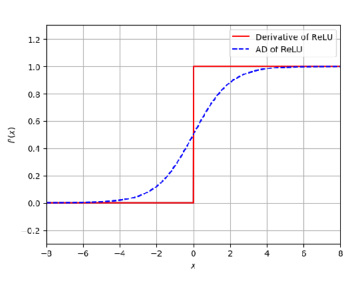
Mathematical Definition:
The tanh function is defined as:

$\mathrm{ReLU}(z)= \max(0, z)$

Key Properties:
. Output range: $[0, \infty)$
. Derivative range: 1 (if $z>0$), 0 (otherwise)
. Sparsity-inducing (reduces overfitting)

Derivation of the Derivative: $d/dz\,\mathrm{ReLU}(z) =$

1 if $z > 0$
0 if $z \le 0$



Limitations
"Dying ReLU" problem: Neurons can get stuck in negative region
Non-differentiable at $z = 0$

The Vanishing/Exploding Gradient Problem

Overview: In deep neural networks, gradients computed during backpropagation involve multiplying derivatives across layers. This can cause:

Exploding Gradients: Gradients grow exponentially if derivatives > 1.
Vanishing Gradients: Gradients shrink to near-zero if derivatives < 1.

Mathematical Explanation:

For a network with n layers, the gradient at the first layer is:

$\partial L/\partial W_1 = \partial L/\partial \hat{y} \cdot \prod_{k=2}^{n} (\partial h_k / \partial h_{k-1})$

$\partial h_k / \partial h_{k-1} > 1$ → Gradient explodes
e.g., $(1.5)^{10} \approx 58$

$\partial h_k / \partial h_{k-1} < 1$ → Gradient vanishes
e.g., $(0.5)^{10} \approx 0.001$

Causes:

Vanishing Gradients
Activation functions with small derivatives (e.g., sigmoid, tanh).
Deep networks amplify the multiplicative effect.

Exploding Gradients
Large initial weights or improper weight initialization.
Unbounded activations (e.g., ReLU without normalization).

Solutions:

For Vanishing Gradients
Use ReLU/Leaky ReLU activations.
Weight initialization (He/Xavier).
Skip connections (ResNet).

For Exploding Gradients
Gradient clipping (torch.nn.utils.clip_grad_norm_).
Batch normalization.
L2 regularization.

Conclusion: The vanishing/exploding gradient problem is critical in deep learning but can be managed with techniques like ReLU, careful initialization, gradient clipping, and architectural innovations (e.g., ResNet). These strategies enable stable training of very deep networks.

Batch Normalization

What is Batch Normalization?

Batch Normalization (BatchNorm) is a technique used in deep neural networks to normalize activations of each layer during training. It improves training stability, prevents vanishing/exploding gradients, and speeds up convergence.

Why is Batch Normalization Needed?

Internal Covariate Shift: Stabilizes shifting activation distributions.
Faster Convergence: Reduces training time by keeping activations consistent.
Prevents Vanishing/Exploding Gradients: Keeps values in a stable range.
Regularization Effect: Acts as a mild form of regularization, reducing reliance on dropout.

Advantages of Batch Normalization

☑ Reduces sensitivity to weight initialization
☑ Speeds up training
☑ Improves generalization
☑ Prevents internal covariate shift
☑ Acts as regularization

Batch Normalization Formula

Compute Mean & Variance for Mini-Batch: $\mu_B = (1/m) \cdot \sum x_i$
$\sigma_B^2 = (1/m) \cdot \sum (x_i - \mu_B)^2$

Normalize the Activations: $\hat{x}_i = (x_i - \mu_B) / \sqrt{\sigma_B^2 + \varepsilon}$

Scale and Shift with Learnable Parameters: $y_i = \gamma \cdot \hat{x}_i + \beta$

Where is Batch Normalization Applied?

Before activation functions (common in CNNs)
After activation functions (common in fully connected networks)
Between layers in deep networks

Gradient Clipping

Overview: Gradient Clipping is a technique used to prevent the problem of exploding gradients, which can occur in deep networks, particularly in recurrent neural networks (RNNs) and long training processes.

Why Is Gradient Clipping Needed?

During backpropagation, gradients can become excessively large, leading to unstable updates in weight parameters.
This can cause:
Loss function divergence (failure to converge).
Numerical instability (NaN values in computations).
Poor generalization due to chaotic weight updates.

Types of Gradient Clipping:

Norm-Based Clipping (Global Norm Clipping)
What: Clips the entire gradient vector if its L2 norm exceeds a threshold c.
Formula: If $\|g\| \ge c$, then: $g \leftarrow (c / \|g\|) \cdot g$
Used when: You want to control the overall gradient magnitude across the model.

Value-Based Clipping (Elementwise Clipping)
What: Clips each individual gradient component to lie within $[-v, +v]$.
Formula: $g_i \leftarrow \mathrm{clip}(g_i, -v, +v)$
Used when: You want to ensure no single gradient value explodes.

Per-Layer Gradient Clipping
What: Apply norm-based or value-based clipping individually per layer.
Why: Helps localize stability without affecting the entire model's gradient structure.

Adaptive Gradient Clipping (AGC)
What: Scales gradients relative to the norm of the parameters (not just gradients).
Formula (simplified): If $\|g\| > \varepsilon \cdot \|\theta\| \Rightarrow$ scale $g \leftarrow (\varepsilon \cdot \|\theta\| / \|g\|) \cdot g$
Used in: Large models (e.g., Transformers, Stable Diffusion) for better stability.

Optimizers:

Optimizers are algorithms that adjust the weights of a neural network to minimize the loss function. The choice of optimizer affects convergence speed, accuracy, and overall model performance.

Gradient Descent (GD)
Formula: $W = W - \eta \nabla L(W)$
Why It Was Introduced: It is the fundamental optimization algorithm used in machine learning, forming the basis for all other optimizers.
☑ Pros: Simple and easy to understand.
Guaranteed to find a minimum if convex loss function.
✕ Cons: Very slow for large datasets (Batch GD).
May get stuck in local minima.

Stochastic Gradient Descent (SGD)
Formula: $W = W - \eta \nabla L(W)$ (for each sample)
Why It Was Introduced: To speed up gradient descent by updating weights after each sample instead of the entire dataset.
☑ Pros: Faster than batch GD.
Can escape local minima due to noise.
✕ Cons: High variance in updates leads to instability.
May not converge to the global minimum.

Momentum Optimizer
Formula: $v_t = \beta v_{t-1} + \eta \nabla L(W)$
$W = W - v_t$
Why It Was Introduced: To reduce the oscillations in SGD and speed up convergence.
☑ Pros: Reduces oscillations in deep valleys.
Faster convergence in certain cases.
✕ Cons: Can overshoot the optimal point.

Nesterov Accelerated Gradient (NAG)
Formula: $v_t = \beta v_{t-1} + \eta \nabla L(W - \beta v_{t-1})$
$W = W - v_t$
Why It Was Introduced: To improve upon momentum by looking ahead before applying updates.
☑ Pros: More accurate updates than momentum.
Less overshooting.
✕ Cons: Slightly more computationally expensive.

Adagrad (Adaptive Gradient Algorithm)
Formula: $W = W - (\eta / \sqrt{G_t + \varepsilon}) \nabla L(W)$
Why It Was Introduced: To adapt learning rates for each parameter, making it useful for sparse data.
☑ Pros: Good for sparse features (e.g., NLP).
Adapts learning rate for each weight.
✕ Cons: Learning rate decreases too much over time.

RMSprop (Root Mean Square Propagation)
Formula: $G_t = \beta G_{t-1} + (1 - \beta) (\nabla L(W))^2$
$W = W - (\eta / \sqrt{G_t + \varepsilon}) \nabla L(W)$
Why It Was Introduced: To solve Adagrad's problem by using an exponentially weighted moving average of past squared gradients.
☑ Pros: Prevents drastic learning rate reduction.
Works well for non-stationary problems (e.g., RNNs).
✕ Cons: Requires careful tuning of hyperparameters.

Adam (Adaptive Moment Estimation)
Formula: $m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla L(W)$
$v_t = \beta_2 v_{t-1} + (1 - \beta_2) (\nabla L(W))^2$
$W = W - (\eta / (\sqrt{v_t} + \varepsilon))\, m_t$
Why It Was Introduced: To combine the benefits of Momentum and RMSprop.
☑ Pros: Works well for most deep learning models.
Combines benefits of Momentum and RMSprop.
✕ Cons: Can sometimes lead to non-converging solutions.

AdamW (Adam with Weight Decay)
Formula: $W = W - (\eta / (\sqrt{v_t} + \varepsilon))\, m_t - \lambda W$
Why It Was Introduced: To improve Adam's generalization by adding weight decay.
☑ Pros: Better generalization than Adam.
✕ Cons: Still requires tuning of hyperparameters.

Nadam (Nesterov-Accelerated Adam)
Formula: $W = W - (\eta / (\sqrt{v_t} + \varepsilon))\, (m_t + \beta_1 m_{t-1})$
Why It Was Introduced: To further improve Adam by incorporating Nesterov momentum.
☑ Pros: Faster convergence than Adam.
✕ Cons: More computationally expensive.

Key Techniques for Improving Neural Network Training:

Dropout

☑ *Purpose:*
Prevent overfitting by introducing stochasticity during training.
☑ *How It Works:*
Randomly "drops" (deactivates) neurons during each training iteration (dropout rate = probability of dropping a neuron).
Example: With a dropout rate of 0.5, 50% of neurons are temporarily ignored in each forward pass.
During inference, all neurons are active, but their outputs are scaled by the dropout rate to maintain expected values.
☑ *Benefits:*
Reduces co-adaptation of neurons (forces redundancy)
Acts as a form of stochastic regularization

Regularization

☑ *Purpose:*
Prevent overfitting by discouraging overly complex models.
☑ *Common Types:*

Comparison:	L1 (Lasso)	L2 (Ridge)
Penalty term:	L1 → $\lambda \sum \lvert w \rvert$	L2 → $\lambda \sum w^2$
Effect:	Sparse (zeros)	Small (non-zero)
Encourages:	Feature selection	Smooth weight decay
Use case:	High-dim data	Overfitting control

☑ *Key Difference:*
L1 creates sparse models, while L2 distributes error across weights.

Batch Normalization:

☑ *Purpose:*
Stabilize and accelerate training by reducing internal covariate shift.
☑ *How It Works:* Normalizes layer inputs to N(0,1) using batch statistics: $\hat{x} = (x - \mathrm{mean}_{batch}) / \sqrt{\mathrm{var}_{batch} + \varepsilon}$
Applies learnable scale (γ) and shift (β) parameters: $y = \gamma\hat{x} + \beta$
☑ *Benefits:*
Allows higher learning rates
Reduces sensitivity to weight initialization
Mild regularization effect due to noise in batch statistics

Key Differences & Interactions:

Technique	Primary Role	When to Use
Dropout	Regularization	Large networks
L1 / L2	Regularization	Feature selection
BatchNorm	Training Stabilization	Deep networks

Weight Initialization Techniques:

Zero Initialization
☑ *Formula:* $W = 0$
☑ *How It Works:*
▶ All weights are set to zero
▶ Causes all neurons to learn the same features → prevents learning

Random Initialization
☑ *Formula:* $W \sim U(-\varepsilon, \varepsilon)$
☑ *How It Works:*
▶ Weights drawn from a uniform distribution
▶ Doesn't account for layer size → may cause vanishing/exploding gradients

Xavier (Glorot) Initialization
☑ *Formula:* Uniform
$W \sim U(-\sqrt{6 / (n_{in} + n_{out})}, \sqrt{6 / (n_{in} + n_{out})})$
Normal
$W \sim N(0, 1 / (n_{in} + n_{out}))$
☑ *How It Works:*
▶ Balances variance → avoids vanishing/exploding gradients
▶ Best for sigmoid and tanh activations

He (Kaiming) Initialization
☑ *Formula:* Uniform
$W \sim U(-\sqrt{6 / n_{in}}, \sqrt{6 / n_{in}})$
Normal
$W \sim N(0, 2/n_{in})$
☑ *How It Works:*
▶ Best for ReLU and Leaky ReLU
▶ Prevents dead neurons and helps deep learning

Choosing the Right Initialization