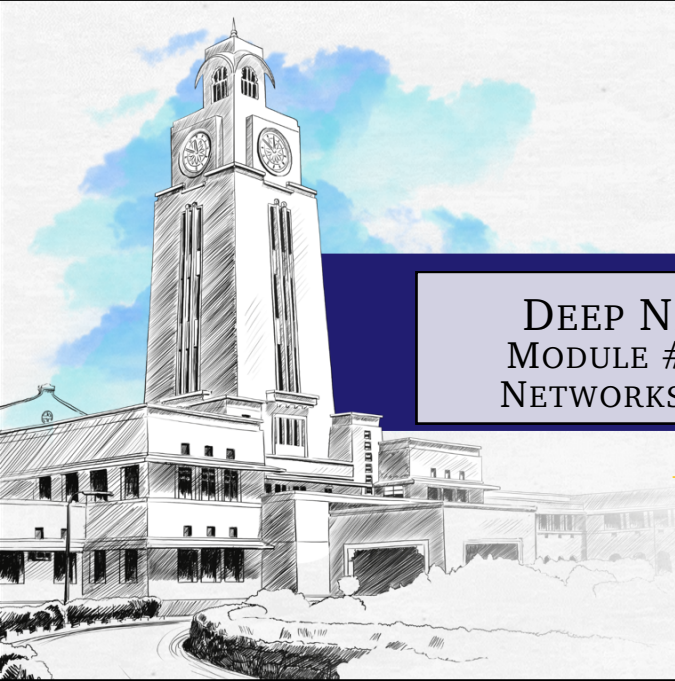


BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Deep Neural Network

Module # 4 : Linear Neural Networks for Classification

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

What is Classification?

Definition

Classification is a supervised learning task that predicts a discrete category or class label based on input features.

Key Characteristics:

- Input: Feature vector $\mathbf{x} \in \mathbb{R}^d$
- Output: Class label $y \in \{1, 2, \ldots, K\}$ (discrete)
- Goal: Learn a function $f : \mathbb{R}^d \to \{1, 2, \ldots, K\}$ that assigns inputs to classes

Difference from Regression:

- Regression: Predicts continuous values (e.g., price, temperature)
- Classification: Predicts discrete categories (e.g., spam/not spam, cat/dog)

Examples of Classification Problems

1. Email Spam Detection
   - Input: Email text, sender information, attachments
   - Output: Spam or Not Spam
2. Medical Diagnosis
   - Input: Patient symptoms, test results, medical history
   - Output: Disease present or absent
3. Image Recognition
   - Input: Pixel values of an image
   - Output: Object category (cat, dog, car, etc.)
4. Sentiment Analysis
   - Input: Customer review text
   - Output: Positive, Negative, or Neutral

Types of Classification

1. Binary Classification
   - Two classes: $y \in \{0, 1\}$ or $\{-1, +1\}$
   - Examples: Spam detection, disease diagnosis, fraud detection
2. Multi-class Classification
   - Multiple classes: $y \in \{1, 2, \ldots, K\}$ where $K > 2$
   - Examples: Digit recognition (0-9), animal classification
   - Each example belongs to exactly one class
3. Multi-label Classification
   - Each example can belong to multiple classes simultaneously
   - Examples: Movie genres (action + comedy), document tagging

Table of Contents

Binary Classification Problem

Definition

Binary classification assigns each input to one of two classes.

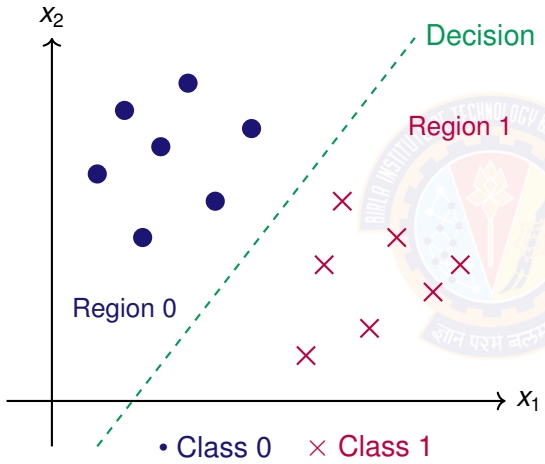
Standard Notation:

- Classes: $y \in \{0, 1\}$
  - $y = 1$: Positive class (e.g., spam, disease present)
  - $y = 0$: Negative class (e.g., not spam, disease absent)
- Dataset: $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$
- Goal: Learn $f(\mathbf{x})$ that outputs probability of class 1

Decision Boundary:

- Predict $\hat{y} = 1$ if $f(\mathbf{x}) \geq 0.5$
- Predict $\hat{y} = 0$ if $f(\mathbf{x}) < 0.5$

Binary Classification: Visual Example

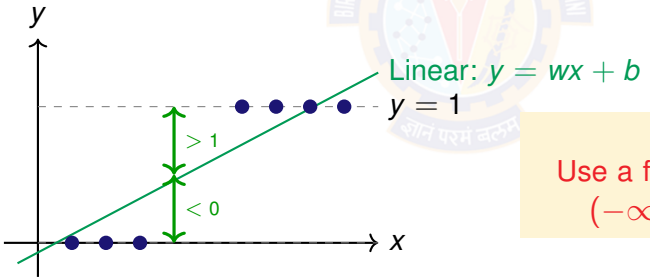


The decision boundary separates the two classes in feature space.

Why Not Use Linear Regression?

Problem with Linear Regression for Classification:

- Linear regression outputs: $\hat{y} \in (-\infty, \infty)$
- We need probabilities: $\hat{y} \in [0, 1]$
- Linear model can predict values < 0 or > 1



Solution:
Use a function that maps
$(-\infty, \infty) \rightarrow [0, 1]$

Table of Contents

What is Logistic Regression?

Definition

Logistic regression is a linear model for binary classification that uses the sigmoid function to output probabilities.

Key Idea:

- Compute linear combination: $z = \mathbf{w}^T\mathbf{x} + w_0$
- Apply sigmoid function: $\hat{y} = \sigma(z)$
- Interpret output as probability: $\hat{y} = P(y = 1|\mathbf{x}) \in [0, 1]$

Mathematical Form:

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T\mathbf{x}}} \tag{1}$$

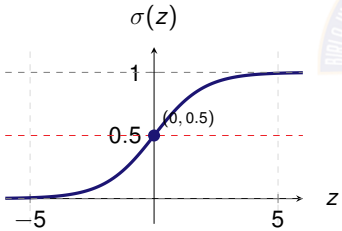
where $\sigma(\cdot)$ is the sigmoid (logistic) function.

The Sigmoid Activation Function

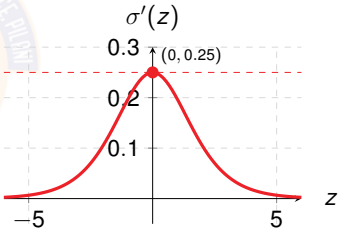
Sigmoid (Logistic) Function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Derivative formula: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Function



Derivative



At $z = 0$: $\sigma(0) = 0.5 \quad \sigma'(0) = 0.25$

Properties of Sigmoid Function

Key Properties:

- Range: $\sigma(z) \in (0, 1) \rightarrow$ perfect for probabilities!
- Monotonic: Always increasing
- Smooth: Differentiable everywhere
- Symmetry: $\sigma(-z) = 1 - \sigma(z)$
- Derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Behavior:

- When $z \rightarrow +\infty$: $\sigma(z) \rightarrow 1$ (confident class 1)
- When $z \rightarrow -\infty$: $\sigma(z) \rightarrow 0$ (confident class 0)
- When $z = 0$: $\sigma(z) = 0.5$ (uncertain)

The derivative property makes sigmoid ideal for gradient-based learning!

Table of Contents

The Four Components

A complete deep learning system for classification consists of:

1. Data: d-dimensional input vectors and binary labels
2. Model: Single neuron with sigmoid activation
3. Objective Function: Binary cross-entropy loss
4. Learning Algorithm: Stochastic gradient descent (SGD)

Note: We use **stochastic** gradient descent instead of batch gradient descent for efficiency. (This is for learning purposes only.)

Matrix Form of Data

For efficient computation, we organize data into matrices:

Input Data = $\mathbf{x}^{(i)} = [1, x_1^{(i)}, \ldots, x_d^{(i)}]^T$

Design Matrix: $\mathbf{X} \in \mathbb{R}^{N \times (d+1)}$ $\mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_d^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \cdots & x_d^{(N)} \end{bmatrix}$

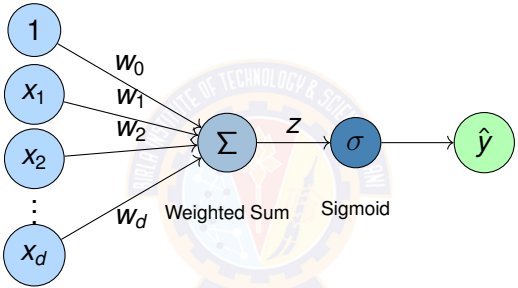
Label Vector: $\mathbf{y} \in \{0, 1\}^N$ $\mathbf{y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \ldots & y^{(N)} \end{bmatrix}^\top$

Weight Vector: $\mathbf{w} \in \mathbb{R}^{d+1}$ $\mathbf{w} = \begin{bmatrix} w_0 & w_1 & w_2 & \ldots & w_d \end{bmatrix}^\top$

Key difference from regression: $y^{(i)} \in \{0, 1\}$ (discrete), not continuous!

Logistic Regression as a Single Neuron

Model binary classification as a single artificial neuron with sigmoid activation:



- Input: $\mathbf{x} \in \mathbb{R}^d$
- Linear combination: $z = \mathbf{w}^T\mathbf{x}$
- Activation: Sigmoid $\hat{y} = \sigma(z)$
- Output: Predicted probability $\hat{y} = P(y = 1|\mathbf{x}) \in [0, 1]$

Model Prediction

Model: A single neuron computes weighted sum and applies sigmoid activation.

Prediction for input **x**:

$$z = \mathbf{w}^T\mathbf{x} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_dx_d \tag{2}$$

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

$$\hat{y} = P(y = 1|\mathbf{x}) \in [0, 1] \tag{4}$$

where:

- z is the logit (pre-activation).
- $\hat{y}$ is the predicted probability.

Making Predictions

Decision Rule:

Given predicted probability $\hat{y} = P(y = 1|\mathbf{x})$:

$$\text{Predicted class} = \begin{cases} 1 & \text{if } \hat{y} \geq 0.5 \\ 0 & \text{if } \hat{y} < 0.5 \end{cases} \tag{5}$$

Equivalently, since $\sigma(z) = 0.5$ when $z = 0$:

$$\text{Predicted class} = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} \geq 0 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x} < 0 \end{cases} \tag{6}$$

The decision boundary is: $\mathbf{w}^T\mathbf{x} = 0$ (a hyperplane in feature space)

Binary Cross-Entropy Loss

We cannot use squared error for classification. Instead, we use cross-entropy loss.

Cross-entropy Loss for a single example:

$$\ell(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = -\left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right] \tag{7}$$

where $\hat{y}^{(i)} = \sigma(\mathbf{w}^T \mathbf{x}^{(i)})$

Interpretation:

- If $y^{(i)} = 1$: Loss $= -\log(\hat{y}^{(i)})$
  - Penalizes if $\hat{y}^{(i)}$ is far from 1
- If $y^{(i)} = 0$: Loss $= -\log(1 - \hat{y}^{(i)})$
  - Penalizes if $\hat{y}^{(i)}$ is far from 0

Total Loss Function

Average Binary Cross-Entropy Loss:

$$J(\mathbf{w}) = \frac{-1}{N} \sum_{i=1}^{N} \left[y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right] \tag{8}$$

Goal:

Find $\mathbf{w}^*$ that minimizes $J(\mathbf{w})$

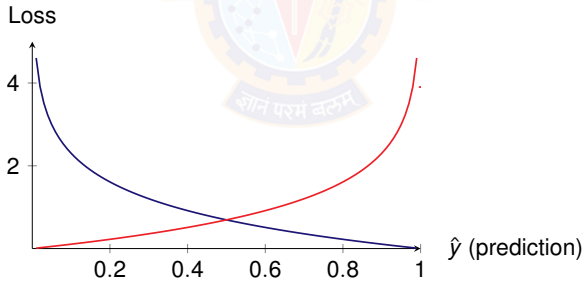
$$\mathbf{w}^* = \arg\min_{\mathbf{w}} J(\mathbf{w}) \tag{9}$$

Why Cross-Entropy Loss?

Advantages:

- Probabilistic interpretation: Derived from maximum likelihood estimation
- Appropriate for probabilities: Heavily penalizes confident wrong predictions
- Convex: Has a single global minimum for logistic regression
- Well-behaved gradients: Works well with sigmoid activation

Loss behavior: Loss increases exponentially as prediction diverges from true label!



From Batch to Stochastic Gradient Descent

Batch Gradient Descent (from Module 3):

- Uses **all** training examples in each update
- Computes gradient: $\nabla J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \nabla \ell(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)})$
- Update: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J(\mathbf{w})$

Problems with Batch GD:

- Slow for large datasets (must process all N examples per update)
- High memory requirements
- Redundant computations when examples are similar

Solution: Stochastic Gradient Descent (SGD)

- Update weights using **one random example** at a time
- Much faster iterations
- Lower memory requirements

Stochastic Gradient Descent (SGD)

Key Idea:

Instead of computing gradient over entire dataset, use gradient from a **single random example**.

SGD Update Rule:

At iteration t:

1. Randomly select one example $(\mathbf{x}^{(i)}, y^{(i)})$
2. Compute gradient for this example: $\nabla \ell(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)})$ (noisy gradient estimate)
3. Update weights: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla \ell(\mathbf{w}^{(t)}; \mathbf{x}^{(i)}, y^{(i)})$

The noise in SGD actually helps escape local minima and often leads to better generalization!

Computing the Gradient for SGD

Gradient of cross-entropy loss for one example:

For example $(\mathbf{x}^{(i)}, y^{(i)})$:

$$\nabla_{\mathbf{w}} \ell(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = \frac{\partial \ell}{\partial w_j} = \left(\hat{y}^{(i)} - y^{(i)}\right) \mathbf{x}^{(i)} \tag{10}$$

where $\hat{y}^{(i)} = \sigma(\mathbf{w}^T \mathbf{x}^{(i)})$ for $j = 0, 1, \ldots, d$ (where $x_0^{(i)} = 1$ for bias)

Remarkably simple form!

- Error: $(\hat{y}^{(i)} - y^{(i)})$
- Direction: $\mathbf{x}^{(i)}$
- Same structure as linear regression gradient!

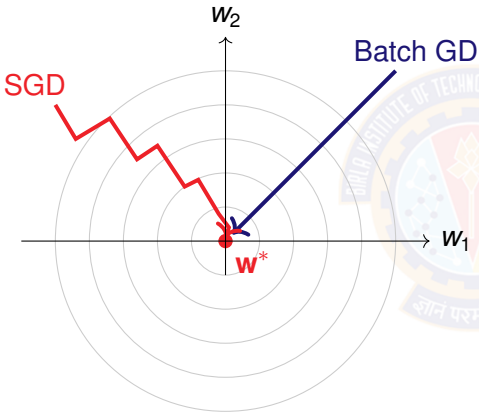
Stochastic Gradient Descent Algorithm

Algorithm 1: Stochastic Gradient Descent for Logistic Regression

Input: Dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{N}$, learning rate η, epochs T
Output: Learned weights $\mathbf{w}$

Initialize $\mathbf{w}^{(0)} = \mathbf{0}$ (or small random values);
for *epoch* $= 1$ *to* T **do**
    Shuffle dataset $\mathcal{D}$;
    for *each example* $(\mathbf{x}^{(i)}, y^{(i)})$ *in* $\mathcal{D}$ **do**
        Compute prediction: $\hat{y}^{(i)} = \sigma(\mathbf{w}^T \mathbf{x}^{(i)})$;
        Compute gradient: $\nabla \ell = (\hat{y}^{(i)} - y^{(i)})\mathbf{x}^{(i)}$;
        Update weights: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \ell$;
    Optional: Compute total loss $J(\mathbf{w})$ *for monitoring*;
return $\mathbf{w}$;

SGD vs Batch GD: Visual Comparison



Batch GD: Smooth, deterministic path
SGD: Noisy, stochastic path but faster iterations

Benefits and Challenges of SGD

Benefits:

- Speed: Much faster per iteration (only one example)
- Memory: Can handle datasets that don't fit in memory
- Online learning: Can update model as new data arrives
- Generalization: Noise helps escape sharp minima
- Scalability: Works well with massive datasets

Challenges:

- Noisy updates: Path to minimum is not smooth
- Learning rate: More sensitive to η choice
- Convergence: May oscillate near minimum
- Hyperparameter tuning: Requires careful tuning

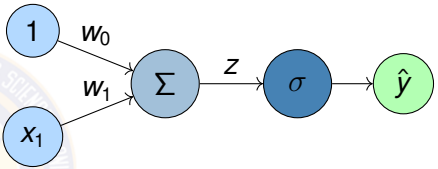
Common compromise: Mini-batch SGD

Table of Contents

Numerical Example: Setup

Tiny Dataset: Classify whether a student passes based on study hours

Hours (x_1)	Pass (y)
1	0
2	0
3	1
4	1



Matrix Formulation:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 1 & 3 \\ 1 & 4 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Numerical Example: Initialization

Hyperparameters: $\eta = 0.5, \quad N = 4$

Initialize weights: $\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Sigmoid function: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$

Loss function: $\ell(\mathbf{w}; \mathbf{x}, y) = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$

Initial predictions with $\mathbf{w}^{(0)} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$: For all examples: $z = 0 \Rightarrow \hat{y} = \sigma(0) = 0.5$

$$\hat{\mathbf{y}}^{(0)} = \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}$$

Numerical Example: First SGD Iteration

Randomly select first example: $(\mathbf{x}^{(1)}, y^{(1)}) = (\begin{bmatrix}1 & 1\end{bmatrix}, 0)$

Step 1 - Forward pass: $z^{(1)} = \mathbf{X}^{(1)}\mathbf{w}^{(0)} = \begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} = 0$

$$\hat{y}^{(1)} = \sigma(0) = 0.5$$

Step 2 - Compute error: $\text{error} = \hat{y}^{(1)} - y^{(1)} = 0.5 - 0 = 0.5$

Step 3 - Compute gradient: $\nabla\ell = (\hat{y}^{(1)} - y^{(1)})\mathbf{x}^{(1)} = 0.5\begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0.5\\0.5\end{bmatrix}$

Step 4 - Update weights: $\mathbf{w}^{(1)} = \mathbf{w}^{(0)} - \eta\nabla\ell$

$$= \begin{bmatrix}0\\0\end{bmatrix} - 0.5\begin{bmatrix}0.5\\0.5\end{bmatrix} = \begin{bmatrix}-0.25\\-0.25\end{bmatrix}$$

Numerical Example: Second SGD Iteration

Randomly select second example: $(\mathbf{x}^{(3)}, y^{(3)}) = (\begin{bmatrix}1 & 3\end{bmatrix}, 1)$

Step 1: Forward pass: $z^{(3)} = \mathbf{X}^{(3)}\mathbf{w}^{(1)} = \begin{bmatrix}1 & 3\end{bmatrix}\begin{bmatrix}-0.25\\-0.25\end{bmatrix} = -1.0$

$$\hat{y}^{(3)} = \sigma(-1.0) = \frac{1}{1+e^{1.0}} \approx 0.269$$

Step 2: Compute error: $\text{error} = 0.269 - 1 = -0.731$

Step 3: Compute gradient: $\nabla \ell = -0.731\begin{bmatrix}1\\3\end{bmatrix} = \begin{bmatrix}-0.731\\-2.193\end{bmatrix}$

Step 4: Update weights: $\mathbf{w}^{(2)} = \begin{bmatrix}-0.25\\-0.25\end{bmatrix} - 0.5\begin{bmatrix}-0.731\\-2.193\end{bmatrix} = \begin{bmatrix}0.116\\0.847\end{bmatrix}$

Numerical Example: Checking Progress

After 2 SGD iterations, let's check predictions:

With $\mathbf{w}^{(2)} = \begin{bmatrix} 0.116 \\ 0.847 \end{bmatrix}$:

$$\text{Example 1:} \quad \hat{y}^{(1)} = \sigma(0.116 + 0.847 \cdot 1) \approx \sigma(0.963) \approx 0.724$$

$$\text{Example 2:} \quad \hat{y}^{(2)} = \sigma(0.116 + 0.847 \cdot 2) \approx \sigma(1.810) \approx 0.859$$

$$\text{Example 3:} \quad \hat{y}^{(3)} = \sigma(0.116 + 0.847 \cdot 3) \approx \sigma(2.657) \approx 0.934$$

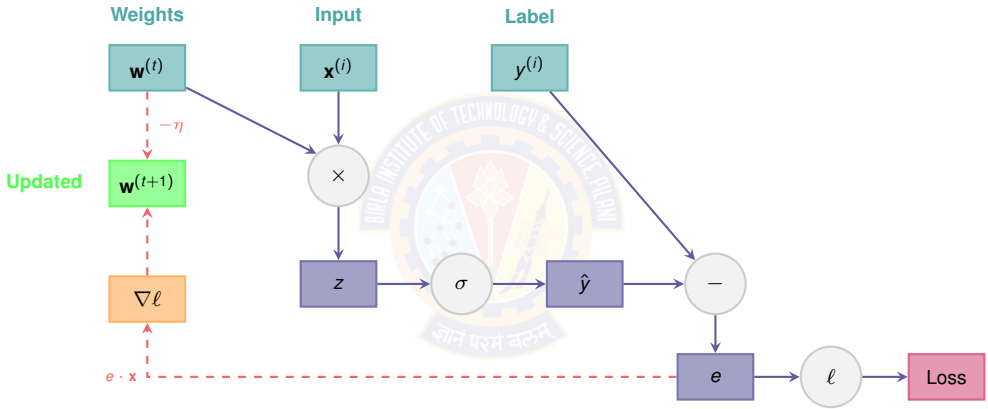
$$\text{Example 4:} \quad \hat{y}^{(4)} = \sigma(0.116 + 0.847 \cdot 4) \approx \sigma(3.504) \approx 0.971$$

Progress:

- Examples 1, 2 (true label 0): predictions moving toward 0 ✓
- Examples 3, 4 (true label 1): predictions moving toward 1 ✓

Continue SGD for multiple epochs until convergence!

Computational Graph for One SGD Step



Single example used per iteration — much faster than batch GD!

Table of Contents

Evaluating Classification Models

Unlike regression, we need different metrics for classification.

Key Questions:

- How often does the model predict correctly? (Accuracy)
- What types of errors does it make? (Confusion Matrix)
- How well does it identify positive cases? (Precision, Recall)

Data Split:

- Training set: Learn weights **w** (typically 90-99%)
- Validation set: Tune hyperparameters (typically 10-15%)
- Test set: Final evaluation (typically 1-10%)

Always evaluate on unseen test data to measure generalization!

Confusion Matrix

The confusion matrix shows all prediction outcomes:

		Predicted	
		Positive (1)	Negative (0)
Actual	Positive (1)	True Positive (TP)	False Negative (FN)
	Negative (0)	False Positive (FP)	True Negative (TN)

Definitions:

- TP: Correctly predicted positive
- TN: Correctly predicted negative
- FP: Incorrectly predicted positive (Type I error)
- FN: Incorrectly predicted negative (Type II error)

Classification Metrics I

1. Accuracy: Overall correctness

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{11}$$

2. Precision: Of predicted positives, how many are correct?

$$\text{Precision} = \frac{TP}{TP + FP} \tag{12}$$

Classification Metrics II

3. Recall (Sensitivity): Of actual positives, how many did we find?

$$\text{Recall} = \frac{TP}{TP + FN} \tag{13}$$

4. F1 Score: Harmonic mean of precision and recall

$$F1 = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{14}$$

When to Use Which Metric?

Accuracy:

- Use when classes are balanced
- Misleading with imbalanced data! (e.g., 95% negative class)

Precision:

- Important when False Positives are costly
- Example: Spam detection (don't want to mark real emails as spam)

Recall:

- Important when False Negatives are costly
- Example: Disease detection (don't want to miss sick patients)

F1 Score:

- Use when you need balance between precision and recall
- Good for imbalanced datasets

Example: Computing Metrics

Given predictions:

True	Pred	True	Pred
1	1 (TP)	0	0 (TN)
1	1 (TP)	0	0 (TN)
1	0 (FN)	0	1 (FP)
1	1 (TP)	0	0 (TN)

Counts: TP = 3, TN = 3, FP = 1, FN = 1

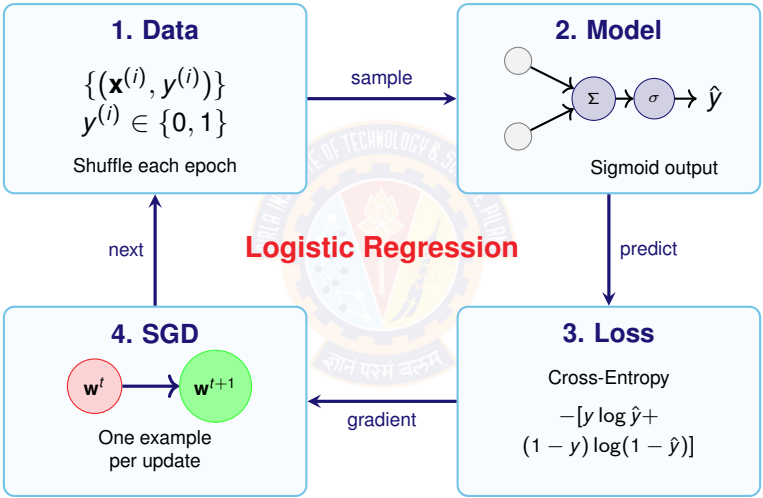
$$\text{Accuracy} = \frac{3+3}{3+3+1+1} = \frac{6}{8} = 0.75$$

$$\text{Precision} = \frac{3}{3+1} = 0.75$$

$$\text{Recall} = \frac{3}{3+1} = 0.75$$

$$F1 = 2 \cdot \frac{0.75 \times 0.75}{0.75 + 0.75} = 0.75$$

Summary: Binary Classification



This framework is the foundation for neural networks!

Key Takeaways

- Binary classification predicts discrete categories using $y \in \{0, 1\}$.
- Sigmoid activation $\sigma(z) = \frac{1}{1+e^{-z}}$ maps to probabilities $[0, 1]$.
- Logistic regression = linear model + sigmoid activation.
- Cross-entropy loss is the appropriate objective for classification.
- Stochastic Gradient Descent (SGD) updates weights one example at a time — faster and more scalable.
- Feature scaling is essential for convergence and numerical stability.
- Evaluation metrics: Accuracy, Precision, Recall, F1 Score, Confusion Matrix.
- This framework extends naturally to multi-class classification and deep networks.

Table of Contents

What is Multi-class Classification?

Definition

Multi-class classification assigns each input to one of K classes, where $K > 2$.

Key Characteristics:

- Classes: $y \in \{1, 2, 3, \ldots, K\}$ (or $\{0, 1, 2, \ldots, K-1\}$)
- Each example belongs to **exactly one** class
- Output: Probability distribution over all K classes
- Probabilities sum to 1: $\sum_{k=1}^{K} P(y = k|\mathbf{x}) = 1$

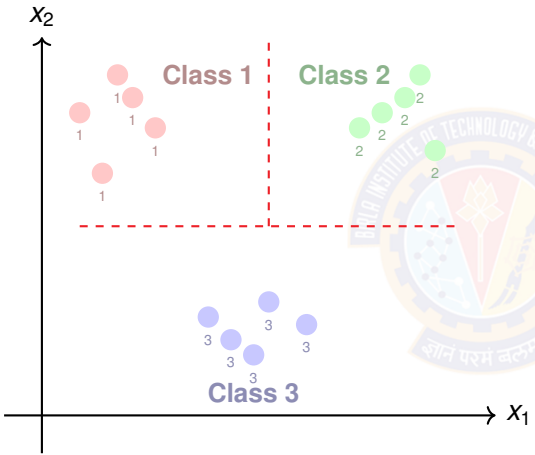
Example:

- Input: Image of a handwritten digit
- Output: Probabilities for each of 10 classes (0, 1, 2, ..., 9)
- Prediction: Class with highest probability

Examples of Multi-class Classification

1. Handwritten Digit Recognition (MNIST)
   - Input: 28×28 pixel image
   - Output: One of 10 digits (0-9)
2. Image Classification (ImageNet)
   - Input: Color image
   - Output: One of 1000 object categories (cat, dog, car, plane, ...)
3. Text Classification
   - Input: News article text
   - Output: Topic category (politics, sports, technology, entertainment, business)
4. Speech Recognition
   - Input: Audio features
   - Output: Phoneme class (multiple phonemes in a language)

Multi-class Classification: Visual Example



Key insight: We need **multiple output neurons** (one per class) and a way to convert their outputs into a valid probability distribution.

Decision boundaries separate $K = 3$ classes in feature space.

Table of Contents

The Four Components for Multi-class

A complete multi-class classification system consists of:

1. Data: Input features and class labels $y \in \{1, 2, \ldots, K\}$
2. Model: Linear model with K output neurons (no hidden layers)
3. Activation: Softmax function to produce probability distribution
4. Objective Function: Categorical cross-entropy loss
5. Learning Algorithm: Mini-batch stochastic gradient descent

Note: We now use **mini-batch SGD** instead of single-example SGD for better efficiency and stability.

Data Representation

Input Features:

$$\mathbf{X} \in \mathbb{R}^{N \times (d+1)} \qquad \mathbf{X} = \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_d^{(1)} \\ 1 & x_1^{(2)} & \cdots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_1^{(N)} & \cdots & x_d^{(N)} \end{bmatrix}$$

Class Labels: Two representations

1. Integer encoding: $\mathbf{y} \in \{1, 2, \ldots, K\}^N$

$$\mathbf{y} = \begin{bmatrix} 3 & 1 & 2 & 1 & 3 \end{bmatrix}^T$$

2. One-hot encoding: $\mathbf{Y} \in \{0, 1\}^{N \times K}$ (preferred for computation)

One-Hot Encoding

Definition: Convert class label to binary vector of length K

For class k: All zeros except 1 at position k

Example with $K = 4$ classes:

$$\text{Class 1:} \quad \mathbf{y} = [1, 0, 0, 0]^T$$
$$\text{Class 2:} \quad \mathbf{y} = [0, 1, 0, 0]^T$$
$$\text{Class 3:} \quad \mathbf{y} = [0, 0, 1, 0]^T$$
$$\text{Class 4:} \quad \mathbf{y} = [0, 0, 0, 1]^T$$

Why one-hot?

- Treats all classes equally (no implicit ordering)
- Matches the format of softmax output probabilities
- Simplifies loss computation
- Standard in deep learning frameworks

Weight Matrix for Multi-class Classificati

For K classes and d input features:

Weight Matrix: $\mathbf{W} \in \mathbb{R}^{(d+1)\times K}$

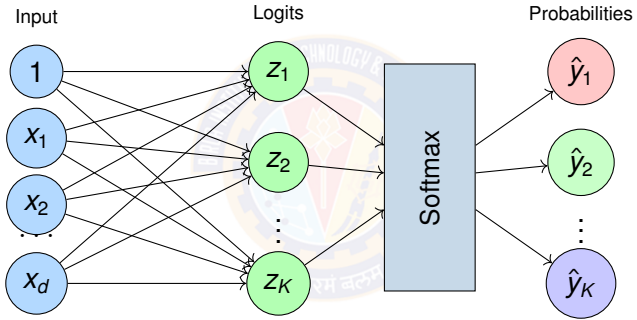
$$\mathbf{W} = \begin{bmatrix} w_{0,1} & w_{0,2} & \cdots & w_{0,K} \\ w_{1,1} & w_{1,2} & \cdots & w_{1,K} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ w_{d,1} & w_{d,2} & \cdots & w_{d,K} \end{bmatrix}$$

- Each **column** $\mathbf{w}_k$ represents weights for class k
- Each class has its own linear model
- Row 0: Bias terms for all classes
- Total parameters: $(d+1) \times K$

Compare to binary: 1 weight vector vs K weight vectors!

Multi-class Model Architecture

Architecture: Multiple output neurons with softmax activation



$$\mathbf{z} = \mathbf{W}^T\mathbf{x} \text{ (linear)} \rightarrow \qquad \hat{\mathbf{y}} = \text{softmax}(\mathbf{z}) \text{ (activation)}$$

Linear Model: Computing Logits

Step 1: Compute logits (pre-activation values)

For a single input $\mathbf{x}$:

$$\mathbf{z} = \mathbf{W}^T\mathbf{x} = \begin{bmatrix} \mathbf{w}_1^T\mathbf{x} \\ \mathbf{w}_2^T\mathbf{x} \\ \vdots \\ \mathbf{w}_K^T\mathbf{x} \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_K \end{bmatrix} \tag{15}$$

where $z_k = \mathbf{w}_k^T\mathbf{x} = w_{0,k} + w_{1,k}x_1 + \cdots + w_{d,k}x_d$

For entire dataset (batch):

$$\mathbf{Z} = \mathbf{X}\mathbf{W} \in \mathbb{R}^{N \times K} \tag{16}$$

Each row of $\mathbf{Z}$ contains logits for one example.

Logits are **unbounded** values in $(-\infty, \infty)$ $\rightarrow$ not probabilities yet!

The Softmax Function

Purpose: Convert K logits into a valid probability distribution

Definition:

For logit vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]^T$

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } k = 1, 2, \ldots, K \tag{17}$$

Output: Probability vector $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_K]^T$

$$\hat{y}_k = P(y = k|\mathbf{x}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{18}$$

Interpretation: $\hat{y}_k$ is the predicted probability for class k

Properties of Softmax

Key Properties:

1. Output range: $\hat{y}_k \in (0, 1)$ for all k
2. Probability distribution: $\sum_{k=1}^{K} \hat{y}_k = 1$
3. Preserves order: If $z_i > z_j$, then $\hat{y}_i > \hat{y}_j$
4. Translation invariant: $\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c)$ for any constant c
5. Differentiable: Enables gradient-based learning
6. Reduces to sigmoid: When $K = 2$, softmax is equivalent to sigmoid

The exponential amplifies differences: larger logits get much larger probabilities!

Softmax Example: Numerical Computation

Given logits: $\mathbf{z} = [2.0, 1.0, 0.1]^T$ for $K = 3$ classes

Step 1: Compute exponentials

$$e^{z_1} = e^{2.0} \approx 7.39$$
$$e^{z_2} = e^{1.0} \approx 2.72$$
$$e^{z_3} = e^{0.1} \approx 1.11$$

Step 2: Compute sum

$$\sum_{j=1}^{3} e^{z_j} = 7.39 + 2.72 + 1.11 = 11.22$$

Step 3: Normalize

$$\hat{y}_1 = \frac{7.39}{11.22} \approx 0.659$$
$$\hat{y}_2 = \frac{2.72}{11.22} \approx 0.242$$
$$\hat{y}_3 = \frac{1.11}{11.22} \approx 0.099$$

Step 4: Predicted class:

$\arg\max_k \hat{y}_k = 1$ (highest probability)

Complete Forward Pass

For a batch of N examples $\mathbf{X}$:

$$\text{Step 1: Compute logits} \quad \mathbf{Z} = \mathbf{X}\mathbf{W} \in \mathbb{R}^{N \times K} \tag{19}$$

$$\text{Step 2: Apply softmax} \quad \hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z}) \in [0,1]^{N \times K} \tag{20}$$

(softmax applied row-wise to each example)

$$\text{Step 3: Predict class} \quad \hat{k} = \arg\max_k \hat{y}_k \tag{21}$$

Each row of $\hat{\mathbf{Y}}$ is a probability distribution over K classes.

Categorical Cross-Entropy Loss

Total loss over dataset:

$$J(\mathbf{W}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{y}_k^{(i)}) \tag{22}$$

Goal: Minimize $J(\mathbf{W})$ to find optimal weights $\mathbf{W}^*$

Understanding Categorical Cross-Entropy

Intuition:

- Loss depends only on predicted probability for the **true class**
- Heavily penalizes **confident wrong predictions**
- Encourages high probability for correct class

Example: True class is 2 ($\mathbf{y} = [0, 1, 0]$)

Prediction	$\hat{y}_1$	$\hat{y}_2$	Loss $= -\log(\hat{y}_2)$
Good	0.1	**0.8**	0.22
Medium	0.3	**0.5**	0.69
Bad	0.7	**0.2**	1.61
Very bad	0.9	**0.05**	3.00

Loss increases exponentially as predicted probability for true class decreases!

Why Categorical Cross-Entropy?

Advantages:

- Probabilistic interpretation: Derived from maximum likelihood
- Appropriate for multi-class: Naturally handles K classes
- Works well with softmax: Numerically stable gradient
- Penalizes wrong confidence: Heavily penalizes confident mistakes
- Convex: For linear models (like ours)

Connection to binary cross-entropy:
When $K = 2$:

$$\text{Categorical: } -[y_1 \log(\hat{y}_1) + y_2 \log(\hat{y}_2)]$$
$$\text{Binary: } -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

They are equivalent! (with $y_2 = y$ and $y_1 = 1 - y$)

Table of Contents

From SGD to Mini-batch SGD

Recap: Gradient Descent Variants

1. Batch Gradient Descent:
   - Uses **all** N examples per update
   - Slow but stable, accurate gradient
2. Stochastic Gradient Descent (SGD):
   - Uses **1** random example per update
   - Fast but noisy, unstable
3. Mini-batch SGD (Best of both worlds):
   - Uses **small batch** of B examples per update ($1 < B \ll N$)
   - Typical batch sizes: $B \in \{32, 64, 128, 256, 512\}$
   - Good balance of speed and stability
   - Standard in modern deep learning!

Why Mini-batch SGD?

Advantages of mini-batches:

1. Computational efficiency:
   - Vectorized operations (matrix multiplication)
   - Better GPU utilization
   - Faster than processing examples one-by-one
2. Gradient estimation:
   - Less noisy than single-example SGD
   - More stable convergence
   - Better approximation of true gradient
3. Generalization:
   - Noise helps escape sharp minima
   - Often leads to better test performance
4. Memory efficiency:
   - Can handle datasets larger than memory
   - Load and process data in batches

Mini-batch SGD: Key Concepts

- Batch size (B): Number of examples per update
  - Small: 32-64 (more noise, faster iterations)
  - Medium: 128-256 (balanced)
  - Large: 512-1024 (less noise, but slower convergence)
- Iteration: One weight update (processing one mini-batch)
- Epoch: One complete pass through the entire dataset

$$\text{Iterations per epoch} = \left\lceil \frac{N}{B} \right\rceil \tag{23}$$

Example: $N = 60{,}000$ samples, $B = 128$

- Iterations per epoch: $\lceil 60000/128 \rceil = 469$
- After 10 epochs: $469 \times 10 = 4690$ weight updates

Computing Gradients for Mini-batch

Gradient for a mini-batch of size B:

Let $\mathcal{B}$ be a mini-batch of B examples: $\mathcal{B} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i \in \mathcal{I}}$

Average gradient over mini-batch:

$$\nabla J_{\mathcal{B}}(\mathbf{W}) = \frac{1}{B} \sum_{i \in \mathcal{B}} \nabla \ell(\mathbf{W}; \mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \tag{24}$$

For categorical cross-entropy with softmax:

$$\nabla_{\mathbf{W}} J_{\mathcal{B}} = \frac{1}{B} \mathbf{X}_{\mathcal{B}}^T (\hat{\mathbf{Y}}_{\mathcal{B}} - \mathbf{Y}_{\mathcal{B}}) \tag{25}$$

where:

- $\mathbf{X}_{\mathcal{B}} \in \mathbb{R}^{B \times (d+1)}$: Mini-batch inputs
- $\hat{\mathbf{Y}}_{\mathcal{B}} \in \mathbb{R}^{B \times K}$: Predicted probabilities
- $\mathbf{Y}_{\mathcal{B}} \in \mathbb{R}^{B \times K}$: One-hot true labels

Mini-batch SGD Update Rule

Update at iteration t:

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \eta \nabla J_{\mathcal{B}}(\mathbf{W}^{(t)}) \tag{26}$$

Expanded form:

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \frac{\eta}{B} \mathbf{X}_{\mathcal{B}}^{T} (\hat{\mathbf{Y}}_{\mathcal{B}} - \mathbf{Y}_{\mathcal{B}}) \tag{27}$$

Mini-batch SGD Algorithm

Algorithm 2: Mini-batch SGD for Multi-class Classification

Input: Dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$, batch size B, learning rate η, epochs T

Output: Learned weight matrix $\mathbf{W}$

Initialize $\mathbf{W} \sim \mathcal{N}(0, 0.01)$ `// Small random values`

for *epoch* $= 1$ *to* T **do**

  Shuffle dataset $\mathcal{D}$;

  for *each mini-batch* $\mathcal{B}$ *of size* B **do**

    `// Forward pass`

    $\mathbf{Z}_{\mathcal{B}} = \mathbf{X}_{\mathcal{B}}\mathbf{W}$ `// Compute logits`

    $\hat{\mathbf{Y}}_{\mathcal{B}} = \text{softmax}(\mathbf{Z}_{\mathcal{B}})$ `// Apply softmax`

    $\nabla J = \frac{1}{B}\mathbf{X}_{\mathcal{B}}^{T}(\hat{\mathbf{Y}}_{\mathcal{B}} - \mathbf{Y}_{\mathcal{B}})$ `// Compute gradient`

    $\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla J$ `// Update weights`

  Optional: Compute total loss for monitoring;

return $\mathbf{W}$;

Choosing Batch Size

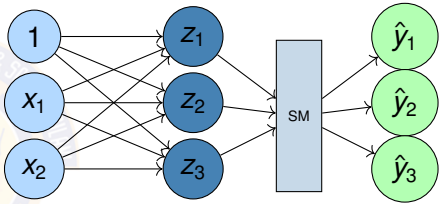
Batch Size	Advantages	Disadvantages
Small (32-64)	• Fast iterations • More updates per epoch • Better generalization • Escapes sharp minima	• Noisy gradients • Unstable training • Poor GPU utilization
Medium (128-256)	• Good balance • Stable training • Good GPU usage • Standard choice	• May need tuning
Large (512-1024)	• Stable gradients • Better GPU utilization • Faster per-epoch time	• Slow convergence • May generalize worse • More memory

Table of Contents

Numerical Example: Setup

Tiny Dataset: Classify 4 data points into 3 classes

x_1	x_2	Class
1	2	1
2	1	2
2	3	1
3	2	3



Parameters:

- $K = 3$ classes, $d = 2$ features, $N = 4$ examples
- Batch size: $B = 2$ (mini-batch)
- Learning rate: $\eta = 0.1$

Numerical Example: Data Representation

Design Matrix: (with bias column) $\mathbf{X} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix} \in \mathbb{R}^{4\times 3}$

One-hot Labels: $\mathbf{Y} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{4\times 3}$ (class 1), (class 2), (class 1), (class 3)

Weight Matrix: Initialize randomly $\mathbf{W}^{(0)} = \begin{bmatrix} 0.1 & -0.1 & 0.2 \\ 0.2 & 0.1 & -0.1 \\ -0.1 & 0.2 & 0.1 \end{bmatrix} \in \mathbb{R}^{3\times 3}$

Numerical Example: First Mini-batch (Exa

Mini-batch: First 2 examples after shuffling

$$\mathbf{X}_{\mathcal{B}} = \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix}, \quad \mathbf{Y}_{\mathcal{B}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Step 1: Compute logits

$$\begin{aligned} \mathbf{Z}_{\mathcal{B}} &= \mathbf{X}_{\mathcal{B}}\mathbf{W}^{(0)} \\ &= \begin{bmatrix} 1 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 0.1 & -0.1 & 0.2 \\ 0.2 & 0.1 & -0.1 \\ -0.1 & 0.2 & 0.1 \end{bmatrix} \\ &= \begin{bmatrix} 0.1 & 0.4 & 0.3 \\ 0.4 & 0.2 & 0.1 \end{bmatrix} \end{aligned}$$

Numerical Example: Softmax Computation

Step 2: Apply softmax to each row

Softmax for example 1: $\hat{\mathbf{y}}^{(1)} \approx \begin{bmatrix} 0.280 \\ 0.378 \\ 0.342 \end{bmatrix}$

Similarly for example 2: $\hat{\mathbf{y}}^{(2)} \approx [0.387, 0.315, 0.298]^T$

For the entire batch: $\hat{\mathbf{Y}}_{\mathcal{B}} = \begin{bmatrix} 0.280 & 0.378 & 0.342 \\ 0.387 & 0.315 & 0.298 \end{bmatrix}$

Numerical Example: Gradient and Update

Step 3: Compute gradient

$$\nabla J = \frac{1}{B}\mathbf{X}_{\mathcal{B}}^{T}(\hat{\mathbf{Y}}_{\mathcal{B}} - \mathbf{Y}_{\mathcal{B}})$$

$$\hat{\mathbf{Y}}_{\mathcal{B}} - \mathbf{Y}_{\mathcal{B}} = \begin{bmatrix} 0.280 & 0.378 & 0.342 \\ 0.387 & 0.315 & 0.298 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} -0.720 & 0.378 & 0.342 \\ 0.387 & -0.685 & 0.298 \end{bmatrix}$$

$$\nabla J = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & 2 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} -0.720 & 0.378 & 0.342 \\ 0.387 & -0.685 & 0.298 \end{bmatrix}$$

$$\approx \begin{bmatrix} -0.167 & -0.154 & 0.320 \\ -0.024 & -0.496 & 0.469 \\ -0.527 & 0.036 & 0.491 \end{bmatrix}$$

Numerical Example: Weight Update

Step 4: Update weights

$$\mathbf{W}^{(1)} = \mathbf{W}^{(0)} - \eta \nabla J$$

$$= \begin{bmatrix} 0.1 & -0.1 & 0.2 \\ 0.2 & 0.1 & -0.1 \\ -0.1 & 0.2 & 0.1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.167 & -0.154 & 0.320 \\ -0.024 & -0.496 & 0.469 \\ -0.527 & 0.036 & 0.491 \end{bmatrix}$$

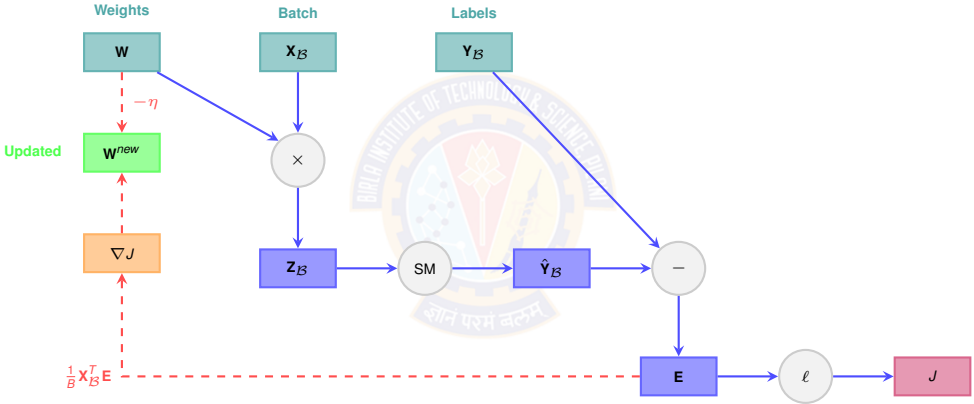
$$= \begin{bmatrix} 0.117 & -0.085 & 0.168 \\ 0.202 & 0.150 & -0.147 \\ -0.047 & 0.196 & 0.051 \end{bmatrix}$$

Progress:

- Weights adjusted based on mini-batch gradient
- Process next mini-batch (examples 3-4) with updated weights
- Continue for multiple epochs until convergence

This completes one iteration of mini-batch SGD!

Computational Graph for Mini-batch Forwa



Mini-batch processes B examples simultaneously — efficient and stable!

Table of Contents

Inference: Making Predictions

After training, use the model to classify new examples.

Inference process for input $\mathbf{x}_{\text{new}}$:

$$\text{Compute logits:} \quad \mathbf{z} = \mathbf{W}^T \mathbf{x}_{\text{new}} \tag{28}$$

$$\text{Apply softmax:} \quad \hat{\mathbf{y}} = \text{softmax}(\mathbf{z}) \tag{29}$$

$$\text{Predict class:} \quad \hat{k} = \arg \max_{k=1,\ldots,K} \hat{y}_k \tag{30}$$

Optional: Report confidence scores (the probabilities $\hat{\mathbf{y}}$)

Inference Example

Trained model with $K = 3$ classes, new input $\mathbf{x} = [1, 2.5, 1.8]^T$

Step 1: Compute logits $\mathbf{z} = \mathbf{W}^T\mathbf{x} = [1.2, 2.5, 0.8]^T$

Step 2: Apply softmax $e^{1.2} \approx 3.32, \quad e^{2.5} \approx 12.18, \quad e^{0.8} \approx 2.23$

$$\text{Sum} = 3.32 + 12.18 + 2.23 = 17.73$$

$$\hat{\mathbf{y}} = \begin{bmatrix} 3.32/17.73 \\ 12.18/17.73 \\ 2.23/17.73 \end{bmatrix} = \begin{bmatrix} 0.187 \\ 0.687 \\ 0.126 \end{bmatrix}$$

Step 3: Predict class $\hat{k} = \arg\max_k \hat{y}_k = 2$ (Class 2 with 68.7% confidence)

Table of Contents

Evaluation Metrics for Multi-class

1. Accuracy: Overall correctness

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{N} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[\hat{y}^{(i)} = y^{(i)}] \tag{31}$$

2. Confusion Matrix: $K \times K$ matrix
   - Row i, Column j: Number of class i examples predicted as class j
   - Diagonal: Correct predictions
   - Off-diagonal: Misclassifications

3. Per-class Precision and Recall:
   - Compute for each class separately (one-vs-all)
   - Average across classes (macro-average or weighted average)

Confusion Matrix for Multi-class

Example: 3-class confusion matrix

		Predicted		
		Class 1	Class 2	Class 3
Actual	Class 1	45	3	2
	Class 2	5	38	7
	Class 3	2	4	44

Interpretation:

- Diagonal (green): Correct predictions (45, 38, 44)
- Off-diagonal (red): Misclassifications
- Row sums: Total actual examples per class (50, 50, 50)
- Column sums: Total predicted examples per class

Accuracy: $(45 + 38 + 44)/150 = 0.847$ or 84.7%

Per-class Metrics

For each class k, compute:

Precision for class k: Of all predictions for class k, what fraction were correct?

$$\text{Precision}_k = \frac{\text{TP}_k}{\text{TP}_k + \text{FP}_k} \tag{32}$$

Recall for class k: Of all actual class k examples, what fraction did we identify?

$$\text{Recall}_k = \frac{\text{TP}_k}{\text{TP}_k + \text{FN}_k} \tag{33}$$

F1 Score for class k:

$$F1_k = 2 \cdot \frac{\text{Precision}_k \times \text{Recall}_k}{\text{Precision}_k + \text{Recall}_k} \tag{34}$$

Averaging Strategies

How to combine per-class metrics into overall score?

1. Macro-average: Simple average across classes

$$\text{Precision}_{\text{macro}} = \frac{1}{K}\sum_{k=1}^{K}\text{Precision}_k \tag{35}$$

- Treats all classes equally
- Good when classes are equally important

2. Weighted average: Weight by class frequency

$$\text{Precision}_{\text{weighted}} = \sum_{k=1}^{K} w_k \cdot \text{Precision}_k \qquad w_k = N_k/N \tag{36}$$

- Accounts for class imbalance
- Good when some classes are more common

Top-K Accuracy

For problems with many classes (e.g., ImageNet with 1000 classes):

Top-K Accuracy: Correct if true class is in top K predictions

$$\text{Top-K Accuracy} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}[y^{(i)} \in \text{Top-K}(\hat{\mathbf{y}}^{(i)})] \tag{37}$$

Example: Image classification with 1000 classes

- Top-1 accuracy: Is the highest probability class correct?
- Top-5 accuracy: Is the true class in the top 5 predictions?

Why useful?

- More forgiving metric for large K
- Captures whether model has correct class among top candidates
- Standard in ImageNet challenge

Summary: Multiclass Classification

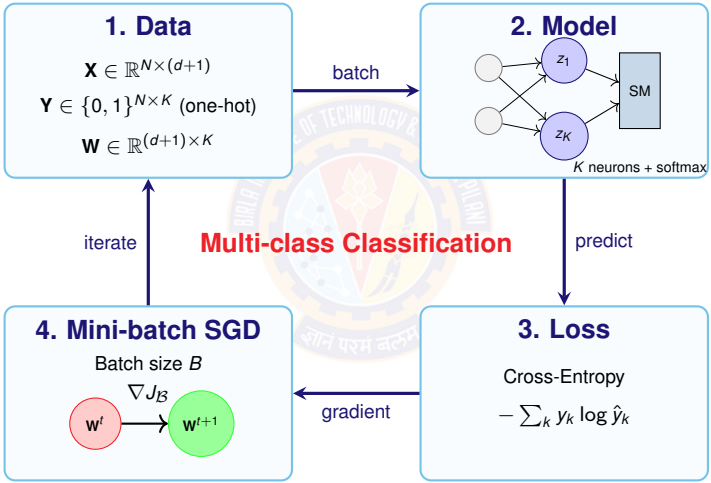


Table of Contents

Implementation Tips

- 1. Learning Rate & Epochs:
  - Start: $\eta \in \{0.01, 0.1, 1.0\}$
  - Typical: 10-100 epochs
  - Use early stopping
- 2. Weight Initialization:
  - $w_j \sim \mathcal{N}(0, 0.01)$
  - Avoid zeros or large values
- 3. Batch Size:
  - Start: $B \in \{64, 128, 256\}$
  - Powers of 2 preferred
  - Larger: stable, slower
  - Smaller: noisy, faster
- 4. Data Handling:
  - **Always shuffle** each epoch
  - Check class balance
- 5. Feature Scaling: **Essential!**
  - Standardization: $x'_j = \frac{x_j - \mu_j}{\sigma_j}$
  - Prevents sigmoid saturation
- 6. Numerical Stability:
  - Use: $\log(\sigma(z))$
  - Stable softmax: $c = \max_k z_k$
  - $\text{SM}(z)_k = \frac{e^{z_k - c}}{\sum_j e^{z_j - c}}$
  - Log-sum-exp trick
  - Prevents overflow/underflow

Debugging Checklist

- Loss is NaN/Inf:
  - η too large $\rightarrow$ decrease
  - Check $\log(0)$ in loss
  - Scale features
  - Use stable softmax
- Predicts same class:
  - Check class balance
  - Verify softmax/loss
  - η might be too small
  - Verify one-hot encoding
- Loss oscillates:
  - η too large
  - Use mini-batches
  - Apply learning rate decay
- Softmax outputs $\neq 1$:
  - Implementation bug
  - Numerical precision
  - Use stable softmax
- Loss not decreasing:
  - Check gradient signs
  - η too small/large
  - Verify feature scaling
  - Verify data shuffling
  - Check one-hot encoding
- High train, low test accuracy:
  - Overfitting
  - Add regularization
  - Get more training data
  - Simpler model
- Sanity checks:
  - Tiny dataset $\rightarrow$ 100% accuracy
  - Verify $\hat{y} \in [0, 1]$
  - Verify $\sum_k \hat{y}_k = 1$
  - Shuffling improves results
  - Print confusion matrix

Table of Contents

Activation Functions

Property	**Identity**	**Sigmoid**	**Softmax**
Formula	$f(z) = z$	$\sigma(z) = \frac{1}{1+e^{-z}}$	$\text{SM}(z)_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$
Output Range	$(-\infty, \infty)$	$[0, 1]$	$[0, 1]^K, \sum_k = 1$
Use Case	Regression	Binary classification	Multi-class
# Outputs	1	1	K (coupled)

Loss Functions

Aspect	MSE	Binary CE	Categorical CE
Formula	$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$	$\frac{-1}{N}\sum_{i=1}^{N}[y^{(i)}\log\hat{y}^{(i)} + (1-y^{(i)})\log(1-\hat{y}^{(i)})]$	$\frac{-1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_k^{(i)}\log\hat{y}_k^{(i)}$
Task	Regression	Binary classification	Multi-class classification
Output Type	Continuous value	Probability (2 classes)	Probability distribution (K classes)
Target y	Real number	$\{0, 1\}$	One-hot vector
Prediction $\hat{y}$	Any real number	Probability $[0, 1]$	Probability vector, $\sum_k \hat{y}_k = 1$
Activation	Identity	Sigmoid	Softmax
Penalizes	Distance from target	Confident wrong predictions	Wrong class probability
Properties	Convex, Quadratic penalty	Convex, Probabilistic, Logarithmic penalty	Convex, Extends binary CE, Multi-class penalty

Gradient Descent Variants

Aspect	Batch GD	Stochastic GD	Mini-batch GD
Examples/update	All N	1	B (batch size)
Gradient	$\nabla J = \frac{1}{N}\sum_{i=1}^{N} \nabla \ell_i$	$\nabla \ell_i$ for random i	$\nabla J_{\mathcal{B}} = \frac{1}{B}\sum_{i \in \mathcal{B}} \nabla \ell_i$
Update rule	$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J$	$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \ell_i$	$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J_{\mathcal{B}}$
Iterations/epoch	1	N	$\lceil N/B \rceil$
Speed per iteration	Slow	Fast	Medium
Convergence	Smooth, deterministic	Noisy, oscillating	Balanced
Memory	High (all data)	Low (1 example)	Medium (B examples)
Gradient quality	Exact	Noisy estimate	Good estimate
Advantages	Stable, Precise gradient, Reproducible	Fast iterations, Online learning, Escapes local minima	**Best balance**, GPU efficient, Stable & fast
Disadvantages	Slow for large N, High memory, NO online	Very noisy, Unstable, Hard to parallelize	Extra hyperparameter (B), Needs tuning
Best for	Small datasets	Online learning	**Deep learning (std)**
Typical B	N (all data)	1	$32, 64, 128, 256$

Comparison of Linear models

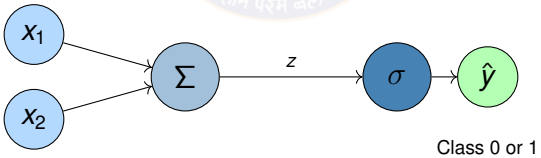
Aspect	Regression	Binary	Multi-class
Output	Continuous $y \in \mathbb{R}$	Discrete $y \in \{0, 1\}$	Discrete $y \in \{1, .., K\}$
Activation	Identity: $f(z) = z$	Sigmoid: $\sigma(z)$	Softmax: SM($\mathbf{z}$)
Loss Function	Mean Squared Error	Binary Cross-Entropy	Categorical Cross-Entropy
Output Range	$(-\infty, \infty)$	$[0, 1]$	$[0, 1]^K$, $\sum_k = 1$
Interpretation	Predicted value	$P(y = 1\|\mathbf{x})$	$P(y = k\|\mathbf{x})$ for all k
Output neurons	1	1	K
Weights	Vector $\mathbf{w} \in \mathbb{R}^{d+1}$	Vector $\mathbf{w} \in \mathbb{R}^{d+1}$	Matrix $\mathbf{W} \in \mathbb{R}^{(d+1)\times K}$
Evaluation	MSE, RMSE, R^2	Accuracy, Precision, Recall, F1	Confusion matrix, Top-K
Learning	Batch GD or SGD	SGD or Mini-batch	Mini-batch SGD
Examples	House prices, temperature	Spam detection, diagnosis	Digit recognition, ImageNet

Recap: Binary Classification

Logistic Regresion

Key insight: Sigmoid maps linear combination to probability; optimized by minimizing cross-entropy loss.

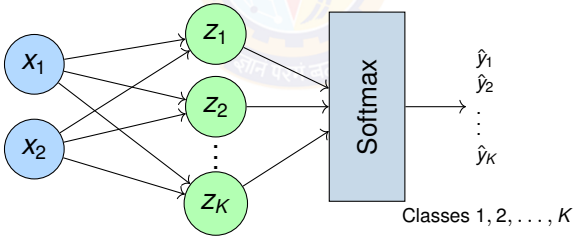
- Problem: Classify into two classes: $y \in \{0, 1\}$
- Model: Single neuron with sigmoid activation
- Output: Probability $\hat{y} = P(y = 1|\mathbf{x}) \in [0, 1]$
- Loss: Binary cross-entropy
- Training: Stochastic Gradient Descent (SGD)



Recap: Multi-class Classification

Key insight: Multiple outputs coupled through softmax create valid probability distribution over all classes.

- Problem: Classify into K classes: $y \in \{1, 2, \ldots, K\}$
- Model: K output neurons with softmax activation
- Output: Probability distribution $\hat{\mathbf{y}} = [\hat{y}_1, \ldots, \hat{y}_K]^T$, where $\sum_{k=1}^{K} \hat{y}_k = 1$
- Loss: Categorical cross-entropy
- Training: Mini-batch Stochastic Gradient Descent



References

- Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2023). *Dive into Deep Learning*. Cambridge University Press. Chapter 4 (Linear Neural Networks for Classification). `https://d2l.ai/`
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Chapter 6 (Deep Feedforward Networks). `https://www.deeplearningbook.org/`
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. Chapter 4 (Linear Models for Classification).
- Murphy, K. P. (2022). *Probabilistic Machine Learning: An Introduction*. MIT Press. Chapter 10 (Logistic Regression). `https://probml.github.io/pml-book/`

Thank You!