THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

DSAI5207 Modern Deep Learning

Linear Model

Xingyi Yang
Hong Kong Polytechnic University

Overview: Building Blocks of Deep Learning

The Foundation

At the heart of Deep Learning lies the **Linear Model**.

$$y = f(\mathbf{w}^\top \mathbf{x} + b)$$

Deep Learning is largely about stacking these linear units with non-linearities.

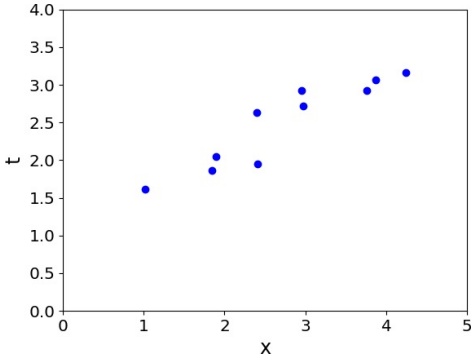
- Here, we will review linear models, some other fundamental concepts (e.g. gradient descent, generalization), and some of the common supervised learning problems:
  - **Regression**: predict a scalar-valued target (e.g. stock price)
  - **Binary classification**: predict a binary label (e.g. spam vs. non-spam email)
  - **Multiclass classification**: predict a discrete label (e.g. object category, from a list)

Outline

- **Linear Regression**
- **Binary Classification**
- **Multiclass Classification**

Contents

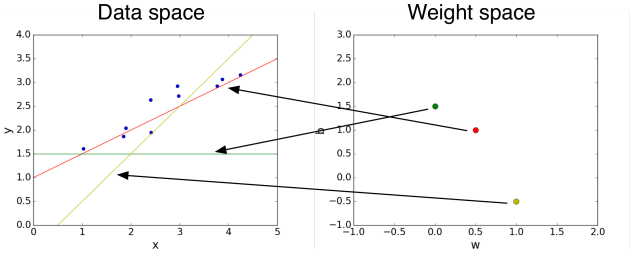
Problem Setup



Goal: Predict scalar t from input vector $\mathbf{x}$.

- **Dataset:** $\mathcal{D} = \{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^{N}$
- **Input:** $\mathbf{x}^{(i)} \in \mathbb{R}^D$
- **Target:** $t^{(i)} \in \mathbb{R}$

The Linear Model



Assumption

The relationship is linear:

$$y = \mathbf{w}^\top \mathbf{x} + b$$

- y: **Prediction**
- $\mathbf{w}$: **Weights** (slope)
- b: **Bias** (intercept)

Learning $\rightarrow$ *Searching for the best* $\mathbf{w}, b$

Objective: Loss and Cost

How do we measure "success"?

1. Loss Function (Single Example)

Measure error on one data point using a squared loss:

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

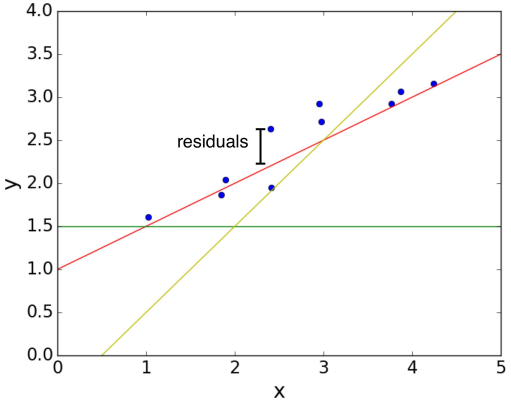
- $(y - t)$ is the **residual**.
- The $\frac{1}{2}$ makes the derivative cleaner later.

2. Cost Function (Entire Dataset)

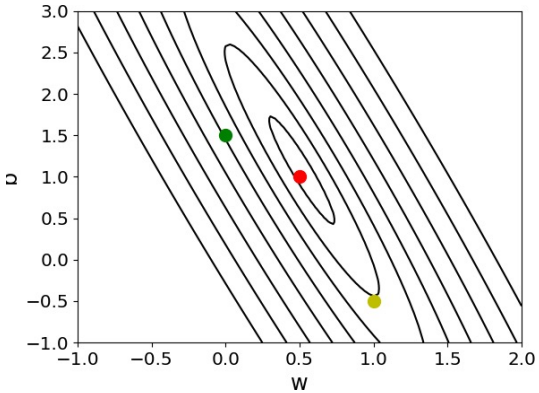
Average the loss over all N examples:

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y^{(i)}, t^{(i)}) = \frac{1}{2N}\sum_{i=1}^{N}(\mathbf{w}^\top \mathbf{x}^{(i)} + b - t^{(i)})^2$$

Visualizing the Cost Surface



Minimizing vertical residuals



Convex “Bowl” Shape

Vectorization

Instead of loops, we use linear algebra.

- $\mathbf{X} \in \mathbb{R}^{N \times D}$: Design Matrix
- $\mathbf{t} \in \mathbb{R}^{N}$: Target Vector

one feature across all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training example (vector)

Batch Prediction

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^\top \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{(N)} + b \end{pmatrix} \in \mathbb{R}^N$$

Vectorization: Math vs. Code

Mathematical Notation

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b$$

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2$$

Python (NumPy) Implementation

```
# Prediction
y = X @ w + b

# MSE Loss
loss = np.sum((y - t)**2) / (2 * N)
```

Solving the Optimization Problem

We want to find $\mathbf{w}^*, b^*$ that minimize $\mathcal{J}$.

Calculus View

The minimum occurs at a **critical point** where $\nabla \mathcal{J} = 0$.

Two Strategies:

1. **Direct Solution (Closed-form):**
   - Solve the system of linear equations directly.
   - Only possible for simple models (like Linear Regression).
2. **Iterative Method (Gradient Descent):**
   - Start random, take steps downhill.
   - Universal method for Deep Learning.

Direct Solution

- Goal: minimize the mean squared error

$$\mathcal{J}(\mathbf{w}, b) = \frac{1}{2N} \|\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t}\|^2$$

- For smooth convex problems, the optimum satisfies:

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \mathbf{0}, \quad \frac{\partial \mathcal{J}}{\partial b} = 0$$

- We compute these derivatives using:
  - **Partial derivatives** (w.r.t. one parameter at a time)
  - **Chain rule** (from loss $\mathcal{L}$ through prediction y)

Direct Solution: Gradients via Chain Rule

- Per example: $\mathcal{L}^{(i)} = \frac{1}{2}(y^{(i)} - t^{(i)})^2, \quad y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$
- Derivatives via **Chain rule**:

$$\frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{w}} = \underbrace{\frac{d\mathcal{L}^{(i)}}{dy^{(i)}}}_{=y^{(i)}-t^{(i)}} \cdot \underbrace{\frac{\partial y^{(i)}}{\partial \mathbf{w}}}_{=\mathbf{x}^{(i)}} = (y^{(i)} - t^{(i)})\, \mathbf{x}^{(i)},$$

$$\frac{\partial \mathcal{L}^{(i)}}{\partial b} = \underbrace{\frac{d\mathcal{L}^{(i)}}{dy^{(i)}}}_{=y^{(i)}-t^{(i)}} \cdot \underbrace{\frac{\partial y^{(i)}}{\partial b}}_{=1} = y^{(i)} - t^{(i)}.$$

- Average over dataset ($\mathcal{J} = \frac{1}{N}\sum_i \mathcal{L}^{(i)}$):

$$\frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)}-t^{(i)})\mathbf{x}^{(i)} = \frac{1}{N}\mathbf{X}^\top(\mathbf{y}-\mathbf{t}), \frac{\partial \mathcal{J}}{\partial b} = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)}-t^{(i)}) = \frac{1}{N}\mathbf{1}^\top(\mathbf{y}-\mathbf{t})$$

Direct Solution: Closed-Form Solution

- Set gradients to zero:

$$\mathbf{X}^\top(\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t}) = \mathbf{0}, \quad \mathbf{1}^\top(\mathbf{X}\mathbf{w} + b\mathbf{1} - \mathbf{t}) = 0$$

- Instead of solving two equations, **augment** the system, so that $\mathbf{y} = \tilde{\mathbf{X}}\tilde{\mathbf{w}}$:

$$\tilde{\mathbf{X}} = \begin{bmatrix}\mathbf{X} & \mathbf{1}\end{bmatrix} \in \mathbb{R}^{N\times(d+1)}, \quad \tilde{\mathbf{w}} = \begin{bmatrix}\mathbf{w} \\ b\end{bmatrix}$$

- The optimal parameters satisfy the **normal equation**:

$$\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}\,\tilde{\mathbf{w}} = \tilde{\mathbf{X}}^\top\mathbf{t}$$

- If $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}$ is invertible:

$$\boxed{\tilde{\mathbf{w}}^* = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^\top\mathbf{t}}$$

Pros: Exact solution. No hyperparameters.
Cons: Matrix inversion is $O(D^3)$. Slow for large D.

Gradient Descent

Concept

Iteratively adjust weights in the **direction of steepest descent** (negative gradient).

- **Initialize** weights (e.g., zeros or random).
- **Update** repeatedly:

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$$

$$w_j \leftarrow w_j - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) x_j^{(i)}$$

- α: **Learning Rate**
- Controls step size.

Gradient descent

- This gets its name from the **gradient**:

$$\nabla \mathcal{J}(\mathbf{w}) = \frac{\partial \mathcal{J}}{\partial \mathbf{w}} = \begin{pmatrix} \frac{\partial \mathcal{J}}{\partial w_1} \\ \vdots \\ \frac{\partial \mathcal{J}}{\partial w_D} \end{pmatrix}$$

- This is the direction of fastest increase in $\mathcal{J}$.
- Update rule in vector form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \mathcal{J}(\mathbf{w}) = \mathbf{w} - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)}) \mathbf{x}^{(i)}$$

- Hence, gradient descent updates the weights in the direction of fastest decrease.

Why Gradient Descent?

If we have the Direct Solution, why use GD?

Scalability
Inverting large matrices is computationally impossible for high-dim data.

Generality
GD works for non-convex, non-linear problems (like Neural Networks).

Implementation
Easy to implement with AutoDiff (PyTorch/TensorFlow).

Feature Mapping

Problem

What if the target is not a linear function of the input?

Solution: Create nonlinear features!

$$y = \mathbf{w}^\top \boldsymbol{\psi}(\mathbf{x})$$

Polynomial Regression Example

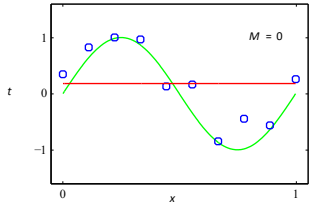
If $\boldsymbol{\psi}(x) = (1, x, x^2, \ldots, x^D)^\top$, then y is a polynomial.

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_D x^D$$

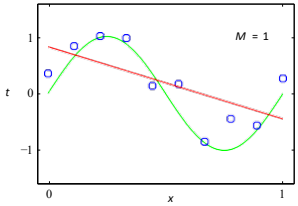
- The model is still linear in $\mathbf{w}$ (Linear Regression still works!).
- Hard part: Choosing the right ψ.

Feature maps

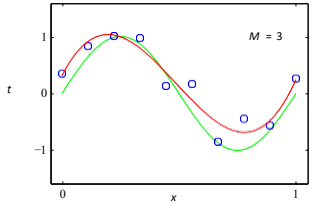
$y = w_0$



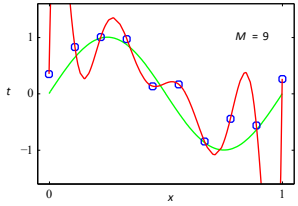
$y = w_0 + w_1 x$



$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$

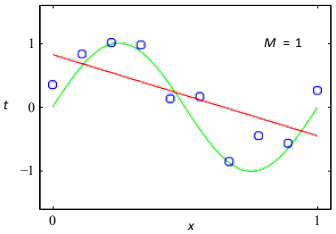


$y = w_0 + w_1 x + \cdots + w_9 x^9$

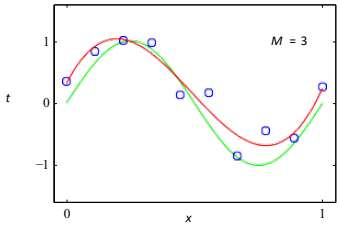


Pattern Recognition and Machine Learning, Christopher Bishop.

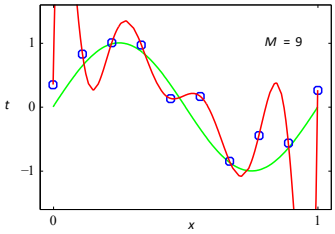
Model Complexity & Generalization



$M = 1$: **Under-fitting**
Model is too simple.
Does not fit well.
Large test error.



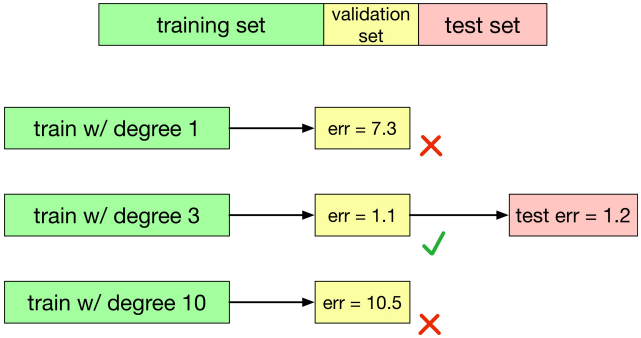
$M = 3$: Good model
Small test error.
Generalizes well.
Small test error.



$M = 9$: **Over-fitting**
Model is too complex.
Perfect Fit.
Large test error.

Generalization

- We would like our models to **generalize** to data they haven't seen before
- The degree of the polynomial is an example of a **hyperparameter**, something we can't include in the training procedure itself
- We can tune hyperparameters using a **validation set**:



Contents

Binary Linear Classification

The Task

- **Goal:** Predict a binary target $t \in \{0, 1\}$ based on input $\mathbf{x}$.
- **Examples:**
  - $t = 1$: **Positive class** (e.g., Spam, Fraud, Disease).
  - $t = 0$: **Negative class** (e.g., Not Spam, Normal, Healthy).

Approach 1: The Linear Threshold Model (Perceptron)

Compute a linear score (logit) z, then apply a hard threshold:

$$z = \mathbf{w}^\top \mathbf{x} + b$$

$$\text{prediction} = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Why not optimize accuracy directly?

The Problem: Vanishing Gradients

We cannot use Gradient Descent on the hard threshold function.

- The step function is **discontinuous** at $z = 0$.
- The gradient is **zero** everywhere else ($\frac{\partial y}{\partial z} = 0$).
- **Consequence:** The weights never update because the model doesn't know which "direction" to move to improve.

The Solution: Soft Classification

Instead of a hard classification, we predict a continuous probability:

$$y \approx P(t = 1 \mid \mathbf{x})$$

This allows us to use a **smooth, differentiable** loss function.

The Logistic (Sigmoid) Function

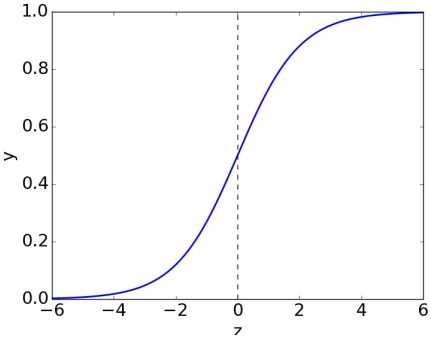
- We need to "squash" the linear score $z \in (-\infty, \infty)$ into a valid probability range $y \in [0, 1]$.

Definition (Sigmoid Activation)

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

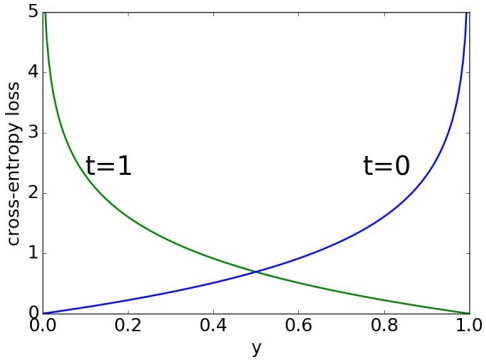
- **Log-Linear Model:**

$$z = \mathbf{w}^\top \mathbf{x} + b, \quad y = \sigma(z)$$



Interpretation:
Large positive $z \to 1$
Large negative $z \to 0$

Loss Function: Binary Cross-Entropy



- We interpret y as the estimated probability $P(t = 1)$.

Intuition

Being **99% wrong** costs drastically more than being **90% wrong**.

Binary Cross-Entropy (BCE)

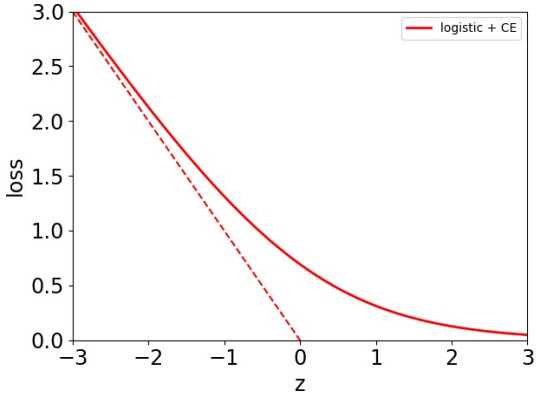
$$\mathcal{L}_{CE} = -t \log(y) - (1 - t) \log(1 - y)$$

- **If** $t = 1$**:** Minimize $-\log(y)$ (Push $y \to 1$)
- **If** $t = 0$**:** Minimize $-\log(1 - y)$ (Push $y \to 0$)

Logistic Regression: Summary

The Full Pipeline:

$$\underbrace{\mathbf{x} \xrightarrow{\mathbf{w},b} z}_{\text{Linear Score}} \xrightarrow{\sigma} \underbrace{y}_{\text{Probability}} \xrightarrow{\mathcal{L}_{CE}} \underbrace{\text{Loss}}_{\text{Scalar Error}}$$



Why this combination?

- When we combine **Sigmoid + BCE**, the gradients cancel out nicely.
- The loss becomes roughly **linear** with respect to z when the prediction is wrong.
- This guarantees a strong error signal (no "saturation").

Gradient Derivation: The Error Signal

- To train parameters $\mathbf{w}, b$, we need the derivative $\frac{\partial \mathcal{L}}{\partial z}$.
- **Chain Rule:** $\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial z}$

1. Loss Gradient

Derivative of BCE w.r.t y:

$$\frac{\partial \mathcal{L}}{\partial y} = \frac{y - t}{y(1 - y)}$$

2. Sigmoid Gradient

Derivative of Sigmoid w.r.t z:

$$\sigma'(z) = y(1 - y)$$

The Beautiful Result

Multiplying them cancels the denominator:

$$\frac{\partial \mathcal{L}}{\partial z} = \left(\frac{y - t}{y(1 - y)} \right) \cdot y(1 - y) = \boxed{y - t}$$

Just like in Linear Regression, the gradient is the **prediction error**!

Parameter Updates

We backpropagate the error signal $\delta = (y - t)$ to update the weights.

Bias Gradient

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = y - t$$

Weight Gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial \mathbf{w}} = (y - t)\mathbf{x}$$

Gradient Descent Step (η = learning rate):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(y - t)\mathbf{x}$$

- **Intuition:** If we predict $y \approx 1$ but target is $t = 0$, the term $(y - t)$ is positive. We **subtract** $\mathbf{x}$ from $\mathbf{w}$ to lower the score z.

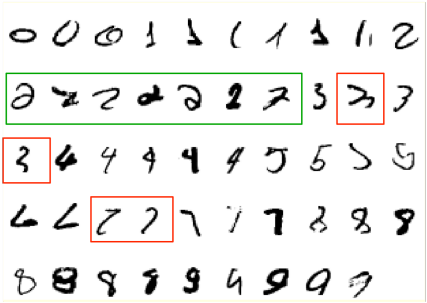
Contents

Multiclass Classification: The Problem

- **Context:** Binary classification handles two options ($K = 2$).
- **Question:** How do we handle tasks with $K > 2$ categories?

Examples:

- Handwritten digits (0-9) $\rightarrow K = 10$
- ImageNet objects $\rightarrow K = 1000$
- Medical diagnosis $\rightarrow K = 3$
  (Healthy, Benign, Malignant)



Representation: One-Hot Encoding

Encoding Targets

- **Raw Data:** Discrete labels $t \in \{1, \dots, K\}$.
- **Model Input:** We strictly use **one-hot vectors**.

- A target vector $\mathbf{t}$ has dimension K:

$$\mathbf{t} = [0, 0, \dots, \mathbf{1}, \dots, 0]^\top$$

  - The k-th entry is 1 if the sample belongs to class k.
  - All other entries are 0.

- **Rationale:** This treats all classes as independent. Using a scalar (e.g., class 2 ¿ class 1) implies an order that usually doesn't exist.

The Linear Model (Logits)

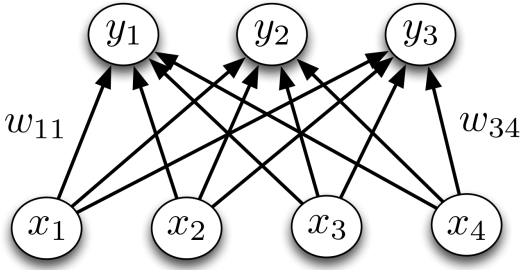
- **Input:** $\mathbf{x} \in \mathbb{R}^D$ (D features).
- **Output:** We need a score for *each* of the K classes.

Matrix Formulation

We stack K linear classifiers:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- **Weights:** $\mathbf{W} \in \mathbb{R}^{K \times D}$
- **Biases:** $\mathbf{b} \in \mathbb{R}^K$
- **Logits:** $\mathbf{z} \in \mathbb{R}^K$ (raw scores)



Single Layer Network

Activation: The Softmax Function

- **Problem:** Logits $\mathbf{z}$ are unbounded $(-\infty, \infty)$. We need a probability distribution.
- **Solution:** The **Softmax function**:

$$y_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Key Properties

- **Positivity:** $e^{z_k} > 0$, so outputs are always positive.
- **Normalization:** $\sum y_k = 1$. It creates a valid distribution.
- **Soft-Argmax:** It amplifies the largest z_k and suppresses smaller ones (winner-takes-all behavior).

Loss Function: Categorical Cross-Entropy

- We compare the predicted distribution $\mathbf{y}$ with the true distribution $\mathbf{t}$.
- **General Formula:**

$$\mathcal{L}_{\text{CE}}(\mathbf{y}, \mathbf{t}) = -\sum_{k=1}^{K} t_k \log y_k$$

Simplification with One-Hot Targets

Since $\mathbf{t}$ has only one non-zero entry (at the correct class c):

$$\mathcal{L}_{\text{CE}} = -\log(y_c)$$

- **Intuition:** We only care about the probability assigned to the correct class. Maximizing y_c (up to 1) minimizes $-\log(y_c)$ (down to 0).

Gradients: The Error Signal

- To train $\mathbf{W}$ and $\mathbf{b}$, we need the gradients.
- **The Chain Rule:**

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \cdot \frac{\partial \mathbf{y}}{\partial \mathbf{z}}$$

- **The Beautiful Result:** For Softmax + Cross-Entropy, the gradient simplifies to:

$$\boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \mathbf{y} - \mathbf{t}}$$

- **Interpretation:** This is the raw **prediction error**.
- If $y \approx t$, the gradient is 0 (no update).
- If y is far from t, the gradient is large.

Final Update Rules

Backpropagating the error signal $(\mathbf{y} - \mathbf{t})$ to the parameters:

Bias Gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{y} - \mathbf{t}$$

Weight Gradient

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = (\mathbf{y} - \mathbf{t})\mathbf{x}^\top$$

Dimensionality Check:

- Error $(\mathbf{y} - \mathbf{t})$ is $K \times 1$.
- Input $\mathbf{x}^\top$ is $1 \times D$.
- Resulting Gradient is $K \times D$ (Same shape as $\mathbf{W}$).