Mathematical Appendix

Complete Derivations for Neural Networks

Neural Networks for Finance

BSc Lecture Series

November 26, 2025

Appendix Contents

Full mathematical derivations for all four modules

A.1 Perceptron Convergence Theorem

Theorem (Novikoff, 1962)

If the training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ is **linearly separable**, the perceptron learning algorithm converges in a finite number of updates.

What This Means:

- The algorithm will find a separating hyperplane
- It will stop updating weights (no more mistakes)
- Convergence is **guaranteed**, not probabilistic

What It Doesn't Say:

- Nothing about non-separable data
- Doesn't guarantee the "best" hyperplane
- Number of steps can be large

Referenced in Module 1

A.2 Setup and Assumptions

Data: Training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ where $y_i \in \{-1, +1\}$

Assumption 1: Linear Separability
There exists $\mathbf{w}^* \in \mathbb{R}^d$ with $\|\mathbf{w}^*\| = 1$ and margin $\gamma > 0$ such that:

$$y_i(\mathbf{w}^{*T}\mathbf{x}_i) \geq \gamma \quad \text{for all } i = 1, \ldots, n$$

Assumption 2: Bounded Data
All data points have bounded norm:

$$\|\mathbf{x}_i\| \leq R \quad \text{for all } i = 1, \ldots, n$$

Key Quantities:

- γ: The "margin" - minimum distance from any point to the hyperplane
- R: Maximum radius - the data lies in a ball of radius R

Defining the margin and bounded data

A.3 Proof: Step 1 - Lower Bound on $\mathbf{w}^{*T}\mathbf{w}^{(t)}$

Goal: Show the inner product $\mathbf{w}^{*T}\mathbf{w}^{(t)}$ grows with each mistake.

Update Rule: When mistake on $(\mathbf{x}_i, y_i)$: $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + y_i\mathbf{x}_i$

Computing the Inner Product:

$$\begin{aligned}
\mathbf{w}^{*T}\mathbf{w}^{(t+1)} &= \mathbf{w}^{*T}(\mathbf{w}^{(t)} + y_i\mathbf{x}_i) \\
&= \mathbf{w}^{*T}\mathbf{w}^{(t)} + y_i(\mathbf{w}^{*T}\mathbf{x}_i) \\
&\geq \mathbf{w}^{*T}\mathbf{w}^{(t)} + \gamma \quad \text{(by separability assumption)}
\end{aligned}$$

After t mistakes (starting from $\mathbf{w}^{(0)} = \mathbf{0}$):

$$\mathbf{w}^{*T}\mathbf{w}^{(t)} \geq t\gamma$$

Showing the inner product grows

A.4 Proof: Step 2 - Upper Bound on $\|\mathbf{w}^{(t)}\|^2$

Goal: Show the squared norm $\|\mathbf{w}^{(t)}\|^2$ grows slowly.

Computing the Squared Norm:

$$\begin{aligned}
\|\mathbf{w}^{(t+1)}\|^2 &= \|\mathbf{w}^{(t)} + y_i \mathbf{x}_i\|^2 \\
&= \|\mathbf{w}^{(t)}\|^2 + 2y_i(\mathbf{w}^{(t)T}\mathbf{x}_i) + \|\mathbf{x}_i\|^2 \\
&\leq \|\mathbf{w}^{(t)}\|^2 + 0 + R^2 \quad (\text{mistake means } y_i(\mathbf{w}^{(t)T}\mathbf{x}_i) \leq 0) \\
&\leq \|\mathbf{w}^{(t)}\|^2 + R^2
\end{aligned}$$

After t mistakes (starting from $\mathbf{w}^{(0)} = \mathbf{0}$):

$$\|\mathbf{w}^{(t)}\|^2 \leq tR^2$$

Showing the norm is bounded

Combining the Two Bounds:

From Step 1: $\mathbf{w}^{*T}\mathbf{w}^{(t)} \geq t\gamma$
From Step 2: $\|\mathbf{w}^{(t)}\|^2 \leq tR^2$, so $\|\mathbf{w}^{(t)}\| \leq \sqrt{t}R$

Using Cauchy-Schwarz:

$$t\gamma \leq \mathbf{w}^{*T}\mathbf{w}^{(t)} \leq \|\mathbf{w}^*\|\|\mathbf{w}^{(t)}\| = 1 \cdot \|\mathbf{w}^{(t)}\| \leq \sqrt{t}R$$

Solving for t:

$$t\gamma \leq \sqrt{t}R \implies \sqrt{t} \leq \frac{R}{\gamma} \implies t \leq \frac{R^2}{\gamma^2}$$

Convergence Bound

$$\text{Number of mistakes} \leq \left(\frac{R}{\gamma}\right)^2$$

Maximum number of mistakes: $(R/\gamma)^2$

A.6 Geometric Interpretation

What the Bound $(R/\gamma)^2$ Tells Us:

- Large margin $\gamma \to$ fewer mistakes
- Larger data spread $R \to$ more mistakes
- Ratio matters, not absolute values

Intuition:

- $\gamma =$ "wiggle room" for the hyperplane
- $R =$ how much ground to cover
- Easy problem: large γ, small R
- Hard problem: small γ, large R

`charts/perceptron_convergence_proof/perceptr`

Limitation: If $\gamma = 0$ (not separable), bound is infinite $\to$ no convergence guarantee.

What the proof means geometrically

B.1 Backpropagation: Overview

Goal: Compute $\frac{\partial \mathcal{L}}{\partial W_{jk}^{(l)}}$ and $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$ for all layers l.

The Challenge:

- Loss depends on weights through many intermediate layers
- Naïve approach: $O(n^2)$ operations per parameter
- Backprop achieves: $O(n)$ total (same as forward pass!)

Key Insight: Reuse intermediate computations via the chain rule.

What We Will Derive:

1. Error signal $\boldsymbol{\delta}^{(L)}$ at output layer
2. Recursion formula for $\boldsymbol{\delta}^{(l)}$ at hidden layers
3. Weight gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$
4. Bias gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$

Referenced in Module 3

B.2 Notation

Network with L layers:

- $\mathbf{h}^{(0)} = \mathbf{x}$: Input (layer 0)
- $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$: Pre-activation (layer l)
- $\mathbf{h}^{(l)} = \phi(\mathbf{z}^{(l)})$: Activation (layer l)
- $\hat{\mathbf{y}} = \mathbf{h}^{(L)}$: Output (layer L)

Dimensions:

- $\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$: Weight matrix
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}$: Bias vector
- n_l: Number of neurons in layer l

Error Signal (key quantity):

$$\boldsymbol{\delta}^{(l)} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \in \mathbb{R}^{n_l}$$

Consistent notation for the derivation

B.3 Chain Rule Review

Univariate Chain Rule:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Multivariate Chain Rule:
If L depends on $z_1, \ldots, z_n$, each depending on w:

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{n} \frac{\partial L}{\partial z_i} \cdot \frac{\partial z_i}{\partial w}$$

Matrix Form:
If $\mathbf{z} = f(\mathbf{h})$ and $L = L(\mathbf{z})$:

$$\frac{\partial L}{\partial \mathbf{h}} = \left(\frac{\partial \mathbf{z}}{\partial \mathbf{h}}\right)^T \frac{\partial L}{\partial \mathbf{z}}$$

Key for Backprop: The Jacobian $\frac{\partial \mathbf{z}}{\partial \mathbf{h}}$ connects layers.

The mathematical foundation of backpropagation

B.4 Forward Pass Equations

Layer-by-Layer Computation:

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

$$\text{Pre-activation:} \quad \mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\text{Activation:} \quad \mathbf{h}^{(l)} = \phi^{(l)}(\mathbf{z}^{(l)})$$

Element-wise:

$$z_j^{(l)} = \sum_{k=1}^{n_{l-1}} W_{jk}^{(l)} h_k^{(l-1)} + b_j^{(l)}$$

$$h_j^{(l)} = \phi(z_j^{(l)})$$

Final Output: $\hat{\mathbf{y}} = \mathbf{h}^{(L)}$
Loss: $\mathcal{L} = \mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$

Computing outputs

B.5 Error Signal Definition

Definition: The error signal at layer l is:

$$\boldsymbol{\delta}^{(l)} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}}$$

Why This Quantity?

- It captures "how much does the loss change if $\mathbf{z}^{(l)}$ changes"
- All weight gradients can be expressed in terms of $\boldsymbol{\delta}^{(l)}$
- Can be computed recursively from layer to layer

Strategy:

1. Compute $\boldsymbol{\delta}^{(L)}$ at output layer (depends on loss function)
2. Propagate backward: $\boldsymbol{\delta}^{(l)} = f(\boldsymbol{\delta}^{(l+1)})$
3. Compute weight gradients: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = g(\boldsymbol{\delta}^{(l)})$

The key quantity for backpropagation

B.6 Output Layer Error Derivation

At the output layer L:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}}$$

Using the Chain Rule:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = \sum_k \frac{\partial \mathcal{L}}{\partial h_k^{(L)}} \cdot \frac{\partial h_k^{(L)}}{\partial z_j^{(L)}}$$

For element-wise activation:

$$\frac{\partial h_k^{(L)}}{\partial z_j^{(L)}} = \phi'(z_j^{(L)}) \cdot \mathbf{1}_{j=k}$$

Result:

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial h_j^{(L)}} \cdot \phi'(z_j^{(L)})$$

In vector form: $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{h}^{(L)}} \mathcal{L} \odot \phi'(\mathbf{z}^{(L)})$

Starting point for backward pass

B.7 Special Case: MSE + Sigmoid

Setup:

- Loss: $\mathcal{L} = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \frac{1}{2}\sum_j (h_j^{(L)} - y_j)^2$
- Activation: $\phi(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
- Derivative: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$

Gradient of Loss:

$$\frac{\partial \mathcal{L}}{\partial h_j^{(L)}} = h_j^{(L)} - y_j = \hat{y}_j - y_j$$

Output Layer Error:

$$\begin{aligned}\delta_j^{(L)} &= (h_j^{(L)} - y_j) \cdot \sigma(z_j^{(L)})(1 - \sigma(z_j^{(L)})) \\ &= (\hat{y}_j - y_j) \cdot \hat{y}_j(1 - \hat{y}_j)\end{aligned}$$

Vector form: $\boldsymbol{\delta}^{(L)} = (\hat{\mathbf{y}} - \mathbf{y}) \odot \hat{\mathbf{y}} \odot (1 - \hat{\mathbf{y}})$

Complete derivation for common case

B.8 Special Case: Cross-Entropy + Softmax

Setup:

- Loss: $\mathcal{L} = -\sum_j y_j \log(\hat{y}_j)$ (one-hot $\mathbf{y}$)
- Activation: $\hat{y}_j = \text{softmax}(z_j^{(L)}) = \frac{e^{z_j^{(L)}}}{\sum_k e^{z_k^{(L)}}}$

Combined Derivative (remarkably simple):

$$\delta_j^{(L)} = \frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = \hat{y}_j - y_j$$

Derivation Sketch:

$$\frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = -\sum_k y_k \frac{\partial \log \hat{y}_k}{\partial z_j^{(L)}} = -\sum_k y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial z_j^{(L)}}$$
$$= -y_j(1 - \hat{y}_j) + \sum_{k \neq j} y_k \hat{y}_j = \hat{y}_j - y_j$$

Vector form: $\boldsymbol{\delta}^{(L)} = \hat{\mathbf{y}} - \mathbf{y}$

Complete derivation for classification - elegantly simple!

B.9 Hidden Layer Error: Setup

Goal: Express $\boldsymbol{\delta}^{(l)}$ in terms of $\boldsymbol{\delta}^{(l+1)}$

The Computational Graph:

$$\mathbf{z}^{(l)} \to \mathbf{h}^{(l)} \to \mathbf{z}^{(l+1)} \to \mathbf{h}^{(l+1)} \to \cdots \to \mathcal{L}$$

Chain Rule:

$$\delta_j^{(l)} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$$

Breaking It Down:

1. $\frac{\partial \mathcal{L}}{\partial z_k^{(l+1)}} = \delta_k^{(l+1)}$ (already computed)
2. $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$ needs to be derived

Applying the chain rule through layers

Computing $\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}$:

Recall:

$$z_k^{(l+1)} = \sum_m W_{km}^{(l+1)} h_m^{(l)} + b_k^{(l+1)}$$

Chain Rule:

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_m W_{km}^{(l+1)} \frac{\partial h_m^{(l)}}{\partial z_j^{(l)}}$$

For element-wise activation:

$$\frac{\partial h_m^{(l)}}{\partial z_j^{(l)}} = \phi'(z_j^{(l)}) \cdot \mathbf{1}_{m=j}$$

Therefore:

$$\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = W_{kj}^{(l+1)} \cdot \phi'(z_j^{(l)})$$

First component of the chain

B.11 Hidden Layer Error: Step 2

Substituting Back:

$$\begin{aligned}
\delta_j^{(l)} &= \sum_k \delta_k^{(l+1)} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \\
&= \sum_k \delta_k^{(l+1)} \cdot W_{kj}^{(l+1)} \cdot \phi'(z_j^{(l)}) \\
&= \phi'(z_j^{(l)}) \cdot \sum_k W_{kj}^{(l+1)} \delta_k^{(l+1)} \\
&= \phi'(z_j^{(l)}) \cdot [\mathbf{W}^{(l+1)T} \boldsymbol{\delta}^{(l+1)}]_j
\end{aligned}$$

Interpretation:

- $\mathbf{W}^{(l+1)T} \boldsymbol{\delta}^{(l+1)}$: Error from next layer, weighted by connection strengths
- $\phi'(z_j^{(l)})$: Scaled by local gradient of activation

Second component of the chain

Common Activation Functions and Their Derivatives:

Activation	$\phi(z)$	$\phi'(z)$
Sigmoid	$\frac{1}{1+e^{-z}}$	$\phi(z)(1-\phi(z))$
Tanh	$\frac{e^z-e^{-z}}{e^z+e^{-z}}$	$1-\phi(z)^2$
ReLU	$\max(0, z)$	$\mathbf{1}_{z>0}$
Leaky ReLU	$\max(\alpha z, z)$	$\alpha\mathbf{1}_{z\leq 0}+\mathbf{1}_{z>0}$

Note: ReLU derivative is 0 for $z \leq 0$, 1 for $z > 0$
This can cause "dead neurons" if z is always negative.

Third component of the chain

Backpropagation Recursion

$$\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)}) \odot \phi'(\mathbf{z}^{(l)})$$

Element-wise:

$$\delta_j^{(l)} = \phi'(z_j^{(l)}) \cdot \sum_k W_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Interpretation:

- **Matrix multiply:** $\mathbf{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)}$ - "pull back" error through weights
- **Element-wise multiply:** $\odot\phi'(\mathbf{z}^{(l)})$ - scale by local gradient

Why "Back" Propagation?

- Errors flow backward: $\boldsymbol{\delta}^{(L)} \to \boldsymbol{\delta}^{(L-1)} \to \cdots \to \boldsymbol{\delta}^{(1)}$
- Uses transpose of forward weights $\mathbf{W}^T$

The backpropagation recursion

B.14 Weight Gradient: Setup

Goal: Compute $\frac{\partial \mathcal{L}}{\partial W_{jk}^{(l)}}$

How $W_{jk}^{(l)}$ Affects the Loss:

$$W_{jk}^{(l)} \to z_j^{(l)} \to h_j^{(l)} \to \cdots \to \mathcal{L}$$

Chain Rule:

$$\frac{\partial \mathcal{L}}{\partial W_{jk}^{(l)}} = \frac{\partial \mathcal{L}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial W_{jk}^{(l)}}$$

We Already Know:

$$\frac{\partial \mathcal{L}}{\partial z_j^{(l)}} = \delta_j^{(l)}$$

Need to Compute:

$$\frac{\partial z_j^{(l)}}{\partial W_{jk}^{(l)}} = ?$$

Computing the gradient with respect to weights

B.15 Weight Gradient: Derivation

Recall:

$$z_j^{(l)} = \sum_m W_{jm}^{(l)} h_m^{(l-1)} + b_j^{(l)}$$

Partial Derivative:

$$\frac{\partial z_j^{(l)}}{\partial W_{jk}^{(l)}} = h_k^{(l-1)}$$

Therefore:

$$\frac{\partial \mathcal{L}}{\partial W_{jk}^{(l)}} = \delta_j^{(l)} \cdot h_k^{(l-1)}$$

Interpretation:

- Error at neuron j ($\delta_j^{(l)}$)
- Times activation of input k ($h_k^{(l-1)}$)
- "Blame" is proportional to both

Applying the chain rule to weights

B.16 Weight Gradient: Complete Formula

Weight Gradient Formula

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\mathbf{h}^{(l-1)})^T$$

Dimensions:

- $\boldsymbol{\delta}^{(l)} \in \mathbb{R}^{n_l}$ (column vector)
- $\mathbf{h}^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ (column vector)
- $\boldsymbol{\delta}^{(l)}(\mathbf{h}^{(l-1)})^T \in \mathbb{R}^{n_l \times n_{l-1}}$ (outer product = matrix)

Element-wise:

$$\left[\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}\right]_{jk} = \delta_j^{(l)} h_k^{(l-1)}$$

Intuition: The gradient is an outer product of error and activation.

The weight update formula

Goal: Compute $\frac{\partial \mathcal{L}}{\partial b_j^{(l)}}$

Recall:

$$z_j^{(l)} = \sum_m W_{jm}^{(l)} h_m^{(l-1)} + b_j^{(l)}$$

Partial Derivative:

$$\frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = 1$$

Therefore:

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)} \cdot 1 = \delta_j^{(l)}$$

Bias Gradient Formula

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$$

Note: The bias gradient is simply the error signal itself!

Simpler than the weight gradient

B.18 Complete Backpropagation Algorithm

Input: Training example $(\mathbf{x}, \mathbf{y})$, network weights $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}$

Forward Pass:

1. Set $\mathbf{h}^{(0)} = \mathbf{x}$
2. For $l = 1$ to L:
   - $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
   - $\mathbf{h}^{(l)} = \phi^{(l)}(\mathbf{z}^{(l)})$
3. Compute loss $\mathcal{L}(\mathbf{h}^{(L)}, \mathbf{y})$

Backward Pass:

1. Compute $\boldsymbol{\delta}^{(L)} = \nabla_{\mathbf{h}^{(L)}}\mathcal{L} \odot \phi'^{(L)}(\mathbf{z}^{(L)})$
2. For $l = L - 1$ to 1:
   - $\boldsymbol{\delta}^{(l)} = (\mathbf{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)}) \odot \phi'^{(l)}(\mathbf{z}^{(l)})$

Gradients: $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)}(\mathbf{h}^{(l-1)})^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$

The full algorithm

B.19 Worked Example: 2-2-1 Network

Network: 2 inputs, 2 hidden neurons (sigmoid), 1 output (sigmoid), MSE loss

Given: $\mathbf{x} = [1, 0.5]^T$, $y = 1$, $\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$, $\mathbf{W}^{(2)} = [0.5, 0.6]$

Forward Pass:

$$
\begin{aligned}
\mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} = [0.2, 0.5]^T \\
\mathbf{h}^{(1)} &= \sigma(\mathbf{z}^{(1)}) = [0.550, 0.622]^T \\
z^{(2)} &= \mathbf{W}^{(2)}\mathbf{h}^{(1)} = 0.648 \\
\hat{y} &= \sigma(z^{(2)}) = 0.656
\end{aligned}
$$

Backward Pass:

$$
\begin{aligned}
\delta^{(2)} &= (\hat{y} - y) \cdot \hat{y}(1 - \hat{y}) = -0.078 \\
\boldsymbol{\delta}^{(1)} &= (\mathbf{W}^{(2)T}\delta^{(2)}) \odot \mathbf{h}^{(1)} \odot (1 - \mathbf{h}^{(1)}) = [-0.010, -0.011]^T
\end{aligned}
$$

Following the numbers through the algorithm

B.20 Computational Complexity Analysis

For a Network with L Layers:

Forward Pass:

- Layer l: Matrix-vector multiply $\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}$
- Cost: $O(n_l \times n_{l-1})$ per layer
- Total: $O(\sum_l n_l n_{l-1}) = O(W)$ where W = total weights

Backward Pass:

- Layer l: Matrix-vector multiply $\mathbf{W}^{(l+1)T}\boldsymbol{\delta}^{(l+1)}$
- Same cost as forward: $O(n_{l+1} \times n_l)$ per layer
- Total: $O(W)$

Key Result:

Backpropagation Complexity

Computing all gradients: $O(W)$ = same as one forward pass!

Naïve finite differences would cost $O(W^2)$ - backprop is $W\times$ faster.

Why backpropagation is efficient

C.1 MSE from Maximum Likelihood

Setup: Regression problem where we model:

$$y = f_\theta(\mathbf{x}) + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

This Implies:

$$p(y|\mathbf{x}, \theta) = \mathcal{N}(y; f_\theta(\mathbf{x}), \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - f_\theta(\mathbf{x}))^2}{2\sigma^2}\right)$$

Log-Likelihood for Dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$:

$$\begin{aligned}\log p(\mathbf{y}|\mathbf{X}, \theta) &= \sum_{i=1}^n \log p(y_i|\mathbf{x}_i, \theta) \\ &= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^n (y_i - f_\theta(\mathbf{x}_i))^2\end{aligned}$$

Why MSE is the natural choice for regression

C.2 MSE Derivation

Maximum Likelihood Estimation:

$$\hat{\theta}_{ML} = \arg\max_{\theta} \log p(\mathbf{y}|\mathbf{X}, \theta)$$

Equivalently (dropping constants):

$$\hat{\theta}_{ML} = \arg\min_{\theta} \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2$$

This is Mean Squared Error (MSE):

$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Key Insight

MSE loss = Maximum likelihood under Gaussian noise assumption

The $\frac{1}{n}$ is for averaging; the $\frac{1}{2}$ often added is for cleaner gradients.

From Gaussian likelihood to squared error

C.3 Cross-Entropy from Maximum Likelihood

Setup: Binary classification where:

$$p(y = 1|\mathbf{x}, \theta) = f_\theta(\mathbf{x}) = \hat{y} \in [0, 1]$$

Bernoulli Distribution:

$$p(y|\mathbf{x}, \theta) = \hat{y}^y (1 - \hat{y})^{1-y}$$

Log-Likelihood:

$$\log p(y|\mathbf{x}, \theta) = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

Negative Log-Likelihood (what we minimize):

$$\mathcal{L} = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

This is the **Binary Cross-Entropy** loss!

Why cross-entropy is natural for classification

Loss Function:

$$\mathcal{L}_{BCE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i)]$$

Gradient with Respect to $\hat{y}$:

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

Combined with Sigmoid Output:
If $\hat{y} = \sigma(z)$, then $\frac{\partial \hat{y}}{\partial z} = \hat{y}(1-\hat{y})$

$$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \cdot \hat{y}(1-\hat{y}) = \hat{y} - y$$

Beautifully simple gradient!

The two-class case

C.5 Categorical Cross-Entropy Derivation

Setup: K-class classification with one-hot encoding $\mathbf{y} \in \{0, 1\}^K$

Categorical Distribution:

$$p(\mathbf{y}|\mathbf{x}, \theta) = \prod_{k=1}^{K} \hat{y}_k^{y_k}$$

Negative Log-Likelihood:

$$\mathcal{L}_{CE} = -\sum_{k=1}^{K} y_k \log \hat{y}_k$$

For Single Sample (one-hot y with $y_c = 1$):

$$\mathcal{L} = -\log \hat{y}_c$$

Interpretation: Minimize negative log of predicted probability for correct class.

The multi-class case

C.6 Softmax Function Derivation

Problem: Convert raw scores $\mathbf{z} \in \mathbb{R}^K$ to probabilities $\hat{\mathbf{y}} \in [0,1]^K$

Requirements:

- $\hat{y}_k \geq 0$ for all k
- $\sum_k \hat{y}_k = 1$
- Larger $z_k \rightarrow$ larger $\hat{y}_k$

Solution - Softmax:

$$\hat{y}_k = \text{softmax}(z_k) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

Properties:

- Exponential ensures positivity
- Normalization ensures sum to 1
- Invariant to adding constant: $\text{softmax}(z_k + c) = \text{softmax}(z_k)$
- In limit: softmax $\rightarrow \arg\max$ ("soft" version)

Converting scores to probabilities

C.7 Softmax Gradient

Computing $\frac{\partial \hat{y}_i}{\partial z_j}$:

Case 1: $i = j$

$$\frac{\partial \hat{y}_i}{\partial z_i} = \frac{e^{z_i} \cdot Z - e^{z_i} \cdot e^{z_i}}{Z^2} = \hat{y}_i - \hat{y}_i^2 = \hat{y}_i(1 - \hat{y}_i)$$

Case 2: $i \neq j$

$$\frac{\partial \hat{y}_i}{\partial z_j} = \frac{0 - e^{z_i} \cdot e^{z_j}}{Z^2} = -\hat{y}_i \hat{y}_j$$

Jacobian Matrix:

$$\frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} = \mathrm{diag}(\hat{\mathbf{y}}) - \hat{\mathbf{y}}\hat{\mathbf{y}}^T$$

Combined with Cross-Entropy: $\frac{\partial \mathcal{L}}{\partial z_j} = \hat{y}_j - y_j$ (same simple form!)

The derivative of softmax

D.1 L2 Regularization: Bayesian View

Bayesian Setup:

- Prior belief about weights: $p(\theta)$
- Likelihood of data given weights: $p(D|\theta)$
- Posterior: $p(\theta|D) \propto p(D|\theta)p(\theta)$

L2 Corresponds to Gaussian Prior:

$$p(\theta) = \mathcal{N}(\mathbf{0}, \sigma_\theta^2 \mathbf{I}) = \prod_i \frac{1}{\sqrt{2\pi\sigma_\theta^2}} \exp\left(-\frac{\theta_i^2}{2\sigma_\theta^2}\right)$$

Log Prior:

$$\log p(\theta) = -\frac{1}{2\sigma_\theta^2} \sum_i \theta_i^2 + \text{const} = -\frac{1}{2\sigma_\theta^2} \|\theta\|_2^2 + \text{const}$$

Interpretation: We believe weights should be small (centered at 0).

Referenced in Module 4

Maximum A Posteriori (MAP) Estimation:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} p(\theta|D) = \arg\max_{\theta} p(D|\theta)p(\theta)$$

Taking Logs:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} [\log p(D|\theta) + \log p(\theta)]$$

With Gaussian Prior and Gaussian Likelihood (MSE):

$$\hat{\theta}_{MAP} = \arg\min_{\theta} \left[\frac{1}{2\sigma^2} \sum_i (y_i - f_\theta(\mathbf{x}_i))^2 + \frac{1}{2\sigma_\theta^2} \|\theta\|_2^2 \right]$$

Defining $\lambda = \frac{\sigma^2}{\sigma_\theta^2}$:

$$\hat{\theta}_{MAP} = \arg\min_{\theta} \left[\mathcal{L}_{MSE} + \lambda \|\theta\|_2^2 \right]$$

This is exactly L2 regularization (weight decay)!

From Gaussian prior to weight decay

D.3 L1 Regularization: Bayesian View

L1 Corresponds to Laplace Prior:

$$p(\theta) = \prod_i \frac{1}{2b} \exp\left(-\frac{|\theta_i|}{b}\right)$$

Log Prior:

$$\log p(\theta) = -\frac{1}{b}\sum_i |\theta_i| + \text{const} = -\frac{1}{b}\|\theta\|_1 + \text{const}$$

Why Laplace Induces Sparsity:

- Laplace has sharp peak at 0
- Higher probability mass near exactly 0
- MAP estimate tends to push weights to exactly 0

Geometric View:

- L2: Ball constraint (smooth, no corners)
- L1: Diamond constraint (corners at axes)
- Solution often lands on corners $\rightarrow$ sparse

Why L1 induces sparsity

D.4 L1 from MAP Estimation

MAP with Laplace Prior:

$$\hat{\theta}_{MAP} = \arg\max_{\theta} \left[\log p(D|\theta) - \frac{1}{b} \|\theta\|_1 \right]$$

Equivalently:

$$\hat{\theta}_{MAP} = \arg\min_{\theta} [\mathcal{L} + \lambda \|\theta\|_1]$$

where $\lambda = \frac{\sigma^2}{b}$ (ratio of noise variance to prior scale).

Why Some Weights Become Exactly Zero:

- L1 gradient is $\pm\lambda$ (constant magnitude)
- Even small weights get constant "push" toward 0
- Eventually cross zero and stay there
- L2 gradient is $\lambda\theta$ (proportional to θ)
- Push decreases as $\theta \to 0$, never reaches 0

From Laplace prior to sparse solutions

D.5 Dropout as Approximate Bayesian Inference

Gal and Ghahramani (2016) Result:
Dropout training approximates variational inference in a deep Gaussian process.

Key Insight:

- Dropout = sampling from approximate posterior
- Each forward pass samples different "network"
- Prediction uncertainty = variance across samples

Monte Carlo Dropout:

1. Keep dropout enabled at test time
2. Run multiple forward passes
3. Mean = prediction, Variance = uncertainty

Practical Benefit:

- Free uncertainty estimates!
- No additional training required
- Useful for detecting out-of-distribution inputs

A theoretical justification for dropout

D.6 Dropout: Mathematical Analysis

Dropout During Training:

$$\tilde{\mathbf{h}}^{(l)} = \mathbf{m}^{(l)} \odot \mathbf{h}^{(l)}, \quad m_j^{(l)} \sim \text{Bernoulli}(1-p)$$

Expected Value:

$$\mathbb{E}[\tilde{h}_j^{(l)}] = (1-p) \cdot h_j^{(l)}$$

At Test Time (Inverted Dropout):
Scale by $(1-p)$ during training: $\tilde{\mathbf{h}}^{(l)} = \frac{1}{1-p}\mathbf{m}^{(l)} \odot \mathbf{h}^{(l)}$
No scaling needed at test time.

Implicit Regularization Effect:

- Prevents co-adaptation of neurons
- Each neuron must be useful independently
- Equivalent to training exponentially many networks
- Final model = average of all subnetworks

Why random dropout improves generalization

End of Mathematical Appendix

Complete Derivations for Neural Networks

For questions about derivations, consult:

Goodfellow, Bengio, Courville - "Deep Learning" (2016)
Bishop - "Pattern Recognition and Machine Learning" (2006)
Nielsen - "Neural Networks and Deep Learning" (online)